Slovenská komisia Matematickej olympiády
FPEDaS ŽU, Veľký diel, 010 26 Žilina

MATEMATICKÁ OLYMPIÁDA NA STREDNÝCH ŠKOLÁCH

53. ročník, školský rok 2003/2004

Riešenia úloh 1. kola kategórie P

Tento pracovný materiál nie je určený priamo študentom — účastníkom olympiády. Má pomôcť učiteľom na školách pri príprave konzultácií a pracovných seminárov pre riešiteľov súťaže, členom krajských výborov MO slúži ako podklad pre opravovanie úloh domáceho kola MO kategórie P. **Študentom možno tieto komentáre poskytnúť až po termíne stanovenom pre odovzdanie riešení úloh domáceho kola MO kategórie P** ako informáciu, ako bolo treba úlohy správne riešiť a pre ich odbornú prípravu na účasť v krajskom kole súťaže.

P–I–1

Na úvod pár slov pre tých, ktorým slová *teória grafov* príliš nehovoria. V našom chápaní graf je niekoľko bodov (ktoré budeme volať vrcholy), niektoré dvojice bodov sú pospájané čiarami (ktoré budeme volať hrany). Alebo ešte raz a formálnejšie, (neorientovaný) graf je dvojica $G = (V, E)$, kde V je množina vrcholov a $E \subseteq \{\{x, y\} \mid x, y \in V\}$ je množina neusporiadaných dvojíc vrcholov = hrán. Práve takýto graf máme v našej úlohe na vstupe – mestá v krajine sú vrcholy grafu a linky sú hrany medzi nimi.

Hovoríme, že graf je súvislý, ak sa po hranách dá prejsť z ľubovoľného vrcholu do ľubovoľného iného. Podľa zadania graf na vstupe je súvislý. Máme zistiť, či odstránenie niektorej hrany jeho súvislosť poruší. Hranu, ktorej odstránenie znesúvislí graf, voláme most.

Ako zistiť, či je graf súvislý? Existuje viacero algoritmov, sú známe pod spoločným názvom *ofarbovanie vrcholov* alebo tiež *prehľadávanie*. Základná myšlienka: začneme v nejakom vrchole grafu a postupne systematicky ofarbujeme všetky vrcholy, kam sa vieme dostať. Keď už nevieme nič ofarbiť, skončili sme. Teraz sa už len stačí pozrieť, či sú ofarbené všetky vrcholy. Samozrejme, treba vrcholy ofarbovať tak, aby sme určite žiaden nevynechali. Teraz si vysvetlíme jeden vhodný postup, nazývaný prehľadávanie do hĺbky.

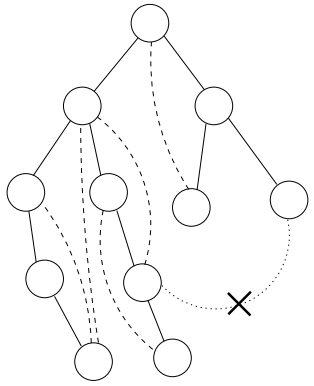
Prehľadávanie do hĺbky je podobné postupu, akým človek skúma neznáme mesto. Začneme tým, že sa postavíme do nejakého vrcholu a ofarbíme ho. Odteraz budeme farbiť vrcholy aj hrany grafu, kade chodíme. Ak z vrcholu, kde sme, vedie ešte nepoužitá hrana, vyberieme sa ňou ďalej. Ak sme prišli do ešte nenavštíveného vrcholu, ofarbíme ho a rekurzívne zavoláme prehľadávanie z neho. (Teda opäť sa snažíme nájsť nepoužitú hranu, atď.) Ak sme prišli do skôr navštíveného vrcholu (t.j. ofarbeného), okamžite sa po hrane, ktorou sme prišli, vrátime späť. No a posledný prípad je keď sme vo vrchole, z ktorého vedú samé ofarbené hrany. V takomto prípade sa len vrátime tou hranou, ktorou sme do neho prvýkrát prišli. Keď sa takto chceme vrátiť z vrcholu, kde sme začínali, prehľadávanie končí.

Zjavne takto prejdeme práve dvakrát (tam a späť) po každej z hrán, ku ktorým sa vieme dostať a navštívime všetky vrcholy, ku ktorým sa vieme dostať zo začiatočného vrcholu. Algoritmus je teda korektný a jeho časová zložitosť je $O(M + N)$. Dá sa ľahko rekurzívne implementovať, viď program na konci riešenia.

Najjednoduchším riešením pôvodnej úlohy by teda bolo postupne vyskúšať každú hranu odstrániť a pozrieť sa, či je výsledný graf ešte stále súvislý. Takéto riešenie by malo časovú zložitosť $O(M(M + N))$ – pre každú hranu potrebujeme spustiť jedno prehľadávanie.

My ukážeme algoritmus, ktorý bude bežať v čase $O(M + N)$ (teda optimálnom) a nájde v grafe všetky mosty.

Naše riešenie bude modifikáciou algoritmu prehľadávania do hĺbky. Predtým, než vysvetlíme samotné riešenie, potrebujeme si ukázať niekoľko vlastností prehľadávania do hĺbky. Spustime ho na našom súvislom grafe zo vstupu. Všimnime si tie hrany grafu, ktorými sme počas prehľadávania prišli do dovtedy nenavštíveného vrcholu. Týchto hrán je zjavne $N - 1$ (jedna pre každý vrchol okrem toho, v ktorom sme začínali). Graf nimi tvorený je strom, lebo je súvislý a neobsahuje kružnice. Tento strom budeme volať DFS strom (DFS = depth-first search = prehľadávanie do hĺbky). Vrchol, z ktorého sme začínali prehľadávať, budeme volať koreň. Z každého iného vrcholu x vedie po stromových hranách (hranách DFS stromu) do koreňa práve jedna cesta. Vrcholy na tejto ceste budeme volať predkami vrcholu x, vrchol x budeme volať ich potomkom. Špeciálne každý vrchol je sám sebe aj predkom, aj potomkom. Všetci potomkovia vrcholu x a stromové hrany medzi nimi tvoria podstrom s koreňom x.



Ostatné hrany teoreticky môžu byť dvoch typov. Ak hrana spája vrchol s nejakým jeho predkom alebo potomkom, budeme ju volať spätná, ostatné hrany budeme volať priečne. (Nech uv je hrana, ktorá nie je stromová. Všimnime si podstromy s koreňmi u, v. Sú dve možnosti – ak je jeden z nich podgrafom druhého, hrana uv je spätná, inak musia tieto podstromy byť disjunktné (prečo?) a hrana uv je priečna.) V DFS strome však žiadne priečne hrany nemôžu byť. Prečo? Sporom. Nech uv je priečna hrana. Bez ujmy na všeobecnosti nech sme počas prehľadávania do u prišli skôr. Všimnime si teraz okamih, keď sa počas prehľadávania ideme vrátiť späť z u. Aby uv bola priečna hrana, nesmeli sme doteraz v navštíviť (ináč by v bol potomok u a hrana uv by bola spätná). Ale v je sused u, preto by sme sa z u ešte nemali vracať späť ale mali by sme sa vybrať do v, spor.

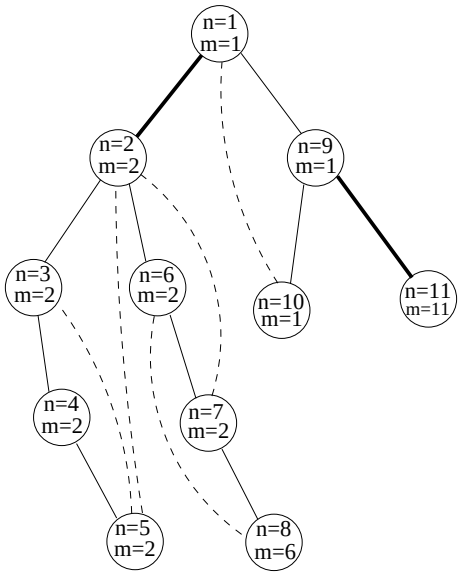
Takže hrany grafu môžeme rozdeliť na stromové a spätné. Zjavne ak hrana leží na nejakej kružnici, po jej odstránení graf zostane súvislý. Každá spätná hrana uv leží na kružnici (tvorenej hranou uv a cestou z u do v po DFS strome). Preto mostami môžu byť len stromové hrany. Každý most rozdeľuje graf na dve časti, v jednej z nich je koreň.

Predstavme si, že náš graf zavesíme za koreň. Teraz sa vyberme z koreňa dodola po stromových hranách. Majme konkrétnu stromovú hranu uv, pričom u je bližšie ku koreňu ako v. Kedy je uv most? Vtedy, keď ju nemáme ako obísť. Inými slovami ak sa z podstromu s koreňom v nevieme dostať do u (alebo ekvivalentne: do u alebo ľubovoľného jeho predka) bez použitia hrany uv.

Budeme teda pre každú hranu uv chcieť určiť, či existuje cesta z v do u alebo jeho predka, ktorá nepoužíva hranu uv. Hľadajme takú cestu, ktorá používa najmenší počet spätných hrán a zo všetkých takých ciest je najkratšia. Čo o nej vieme povedať? Posledná jej hrana bude určite spätná, lebo po stromových sa nad u nedostaneme. Všetky vrcholy na nej okrem posledného budú ležať v podstrome s koreňom v, lebo akonáhle sa dostaneme nad u, končíme. Do všetkých vrcholov v podstrome s koreňom v sa ale vieme dostať z v stromovými hranami. Ukázali sme teda nasledovné tvrdenie: AK nejaká hľadaná cesta existuje, TAK existuje aj taká, pri ktorej ideme najskôr niekoľkými stromovými a potom jednou spätnou hranou. Stačí nám teda pre každú hranu overiť, či existuje takáto cesta. Ako na to?

Počas prehľadávania číslujme vrcholy v poradí, v akom do nich prichádzame. Číslo vrcholu x budeme značiť $n(x)$. Zjavne všetky vrcholy v podstrome s koreňom v majú číslo väčšie ako

$n(u)$. Na druhej strane všetci predkovia u majú číslo menšie ako $n(u)$. Keby sme pre v vedeli najmenšie číslo vrcholu, do ktorého sa vieme dostať bez použitia hrany uv, vyhrali sme – uv je most práve vtedy, ak je toto číslo väčšie ako $n(u)$. Ukázali sme si ale, že nám stačí uvažovať cesty, ktoré idú najskôr niekoľkými stromovými hranami "dodola" a potom jednou spätnou "dohora". Budeme si teda pre každý vrchol priamo počas prehľadávania počítať najmenšie číslo vrcholu, do ktorého sa vieme z neho dostať takouto cestou.



Tým už máme výsledný algoritmus takmer hotový, zostáva si to už len celé zhrnúť. Prehľadávame náš graf do hĺbky a zároveň si pre každý vrchol x počítame dve čísla: $n(x)$ (koľký objavený vrchol to je) a $m(x) = \min\{n(y) \mid$ do y vedie z x cesta vyššie uvedeného tvaru$\}$. Ako počítať hodnotu $n(x)$ je zjavné. Hodnota $m(x)$ je minimum z $n(x)$, zo všetkých hodnôt $m(x_i)$ pre synov vrcholu x a zo všetkých hodnôt $n(y_i)$ vrcholov, do ktorých vedie z x spätná hrana. Hodnotu $m(x)$ teda vieme spočítať v okamihu, keď sa počas prehľadávania vraciame z vrcholu x. V tomto okamihu vieme aj rozhodnúť o hrane z x do jeho otca y, či je most – stačí porovnať hodnoty $m(x)$ a $n(y)$ (resp. $m(x)$ a $n(x)$).

```
program Mosty;

var G : array[1..100, 1..100] of integer;    { graf }
    deg, num, up : array[1..100] of integer; { stupne vrcholov a obe cisla pre ne }
    visited : array[1..100] of boolean;      { bol som uz v tomto vrchole? }
    N, M, C : integer;                       { pocet vrcholov, hran, navstivenych vrcholov }
    ok : boolean;

procedure Load;
var i, x, y : integer;
begin
  read(N, M); fillchar(deg, sizeof(deg), 0);
  for i := 1 to M do begin
    read(x, y); inc(deg[x]); G[x][deg[x]] := y; inc(deg[y]); G[y][deg[y]] := x;
  end;
end;

procedure DFS(v, parent : integer);
var i : integer;
begin
  visited[v] := true;
  num[v] := C; up[v] := C; inc(C); { nastavime obe cisla vo vrchole }
  for i := 1 to deg[v] do if not visited[G[v][i]] then begin
    DFS(G[v][i], v);
    if up[G[v][i]] < up[v] then up[v] := up[G[v][i]];
  end else begin { spaetna hrana }
    if G[v][i] <> parent then
```

```
        if num[G[v][i]] < up[v] then up[v] := num[G[v][i]];
    end;
    if (num[v] = up[v]) then ok := false; { hrana v-parent je most }
end;

begin
    Load;
    fillchar(visited, sizeof(visited), 0); C := 1; ok := true;
    DFS(1, 1);
    if ok then writeln('ANO') else writeln('NIE');
end.
```

P–I–2

Uvažujme ľubovolné poradie, v ktorom budú programátori pracovať, a pozrime sa na dva po sebe napísané programy, nech sú to i a j. Napísanie programu budeme odteraz volať udalosť. Prvá z našich udalostí, teda i, začne v čase T_0, bude trvať čas t_i a Janko za ňu zaplatí sumu $(T_0 + t_i)m_i$, druhá udalosť, j, začne v čase $T_0 + t_i$ (teda hneď po skončení i) a bude stáť $(T_0 + t_i + t_j)m_j$ – každého programátora platíme nielen za čas, kedy pracuje, ale od úplneho začiatku.

Keď to všetko spočítame, zistíme, že ak sa udalosti vykonajú v poradí i, j, Janko bude musieť za ne zaplatiť sumu $S_{i,j} = T_0(m_i + m_j) + t_i m_i + (t_i + t_j)m_j$.

Čo by sa stalo, keby sme v tomto poradí vymenili udalosti i a j? Podobne ako v predošlom prípade môžeme spočítať, koľko bude musieť Janko zaplatiť za tieto 2 udalosti. Za prvú (teda j) to bude $(T_0+t_j)m_j$ a za druhú $(T_0+t_j+t_i)m_i$, čo je spolu $S_{j,i} = T_0(m_i+m_j)+t_j m_j+(t_j+t_i)m_i$.

Porovnajme teraz tieto 2 výsledky. Označme si pre jednoduchosť ich spoločnú časť $A := T_0(m_i + m_j) + t_i m_i + t_j m_j$ a po triviálnych úpravách dostávame:

$$S_{i,j} = A + m_i m_j \frac{t_i}{m_i} \qquad S_{j,i} = A + m_i m_j \frac{t_j}{m_j}$$

Nás zaujíma to, ktorá z týchto hodnôt je menšia, ale to je zjavne tá, ktorá má menší pomer $\frac{t_k}{m_k}$. To teda znamená, že ak $\frac{t_i}{m_i} > \frac{t_j}{m_j}$, výmenou poradia týchto udalostí dosiahneme nižšiu výslednú sumu. (Zjavne zmena poradia 2 po sebe nasledujúcich udalostí neovplyvní sumu, ktorú zaplatíme ostatným programátorom.)

Z uvedeného vyplýva, že ak v nejakom poradí udalostí nájdeme 2 po sebe idúce, pričom tá prvá má väčší pomer $\frac{t_k}{m_k}$ ako tá druhá, ich výmenou získame nové poradie udalostí, ktoré je lacnejšie. Teda optimálne poradie bude také, v ktorom sú pomery $\frac{t_k}{m_k}$ usporiadané od najmenšieho po najväčší.

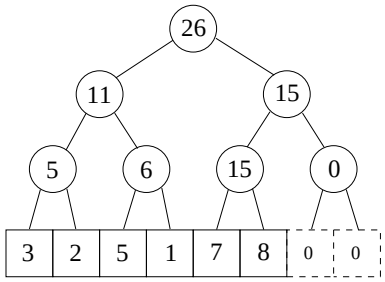
Samotný program je potom už jednoduchý – stačí udalosti utriediť vzostupne podľa pomeru $\frac{t_k}{m_k}$, čo vieme realizovať v čase $O(n \log n)$ napríklad algoritmom QuickSort.

```
program Scheduling;
type Tprg = record    m, t, idx : integer;      tm : real;    end;
var N, i, sum, t : integer;
    prg : array[1..10000] of Tprg;

procedure QSort(l, r : integer);
var y : Tprg;
    x : real;
```

```
    i, j : integer;
begin
    i := l; j := r; x := prg[(l + r) div 2].tm;
    repeat
        while prg[i].tm < x do inc(i); while x < prg[j].tm do dec(j);
        if i <= j then begin y := prg[i]; prg[i] := prg[j]; prg[j] := y; inc(i); dec(j); end;
    until i > j;
    if l < j then QSort(l, j); if i < r then QSort(i, r);
end;

begin
    read(N);
    for i := 1 to N do begin
        read(prg[i].m, prg[i].t);   prg[i].idx := i;   prg[i].tm := prg[i].t/prg[i].m;
    end;
    QSort(1, N);
    for i := 1 to N do writeln(prg[i].idx);
end.
```

P–I–3

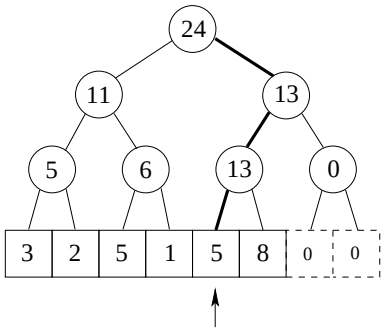
Pre jednoduchosť ďalších úvah zväčšme najskôr pole A tak, aby jeho veľkosť bola najbližšia väčšia mocnina dvoch. Tým sa pole A nanajvýš dvakrát predĺži, takže na časovej zložitosti výsledného algoritmu sa to neodrazí. Nech odteraz $N = 2^K$ je dĺžka predĺženého poľa.



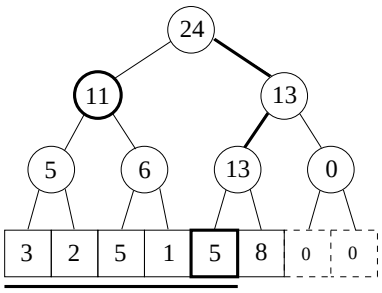
Predstavme si, že nad poľom A vybudujeme úplný binárny strom. Jeho listy budú zodpovedať jednotlivým políčkam poľa A, každý vyšší vrchol tohoto stromu bude zodpovedať nejakému intervalu v poli A (presnejšie bude zodpovedať políčkam, určeným listami z jeho podstromu). V každom vrchole stromu si budeme pamätať súčet čísel v príslušnom intervale poľa. Takejto dátovej štruktúre budeme hovoriť intervalový strom.

V najspodnejšej vrstve nášho stromu je N vrcholov, vo vyššej ich je $N/2$, v tretej odspodu $N/4$, atď. V celom našom strome je teda $2N-1$ vrcholov, preto budeme potrebovať na jeho zapamätanie pamäť veľkosti $\Theta(N)$ (čítaj: lineárnu). Pri predspracovaní poľa A budeme potrebovať túto pamäť naplniť, preto predspracovanie musí bežať v čase $\Omega(N)$ (čítaj: aspoň lineárnom). Ľahko nahliadneme, že v lineárnom čase vieme náš strom skutočne vytvoriť – stačí ho vytvárať zdola nahor.

Čo sa stane s naším stromom, keď zmeníme hodnotu prvku $A[j]$? Musíme zmeniť zapamätané hodnoty pre všetky intervaly, ktoré zmenený prvok obsahujú. Tie ale zodpovedajú práve vrcholom na ceste z j-teho listu do koreňa. Je ich teda $K + 1 = O(\log N)$. Zmeniť hodnotu v poli A teda vieme v logaritmickom čase.



Zmena hodnoty.



Počítanie súčtu.

Ešte ostáva ukázať, ako pomocou nášho stromu odpovedať na otázky zo zadania. Riešme najskôr jednoduchšiu úlohu: Akú hodnotu má súčet $S(x) = A[1] + \ldots + A[x]$? Pozrime sa do koreňa nášho stromu. Sú dve možnosti: Ak interval od 1 po x leží celý v ľavom podstrome, zavoláme sa rekurzívne naň. Ak nie, tak tento interval zaberá celý ľavý podstrom a kúsok pravého. Tak zoberieme súčet všetkých políčok v ľavom podstrome (ten máme spočítaný v ľavom synovi) a rekurzívne sa zavoláme na pravého syna a zvyšok intervalu.

Takto postupne v našom strome schádzame dole po ceste od koreňa do x-tého listu, na každej úrovni urobíme len konštantný počet operácií. Preto pre ľubovoľné x vieme hodnotu $S(x)$ spočítať v čase $O(\log N)$. To je ale všetko, čo potrebujeme vedieť – totiž $A[x]+\ldots+A[y] = S(y) - S(x-1)$ (pričom definujeme $S(0) = 0$).

Za pomoci intervalového stromu vieme teda každý príkaz spracovať v logaritmickom čase. Naše riešenie potrebuje lineárnu pamäť a lineárny čas na predspracovanie.

Najjednoduchšia implementácia intervalového stromu je uložiť ho v jednom poli, podobne ako haldu. Teda synovia vrcholu x sú na políčkach $2x$ a $2x+1$, pôvodné pole A začína na pozícii N. V praxi sa občas pamäťová zložitosť znižuje na polovicu tým, že si pamätáme len súčty v ľavých synoch, implementácia je potom ale trochu náročnejšia.

```
program Intervalovy_Strom;

var T : array[1..10000] of longint; { strom }
    oldN, N, prikaz, i : longint;
    x, y, pom : longint;

function Sucet(dlzka, koren, interval : longint) : longint;
{dlzka - dlzka intervalu, ktoreho sucet ratame
 koren - koren podstromu, v ktorom ho ratame
 interval - dlzka intervalu zodpovedajuceho podstromu (aby sme ju nemuseli ratat)}
begin
   if (dlzka = 0) then begin Sucet := 0; exit; end;
   if (interval = 1) then begin Sucet := T[koren]; exit; end;
   if (dlzka <= (interval div 2))
      then Sucet := Sucet(dlzka, 2 * koren, interval div 2)
      else Sucet := T[2 * koren]+
          Sucet(dlzka - (interval div 2), 2 * koren + 1, interval div 2);
end;

begin
   fillchar(T, sizeof(T), 0);
   read(oldN); N := 1; while (N < oldN) do N := N * 2; { upravime velkost pola }
   for i := 1 to oldN do read(T[N + i - 1]);
```

```
   for i := N − 1 downto 1 do T[i] := T[2 ∗ i] + T[2 ∗ i + 1];
   read(prikaz);
   while (prikaz > 0) do begin
      if (prikaz = 1) then begin        { menime hodnotu }
         read(x, y); i := x + N − 1; pom := y − T[i];
         while (i >= 1) do begin Inc(T[i], pom); i := i div 2; end;
      end else begin          { ratame sucet }
         read(x, y); writeln(Sucet(y, 1, N) − Sucet(x − 1, 1, N));
      end;
      read(prikaz);
   end;
end.
```

P–I–4

Najjednoduchšie riešenie je použiť štyri registre, v každom si počítať počet písmen jedného typu. Keď dočítame slovo, v R_0 máme počet prečítaných písmen a, v R_1 počet b, atď. Teraz budeme naraz zmenšovať hodnoty vo všetkých štyroch registroch. **Accept** zavoláme práve vtedy, ak register R_0 ostane najdlhšie neprázdny.

Počet použitých registrov sa dá ľahko zmenšiť na 3: Nech sme doteraz prečítali α písmen a, β písmen b, γ písmen c a δ písmen d. V registroch si budeme pamätať absolútne hodnoty výrazov $\alpha - \beta$, $\alpha - \gamma$, $\alpha - \delta$, v troch premenných si budeme pamätať ich znamienka. (Napr. 0 ak je v príslušnom registri nula, 1 ak je tam kladné číslo a 255 ak je záporné.) Takto v každom okamihu výpočtu vieme povedať, či bolo doteraz písmen a najviac – to platí práve vtedy, keď sú všetky tri znamienka, čiže všetky tri pamätané hodnoty kladné.

Naše riešenie bude potrebovať len dva registre. Dá sa ukázať (v tomto vzorovom riešení to nespravíme) že jeden register na riešenie tejto úlohy nestačí, preto naše riešenie bude vzhľadom na počet registrov optimálne.

V priebehu výpočtu si v R_0 budeme pamätať číslo $2^\alpha 3^\beta 5^\gamma 7^\delta$, register R_1 budeme používať na pomocné výpočty. Keď napríklad prečítame ďalšie písmeno b, za pomoci registra R_1 vynásobíme obsah registra R_0 tromi. Keď dočítame vstup, potrebovali by sme porovnať hodnoty α, β, γ a δ. Podobne ako v prvom riešení ich budeme naraz zmenšovať (to v našom prípade znamená deliť obsah R_0 vhodným číslom) a akceptujeme práve vtedy, ak nám na konci ostane kladná mocnina 2.

Samotný program je trochu dlhší, ale je priamočiarou implementáciou tejto myšlienky.

```
var vstup : char;
    i, zvysok, delitel : byte;
    esteA, esteB, esteC, esteD : byte; { ktore pismena este mame? }
begin
   Read(vstup);
   Inc(R_0); { na zaciatku su vsetky pocty 0, teda v R_0 mame 1 }
   while (vstup <> '$') do begin
      case vstup of
         'a' : while not Zero(R_0) do begin Dec(R_0); for i := 1 to 2 do Inc(R_1); end;
         'b' : while not Zero(R_0) do begin Dec(R_0); for i := 1 to 3 do Inc(R_1); end;
         'c' : while not Zero(R_0) do begin Dec(R_0); for i := 1 to 5 do Inc(R_1); end;
         'd' : while not Zero(R_0) do begin Dec(R_0); for i := 1 to 7 do Inc(R_1); end;
      end;
```

```
    while not Zero(R_1) do begin Dec(R_1); Inc(R_0); end;
    Read(vstup);
  end;

  esteA := 1; esteB := 1; esteC := 1; esteD := 1; { co este mame }
  while (esteA + esteB + esteC + esteD > 1) do begin { kym mame aspon 2 typy pismen }
    { skusime vydelit naraz vsetkymi prvocislami ktorymi moze byt delitelne }
    delitel := 1;  { 2*3*5*7 = 210 < 255, takze 1 bajt staci }
    if (esteA = 1) then delitel := delitel * 2; if (esteB = 1) then delitel := delitel * 3;
    if (esteC = 1) then delitel := delitel * 5; if (esteD = 1) then delitel := delitel * 7;

    zvysok := 0;
    while not Zero(R_0) do begin
      Dec(R_0); zvysok := (zvysok + 1) mod delitel; if (zvysok = 0) then Inc(R_1);
    end;
    { zistime cim bolo este delitelne – uz nulove esteX nas netrapia }
    if (zvysok mod 2 > 0) then esteA := 0; if (zvysok mod 3 > 0) then esteB := 0;
    if (zvysok mod 5 > 0) then esteC := 0; if (zvysok mod 7 > 0) then esteD := 0;
    { len na urychlenie: ak sa minuli pismena "a", nie je ich najviac }
    if (esteA = 0) then Reject;
    { vratime spaet do R_0 povodnu hodnotu }
    while (zvysok > 0) do begin Inc(R_0); zvysok := zvysok - 1; end;
    while not Zero(R_1) do begin Dec(R_1); for i := 1 to delitel do Inc(R_0); end;
    { vydelime R_0 tym cim je naozaj delitelne }
    delitel := 1; zvysok := 0;
    if (esteA = 1) then delitel := delitel * 2; if (esteB = 1) then delitel := delitel * 3;
    if (esteC = 1) then delitel := delitel * 5; if (esteD = 1) then delitel := delitel * 7;
    while not Zero(R_0) do begin
      Dec(R_0); zvysok := (zvysok + 1) mod delitel; if (zvysok = 0) then Inc(R_1);
    end;
    while not Zero(R_1) do begin Dec(R_1); Inc(R_0); end;
  end;
  { ostal najviac jeden typ pismen, je to "a"? }
  if (esteA = 1) then Accept;
end.
```

SLOVENSKÁ KOMISIA MATEMATICKEJ OLYMPIÁDY

53. ROČNÍK MATEMATICKEJ OLYMPIÁDY

Riešenia 1. kola kategórie P

Vydala IUVENTA
pre vnútornú potrebu Ministerstva školstva SR
Sadzba programom LaTeX
Zodpovedný redaktor M. Forišek