MATEMATICKÁ OLYMPIÁDA
NA STREDNÝCH ŠKOLÁCH

52. ročník, školský rok 2002/2003

Riešenia úloh 3. kola kategórie P

1. súťažný deň

P-III-1

Pri riešení úlohy si najskôr uvedomíme, že zo zadaných čísel hračiek môžeme jednoducho zistiť, ktoré dieťa chce hračku po ktorom dieťati. Situáciu si predstavíme ako orientovaný graf, v ktorom vrcholy zodpovedajú deťom a z vrcholu i vedie hrana do vrcholu j, ak dieťa i chce hračku po dieťati j. Pretože dieťa je ochotné vymeniť hračku iba ak dostane tú svoju vytúženú, deti si môžu vymieňať hračky iba v cykloch – aby sa dieťa i_1 vzdalo svojej hračky, musí dostať hračku od i_2, to od i_3 a tak ďalej, až nejaké dieťa dostane hračku od i_1. Chceme teda nájsť v grafe množinu disjunktných cyklov (na zjednodušenie vyjadrovania budeme za cyklus považovať aj vrchol so slučkou), ktoré spolu obsahujú čo najviac vrcholov. Hľadanie týchto cyklov je uľahčené tým, že každé dva cykly v našom grafe sú disjunktné – keby nejaké dva cykly mali spoločný vrchol, museli by sa niekde tiež od seba oddeľovať. Z príslušného vrcholu by teda museli viesť dve hrany, čo ale v našom grafe nie je možné. Keďže sú cykly navzájom disjunktné, chceme vlastne vypísať všetky cykly.

A teraz ako budeme cykly hľadať: Začneme v ľubovoľnom vrchole (napríklad v prvom) a pôjdeme po hranách (z každého vrcholu vedie práve jedna hrana, takže postup je jednoznačný), až kým sa nevrátime do nejakého vrcholu, v ktorom sme už boli (to spoznáme jednoducho, ak si budeme označovať navštívené vrcholy). Tým sme v grafe našli cyklus, ten môžeme vypísať a jeho vrcholy označiť za vyriešené. Vrcholy, ktoré sme prešli predtým, ako sme sa dostali na cyklus, pre zmenu určite na žiadnom cykle neležia (inak by z niektorého vrcholu museli viesť aspoň dve hrany). Preto sa týmito vrcholmi už nikdy nemusíme zaoberať a môžeme ich tiež označiť za vyriešené. Teraz vezmeme ďalší zatiaľ nevyriešený vrchol a opäť sa z neho vydáme hľadať cyklus. Ak narazíme na nejaký už vyriešený vrchol, hľadanie ukončíme a prejdené vrcholy označíme ako vyriešené – zjavne totiž nemôžu ležať na žiadnom cykle. Keď už nezostane žiadny nevyriešený vrchol, máme nájdené všetky cykly a výpočet ukončíme. Jediným nevyriešeným problémom zostáva, ako rýchlo hľadať zatiaľ nevyriešené vrcholy. To môžeme spraviť tak, že pri hľadaní ďalšieho nevyriešeného vrcholu začneme hľadať od posledne nájdeného vrcholu (pred ním isto žiadne nevyriešené vrcholy už nie sú). Vďaka tomu s hľadaním vrcholov strávime spolu čas $O(N)$ a pretože na nájdenie cyklov potrebujeme spolu tiež $O(N)$ (každú hranu prejdeme najviac raz), je celková časová zložitosť $O(N)$. Pamäťová zložitosť je tiež $O(N)$.

```
/* Hračkárstvo */
#include <stdio.h>

#define MAXD 10          /* Maximálny počet detí */

int N;                   /* Počet detí */
int Ma[MAXD];            /* Číslo hračky, ktorú príslušné dieťa má */
int Vlastni[MAXD];       /* Dieťa, ktoré vlastní príslušnú hračku */
int Chce[MAXD];          /* Dieťa, ktorého hračku toto dieťa chce */
```

```
int Hotovo[MAXD];      /* Už sme dieťa riešili? */

/* Načíta vstup */
void nacitaj(void)
{
  int i;

  scanf("%d", &N);
  for (i = 0; i < N; i++) {
    printf("Dieta %d: ", i+1);
    scanf("%d %d", &Ma[i], &Chce[i]);
    Ma[i]--; Chce[i]--;
    Vlastni[Ma[i]] = i;
  }
  /* Prevedieme odkazy na hračky na odkazy na deti */
  for (i = 0; i < N; i++)
    Chce[i] = Vlastni[Chce[i]];
}

/* Nájde cyklus začínajúc vo vrchole act. Vypíše cyklus
 * a označí prejdené vrcholy za vyriešené. */
void najdi_cyklus(int act)
{
  int start = act;

  /* Prejde deti a nájde cyklus */
  while (!Hotovo[act]) {
    Hotovo[act] = 1;
    act = Chce[act];
  }
  /* Vypíše cyklus */
  while (Hotovo[act] != 2) {
    Hotovo[act] = 2;
    printf(" %d", act+1);
    act = Chce[act];
  }
  /* Ešte označíme zvyšné prejdené vrcholy */
  act = start;
  while (Hotovo[act] != 2) {
    Hotovo[act] = 2;
    act = Chce[act];
  }
}

int main(void)
{
  int i;

  nacitaj();
```

```
  printf("Spokojne deti:");
  for (i = 0; i < N; i++)
    if (!Hotovo[i])                    /* Zatiaľ sme dieťa neriešili? */
      najdi_cyklus(i);
  printf("\n");
  return 0;
}
```

P-III-2

Základom nášho riešenia bude funkcia `existuje(s:integer)`, ktorá pre zadanú šírku s rozhodne, či existuje skriňa tejto šírky s P poličkami, do ktorej je možné umiestniť všetkých N kníh. Optimálnu šírku skrine s_0 potom môžeme nájsť binárnym vyhľadávaním pomocou $O(\log \sum_{i=1}^{N} h_i)$ volaní funkcie `existuje(s:integer)`, lebo zrejme šírka skrine bude nanajvýš rovná súčtu hrúbok všetkých kníh, $\sum_{i=1}^{N} h_i$. Poznamenajme, že ak by platilo $\log \sum_{i=1}^{N} h_i > N^2$ (čo ale vďaka obmedzeniu na hrúbku kníh nie je práve náš prípad), bolo by výhodnejšie spočítať $O(N^2)$ súčtov hrúbok i-tej až j-tej knihy ($i \leq j$), utriediť ich (na to potrebujeme čas $O(N^2 \log N)$), a potom hľadať optimálnu šírku knižnice iba medzi týmito súčtami binárnym vyhľadávaním.[1]

Samotná funkcia `existuje` bude fungovať nasledovne: Pre zadané s nájde najväčšie i_1 také, že $\sum_{i=1}^{i_1} h_i \leq s$; je jasné, že i_1 je maximálny možný počet kníh, ktoré sa dajú umiestniť do prvej poličky. Potom nájdeme najväčšie i_2 také, že $\sum_{i=i_1+1}^{i_2} h_i \leq s$, teda najväčší možný počet kníh i_2, ktoré sa dajú umiestniť do prvých dvoch poličiek, atď. Ak sa nám podarí umiestniť všetky knihy, t.j. $i_P = N$, tak existuje skriňa šírky s, do ktorej sa dajú všetky knihy uložiť; v opačnom prípade taká skriňa zjavne neexistuje.

Zostáva domyslieť, ako rýchlo hľadať čísla i_k, $1 \leq k \leq P$, vo funkcii `existuje`. Na to si najprv vytvoríme pomocné pole, v ktorom budú uložené súčty hrúbok prvých j kníh pre $1 \leq j \leq N$. Pri počítaní hodnoty i_k binárnym vyhľadávaním nájdeme v tomto pomocnom poli najväčšie číslo i' také, že $\sum_{i=1}^{i'} h_i - \sum_{i=1}^{i_{k-1}} h_i \leq s$; zjavne i' je hľadaná hodnota i_k.

Teraz odhadnime časovú a pamäťovú zložitosť nášho algoritmu. Funkcia `existuje` vykoná P vyhľadávaní v poli veľkosti N, t.j. pracuje v čase $O(P \log N)$. Celkový čas výpočtu nášho programu je teda $O(N + P \log N \log \sum_{i=1}^{N} h_i)$; čas $O(N)$ spotrebujeme okrem načítania dát tiež na vytvorenie pomocného poľa popísaného v predchádzajúcom odstavci. Pamäťová zložitosť je $O(N)$ – pole veľkosti N potrebujeme na uloženie hrúbok jednotlivých kníh a rovnaká je aj veľkosť pomocného poľa.

```
program p_3_2;
const MAXN=1000;
var hrubka: array[1..MAXN] of word;      { hrúbky kníh }
    sucet: array[0..MAXN] of word;       { súčty hrúbok kníh }
    n: word;                             { počet kníh }
    p: word;                             { počet poličiek }
function najdi(s: word): word;
{ Binárnym vyhľadávaním nájde najväčší index i
  taký, že sucet[i]<=s. }
var i1,i2 : word;
```

[1] Alebo ešte lepšie: na binárne vyhľadávanie nepotrebujeme mať súčty utriedené, stačí zakaždým nájsť medián hodnôt v prehľadávanom intervale.

```
begin
    i1:=0; i2:=n;
    while i1<i2 do
        if sucet[(i1+i2+1) div 2]>s then
            i2:=(i1+i2+1) div 2-1
        else
            i1:=(i1+i2+1) div 2;
    najdi:=i1
end;
function existuje(sirka: word):boolean;
{ Zistí, či existuje skriňa so šírkou sirka }
var i,j : word;
begin
    i:=0;
    for j:=1 to p do i:=najdi(sucet[i]+sirka);
    existuje:=(i=n);
end;
var i      : word;
    s1, s2 : word;
begin
    readln(n,p);
    for i:=1 to n do read(hrubka[i]);
    sucet[0]:=0;
    for i:=1 to n do sucet[i]:=sucet[i-1]+hrubka[i];
    { binárnym vyhľadávaním nájdeme šírku skrine }
    s1:=1;
    s2:=sucet[n];
    while s1<s2 do
        if existuje((s1+s2) div 2) then
            s2:=(s1+s2) div 2
        else
            s1:=(s1+s2) div 2+1;
    { výpis výsledku }
    writeln('Optimálna šírka skrine: ',s1,' mm');
    i:=1;
    while i<=n do begin
        write('Knihy na poličke:');
        s2:=0;
        while (i<=n) and (s2+hrubka[i]<=s1) do begin
            write(' ',i,'(',hrubka[i],' mm)');
            s2:=s2+hrubka[i];
            inc(i);
        end;
        writeln;
    end
end.
```

P-III-3

Na úvod si uvedomme, že klasický algoritmus na sčítanie čísel (napíšeme pod seba a postupne sprava doľava sčitujeme cifry a prenos) funguje bez ohľadu na použitú číselnú sústavu. Ak by sme teda vedeli spočítať prenosy P_i medzi jednotlivými rádmi, sčítanie by už bolo triviálne: $A'_i = A_i \text{ xor } B_i \text{ xor } P_i$. (Pritom P_i je prenos z $(i-1)$-ého rádu do i-teho, xor vlastne robí sčítanie modulo 2.) Ak sme ochotní obetovať pamäť na všetky P_i, môžeme ich spočítať nasledovne: zjavne $P_0 = 0$, pre $i > 0$ je $P_i = 1$ práve vtedy, keď spomedzi bitov A_{i-1}, B_{i-1} a P_{i-1} majú aspoň 2 hodnotu 1. Takto okamžite dostávame program s lineárnou časovou aj pamäťovou zložitosťou:

```
procedure Add(var n:word; var A,B:array [0..n-1] of bit);
var P:array [0..n] of bit;
begin
    wrap
        for var i = 0 to n-1 do
            P[i+1] ^= (A[i] and B[i]) or ((A[i] or B[i]) and P[i])
    on
        for var i = 0 to n-1 do
            A[i] ^= B[i] xor P[i]
end;
```

Aby sme dosiahli lepšiu ako lineárnu pamäť, nemôžeme si pamätať všetky hodnoty prenosov. Celý problém je v tom, že na vypočítanie P_i potrebujeme poznať hodnotu P_{i-1}. Navyše hodnotu P_{i-1} budeme potrebovať aj keď budeme chcieť P_i na konci výpočtu odpočítať. (Nestačí nám na to hodnota P_{i+1}, totiž medzičasom sme si jeden zo sčítancov prepísali výsledkom a vo všeobecnosti nevieme povedať, či jednotka vo výsledku pochádza z jednotky v prepísanom sčítanci alebo z prenosu.)

Takže hodnotu P_{i-1} si musíme celý čas pamätať. Preto pamäťová zložitosť nemôže byť lepšia ako lineárna a predchádzajúce riešenie je optimálne. Alebo predsa len nie? Nedalo by sa na P_{i-1} zabudnúť a keď ho budeme na konci potrebovať, spočítať si ho znovu? Fungovalo by to, len ho budeme musieť počítať šikovne, aby sme nespotrebovali priveľa (lineárne veľa alebo dokonca viac) pamäti na rekurzívne volania na výpočet predchádzajúcich P_j.

Zostrojíme procedúru `Prenos(i,l,p_in,p_out)`, ktorá sa pozrie na bity i až $i+l-1$ v číslach A, B, spočíta prenos P_{i+l} do vyšších rádov (pričom $P_i = p\_in$) a prixoruje ho k premennej p_out. Ak je dĺžka úseku $l = 1$, spraví to známym spôsobom z predchádzajúceho riešenia. Ak je úsek dlhší, nebudeme prenosy rátať postupne. Aby sme ušetrili pamäť, spravíme to nasledovne:

Úsek si rozdelíme na dve polovice. Rekurzívne sa zavoláme na nižšiu polovicu a spočítame si prenos *mid* z nej do vyššej polovice. Teraz sa rekurzívne zavoláme na vyššiu polovicu a spočítame výsledok. Ešte potrebujeme hodnotu *mid* odpočítať, na čo sa opäť rekurzívne zavoláme na nižšiu polovicu úseku. Ako obvykle nám pri zápise pomôže príkaz `wrap`.

```
procedure Prenos(var i,l:word; var p_in,p_out:bit);
var l1,l2,j:word;
var mid:bit;
begin
    if l=1 then                    { jednobitový prenos }
        p_out ^= (A[i] and B[i]) or ((A[i] or B[i]) and p_in)
    else wrap begin
        l1 += l div 2;             { l1=dĺžka dolnej polovice }
        l2 += l-l1;                { l2=dĺžka hornej polovice }
```

```
        j += i+l1;                  { j=začiatok hornej polovice }
        Prenos(i,l1,p_in,mid)       { prenos z dolnej polovice }
        end
    on  Prenos(j,l2,mid,p_out)      { prenos z hornej polovice }
end;
```

Keďže pri každom rekurzívnom volaní zmenšíme dĺžku úseku l približne na polovicu, hĺbka rekurzie je nanajvýš $\lceil \log_2 l \rceil$, takže procedúra `Prenos` má pamäťovú zložitosť $O(\log l)$. S časovou zložitosťou je to trochu ťažšie. Ak označíme $T(l)$ čas, ktorý procedúra potrebuje, keď ju spustíme na úsek dĺžky l, tak platí $T(l) = 1 + 3 \cdot T(l/2)$ – procedúra vykoná konštantný kus práce (keďže nás výsledná zložitosť zaujíma len asymptoticky, môžeme predpokladať, že jednotkovú) a následne trikrát zavolá sama seba na vstup polovičnej dĺžky.

Keď tento vzťah dosadíme sám do seba, dostaneme $T(l) = 1+3\cdot(1+3\cdot T(l/4)) = 1+3+9\cdot T(l/4)$ a keď tento postup zopakujeme ešte niekoľkokrát, dostaneme $T(l) = 1+3+\ldots+3^{k-1}+3^k \cdot T(l/2^k)$. Zoberme teraz $k = \log_2 l$, čo je hĺbka našej rekurzie. Potom $T(l/2^k) = T(1) = 1$, dosadením dostávame $T(l) = 1 + 3 + \ldots + 3^{\log l - 1} + 3^{\log_2 l}$. To je geometrický rad so súčtom $(3^{k+1}-1)/2 = O(3^k) = O(3^{\log_2 l})$ ktorý môžeme ďalej zjednodušiť: $3^{\log_2 l} = (2^{\log_2 3})^{\log_2 l} = 2^{\log_2 3 \cdot \log_2 l} = (2^{\log_2 l})^{\log_2 3} = l^{\log_2 3} \leq l^{1.59}$. Časová zložitosť procedúry `Prenos` je teda $O(l^{1.59})$.

Použitím procedúry `Prenos` priamočiaro dostávame riešenie s logaritmickou pamäťovou zložitosťou – vždy, keď potrebujeme vedieť hodnotu prenosu z nižších rádov, zavoláme vyššie popísanú funkciu. Musíme samozrejme bity sčitovať v opačnom poradí, teda najskôr najvyššie, aby sme si výsledkom neprepísali hodnoty z A, B, ktoré ešte budeme potrebovať. Toto riešenie n-krát volá procedúru `Prenos`, teda jeho časová zložitosť je $O(n \cdot n^{1.59}) = O(n^{2.59})$.

Ešte stále je čo zlepšovať. Uvedomme si, že na niektoré úseky sme spúšťali procedúru `Prenos` zbytočne veľakrát. Dá sa s tým niečo robiť? Nuž, nepýtali by sme sa tak, keby sa nedalo :-)

Zostrojme rekurzívnu procedúru `Scitaj`, ktorá zároveň s počítaním prenosu príslušné úseky rovno sčíta. Bude mať rovnaké parametre ako `Prenos`. Procedúra najskôr zavolá procedúru `Prenos`, ktorá spočíta do premennej *mid* prenos z dolnej polovice do hornej. Následne rekurzívnym zavolaním samej seba spočíta horné polovice úsekov. Na záver sa rekurzívne zavolá na dolné polovice úsekov, čím ich nielen spočíta, ale aj vynuluje predtým spočítanú premennú *mid*.

```
procedure Scitaj(var i,l:word; var p_in,p_out:bit);
var l1,l2,j:word;
var mid:bit;
begin
    if l=1 then begin               { jednobitové sčítanie }
        p_out ^= (A[i] and B[i]) or ((A[i] or B[i]) and p_in);
        A[i] ^= B[i] xor p_in
        end
    else wrap begin
        l1 += l div 2;              { opäť počítame, kde sú polovice }
        l2 += l-l1;
        j += i+l1
        end
    on begin
        Prenos(i,l1,p_in,mid);      { prenos z dolnej polovice }
        Scitaj(j,l2,mid,p_out);     { sčítame hornú polovicu }
        Scitaj(i,l1,p_in,mid)       { sčítame dolnú a odpočítame prenos }
        end
end;
```

Časová a pamäťová zložitosť našej sčítacej procedúry bude zjavne rovnaká ako u procedúry `Prenos`. Vieme teda sčítať v pamäti $O(\log n)$ a čase $O(n^{1.59})$. Samotný program bude vyzerať nasledovne:

```
procedure Add(var n:word; var A,B:array [0..n-1] of bit);
{ Tu sú vložené procedúry Prenos a Scitaj }
var zero:word;
var p_in,p_out:bit;
begin
    Scitaj(zero,n,p_in,p_out);   { p_in=0, vieme, že p_out bude 0 }
end;
```

SLOVENSKÁ KOMISIA MATEMATICKEJ OLYMPIÁDY

52. ROČNÍK MATEMATICKEJ OLYMPIÁDY

Riešenia 3. kola kategórie P

1. súťažný deň

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

MATEMATICKÁ OLYMPIÁDA NA STREDNÝCH ŠKOLÁCH

52. ročník, školský rok 2002/2003

Riešenia úloh 3. kola kategórie P

2. súťažný deň

P-III-4

Zadanie si najskôr preložíme do matematickej reči. Daný je konvexný N-uholník a M jeho nepretínajúcich sa tetív, ktoré ho delia na niekoľko dielov. Nájdite najväčší počet dielov, z ktorých žiadne dva nemajú spoločnú stranu.

Zostrojme graf G nasledovne: vrcholy budú diely mapy, dva vrcholy budú spojené hranou práve vtedy, ak príslušné diely majú spoločnú stranu. Graf G je zjavne súvislý. Keďže žiadne dve tetivy sa nepretínajú, rozdelia N-uholník na presne $M + 1$ dielov, pričom každá tetiva oddeľuje od seba niektoré dva z nich. Preto G má $M + 1$ vrcholov a M hrán.

Keďže G je súvislý, ľahko nahliadneme, že nemôže obsahovať žiadnu kružnicu – nemáme na to dosť hrán. Takémuto grafu (súvislému a neobsahujúcemu kružnicu) hovoríme strom. Našou úlohou je nájsť v našom strome najväčšiu nezávislú množinu vrcholov. (Množina vrcholov je nezávislá, ak žiadne dva vrcholy z nej nie sú spojené hranou. To je práve to, čo chceme dosiahnuť.) Naše riešenie bude využívať prehľadávanie do hĺbky. Na začiatku označíme všetky vrcholy ako vybrané. Začneme graf prehľadávať z ľubovoľného vrcholu. Keď sa počas prehľadávania vraciame z vrcholu, ktorý je označený ako vybraný, jeho otcovi (tomu vrcholu, kam sme sa vrátili) túto značku zrušíme.

Je zrejmé, že takto dostaneme nezávislú množinu vrcholov. Ale bude naozaj najväčšia? Ukážeme, že áno. Sporom. Nech A je množina, ktorú sme zostrojili, nech B je taká najväčšia nezávislá množina, ktorá má s A najviac vrcholov spoločných. Pozrime sa na vrcholy, v ktorých sa líšia a vyberme si z nich taký, ktorý je najďalej od vrcholu, z ktorého sme spustili prehľadávanie. Označme vybraný vrchol v. Rozoberme dva prípady.

Nech $v \in B$, $v \notin A$. Keďže v nepatrí do A, znamená to, že niektorý vrchol u, do ktorého sme prišli z v, leží v A. (Z A sme vrchol vyhodili len ak bol v A nejaký jeho syn, toho syna sme už neskôr z A vyhodiť nemohli.) Keďže v je najvzdialenejší vrchol, v ktorom sa A a B líšia, nemôžu sa líšiť v u. Preto $u \in B$. To je ale spor, lebo v B máme dva vrcholy u, v spojené hranou.

Preto nutne $v \in A$, $v \notin B$. Pridajme v do B. Keďže v je v A a na vrcholoch ďalej od v sa A a B zhodujú, nie je v B ani jeden zo synov vrcholu v. Preto jediný problém môže byť ak do B patrí otec vrcholu v. V takomto prípade otca vrcholu v z B vyhodíme. Takto sme z B dostali rovnako veľkú nezávislú množinu vrcholov, ktorá má ale s A spoločných viac vrcholov ako B. To je ale spor (s výberom B).

Iný pohľad: Spustením prehľadávania do hĺbky sme si náš strom "zakorenili". Vrchol, v ktorom sme začali prehľadávať, nazveme koreň. Zakorenený strom dostaneme tak, že zoberieme strom za koreň a zatrasieme. Vrcholy budú tým nižšie, čím sú ďalej od koreňa. Navyše z každého vrcholu pôjde práve jedna hrana dohora (tou sme doň počas prehľadávania prišli; vrchol na jej hornom konci voláme otcom vrcholu na dolnom konci). Predstavme si, že postupne spracúvame vrcholy oddola nahor, pričom pre každý chceme povedať, koľko najviac nezávislých vrcholov vieme nájsť na strome visiacom pod ním. Potom zjavne odpoveď pre koreň stromu je riešením úlohy.

Keď teraz spracúvame vrchol, vieme už odpoveď pre všetky vrcholy visiace pod ním. Pre najväčšiu množinu máme dve možnosti – buď spracúvaný vrchol vyberieme (a už nesmieme vybrať žiadneho jeho syna), alebo nie. Zjavne prvá možnosť sa oplatí ***iba*** *vtedy, ak v podstrome každého syna existuje najlepšie riešenie, v ktorom tento syn nie je vybraný. Inak dá totiž druhá možnosť aspoň tak dobré riešenie, ktoré je navyše o to lepšie, že práve spracovaný vrchol nie je vybraný. Vyššie popísané upravené prehľadávanie robí presne toto isté, len to robí už priamo počas prehľadávania.*

V našej úlohe zostrojiť vyššie popísaný graf G je zbytočná robota navyše. Ukážeme, ako ho prehľadať bez toho, aby sme ho museli najskôr explicitne zostrojiť. Orientujme každú tetivu tak, aby jej prvý koniec mal menšie číslo ako druhý. Pridajme k nim ako zarážku hranu z 1 do N. Keďže konce tetív sú čísla od 1 do N, vieme ich priehradkovým triedením (BucketSort, resp. CountSort) zoradiť podľa začiatočného vrcholu, tetivy s rovnakým začiatočným vrcholom zoradíme **klesajúco** podľa koncového. Teraz budeme prechádzať vrcholy mnohouholníka v poradí od 1 do N, pričom v zásobníku si udržiavame zoznam tetív, ktorých začiatok sme už prešli, ale koniec ešte nie. (Keďže tetivy sa nepretínajú, na ich konce naozaj narazíme v opačnom poradí ako na začiatky, preto je zásobník vhodná dátová štruktúra.)

Každá tetiva teraz zodpovedá jednému dielu mnohouholníka – pridaná zarážka zodpovedá dielu, v ktorom sme začali prehľadávať, každá iná leží medzi dvoma dielmi a odteraz bude zodpovedať tomu z nich, ktorý leží ďalej od začiatočného dielu. Pri každej tetive si budeme pamätať, či jej zodpovedajúci diel je ešte vybraný, alebo už nie. Spracovanie vrcholu mnohouholníka bude vyzerať nasledovne: Najskôr zo zásobníka postupne odstránime všetky tetivy, ktoré v ňom končia. Pritom ak diel zodpovedajúci odstraňovanej tetive máme stále označený ako vybraný, zvýšime si počet vybraných dielov a odznačíme tetivu, ktorá leží v zásobníku pod práve odstránenou. (Jej zodpovedajúci diel už nemôžeme vybrať.) Teraz postupne pridáme všetky tetivy, ktoré v tomto vrchole začínajú a pri každej z nich si zapamätáme, že jej zodpovedajúci diel je zatiaľ vybraný. Výpočet končí spracovaním N-tého vrcholu mnohouholníka.

Počas výpočtu raz spracujeme každý vrchol mnohouholníka. Každú tetivu raz vložíme do zásobníka a raz ju vyberieme. Keďže sa tetivy nepretínajú, je ich najviac $N - 3$. Preto časová aj pamäťová zložitosť nášho algoritmu sú $O(N)$.

```
#include <stdio.h>
#include <stdlib.h>

#define MAXV 30000        /* Maximálny počet vrcholov */
#define MAXR 30000        /* Maximálny počet tetív */

/* Štruktúra pre jednu tetivu */
struct tetiva {
  int a, b;
};

int tetiv, vrcholov;      /* Počet tetív a vrcholov */
struct tetiva t[MAXR];    /* Jednotlivé tetivy */
int vpoc[MAXV];           /* Počty tetív v jednotlivých vrcholoch */

/* Načíta vstup */
void nacitaj(void)
{
  int pom, i;
```

```
  FILE *vstup;

  if (!(vstup = fopen("poklad.in", "r")))
    exit(1);
  fscanf(vstup, "%d %d", &vrcholov, &tetiv);
  for (i = 0; i < tetiv; i++) {
    fscanf(vstup, "%d %d", &t[i].a, &t[i].b);
    t[i].a--; t[i].b--;
    if (t[i].a > t[i].b) {
      pom = t[i].a;
      t[i].a = t[i].b;
      t[i].b = pom;
    }
  }
  fclose(vstup);
  /* Pridáme ešte fiktívnu tetivu medzi vrcholmi 1, N */
  t[tetiv].a = 0;
  t[tetiv].b = vrcholov-1;
  tetiv++;
}

/* Zotriedi tetivy */
void zotried(void)
{
  struct tetiva t1[MAXR];        /* Jednotlivé tetivy */
  int vrchind[MAXV];     /* Index, kde začínajú tetivy z vrcholu */
  int vrchpoc[MAXV];     /* Počet tetiv z vrcholu */
  int i;

  /* Prvý prechod triedenia */
  for (i = 0; i < vrcholov; i++)
    vrchpoc[i] = 0;
  /* Spočítame počty tetív pre jednotlivé vrcholy */
  for (i = 0; i < tetiv; i++)
    vrchpoc[t[i].b]++;
  vrchind[0] = 0;
  for (i = 1; i < vrcholov; i++)
    vrchind[i] = vrchind[i-1] + vrchpoc[i-1];
  /* Preusporiadame tetivy podľa koncového vrcholu */
  for (i = 0; i < tetiv; i++)
    t1[vrchind[t[i].b]++] = t[i];

  /* Druhý prechod triedenia */
  for (i = 0; i < vrcholov; i++)
    vrchpoc[i] = 0;
  for (i = 0; i < tetiv; i++)
    vrchpoc[t1[i].a]++;
  vrchind[0] = 0;
  for (i = 1; i < vrcholov; i++)
```

```
    vrchind[i] = vrchind[i-1] + vrchpoc[i-1];
  /* Preusporiadame hrany podľa začiatočného vrcholu
   * (berieme ich zostupne podľa koncového vrcholu) */
  for (i = tetiv; i >= 0; i--)
    t[vrchind[t1[i].a]++] = t1[i];
}

/* Spočíta koľko kusov mapy sa dá rozdať */
int spocitaj(void)
{
  int casti = 0;          /* Počet častí */
  int zasvrch = 0;        /* Vrchol zásobníka */
  int zas[MAXR];          /* Zásobník na spracovávané tetivy */
  int vybrat[MAXR];       /* Značka, že prísl. časť môžeme vybrať */
  int actvrch = 0, acttetiva = 0; /* Aktuálny vrchol a tetiva */

  while (actvrch < vrcholov) {
    while (zasvrch && t[zas[zasvrch-1]].b == actvrch) {
      if (vybrat[zasvrch-1]) {
        casti++;
        vybrat[zasvrch-2] = 0;
      }
      zasvrch--;
    }
    while (acttetiva < tetiv && t[acttetiva].a == actvrch) {
      zas[zasvrch] = acttetiva++;
      vybrat[zasvrch++] = 1;
    }
    actvrch++;
  }
  return casti;
}

int main(void)
{
  FILE *vystup;

  nacitaj();
  zotried();

  if (!(vystup = fopen("poklad.out", "w")))
    exit(1);
  fprintf(vystup, "%d\n", spocitaj());
  fclose(vystup);
  return 0;
}
```

P-III-5

Na riešenie úlohy zjavne potrebujeme vhodne upraviť niektorý triediaci algoritmus. Keďže chceme mať optimálnu časovú zložitosť, vyberieme si MergeSort, ktorý aj v najhoršom prípade potrebuje $O(N \log N)$ operácií (kde N je počet mincí). Daňou za to bude, že jeho implementácia nebude práve najjednoduchšia. Poznamenávame, že existuje viacero iných riešení tejto úlohy (založených na iných triediacich algoritmoch), asi najjednoduchšiu implementáciu má upravený QuickSort.[2]

Výstupom algoritmu bude (jednosmerný) spájaný zoznam, pričom každému prvku v ňom zodpovedá jedna váha mincí a tieto váhy tvoria rastúcu postupnosť. Na každom mieste v spájanom zozname si navyše budeme pamätať (neskôr upresníme ako) všetky mince príslušnej váhy.

Základom programu bude rekurzívna procedúra `Vytvor(prvy,posledny)`, ktorá zoberie mince od *prvy* po *posledny* a vytvorí z nich takýto spájaný zoznam. Ak $prvy = posledny$, je to triviálne. Ak je mincí viac, procedúra tento interval rozdelí na dve približne rovnaké časti a na obe sa rekurzívne zavolá. Takto dostane dva spájané zoznamy, ktoré potrebuje spojiť do jedného.

Zoberme si prvé prvky oboch zoznamov. Každý z nich predstavuje množinu mincí s nejakou hmotnosťou. Tieto hmotnosti potrebujeme porovnať, aby sme vedeli, ktorý z týchto prvkov bude na začiatku výsledného zoznamu. **Vyberieme** teda z každej množiny jednu ľubovoľnú mincu a opýtame sa na ich váhy. Ak je ľahšia prvá, odstránime prvý prvok z prvého zoznamu a spravíme z neho prvý prvok výsledného zoznamu. Analogicky ak je ľahšia druhá minca. Ak sú váhy mincí rovnaké, odstránime prvé prvky z oboch spracúvaných zoznamov, príslušné množiny mincí **spojíme** a z výslednej množiny spravíme začiatok výsledného zoznamu.

Zvyšok oboch zoznamov spracujeme analogicky, teda vždy porovnáme váhy dvoch potenciálne najľahších množín a ľahšiu z nich vložíme na koniec hotovej časti výsledného zoznamu.

Ostáva vyriešiť, ako si pamätať množiny mincí, aby sme vedeli vyššie zvýraznené operácie robiť efektívne, t.j. v konštantnom čase. Jedno možné riešenie je použiť binárne stromy (nebudú musieť byť vyvážené). Prvky množiny (čísla mincí) si budeme pamätať v listoch stromu. V každom vnútornom vrchole si budeme pamätať to isté číslo, ako v ľubovoľnom z jeho synov. Takto keď chceme číslo niektorej mince v množine, stačí pozrieť na číslo v koreni stromu. Takisto spojiť dva stromy do jedného je triviálne – stačí pridať nový koreň, ktorého synmi budú pôvodné dva korene a uložiť doň hodnotu z jedného z nich.

V prvku spájaného zoznamu si budeme pamätať dva ukazovatele – jeden na nasledujúci prvok, jeden na koreň stromu, v ktorom si pamätáme príslušnú množinu mincí.

Na záver algoritmu len prejdeme postupne celý výsledný spájaný zoznam a každú množinu (listy každého stromu) vypíšeme na samostatný riadok. Ľahko nahliadneme, že ak budeme spájať stromy tak, že ako ľavý podstrom vždy zoberieme ten, v ktorom sú menšie čísla mincí (a na záver vypíšeme listy stromu v poradí zľava doprava), budú vypísané čísla rovno utriedené.

Indukciou ukážme, že náš algoritmus nevolá funkciu `porovnaj` zbytočne. Procedúra `Vytvor` volá `porovnaj` len na mince z príslušného intervalu. Ak je tento interval jednoprvkový, tvrdenie triviálne platí. V opačnom prípade sa procedúra dvakrát rekurzívne zavolá a potom vytvorené dva zoznamy spojí do jedného. Z indukčného predpokladu pri rekurzívnych volaniach zbytočné volania nerobíme. Keď teraz spájame zoznamy do jedného, porovnávame vždy mince z rôznych dvoch zoznamov. Výsledok tohto váženia nemôže teda byť určený váženiami počas rekurzívnych volaní. Takisto ale nemôže byť určený ani predchádzajúcimi váženiami počas spájania týchto

[2]Vyberieme mincu-pivota, preusporiadame pole tak, aby v prvej časti boli ľahšie, v druhej rovnako ťažké a v tretej ťažšie mince a rekurzívne sa zavoláme na prvú a tretiu časť. Keď navyše použijeme randomizovanú verziu, t.j. pivota vyberáme náhodne, dostaneme očakávanú časovú zložitosť $O(N \log N)$.

dvoch zoznamov – z tých vieme len toľko, že mince doteraz presunuté do výsledného zoznamu sú ľahšie od mincí v oboch práve porovnávaných množinách.

Hĺbka rekurzie pri volaní procedúry `Vytvor` je $O(\log N)$, lebo pri každom volaní sa dĺžka úseku, ktorý treba spracovať, zmenší približne na polovicu. Na spojenie dvoch zoznamov je potrebný čas úmerný dĺžke výsledného zoznamu, tú môžeme zhora odhadnúť počtom mincí v ňom. Na každej úrovni rekurzie sa každá minca vyskytuje v práve jednom zozname, preto celkový čas spotrebovaný volaniami procedúry `Vytvor` v jednej úrovni rekurzie je $O(N)$. Celková časová zložitosť je preto $O(N \log N)$. Z podobných argumentov (počas behu algoritmu je v každom okamihu každá minca v najviac 1 zozname) vyplýva, že pamäťová zložitosť nášho algoritmu je lineárna.

```
#include <stdio.h>
#include <stdlib.h>
#include "vahy_lib.h"

struct tUzol {
  int prvok;
  struct tUzol *lavy, *pravy;
};

struct tZoznam {
  struct tUzol *strom;
  struct tZoznam *dalsi;
};

struct tZoznam *vytvor(int prvy, int posledny) {
  struct tZoznam *vysledok, *zoznam1, *zoznam2, **chvost, *pomocna;
  if (prvy==posledny) {
    vysledok=malloc(sizeof(struct tZoznam));
    vysledok->dalsi=NULL;
    vysledok->strom=malloc(sizeof(struct tUzol));
    vysledok->strom->prvok=prvy;
    vysledok->strom->lavy=vysledok->strom->pravy=NULL;
    return vysledok;
  }
  zoznam1=vytvor(prvy,(prvy+posledny)/2);
  zoznam2=vytvor((prvy+posledny)/2+1,posledny);
  chvost=&vysledok;
  while (zoznam1&&zoznam2) {
    switch (porovnaj(zoznam1->strom->prvok,zoznam2->strom->prvok)) {
    case 0:
      (*chvost)=malloc(sizeof(struct tZoznam));
      (*chvost)->strom=malloc(sizeof(struct tUzol));
      (*chvost)->strom->prvok=zoznam1->strom->prvok;
      (*chvost)->strom->lavy=zoznam1->strom;
      (*chvost)->strom->pravy=zoznam2->strom;
      pomocna=zoznam1; zoznam1=zoznam1->dalsi; free(pomocna);
      pomocna=zoznam2; zoznam2=zoznam2->dalsi; free(pomocna);
      break;
```

```
      case 1:
        *chvost=zoznam1; zoznam1=zoznam1->dalsi;
        break;
      case -1:
        *chvost=zoznam2; zoznam2=zoznam2->dalsi;
        break;
      }
      chvost=&((*chvost)->dalsi);
    }
    *chvost=zoznam1?zoznam1:(zoznam2?zoznam2:NULL);
    return vysledok;
}

void vypis_strom(FILE *subor, struct tUzol *uzol) {
    if (uzol->lavy) {
      vypis_strom(subor,uzol->lavy);
      vypis_strom(subor,uzol->pravy);
    }
    else
      fprintf(subor,"%d ",uzol->prvok);
}

void vypis_zoznam(FILE *subor, struct tZoznam *zoznam) {
    while (zoznam) {
      vypis_strom(subor, zoznam->strom);
      fprintf(subor,"\n");
      zoznam=zoznam->dalsi;
    }
}

int main() {
    FILE *subor;
    int N;
    struct tZoznam *zoznam;

    subor=fopen("vahy.in","rt");
    fscanf(subor,"%d",&N);
    fclose(subor);
    zoznam=vytvor(1,N);
    subor=fopen("vahy.out","wt");
    vypis_zoznam(subor,zoznam);
    fclose(subor);
    return 0;
}
```

SLOVENSKÁ KOMISIA MATEMATICKEJ OLYMPIÁDY

52. ROČNÍK MATEMATICKEJ OLYMPIÁDY

Riešenia 3. kola kategórie P

2. súťažný deň

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