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

MATEMATICKÁ OLYMPIÁDA NA STREDNÝCH ŠKOLÁCH

53. ročník, školský rok 2003/2004

Riešenia úloh II. kola kategórie P

P-II-1

Najprv zadefinujme našu úlohu v jazyku teórie grafov. *Vrcholmi* nášho grafu budú budovy firmy, kým *hrany* budú reprezentovať možné prepojenia optickým káblom. Vyriešme našu úlohu najprv pre $K = 1$. V takom prípade je našou úlohou vybrať takú množinu hrán, aby všetky vrcholy boli navzájom poprepájané (nie nutne priamo). Takáto množina hrán sa nazýva *kostra grafu* a keďže chceme, aby súčet cien hrán v kostre bol čo najmenší, máme problém hľadania *najlacnejšej kostry*.

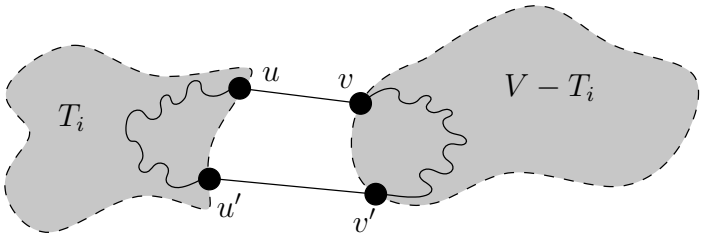
Hľadanie najlacnejšej kostry (Primov algoritmus). Algoritmus je založený na nasledujúcej myšlienke. Vrcholy grafu rozdelíme na dve skupiny: na pripojené a nepripojené. Na začiatku algoritmu si zvolíme ľubovoľný vrchol a prehlásime ho za pripojený. Ostatné vrcholy sú zatiaľ nepripojené. V každom kroku algoritmu pripojíme jeden vrchol k už doteraz vytvorenej sieti nasledujúcim spôsobom. Nájdeme najkratšiu hranu spájajúcu pripojený a nepripojený vrchol. Túto hranu pridáme do siete a jej druhý koniec sa stane pripojeným vrcholom. Skončíme, keď sú všetky vrcholy pripojené.

Aby náš algoritmus bol efektívny, potrebujeme vedieť rýchlo nájsť najkratšiu hranu spájajúcu pripojený a nepripojený vrchol. To spravíme tak, že pre každý nepripojený vrchol si pamätáme, ktorý z pripojených vrcholov je k nemu pripojený najkratšou hranou. Vždy, keď pridáme ku pripojeným vrcholom ďalší vrchol, musíme si informáciu o najbližších pripojených vrcholoch aktualizovať, tak že prezrieme nepripojené vrcholy a ak je novopripojený vrchol bližšie, našu informáciu zmeníme.

To, že výsledná množina hrán tvorí kostru, je zrejmé. Treba ale dokázať, že táto kostra je minimálna. Predstavme si ľubovoľnú minimálnu kostru (ďalej ju budeme nazývať MK) a porovnávajme ju s výsledkom nášho algoritmu (ďalej VNA).

Ak MK a VNA sú zhodné, VNA je minimálna kostra. Predpokladajme teda, že MK a VNA nie sú zhodné. Nech T_i je množina pripojených vrcholov po i-tom kroku nášho algoritmu. Zoraďme hrany vo VNA podľa toho, ako sme ich pridávali a nájdime prvú hranu, ktorá sa vyskytuje vo VNA, ale nevyskytuje sa v MK. Nech táto hrana bola pridaná v kroku $i + 1$ a nech spája vrchol $u \in T_i$ a vrchol $v \notin T_i$.

Pridajme hranu (u, v) do MK. Tým vznikne v MK cyklus, ktorý začína v T_i, prejde po hrane (u, v) von z T_i a potom sa vráti nejakou cestou späť do T_i (pozri obrázok 1). Na tejto ceste musí existovať aspoň jedna hrana (u', v'), ktorá má jeden koniec v T_i a druhý koniec mimo T_i. Cena tejto hrany musí byť aspoň taká, ako je cena hrany (u, v). V opačnom prípade by si náš algoritmus v kroku $i + 1$ musel vybrať hranu (u', v') namiesto hrany (u, v). Preto ak hranu (u', v') odoberieme z MK a pridáme hranu (u, v), cena MK sa nezvýši a MK bude naďalej minimálnou kostrou v grafe. Navyše VNA a MK sa teraz zhodujú na prvých $i + 1$ hranách. Takýmto spôsobom postupne prerobíme MK na VNA, pričom nezvýšime jej cenu, a teda VNA musí byť minimálna kostra.



Obr. 1: Pridaním hrany (u, v) vznikne v MK cyklus, ktorý začína v T_i, prejde po hrane (u, v) von z T_i a potom sa vráti nejakou cestou späť do T_i.

Riešenie pre všeobecné K. Doteraz sme predpokladali $K = 1$. Ak $K > 1$, nemusíme hranami pospájať všetky vrcholy. Na komunikáciu totiž môžeme využiť aj internet. Stačí, ak naša sieť bude pozostávať z K súvislých častí: v každej takejto súvislej časti vyberieme jeden vrchol, ktorý pripojíme na internet, a tak každý vrchol s každým bude môcť komunikovať.

Takúto sieť môžeme dostať napríklad tak, že z minimálnej kostry MK uberieme $K-1$ najdrahších hrán. Tým sa nám totiž MK rozpadne práve na K súvislých častí. Jediným problémom je ukázať, že takéto riešenie je skutočne najlacnejšie možné.

Označme teda P množinu $K - 1$ najdrahších hrán kostry MK. Po ich odobratí z MK dostaneme množinu hrán Q, ktorá pozostáva z K súvislých častí. Nech existuje lacnejšia množina hrán T, ktorá takisto tvorí sieť pozostávajúcu z K súvislých častí. Takúto množinu môžeme ľahko doplniť na kostru tak, že jednotlivé časti poprepájame pomocou $K - 1$ hrán. Navyše to ide aj tak, že použijeme iba hrany z kostry MK (zjednotenie kostry MK a množiny P je totiž súvislý graf) – označme túto množinu hrán S.

Všimnime si nasledujúce dva fakty:

- Množina hrán S určite nie je drahšia ako P, lebo obe obsahujú $K - 1$ hrán z kostry MK, ale P sme vybrali tak, aby obsahovala najdrahšie hrany.
- Z nášho predpokladu, množina hrán T je lacnejšia ako množina hrán Q.

Z toho ale vyplýva, že kostra $S \cup T$ je lacnejšia ako MK $= P \cup Q$, čo je spor s tým, že MK je minimálna kostra. Tým sme ukázali, že na vyriešenie úlohy stačí z MK odobrať $K - 1$ najdrahších hrán.

Časová zložitosť. Hľadanie najlacnejšej kostry v každom kroku pridá jeden vrchol do množiny pripojených, teda spraví celkovo $N-1$ krokov. V každom kroku najprv v čase $O(N)$ nájde najkratšiu hranu spájajúcu pripojený a nepripojený vrchol. Potom aktualizuje informáciu o najbližšom pripojenom vrchole pre všetky nepripojené vrcholy. Táto aktualizácia nás stojí opäť $O(N)$ operácií. Celková zložitosť je teda kvadratická, t.j. $O(N^2)$.

Z výslednej kostry potom potrebujeme ubrať $K - 1$ najdrahších hrán. To môžeme urobiť tak, že hrany kostry zostriedime (triedenie možno robiť v čase $O(N \log N)$, v tomto prípade nám ale stačí aj triedenie v čase $O(N^2)$). Celková časová zložitosť je teda $O(N^2)$.

P-II-2

Uvažujme, ktorý zo študentov má pracovať pri počítači v danom okamžiku t. Zrejme to musí byť jeden z tých študentov, ktorí už prišli do firmy, ale ešte nedokončili svoj program. O týchto študentoch budeme vravieť, že sú *aktívni* v čase t. Dokážme teraz, že pracovať vždy môžeme poslať toho z aktívnych študentov, ktorý musí odísť najskôr (nech je to študent a). Ak by totiž existoval rozvrh, v ktorom v čase t pracuje nejaký iný študent b, tak študent a sa ešte dostať

k počítaču v čase t' medzi t a časom odchodu t_a. Študent b však odchádza v čase $t_b \geq t_a$. Preto môžeme zostrojiť nový rozvrh, v ktorom necháme študenta a chvíľu pracovať na počítači v čase t a taký istý dlhý čas potom necháme študenta b pracovať na počítači v čase t'. Ak bol pôvodný rozvrh správny, aj zmodifikovaný rozvrh je správny, lebo každý pracuje rovnako dlho ako v pôvodnom rozvrhu a každý pracuje pred svojim odchodom.

Dokázali sme teda, že v každom okamžiku môže pracovať ten z aktívnych študentov, kto musí najskôr odísť. Kedy teda môže dôjsť k zmene obsadenia počítača? Buď keď príde nový študent a musí odísť skôr ako študent, ktorý práve pracuje (v tom prípade sa vystriedajú pri počítači), alebo ak študent, ktorý práve pracuje, dokončí svoj program a uvoľní počítač.

Náš algoritmus bude zostrojovať rozvrh obsadenia počítača od začiatku do konca. Študentov si utriedime podľa času príchodu a pre každého si pamätáme, koľko ešte potrebuje pracovať. Tiež si udržujeme množinu aktívnych študentov. V každom kroku algoritmu nájdeme najbližšiu udalosť, ktorá môže ovplyvniť rozvrh. Takouto udalosťou je buď príchod študenta alebo ukončenie práce práve pracujúceho študenta. V oboch prípadoch zaktualizujeme dátové štruktúry a potom nájdeme aktívneho študenta s najskorším odchodom a priradíme mu počítač. Ak tento študent už nemá dosť času na dokončenie programu, vyhlásime, že všetky programy sa nedajú dokončiť.

Triedenie študentov je možné spraviť v čase $O(N \log N)$. Počet udalostí je $2N$, lebo každý študent raz príde a raz dokončí program. Pre každú udalosť potrebujeme nájsť aktívneho študenta s najskorším odchodom. Ak musíme kvôli tomu porovnať všetkých aktívnych študentov, dostaneme algoritmus s časovou zložitosťou $O(N^2)$. Algoritmus sa však dá zefektívniť ak uložíme aktívnych študentov do binárnej haldy podľa času odchodu. V takom prípade spracovanie jednej udalosti trvá len $O(\log N)$ – ak nám niekto prišiel, vložíme ho do haldy, ak niekto skončil prácu, z haldy ho vyberieme. Následne sa pozrieme na minimum v halde – študenta, ktorý ide odteraz pracovať. Celková zložitosť algoritmu s použitím haldy je teda iba $O(N \log N)$.

```
program P_II_2;
type student =
   record
      prichod, odchod, zostalo : integer;
   end;
const Nekonecno = 10000;
var A          : array [1..1000] of student; { pole študentov }
   N           : integer;                    { počet študentov }
   Halda       : array [1..1000] of integer; { halda podľa odchodu }
   Halda_N     : integer;                    { počet študentov v halde }


procedure tried;
begin
   { vynechané kvôli úspore miesta }
end; { tried }

procedure vloz_do_haldy(student : integer);
var i, rodic, tmp : integer;
begin
   { vlož študenta na koniec haldy a posúvaj ho hore až kým
    nie je väčší ako jeho rodič }
   Halda_N := Halda_N + 1;
```

```
    Halda[Halda_N] := student;
    i := Halda_N;
    while i > 1 do begin
      rodic := i div 2;
      if A[Halda[i]].odchod < A[Halda[rodic]].odchod then begin
        tmp := Halda[i];
        Halda[i] := Halda[rodic];
        Halda[rodic] := tmp;
      end;
      i := rodic;
    end;
end; { vloz_do_haldy }

procedure vyber_z_haldy;
var i, dieta, tmp : integer;
begin
    { prvý prvok nahraď posledným a posúvaj ho dole až kým nie je
     menší ako obe deti }
    Halda[1] := Halda[Halda_N];
    Halda_N := Halda_N - 1;
    i := 1;
    while i * 2 <= Halda_N do begin
      dieta := i * 2;
      if (i * 2 + 1 <= Halda_N)
        and (A[Halda[i * 2 + 1]].odchod < A[Halda[dieta]].odchod) then begin
          dieta := i * 2 + 1;
        end;
      if A[Halda[i]].odchod > A[Halda[dieta]].odchod then begin
        tmp := Halda[i];
        Halda[i] := Halda[dieta];
        Halda[dieta] := tmp;
      end;
      i := dieta;
    end;
end;

procedure nacitaj;
var i : integer;
begin
    readln(N);
    for i := 1 to N do begin
      readln(A[i].prichod, A[i].odchod, A[i].zostalo);
    end;
end; { nacitaj }

function min(a, b : integer) : integer;
begin
    if a < b then min := a
    else min := b;
```

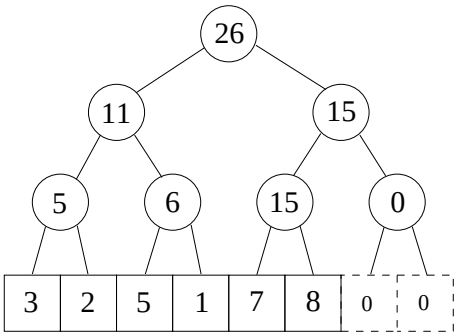
```
end;

var Pracuje, Skonci     : integer; { kto práve pracuje a kedy skončí }
    Stary_cas, Novy_cas : integer; { aktuálna a predchádzajúca udalosť }
    i                   : integer; { index do poľa A }

begin
    nacitaj;  { načítaj študentov do poľa A }
    A[N + 1].prichod := Nekonecno; { zarážka }
    tried;  { utrieď A podla položky „prichod“ }
    Pracuje := -1;      { nikto nepracuje }
    Skonci := Nekonecno;
    Halda_N := 0;  { inicializácia haldy }
    i := 1;
    Stary_cas := 0;
    while (i <= N) or (i = N + 1) and (Pracuje > 0) do begin
        { Nájdi novú udalosť }
        Novy_cas := min(A[i].prichod, Skonci);
        writeln('Cas ', Novy_cas);
        { Ak niekto pracoval pri počítači, vyhoď ho }
        if Pracuje > 0 then begin
            writeln(Novy_cas, ': študent ', Pracuje, ' od počítača.');
            A[Pracuje].zostalo := A[Pracuje].zostalo - (Novy_cas - Stary_cas);
            if A[Pracuje].zostalo = 0 then vyber_z_haldy;
        end;

        { Ak udalosť je príchod, vlož do haldy a choď na další príchod }
        if (i <= N) and (A[i].prichod < Skonci) then begin
            vloz_do_haldy(i);
            i := i + 1;
        end;

        { Nájdi nového študenta k počítaču }
        if Halda_N > 0 then begin
            Pracuje := Halda[1];
            Skonci := Novy_cas + A[Pracuje].zostalo;
            if Skonci > A[Pracuje].odchod then begin
                writeln('Rozvrh neexistuje!');
                exit;
            end;
            writeln(Novy_cas, ': študent ', Pracuje, ' k počítaču.');
        end
        else begin
            Pracuje := -1;
            Skonci := Nekonecno;
        end;
        Stary_cas := Novy_cas;
    end;
end.
```

P-II-3

Na príkaz `KOLKO c` by náš program mal odpovedať, v koľkých dovtedy zadaných intervaloch c leží. Na úvod uvedieme dve triviálne riešenia. Prvé je jednoducho si pamätať všetky dovtedy zadané intervaly a pri každom príkaze `KOLKO` ich všetky prejdeme. Pamäťová zložitosť takéhoto riešenia je $O(P)$, časová v najhoršom prípade až $O(P^2)$. (Príkaz `PRIDAJ` vieme spracovať v $O(1)$, ale na `KOLKO` potrebujeme v najhoršom až $O(P)$.) Trochu lepšie je použiť pomocné pole veľkosti $N+1$, v ktorom si pre každú pozíciu budeme pamätať počet intervalov, ktoré ju obsahujú. Jeden interval pridáme v čase $O(N)$, na otázku odpovieme v čase $O(1)$. Výsledná časová zložitosť je $O(NP)$, pamäťová $O(N)$.

Uvedomme si, čo vlastne potrebujeme zistiť, keď nám príde príkaz `KOLKO c`. Potrebujeme nájsť S – počet intervalov, ktoré začínajú na pozícií $\leq c$ a končia na pozícií $\geq c$. Nech $Z(x)$ je počet intervalov, ktoré začínajú na pozícií $\leq x$ a $K(x)$ je počet intervalov, ktoré končia na pozícií $\leq x$. Potom $S = Z(c) - K(c-1)$. (Totiž zlé intervaly, ktoré končia pred c sú zarátané aj v $Z(c)$, aj v $K(c-1)$, a teda ich do S nezarátame.) Stačilo by nám teda vedieť rýchlo zisťovať hodnoty $Z(x)$ a $K(x)$.

Budeme využívať myšlienku z domáceho kola – dátovú štruktúru, ktorú sme nazvali *intervalový strom*[1]. Pripomeňme si, o čo išlo: Predstavme si, že nad poľom A (ktorého dĺžku N sme zväčšili na najbližšiu mocninu dvoch) 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. Zmeniť hodnotu v poli A (a príslušne upraviť súčty vo vrcholoch stromu) vieme v čase $O(\log N)$, zistiť súčet ľubovoľného intervalu v poli A vieme takisto v čase $O(\log N)$.



$Z(c)$ je vlastne súčet počtov intervalov začínajúcich na pozíciách 0, 1, 2, ..., c. Budeme mať pole, v ktorom si tieto počty budeme pamätať a nad ním vybudovaný *intervalový strom*. Každé pridanie intervalu zmení jednu hodnotu v poli, túto zmenu vieme uskutočniť v čase $O(\log N)$. Analogicky budeme používať druhé pole (a druhý *intervalový strom*) pre počty intervalov, ktoré na jednotlivých pozíciách končia. Pomocou týchto dátových štruktúr vieme každú hodnotu Z aj K spočítať v čase $O(\log N)$.

Detailnejší popis oboch operácií s *intervalovým stromom* a jeho implementácie v poli nájdete vo vzorových riešeniach domáceho kola. Časová zložitosť nášho vzorového riešenia je $O(P \log N)$ a pamäťová $O(N)$. Všimnite si, že by nám stačilo udržiavať jedno pole, pridanie intervalu $\langle a, b \rangle$ by zodpovedalo napr. zväčšeniu hodnoty na pozícií a a zmenšeniu hodnoty na pozícií $b+1$.

[1] Nepliesť si s intervalmi zo zadania!

```
program Intervalovy_Strom_II;

var ZZ, KK : array[0..3000047] of longint; { stromy pre Z a K }
    N, oldN, a, b, c, kde : longint;
    prikaz, pom : char;

{T - pole, v ktorom ratame sucty ( vsimnite si: „var T“, nie „T“ – preco? )
 dlzka - dlzka intervalu, ktoreho sucet ratame
 koren - koren podstromu, v ktorom ho ratame
 interval - dlzka intervalu zodpovedajuceho korenu (aby sme ju nemuseli ratat)}
function Sucet(var T : array of longint; dlzka, koren, interval : longint) : longint;
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(T, dlzka, 2 * koren, interval div 2)
    else Sucet := T[2 * koren]+
      Sucet(T, dlzka - (interval div 2), 2 * koren + 1, interval div 2);
end;

begin
  fillchar(ZZ, sizeof(ZZ), 0);
  fillchar(KK, sizeof(KK), 0);
  readln(oldN); N := 1; while (N < oldN + 1) do N := N * 2; { upravime velkost pola }

  while not eof do begin
    read(prikaz); pom := prikaz; while (pom <> ' ') do read(pom);
    if (prikaz = 'P') then begin
      readln(a, b);
      kde := a + N; while (kde >= 1) do begin Inc(ZZ[kde]); kde := kde div 2; end;
      kde := b + N; while (kde >= 1) do begin Inc(KK[kde]); kde := kde div 2; end;
    end else begin
      readln(c);
      writeln(Sucet(ZZ, c + 1, 1, N) - Sucet(KK, c, 1, N));
    end;
  end;
end.
```

P–II–4

Predstavme si, že by sme okrem registrov mali k dispozícií jeden zásobník.[2] Potom by sme už úlohu ľahko vyriešili: Ideme po vstupnom slove zľava doprava, prečítané písmená vkladáme na zásobník. Keď teraz budeme vyberať písmená zo zásobníka, vychádzať budú v opačnom poradí ako boli vložené. Preto sa vrátime na začiatok slova a ideme porovnávať, či je slovo rovnaké odpredu aj odzadu. Vždy prečítame jedno písmeno zo vstupu, vyberieme jedno zo zásobníka a porovnáme ich. Skončíme, keď niekedy dostaneme dve rôzne písmená (slovo je zlé) alebo keď dočítame celé vstupné slovo (a teda je dobré).

[2] Zásobník je dátová štruktúra, ktorá podporuje operácie „vlož prvok“ a „vyber naposledy vložený prvok“.

Keby sme teda mali k dispozícií zásobník, máme úlohu vyriešenú. Zásobník si však vieme simulovať v jednom registri (za pomoci druhého)! Ako na to? Odteraz budú písmená a, b, c, d zodpovedať číslam 1, 2, 3, 4. Číslo v registri R_1 bude predstavovať náš zásobník – keď ho zapíšeme v sústave so základom 5, jednotlivé cifry budú predstavovať vložené hodnoty (cifra na mieste jednotiek bude naposledy vložená hodnota). Teda keď do prázdneho zásobníka uložíme postupne písmená a, c, b, a, bude v R_1 hodnota $a\times5^3+c\times5^2+b\times5+a = 1\times5^3+3\times5^2+2\times5+1 = 125+75+10+1 = 211$.

Ako ale s takýmto registrom-zásobníkom pracovať? Vložiť novú hodnotu x je jednoduché – za pomoci registra R_2 vynásobíme obsah R_1 piatimi a potom ho x-krát zväčšíme o 1. Takisto vybrať naposledy vloženú hodnotu nie je ťažké – je to presne opačná operácia. Vydelíme obsah registra R_1 piatimi. Zvyšok po delení je naposledy vložená hodnota, podiel (ktorý dostaneme v R_2) je zásobník bez tejto hodnoty.

Máme teda funkčné riešenie, ktoré potrebuje dva registre. Nejde to predsa len lepšie? S iba jedným registrom sa nám už nepodarí simulovať zásobník, musíme vymyslieť niečo iné.

Na úvod trocha terminológie: aktuálne písmeno sa bude pri našom riešení pohybovať sem a tam po vstupnom slove. Kvôli názornosti budeme namiesto „aktuálne je i-te písmeno vstupného slova“, resp. „presunieme akt. písmeno doľava/doprava“ hovoriť „stojíme na pozícií i“, resp. „ideme doľava/doprava“. Dĺžku vstupného slova budeme značiť n.

Predstavme si, že stojíme na pozícií i (pričom ale i si nepamätáme, v R_1 je nula). Chceli by sme písmeno na tejto pozícií porovnať s jemu zodpovedajúcim písmenom na pozícií $n+1-i$. Náš program však nevie n ani i. Ako na to? Písmeno na našej pozícií si zapamätáme v premennej. Teraz si zistíme i. Ideme doľava, kým neprídeme na začiatok vstupného slova a zvyšujeme R_1. Zodpovedajúce písmeno je i-te od konca. Nie je nič ľahšie ako prísť naň – prejdeme na koniec slova, potom zmenšujeme R_i a ideme doľava, kým v ňom nie je nula. Písmená porovnáme. Ak nesedia, končíme. Inak by sme sa chceli vrátiť späť na pozíciu, kde sme začínali. Na to použijeme presne rovnaký postup: Idúc doprava spočítame v R_1 potrebný počet krokov, presunieme sa na začiatok a spravíme rovnako veľa krokov. Skončili sme teda v rovnakej situácií ako sme začínali, len máme porovnané aktuálne písmeno s jemu zodpovedajúcim. Celý tento postup budeme volať *porovnanie*.

My by sme ale chceli postupne porovnať všetky navzájom si prislúchajúce dvojice písmen. To ale nie je problém. Začíname na prvom písmene vstupu. Spravíme *porovnanie*. Ak nie je prvé písmeno rovnaké ako posledné, skončili sme, inak pokračujeme. Presunieme sa doprava (na druhé písmeno) a spravíme ďalšie *porovnanie*. Takto pokračujeme až kým neporovnáme n-té písmeno s prvým (a nezistíme, že už nemáme porovnávané písmeno kam posunúť).

```
var vstup : char;
    pismeno : byte;
begin
  while (vstup <> '$') do begin
        { v R_1 mame nulu, zaciname porovnanie }
    if (vstup = 'a') then pismeno := 1;  if (vstup = 'b') then pismeno := 2;
    if (vstup = 'c') then pismeno := 3;  if (vstup = 'd') then pismeno := 4;
        { spocitame kde sme }
    while (vstup <> '$') do begin Inc(R_1); Left; end;
        { prejdeme na pravy koniec }
    Right; while (vstup <> '$') do Right;
        { prideme na zodpovedajuce policko }
    while not Zero(R_1) do begin Dec(R_1); Left; end;
        { kontrola }
```

```
      if (pismeno = 1) and (vstup <> 'a') then Reject;
      if (pismeno = 2) and (vstup <> 'b') then Reject;
      if (pismeno = 3) and (vstup <> 'c') then Reject;
      if (pismeno = 4) and (vstup <> 'd') then Reject;
         { navrat spaet }
      while (vstup <> '$') do begin Inc(R_1); Right; end;
      Left; while (vstup <> '$') do Left;
      while not Zero(R_1) do begin Dec(R_1); Right; end;
         { a posun na dalsie pismeno, ktore treba skontrolovat }
      Right;
   end;
   Accept; { vsetko sedelo }
end.
```

SLOVENSKÁ KOMISIA MATEMATICKEJ OLYMPIÁDY
53. ROČNÍK MATEMATICKEJ OLYMPIÁDY
Vzorové riešenia II. kola kategórie P

Autori príkladov: B. Brejová, M. Forišek, M. Pál, T. Vinař
Zodpovedný redaktor: M. Forišek
Náklad: vopred neznámy
Sadzba programom LaTeX