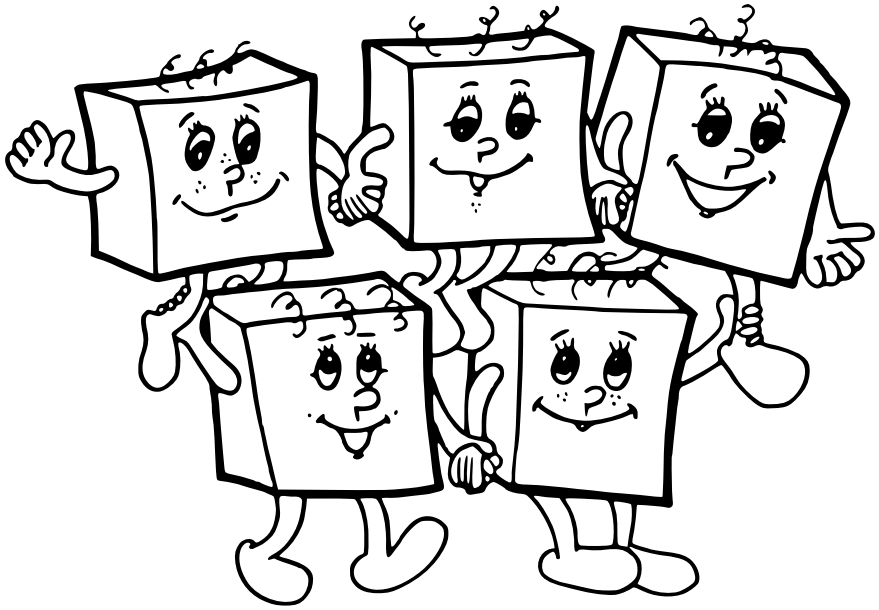
OLYMPIÁDA V INFORMATIKE
NA STREDNÝCH ŠKOLÁCH



23. ročník
školský rok 2007/08

riešenia domáceho kola
kategória B

- **Olympiáda v informatike** je od školského roku 2006/07 samostatnou súťažou. Predchádzajúcich 21 ročníkov tejto súťaže prebiehalo pod názvom **Matematická olympiáda, kategória P** (programovanie).
- Oficiálnu **webstránku** súťaže nájdete na `http://www.ksp.sk/oi/`.

Riešenia kategórie B

Tento materiál má pomôcť učiteľom na školách pri príprave riešiteľov domáceho kola. Taktiež slúži ako podklad pre opravovanie úloh členmi krajských výborov OI.

Žiakom možno tento materiál poskytnúť až po termíne stanovenom pre odovzdanie riešení úloh (15. novembra 2007). Po tomto termíne bude aj zverejnený na webstránke súťaže.

B-I-1 O spájaní polí

Pomalšie riešenie

Prvé, čo nás môže napadnúť, je začať tým, že presunieme prvky z B do A, a následne sa pokúsime pole A utriediť.

Ak by sme na to použili nejaký štandardný algoritmus na triedenie (napr. QuickSort), najlepšia časová zložitosť, akú môže náš program mať, by bola $O\Big((M+N)\cdot\log(M+N)\Big)$. A navyše by sme si museli pri implementácii triedenia poradiť bez pomocnej pamäte. V našom prípade však existujú aj efektívnejšie riešenia.

Vzorové riešenie

S úlohou „spojte dve utriedené postupnosti do jednej“ sa stretáme napríklad pri známom triediacom algoritme MergeSort.

Keby sme mohli použiť nové pole, bolo by to jednoduché. Predstavme si triedené prvky ako ľudí usporiadaných podľa výšky. Máme teda dva rady, z každého najmenší stojí úplne vpredu. Kto je teda najmenší celkovo? Predsa menší z tých dvoch predných. Toho teda pošleme, nech sa ide postaviť do výsledného poradia na prvé miesto. A pokračujeme: kto je najmenší z tých, čo zostali? No predsa jeden z tých dvoch, ktorí teraz stoja ako prví v radoch. Tak ich opäť porovnáme, menšieho pošleme „na výstup“, a tak dokola, až kým sa nám niektorý

rad neminie. Potom už len pridáme zvyšok druhého radu na koniec výslednej postupnosti.

Takéto riešenie by malo časovú zložitosť $O(M+N)$, teda lineárnu od počtu spracúvaných prvkov. (To preto, že každý krok vieme spraviť v konštantnom čase, a v každom kroku pošleme jeden prvok na výstup.)

Ak by sme ale chceli tento postup použiť na našu úlohu, narazíme na problém – nemáme nové pole, kam by sme výstup dávali.

Máme ale nejaké voľné miesto, a to na konci poľa A. Pomôžeme si teda tak, že výslednú postupnosť budeme zostrojovať od konca. Namiesto toho, aby sme porovnávali dva najmenšie (ešte nespracované) prvky, budeme vždy porovnávať dva najväčšie, a „výhercov“ budeme ukladať do poľa A od konca.

Uvedomte si, že voľné miesto v poli A sa nám pri takomto postupe predčasne neminie. Dokonca to vieme povedať aj presnejšie: V každom okamihu bude v poli A toľko voľného miesta, koľko ešte ostáva v poli B nespracovaných prvkov.

Poznámka

Úloha by bola riešiteľná v lineárnom čase dokonca aj vtedy, ak by sme nemali k dispozícii ani len to voľné miesto v poli A. Teda zadanie by bolo: „v poli A sú za sebou dve utriedené postupnosti, prvá má M prvkov a druhá N, bez použitia pomocných polí ich spojte do jednej“.

Tento problém je známy pod názvom Merge-in-place a je známych viacero rôznych algoritmov, ktoré ho riešia v lineárnom čase. Kvôli prísnejšiemu obmedzeniu na použitú pamäť však vo všetkých prípadoch však ide o algoritmy neporovnateľne komplikovanejšie ako naše riešenie.

Listing programu:

```
procedure merge(var A,B : array of longint; M,N : longint);
var kdeA,kdeB,kdeC : longint;
begin
   kdeA := M-1; kdeB := N-1; kdeC := M+N-1;
   { kym mame co porovnavat, porovnavame }
   while (kdeA >= 0) and (kdeB >= 0) do begin
      if (A[kdeA] > B[kdeB]) then begin
         A[kdeC] := A[kdeA]; dec(kdeA); dec(kdeC);
      end else begin
         A[kdeC] := B[kdeB]; dec(kdeB); dec(kdeC);
      end;
```

```
   end;
   { ked uz nemame co porovnavat, dopresuvame zvysok B }
   while (kdeB >= 0) do begin
      A[kdeC] := B[kdeB]; dec(kdeB); dec(kdeC);
   end;
end;

{ pre nazornost uvedieme aj program, ktory nasu proceduru pouziva }
var
   _A,_B:array[0..200000] of longint;
   i,_M,_N:longint;
begin
   read(_M); for i:=0 to _M-1 do read(_A[i]);
   read(_N); for i:=0 to _N-1 do read(_B[i]);
   merge(_A,_B,_M,_N);
   for i:=0 to _M+_N-1 do writeln(_A[i]);
end.
```

B-I-2 Krtkovou norou

Ak nevieme riešiť zadanú úlohu, skúsme najskôr vyriešiť jednoduchšiu. Zaoberajme sa teda najskôr prípadom, kde sú všetky chodbičky rovnako dlhé.

Riešenie pre chodbičky rovnakej dĺžky

V takomto prípade môžeme na nájdenie najvzdialenejších brlôžkov použiť postup známy pod menom *prehľadávanie do šírky*.

Myšlienka tohoto postupu je jednoduchá. Najskôr postupne spracujeme všetky brlôžky, ktoré sú od začiatočného vo vzdialenosti 1, potom všetky, ktoré sú vo vzdialenosti 2, a tak ďalej.

Presnejšie to bude fungovať takto: Budeme mať jedno pole, v ktorom budeme o každom brlôžku mať zaznamenané, či už vieme, ako je ďaleko. Okrem toho si budeme pamätať zoznam brlôžkov, ktoré čakajú na spracovanie.

Na začiatku si zaznamenáme, že do brlôžku krtka Vítka je vzdialenosť 0 a zaradíme tento brlôžok do zoznamu na spracovanie.

Teraz dokola opakujeme nasledujúci postup: Vyberieme zo zoznamu brlôžok, ktorý je v ňom najdlhšie, a ideme ho spracovať. Nech je tento brlôžok vo vzdialenosti d od brlôžku, kde sme začínali. Jeho spracovanie bude vyzerať nasledovne: Pozrieme sa postupne na všetky brlôžky, ktoré s ním susedia. Pre niektoré z nich už vzdialenosť vieme, tie necháme na pokoji. Ostatným brlôž-

kom (t.j. tým, do ktorých sme sa pri prehľadávaní doteraz nedostali) nastavíme vzdialenosť na $d + 1$ a zaradíme ich do zoznamu na spracovanie.

Zoznam brlôžkov na spracovanie si môžeme pamätať napríklad ako spájaný zoznam, ale najjednoduchšie je uložiť si ich v poli ako súvislý úsek. Dátová štruktúra, ktorá funguje ako náš zoznam brlôžkov (t.j. vždy vyberáme prvok, ktorý je v nej najdlhšie), sa volá *fronta*. Jej implementáciu pomocou poľa nájdete v listingu vzorového programu.

Aké efektívne je toto riešenie? Každý z N brlôžkov práve raz zaradíme do zoznamu, raz ho odtiaľ vyberieme a raz spracujeme. A každú z M chodbičiek spracujeme dvakrát (po jednom raze v každom koncovom brlôžku). Preto je časová zložitosť tohoto riešenia $O(N + M)$.

Listing programu:

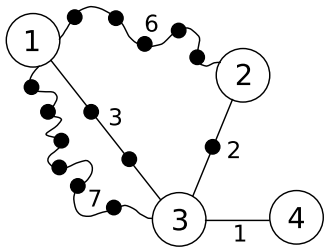
```
var G : array[1..1000,1..1000] of longint;
    stupne,fronta,vzdialenost : array[1..1000] of longint;
    bol : array[1..1000] of boolean;
    N,M,dalsi,i,a,b,h,kde,kam,zac,kon,max : longint;

begin
   readln(N,M);
   for i:=1 to M do begin
      readln(a,b);
      inc(stupne[a]); inc(stupne[b]);
      G[a][ stupne[a] ] := b;
      G[b][ stupne[b] ] := a;
   end;
   { pustime prehladavanie do sirky }
   { v kazdom okamihu plati, ze brlozky na spracovanie su v poli fronta[]
     na poziciach zac az (kon-1) }
   zac:=1; kon:=2; fronta[1]:=1; bol[1]:=true; vzdialenost[1]:=0;
   while (zac < kon) do begin
      kde := fronta[zac]; inc(zac);
      for i:=1 to stupne[kde] do begin
         kam := G[kde][i];
         if (not bol[kam]) then begin
            bol[kam]:=true; vzdialenost[kam]:=vzdialenost[kde]+1;
            fronta[kon]:=kam; inc(kon);
         end;
      end;
   end;
   { najdeme a vypiseme najvzdialenejsie brlozky }
   max:=0; for i:=1 to N do if (vzdialenost[i]>max) then max:=vzdialenost[i];
   for i:=1 to N do if (vzdialenost[i]=max) then writeln(i);
end.
```

Riešenie pôvodnej úlohy

V pôvodnej úlohe môžu mať chodbičky rôzne dĺžky, a to od 1 do 7. Tým sa ale nedáme zastrašiť. Presvedčíme krtkov, aby na každej chodbičke, ktorá je dlhšia ako 1, vyhrabali nové brlôžky, ktoré ju rozdelia na úseky dĺžky 1.

Napríklad pre sieť z príkladu v zadaní by výsledok snaženia krtkov vyzeral takto:



Takto sme dostali novú sieť chodbičiek, kde už majú všetky chodbičky rovnakú dĺžku. A pre túto sieť už zadanú úlohu vieme riešiť. (Len si treba dať pozor na to, že vypisovať sa majú len pôvodné komôrky, nie tie nové.)

Posledná otázka znie: nepokazili sme týmto trikom časovú zložitosť riešenia? Nuž, našťastie nie. Prečo? Nová sieť má nanajvýš $7\times$ toľko chodbičiek ako pôvodná. A na každej chodbičke pribudlo najviac 6 nových brlôžkov.

Brlôžkov je teda dokopy nanajvýš $N+6M$ a chodbičiek $7M$. Náš algoritmus teda bude mať časovú zložitosť $O(N+13M) = O(N+M)$.

Listing programu:

```
var G : array[1..1000,1..1000] of longint;
    stupne,fronta,vzdialenost : array[1..1000] of longint;
    bol : array[1..1000] of boolean;
    N,M,dalsi,i,a,b,h,kde,kam,zac,kon,max : longint;

begin
   readln(N,M);
   dalsi := N+1; { cislo, ktore dostane dalsi brlozok }
   for i:=1 to M do begin
      readln(a,b,h);
      kde := a;
      while (h>1) do begin { kopeme novy brlozok }
         inc(stupne[kde]); inc(stupne[dalsi]);
         G[kde][ stupne[kde] ] := dalsi;
         G[dalsi][ stupne[dalsi] ] := kde;
         kde := dalsi; inc(dalsi);
```

```
         dec(h);
      end;
      inc(stupne[kde]); inc(stupne[b]);
      G[kde][ stupne[kde] ] := b;
      G[b][ stupne[b] ] := kde;
   end;
   { pustime prehladavanie do sirky }
   zac:=1; kon:=2; fronta[1]:=1; bol[1]:=true; vzdialenost[1]:=0;
   while (zac < kon) do begin
      kde := fronta[zac]; inc(zac);
      for i:=1 to stupne[kde] do begin
         kam := G[kde][i];
         if (not bol[kam]) then begin
            bol[kam]:=true; vzdialenost[kam]:=vzdialenost[kde]+1;
            fronta[kon]:=kam; inc(kon);
         end;
      end;
   end;
   { najdeme a vypiseme najvzdialenejsie brlozky }
   max:=0; for i:=1 to N do if (vzdialenost[i]>max) then max:=vzdialenost[i];
   for i:=1 to N do if (vzdialenost[i]=max) then writeln(i);
end.
```

B-I-3 Kontrola XML

V prvom rade nás bude zaujímať len postupnosť tagov na vstupe, text medzi nimi môžeme pokojne ignorovať.

Na otváracie a zatváracie tagy sa môžeme dívať ako na ľavé a pravé zátvorky rôznych typov. Potom podmienka, že tagy sa nesmú prekrývať, nadobudne jasnejší význam: XML dokumenty sa podobajú dobre uzátvorkovaným výrazom.

Ako teda budeme kontrolovať, či je náš XML dokument korektný? Tagy budeme spracúvať v poradí, v akom sa v ňom vyskytujú. V každom okamihu si budeme pamätať, ktoré tagy sme už otvorili a ešte nezavreli. Ak je nasledujúci spracúvaný tag otvárací, len ho pridáme na koniec pamätaného zoznamu. Naopak, ak je zatvárací, môže nastať hneď niekoľko problémov.

V prvom rade ten najbežnejší problém: posledný otvorený tag bol iný ako ten, ktorý práve spracúvame (`<a></b>`).

Mohlo sa tiež stať, že nie je otvorený vôbec žiaden tag (`<a></a></x>`).

Jediná dobrá situácia je, ak posledný otvorený a ešte nezavretý tag sedí so zatváracím tagom, ktorý práve spracúvame. Vtedy tento tag odstránime zo zoznamu, ktorý si pamätáme (`<a><b><c></c></b>`).

Ak sa takto dočítame až na koniec XML dokumentu, ešte potrebujeme skontrolovať jednu vec: či nezostali niektoré tagy otvorené (<a>).

Zoznam otvorených tagov

Ako si má náš program pamätať zoznam otvorených tagov tak, aby sme nové tagy vedeli spracúvať čo najefektívnejšie?
Všimnime si, že potrebujeme robiť nasledujúce operácie:

– pridaj nový tag na koniec zoznamu
– zisti, aký tag je na konci zoznamu
– odstráň posledný tag zo zoznamu

Na toto je ako stvorená dátová štruktúra *zásobník*. V našom programe na uloženie zásobníka používame statické pole a jednu premennú, v ktorej si pamätáme aktuálny počet uložených tagov.

Pár slov k implementácii

Náš program spracúva vstup po znakoch. Vždy, keď narazí na znak <, zavolá procedúru, ktorá sa pokúsi sa načítať názov tagu. Ak sa jej to nepodarí, program okamžite ukončíme. Ak áno, tag spracujeme podľa vyššie uvedeného postupu.

Všimnite si, že vďaka použitiu tejto procedúry sa hlavný program zjednodušil v podstate len na samotné spracúvanie tagov.

Listing programu:

```
function je_znak(ch : char) : boolean;
begin je_znak := (ch>='a') and (ch<='z'); end;

procedure urcite_nacitaj(var ch : char);
begin
   if eof then begin writeln('nie'); halt; end;
   read(ch);
end;

procedure nacitaj_tag(var zaciatok : boolean; var nazov : string);
var ch : char;
begin
   nazov:=''; zaciatok:=true;
   urcite_nacitaj(ch);
   if (ch='/') then begin zaciatok:=false; urcite_nacitaj(ch); end;
   while true do begin
      if (ch='>') then break;
```

```
      if (not je_znak(ch)) then begin writeln('nie'); halt; end;
      nazov := nazov + ch;
      if (length(nazov) > 8) then begin writeln('nie'); halt; end;
      urcite_nacitaj(ch);
   end;
   if (nazov='') then begin writeln('nie'); halt; end;
end;

var zasobnik : array[1..10000] of string;
    pocet : longint;
    ch : char;
    zaciatok : boolean;
    nazov : string;

begin
   pocet := 0;
   while not eof do begin
      read(ch);
      if (ch='<') then begin
         nacitaj_tag( zaciatok, nazov );
         if (zaciatok) then begin
            { vlozime novy tag na zasobnik }
            inc(pocet);
            zasobnik[pocet] := nazov;
         end else begin
            { porovname tag s vrchom zasobnika }
            if (pocet=0) then begin writeln('nie'); halt; end;
            if (zasobnik[pocet]<>nazov) then begin writeln('nie'); halt; end;
            dec(pocet);
         end;
      end;
   end;
   if (pocet>0) then begin writeln('nie'); halt; end;
   writeln('ano');
end.
```

B-I-4 Rákosie

Úlohou je nájsť odrodu rákosia, ktorej steblá majú 2^n (mŕtvych) buniek. Asi prvá myšlienka, ktorá nám môže napadnúť, je použiť pravidlo $A \to AA$, aby sa nám bunky množili. Rákosie by mohlo rásť vo fázach, pričom v každej ďalšej by sa zdvojnásobil počet áčok.

Jediným problémom je teda dohliadnuť na to, aby sa v každej fáze počet áčok naozaj zdvojnásobil. Všimnite si, že samotné pravidlo $A \to AA$ je problematické, pretože nemôžeme ovplyvniť, na ktoré áčka sa použije. Môže sa použiť na každé áčko raz (tak by sme to chceli), ale môže sa napríklad použiť aj opakovane vždy na prvé áčko (tak o chvíľu stratíme pojem o ich počte).

Aby sme do rastu zaviedli trochu disciplíny, budeme mať jednu špeciálnu bunku, ktorá sa bude po steble „pohybovať“. Návod, ako na to, nám dáva už druhý príklad zo študijného materiálu. Vezmime si také pravidlo $AB \to BA$. Presvedčte sa, že keby sme mali steblo $ABBBBB$, použitím tohto pravidla by A postupne preliezlo cez všetky béčka.

Naša špeciálna bunka sa bude volať M, pretože pri pohybe „doprava“ (od koreňa) bude navyše *množiť* áčka. Dosiahneme to pravidlom $Ma \to aaM$. Napríklad, ak sú na začiatku áčka štyri, po prechode bude áčok osem:

$$Maaaa \Rightarrow aaMaaa \Rightarrow aaaaMaa \Rightarrow aaaaaaMa \Rightarrow aaaaaaaaM$$

Čo sa stane, keď M príde na koniec? Máme dve možnosti: buď nám už vyrástlo dosť dlhé steblo a chceli by sme skončiť, alebo by sme sa mali vrátiť a znovu a znovu dĺžku zdvojnásobovať.

Ako ale vôbec zistíme, či už sme na konci? Odpoveď je jednoduchá: na začiatku použijeme pravidlo $Z \to BaME$, ktorým vytvoríme prvé áčko, špeciálnu „množiacu“ bunku a navyše si aj „označíme“ začiatok a koniec stebla (B ako begin a E ako end). Teraz už budeme vedieť, či sme na konci stebla.

Ako sa vrátime? Nemôžeme pridať pravidlo $aM \to Ma$, pretože potom by M mohlo chodiť striedavo aj doprava aj doľava (a nevedeli by sme ho „donútiť“ počet buniek práve zdvojnásobiť; M-ko by si „mohlo robiť, čo chce“; ak ideme doprava, musíme ísť až na koniec; ak sa vraciame, musíme sa vrátiť až na začiatok). Preto M-ko, ktoré je na konci (označenom E-čkom) zmeníme na R, ktoré bude chodiť *iba* doľava. Inými slovami, písmenom R budeme označovať našu špeciálnu bunku M, keď sa vracia. Keď R príde na začiatok (označený B-čkom), zmení sa späť na M. Máme teda pravidlá

$$Z \to BaME \quad Ma \to aaM \quad ME \to RE \quad aR \to Ra \quad BR \to BM$$

Čo urobíme, keď budeme chcieť skončiť? Po predchádzajúcej úvahe by to už nemal byť problém: Zmeníme M napríklad na H (ako halt). Bunka H sa postará o to, aby v steble ostali iba áčka, t.j. „zmaže“ B, E, a nakoniec aj samu seba. Dosiahneme to pomocou pravidiel

$$ME \to H \quad aH \to Ha \quad BH \to \varepsilon.$$

Úplne všetky genetické pravidlá našej odrody teda vyzerajú takto:

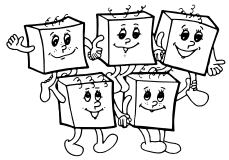
Z	$\rightarrow$	$BaME$	(označíme začiatok a koniec)
Ma	$\rightarrow$	aaM	(množíme áčka)
ME	$\rightarrow$	$RE \mid H$	(na konci sa otočíme, alebo končíme)
aR	$\rightarrow$	Ra	(vraciame sa)
BR	$\rightarrow$	BM	(otočíme sa, ideme množiť)
aH	$\rightarrow$	Ha	(ideme zmazať B)
BH	$\rightarrow$	ε	(ostanú len áčka)

Ukážme si príklad, ako steblo narastie na 4 áčka:

$$\begin{aligned} Z &\Rightarrow BaME \Rightarrow BaRE \Rightarrow BRaE \Rightarrow BMaE \\ &\Rightarrow BaaME \Rightarrow BaaRE \Rightarrow BaRaE \Rightarrow BRaaE \\ &\Rightarrow BMaaE \Rightarrow BaaMaE \Rightarrow BaaaaME \Rightarrow BaaaaH \\ &\Rightarrow BaaaHa \Rightarrow BaaHaa \Rightarrow BaHaaa \Rightarrow BHaaaa \Rightarrow aaaa \end{aligned}$$

Na záver len malú poznámku: na trojicu M, R, H sa dá pozerať ako na *jednu* špeciálnu bunku, ktorá behá po steble. Pri tom si „pamätá“, čo práve robí (či množí bunky, vracia sa, alebo ukončuje steblo). V našich genetických pravidlách sme určili, ako túto činnosť vykonáva, a ako a kedy svoju činnosť mení.

Olympiáda
v Informatike



2007/08
domáce kolo
riešenia kategórie B

SLOVENSKÁ KOMISIA OLYMPIÁDY V INFORMATIKE

DVADSIATY TRETÍ ROČNÍK OLYMPIÁDY V INFORMATIKE

Vydala IUVENTA s finančnou podporou Ministerstva školstva SR
Náklad: 400 výtlačkov
Zodpovedný redaktor: Michal Forišek
Sadzba programom LaTeX