52. ročník Matematické olympiády – 2002/2003

Řešení úloh krajského kola kategorie P

P-II-1 Tvůrci hvězd

Řešení této úlohy se skládá ze dvou částí. Nejdříve spočítáme, kde se musí nacházet střed symetrie, pokud jsou hvězdy skutečně rozmístěny symetricky. V druhé části pak ověříme, zda jsou hvězdy skutečně symetrické podle spočteného středu.

Střed symetrie spočteme velmi snadno. Pokud střed symetrie má souřadnice (x_s, y_s, z_s), tak se hvězda o souřadnicích (x, y, z) musí zobrazit na souřadnice $(2 \cdot x_s - x, 2 \cdot y_s - y, 2 \cdot z_s - z)$ (aby střed symetrie ležel uprostřed úsečky mezi hvězdou a jejím obrazem). Když sečteme odpovídající souřadnice hvězdy a jejího obrazu, dostaneme $(2 \cdot x_s, 2 \cdot y_s, 2 \cdot z_s)$. Pokud jsou hvězdy rozloženy symetricky, tak tedy sečtením odpovídajících souřadnic všech N hvězd dostaneme $(N \cdot x_s, N \cdot y_s, N \cdot z_s)$ (se souřadnicemi každé hvězdy jsme přičetli i souřadnice hvězdy, která byla jejím obrazem), z čehož již triviálně spočteme očekávaný střed symetrie (x_s, y_s, z_s).

Nyní potřebujeme ještě ověřit, že hvězdy jsou skutečně symetrické podle spočteného středu. Abychom ověřování zjednodušili (a zrychlili), setřídíme si hvězdy lexikograficky podle jejich souřadnic (tzn. v takovém pořadí, že $(x_1, y_1, z_1) < (x_2, y_2, z_2)$ pokud $x_1 < x_2$ nebo $x_1 = x_2$ a $y_1 < y_2$ nebo $x_1 = x_2, y_1 = y_2$ a $z_1 < z_2$). Nyní si všimněme, že pokud jsou hvězdy symetricky rozloženy, tak díky vztahům mezi souřadnicemi hvězdy a jejího obrazu platí, že i-tá hvězda se musí zobrazit na $N - i$-tou hvězdu. Pro ověření symetrie tedy stačí projít seřazené pole hvězd a ověřit, že i-tá hvězda se skutečně zobrazí na $N - i$-tou hvězdu.

Zjištění středu symetrie nám zřejmě zabere čas $O(N)$, třídění pole s hvězdami $O(N \log N)$ a ověřování symetrie hvězd $O(N)$. Celkově má tedy algoritmus časovou složitost $O(N \log N)$. Paměťová složitost algoritmu je $O(N)$.

```
program Hvezdy;
const
  MAXN = 100;
type
  Hvezda = record
           x, y, z : Integer;
         end;
  PoleHvezd = Array[1..MAXN] of Hvezda;
var
  N : Integer;                  {Pocet Hvezd}
  H : PoleHvezd;                {Jednotlive hvezdy}
  Stred : Hvezda;               {Spocitany stred symetrie}

{Nacte vstup}
procedure Nacti;
var
  i : Integer;
begin
  Write('Pocet hvezd: ');
  ReadLn(N);

  for i := 1 to N do begin
    Write('Hvezda: ');
    ReadLn(H[i].x, H[i].y, H[i].z);
  end;
end;

{Nalezne stred symetrie}
procedure NajdiStred(var Stred : Hvezda);
var
  i : Integer;
```

```
  xs, ys, zs : Integer;
begin
  xs := 0;
  ys := 0;
  zs := 0;
  {Spocteme prumer z kazde souradnice}
  for i := 1 to N do begin
    xs := xs + H[i].x;
    ys := ys + H[i].y;
    zs := zs + H[i].z;
  end;
  Stred.x := xs div N;
  Stred.y := ys div N;
  Stred.z := zs div N;
end;

{Porovna souradnice dvou hvezd}
function PorovnejHvezdy(A, B : Hvezda) : Integer;
begin
  if A.x <> B.x then begin
    PorovnejHvezdy := A.x - B.x;
    Exit;
  end;
  if A.y <> B.y then begin
    PorovnejHvezdy := A.y - B.y;
    Exit;
  end;
  if A.z <> B.z then begin
    PorovnejHvezdy := A.z - B.z;
    Exit;
  end;
  PorovnejHvezdy := 0;
end;

{Prohodi dva prvky haldy}
procedure Prohod(var Halda : PoleHvezd; A,B : Integer);
var
  Tmp : Hvezda;
begin
  Tmp := Halda[A];
  Halda[A] := Halda[B];
  Halda[B] := Tmp;
end;

{Vlozi prvek H do haldy}
procedure Vloz(var HT : Integer; var Halda : PoleHvezd; H : Hvezda);
var
  S : Integer;
begin
  Inc(HT);
  Halda[HT] := H;
  S := HT;
  while (S > 1) and (PorovnejHvezdy(Halda[S], Halda[S div 2]) < 0) do begin
    Prohod(Halda, S, S div 2);
    S := S div 2;
```

```
  end;
end;

{Vybere prvni hvezdu z haldy}
function Vyber(var HT : Integer; var Halda : PoleHvezd) : Hvezda;
var
  S, T : Integer;
begin
  Vyber := Halda[1];
  Halda[1] := Halda[HT];
  Dec(HT);
  S := 1;
  while 2*S <= HT do begin
    T := 0;
    if PorovnejHvezdy(Halda[S], Halda[2*S]) > 0 then
      T := 2*S;
    if (2*S+1 <= HT) and (PorovnejHvezdy(Halda[S], Halda[2*S+1]) > 0) then
      if PorovnejHvezdy(Halda[2*S], Halda[2*S+1]) > 0 then
        T := 2*S+1;
    if T > 0 then begin
      Prohod(Halda, S, T);
      S := T;
    end
    else
      S := HT;
  end;
end;

{Setridi hvezdy podle souradnic}
procedure Setrid;
var
  Halda : PoleHvezd;    {Halda na trideni}
  HT : Integer;         {Pocet prvku v halde}
  i : Integer;
begin
  HT := 0;
  {Vlozime prvky do Haldy}
  for i := 1 to N do
    Vloz(HT, Halda, H[i]);
  {Nyni je vybereme v setridenem poradi}
  for i := 1 to N do
    H[i] := Vyber(HT, Halda);
end;

{Overi, zda jsou hvezdy symetricke podle daneho stredu}
function Over(Stred : Hvezda) : Boolean;
var
  i : Integer;
begin
  Setrid;       {Setridi hvezdy podle souradnic}

  for i := 1 to (N+1) div 2 do begin
    {Je odpovidajici hvezda polozena symetricky?}
    if (Stred.x - H[i].x <> H[N-i+1].x - Stred.x) or
       (Stred.y - H[i].y <> H[N-i+1].y - Stred.y) or
```

```
       (Stred.z - H[i].z <> H[N-i+1].z - Stred.z) then begin
      Over := False;
      Exit;
    end;
  end;
  Over := True;
end;

begin
  Nacti;
  NajdiStred(Stred);
  if Over(Stred) then
    WriteLn('Stred symetrie lezi na pozici ', Stred.x, ', ', Stred.y, ', ', Stred.z, '.')
  else
    WriteLn('Hvezdy nejsou symetricke podle zadneho stredu.');
end.
```

P-II-2 Knihovna

Nejprve učiňme následující pozorování: Nechť s_0 je šířka optimální skříně a nechť má tato skříň p poliček. Potom existují výšky $w_1 \geq \ldots \geq w_p$ poliček a rozmístění knih do skříně se šířkou s_0 a poličkami výšky $w_1, \ldots, w_p$ takové, že výšky knih v této skříni v pořadí zeshora dolů a v každé poličce zleva doprava tvoří nerostoucí posloupnost (první polička je ta nejvýše umístěná).

První část pozorování, o existenci výšek $w_1 \geq \ldots \geq w_p$, je jednoduchá – pokud výšky poliček ve skříni seshora dolů netvoří nerostoucí posloupnost, stačí poličky (i s jejich obsahem) ve skříni přeuspořádat. Nyní dokážeme, že existuje rozmístění knih ve skříni takové, že výšky knih tvoří nerostoucí posloupnost. Bez újmy na obecnosti můžeme předpokládat, že $v_1 \geq \ldots \geq v_N$. Uvažme rozmístění knih do skříně takové, že první polička obsahuje s_0 nejvyšších knih, druhá s_0 nejvyšších knih mezi zbylými knihami, atd., a v každé z poliček výšky knih tvoří nerostoucí posloupnost. Tvrdíme, že výška nejvyšší knihy v i–té poličce je nejvýše w_i, tj. $v_{(i-1)s_0+1} \leq w_i$. Pokud tomu tak není, pak $(i-1)s_0+1$-tá kniha musí být v optimálním řešení na jedné z prvních $i-1$ poliček, ale pak některá z $(i-1)s_0$ nejvyšších knih (řekněme ta s výškou v_k, $1 \leq k \leq (i-1)s_0$) není v optimálním řešení na jedné z prvních $i-1$ poliček – je tedy na j-té poličce, $j \geq i$. Potom ale $w_j \leq v_k$ a tedy $w_i \leq v_{(i-1)s_0+1}$, což je požadovaná nerovnost.

Všimněme si, že jsme v předchozím odstavci vlastně dokázali, že ve výše popsaném optimálním řešení jsou všechny poličky až na tu poslední plné, tj. obsahují přesně s_0 knih. Základem našeho programu bude funkce `existuje(s:integer)`, která pro danou šířku s rozhodne, zda existuje knihovna maximální výšky 250 cm a šířky s, do které lze umístit všechny knihy. Optimální hodnotu s_0 nalezneme pak metodou půlení intervalu, kterou lze nalézt v popisu řešení úlohy P-I-2 domácího kola. Samotná funkce zvolí za výšku i-té poličky výšku $v_{(i-1)s_0+1}$, což je výška nejvyšší knihy, kterou uložíme do i–té poličky v řešení popsaném v minulém odstavci. Naše funkce z výšek jednotlivých poliček snadno spočte výšku celé knihovny a ověří, zda je nejvýše 250 cm.

Nyní odhadněme časové a paměťové nároky výše popsaného programu. Nejprve potřebujeme setřídit N čísel, což lze učinit užitím některého ze standardních algoritmů v čase $O(N \log N)$. Časová složitost funkce `existuje` je $O(N/s)$, neboť je v ní potřeba sečíst $\lceil N/s \rceil$ čísel. Odtud již plyne, že časové nároky celého našeho algorithmu jsou majorizovány funkcí $O(N \log N)$. Pokud si uvědomíme, že $s = N/2$ při prvním volání funkce `existuje`, $s = N/4$ při druhém, atd., pak lze časové nároky algoritmu bez úvodního setřídění výšek knih dokonce odhadnout funkcí $O(N)$. Paměťové nároky algoritmu lze odhadnout funkcí $O(N)$, neboť potřebujeme pole velikosti N na uložení výšek jednotlivých knih.

```
program knihovna;
const MAXN=100;
      VYSKA_MISTNOSTI=250;
var vyska: array[1..MAXN] of word;      { výšky knih }
    n: word;                            { počet knih }
procedure utrid_vysky(i1,i2:word);
  { quicksort }
  var pivot: word;
      w: word;
```

```
      j1, j2: word;
  begin
    if i1>=i2 then exit;
    pivot:=vyska[(i1+i2) div 2];
    j1:=i1; j2:=i2;
    while (j1<j2) do
      begin
        while (vyska[j1]>pivot) do inc(j1);
        while (vyska[j2]<pivot) do dec(j2);
        w:=vyska[j1]; vyska[j1]:=vyska[j2]; vyska[j2]:=w;
        inc(j1); dec(j2);
      end;
    utrid_vysky(i1,j2);
    utrid_vysky(j1,i2);
  end;
function existuje(s:word):boolean;
  var v:word;
      i:word;
  begin
    v:=1;
    i:=1;
    repeat
      v:=v+vyska[i]+1;
      i:=i+s;
    until i>n;
    existuje:=v<=VYSKA_MISTNOSTI
  end;
var i:word;
    s1,s2:word;
    v:word;
begin
  readln(n);
  for i:=1 to n do read(vyska[i]);
  utrid_vysky(1,n);
  if vyska[1]>VYSKA_MISTNOSTI-2 then
    begin
      writeln('Pro zadané rozměry knih neexistuje knihovna!');
      halt;
    end;
  s1:=1; s2:=n;
  while s1<s2 do
    if existuje((s1+s2) div 2) then
      s2:=(s1+s2) div 2
    else
      s1:=(s1+s2) div 2+1;
  writeln('Optimální šířka skříně je ',s1,' cm.');
  writeln('Počet poliček ve skříni: ',(n+s1-1) div s1);
  i:=1; v:=1;
  while (i<=n) do
    begin
      v:=v+vyska[i]+1;
      writeln('Výška poličky: ',vyska[i],' cm');
      write('Výšky knih na poličce:');
      repeat
        if (i>n) then break;
```

```
        write(' ',vyska[i],' cm');
        inc(i);
      until (i mod s1)=1;
      writeln;
    end;
  writeln('Výška skříně: ',v,' cm');
end.
```

P-II-3 Transformace

Použijeme myšlenku podobnou té z řešení úlohy P-I-3 domácího kola; problém ovšem je, že když se nám nyní mohou písmena opakovat, následníci nemusí být jednoznačně určeni. Provedeme následující úvahu:

Máme dán poslední sloupec, jeho setříděním dostaneme první sloupec. Dále máme dánu pozici slova, které bylo zakódováno, v setříděné tabulce, tedy známe jeho první písmeno; nechť je to x. Toto písmeno se nám může v prvním sloupci vyskytovat vícekrát, na pozicích odpovídajících slovům xv_1, xv_2, ..., xv_k, kde $xv_1 \leq xv_2 \leq \ldots \leq xv_k$. Z toho ovšem plyne také $v_1x \leq v_2x \leq \ldots \leq v_kx$, a tedy je-li xv_j zakódované slovo, wx j-té (v abecedním pořadí) slovo končící na x, musí platit $w = v_j$. Nyní můžeme celý postup opakovat (pozice, na níž je první písmeno zbytku zakódovaného slova, je ta, na níž je v posledním sloupci j-té písmeno x).

Algoritmus je již pouze přímočarým přepisem této myšlenky. Implementace tohoto algoritmu je poměrně jednoduchá; místo komplikované práce s dvojicemi (písmeno, pozice) je výhodnější si písmena v posledním sloupci očíslovat (písmenu přiřadíme jeho index v posledním sloupci) a po setřídění (přihrádkovým tříděním, abychom dosáhli linearní časové složitosti) pracovat pouze s těmito indexy.

Časová i paměťová složitost algoritmu jsou opět lineární.

```
program transformace;
const MAX = 10000;
var prvni_sloupec : array[1 .. MAX] of integer;
    posledni_sloupec : string;
    radek, delka, i, l : integer;
    buckets: array[char] of integer;
    ch : char;
begin
  {nacteni a ocislovani}
  readln (posledni_sloupec);
  readln (radek);
  delka := length (posledni_sloupec);
  for ch := #0 to #255 do buckets[ch] := 0;
  for i := 1 to delka do
    inc (buckets[posledni_sloupec[i]]);

  {setrideni}
  l := 1;
  for ch := #0 to #255 do
    begin
      i := l;
      inc (l, buckets[ch]);
      buckets[ch] := i;
    end;
  for i := 1 to delka do
    begin
      ch := posledni_sloupec[i];
      l := buckets[ch];
      inc (buckets[ch]);
      prvni_sloupec[l] := i;
    end;
```

```
  {vypis}
  for i:=1 to delka do
    begin
      write (posledni_sloupec[prvni_sloupec[radek]]);
      radek := prvni_sloupec[radek];
    end;
  writeln;
end.
```

P-II-4 Reverzibilní výpočty: Ouřad

Podobnost úlohy s počítáním vzdálenosti vrcholů (tj. délky nejkratší cesty mezi nimi) v orientovaném grafu jistě není náhodná, držme se proto i my grafové analogie: Jednotlivé budovy Ouřadu jsou pro nás vrcholy, potrubí mezi nimi orientovanými hranami grafu a A není ničím jiným než maticí sousednosti grafu. Nabízí se použít prohledávání grafu do šířky, ovšem musíme je náležitě upravit, aby bylo reverzibilní.

Vrcholy grafu si rozdělíme do *vrstev*: i-tá vrstva W_i bude obsahovat právě ty vrcholy, jejichž vzdálenost od vrcholu x je rovna i. Vrstev je proto nejvýše n a můžeme je snadno zkonstruovat indukcí: do W_0 padne vrchol x a žádný další; když máme sestrojeny vrstvy W_0 až W_{i-1}, tak do W_i patří právě ty vrcholy w, do kterých vede hrana z nějakého vrcholu $v \in W_{i-1}$ (tedy existuje cesta délky i z x do w) a $w \notin W_j$ pro $j < i$ (neexistuje žádná kratší cesta).

To je bezpochyby reverzibilní postup – při konstrukci vrstvy nijak neměníme vrstvy už spočítané; nakonec najdeme číslo vrstvy, do které padl vrchol y, to vydáme jako výsledek a všechny informace o vrstvách opět odpočítáme. Tak dostaneme řešení s časovou složitostí $O(n^3)$ a prostorovou složitostí $O(n^2)$. Všimněme si ještě dvou drobností:

(1) Ačkoliv vrstev může být až n a v každé z nich až $n-1$ vrcholů, lze je uložit efektivněji, protože ve všech vrstvách dohromady je nejvýše n vrcholů. Stačí je všechny naskládat za sebe do jednoho pole (říkejme mu třeba V) a nechat druhé pole S ukazovat, kde v poli V která vrstva začíná. Vrcholy ve vrstvě W_i tedy budou uloženy v prvcích V_{S_i} až $V_{S_{i+1}-1}$.

(2) Reverzibilita programu není přiliš nakloněna značkování vrcholů. Když si totiž budeme v nějakém poli pro každý vrchol pamatovat, zda jsme v něm již byli, a případně jej pak označkujeme, řekněme takto:

```
if UžJsemTamByl[i]=0 then begin
   { objevil jsem nový vrchol a někam si ho zapíšu }
   UžJsemTamByl[i] += 1;
   end;
```

dostaneme se do sporu s reverzibilitou podmínek: po ukončení příkazu `if` nepoznáme, zda byla podmínka splněna či nikoliv, protože `UžJsemTamByl[i]` bude vždycky jednička. To přesně náš jazyk zakazuje. Naštěstí nás zachrání jednoduchý trik: pokud dokážeme zajistit, abychom v rámci jedné vrstvy na každý vrchol narazili nejvýše jednou, stačí si u každého vrcholu zapamatovat (k tomu budeme používat pole L), ve které vrstvě byl objeven, a pokud dosud objeven nebyl, tak nějaké dostatečně velké číslo `inf`. Test se změní na:

```
if L[i] >= TatoVrstva then begin
   { objevil jsem nový vrchol a někam si ho zapíšu }
   L[i] -= inf - TatoVrstva;
   end;
```

a to už je korektní: platnost podmínky v této vrstvě se totiž přenastavením `L[i]` nezmění, ale v dalších vrstvách již správně poznáme, že vrchol byl zpracován.

Zde je program využívající oba popsané triky:

```
procedure Zkoumej(var n:word; var A:array [1..n] of array [1..n] of bit; var x,y,d:word);
var inf,cnt:word;
var L,V,S:array [0..n] of word;
begin
   wrap begin
      inf += n+1;                                    { "nekonečná vzdálenost" }
```

```
      for var i = 1 to n do                              { L[i] = inf }
         L[i] += inf;
      V[0] += x;                                         { nultá vrstva: vrchol x ... }
      L[x] -= inf;                                       { ... ve vzdálenosti 0 ... }
      S[1] += 1;                                         { ... a žádný další }
      for var i = 1 to n-1 do begin                      { hledáme další vrstvy }
         S[i+1] += S[i];                                 { zatím prázdná }
         for var w = 1 to n do
            if L[w] >= i then                            { nezařazený vrchol }
               wrap
                  for var j = S[i-1] to S[i]-1 do        { vede do něj hrana z vrstvy i-1? }
                     if A[V[j]][w]=1 then
                        cnt += 1
               on if cnt>0 then begin                    { ano => přidat do i-té vrstvy }
                     V[S[i+1]] += w;
                     S[i+1] += 1;
                     L[w] -= inf-i                       { L[w] >= i stále platí }
                     end
         end
      end
   on d += L[y]                                          { vrátíme výsledek }
end;
```

Zbývá ještě dodat, že prostorová složitost procedury je lineární a časová kvadratická (inicializace je lineární, vše mimo cyklu řízeného proměnnou j kvadratické a vnitřek zbylého cyklu se provede pro každý vrchol j právě n-krát, takže je dohromady také kvadratický).

Poznámka: Pokud bychom se vzdali polynomiální časové složitosti, existovala by prostorově ještě efektivnější řešení. Jedno z nich je založeno na následující úvaze: hledám-li cestu délky l z x do y, pak je buďto $l < 2$ (tehdy je úloha triviální) nebo cesta musí mít nějaký střední vrchol ve vzdálenosti $\lfloor l/2 \rfloor$. Vyzkouším proto postupně všechny vrcholy a pro každý z nich si rekurzivním zavoláním téže funkce pro obě poloviny cesty a poloviční l ověřím, zda existuje příslušná polovina cesty. Hloubka rekurze je maximálně $\lceil \log_2 l \rceil = O(\log n)$, dosáhneme tedy prostorové složitosti $O(\log n)$ za cenu drastického zpomalení na $n^{O(\log n)}$.