57. ročník Matematické olympiády – 2007/2008

Řešení úloh ústředního kola kategorie P – 1. soutěžní den

P-III-1 Karpaty

Jedním z nejjednodušších (a také nejpomalejších) způsobů, jak úlohu řešit, je projít postupně všechny možnosti, na které kopce byste vystoupili. V jednom průchodu bychom našli délku nejdelší rostoucí podposloupnosti a v druhém průchodu bychom pak zjistili, které vrcholy se vyskytují ve všech nejdelších podposloupnostech, které jen v některých a na které se vystoupit nesmí. Bohužel, časová složitost takového algoritmu bude řádově $\mathcal{O}(2^N)$, protože tolik je možností výstupu na jednotlivé vrcholy (u každého si vyberete, zda na něj vystoupíte nebo ne).

Zkušený řešitel olympiády si však hravě všimne, že minimálně délku nejdelší rostoucí podposloupnosti lze zjistit výrazně rychleji metodou dynamického programování. To jsme už potkali v řešení úlohy P-I-1, nyní si ukážeme trochu jiný postup, který půjde lépe zobecnit.

Označme si d_i délku nejdelší rostoucí podposloupnosti, jejímž posledním členem je v_i (výška i-té hory). Hledaná délka pak bude maximum z čísel d_i. A jak lze hodnoty d_i spočítat? Podívejme se, jak nějaká taková nejdelší podposloupnost P končící vrcholem i vypadá: Buď obsahuje jen vrchol i (a pak má tedy délku 1) nebo jejím předposledním vrcholem je nějaký vrchol j, pro který platí, že $j < i$ a $v_j < v_i$. Všimněme si, že v druhém případě určitě platí, že pokud z P odstraníme poslední prvek (tedy vrchol i), získáme podposloupnost P' končící vrcholem j, která je zároveň jednou z nejdelších podposloupností, které končí v tomto vrcholu. Kdyby tomu tak nebylo, tak bychom přidáním vrcholu i na konec některé z nejdelších podposloupností končících vrcholem j získali podposloupnost, která by končila vrcholem i, ale její délka by byla větší než délka P. Takže stačí postupně počítat d_i v cyklu od 1 do N, pro každé i se podíváme na hodnoty d_j pro $j = 1, \ldots, i-1$ takové, že $v_j < v_i$ a, pokud nějaké najdeme, vezmeme za d_i jejich maximum zvýšené o jedna; pokud žádné takové v_j neexistuje, nastavíme d_i na 1. Přímočarou implementací tohoto postupu dostaneme algoritmus s časovou složitostí $\mathcal{O}(N^2)$.

Když jsme se takto zvládli zbavit exponenciálního času výpočtu délky nejdelší rostoucí podposloupnosti, nemohli bychom podobný princip použít i pro druhou část úlohy? Odpověď na tuto otázku je ano, ale bude to vyžadovat malinko více práce. Nejprve si kromě hodnot d_i spočteme i hodnoty z_i, kde z_i bude délka nejdelší rostoucí podposloupnosti začínající prvkem v_i. Spočítat tyto hodnoty zvládneme obdobným způsobem jako u hodnot d_i (místo od začátku budeme posloupnost procházet od konce). Nyní nahlédneme, že délka nejdelší rostoucí podposloupnosti obsahující i-tý vrchol je $d_i + z_i - 1$. To proto, že pokud by P byla nějaká taková nejdelší podposloupnost a přitom její část od začátku po i-tý vrchol by nebyla nejdelší rostoucí podposloupností končící i-tým vrcholem, mohli bychom P prodloužit výměnou úvodní části za nějakou delší. Naopak spojením nejdelších podposloupností z a do i-tého vrcholu dostaneme rostoucí podposloupnost požadované délky.

Označme maximum z čísel $d_i + z_i - 1$ tedy L. Již víme, jak poznat vrcholy, na které vystoupit nesmíme – podíváme se, zda $d_i + z_i - 1$ je rovno L (pak na tento vrchol můžeme a možná i musíme vystoupit) nebo ne (a pak na tento vrchol nesmíme vystoupit). Zbývá rozpoznat ty vrcholy, na které musíme vystoupit, od těch, na které „jen můžeme“ vystoupit. Pokud je pro vrchol v_i hodnota $d_i + z_i - 1$ rovna L, pak v_i je d_i-tým vrcholem v každé nejdelší rostoucí podposloupnosti, která ho obsahuje. Vrchol v_i je tedy v každé nejdelší rostoucí podposloupnosti právě tehdy, když neexistuje jiný vrchol v_j s $d_j + z_j - 1 = L$ a $d_i = d_j$.

Pokud implementujeme výše uvedený postup, dostaneme algoritmus s kvadratickou časovou složitostí, kde všechny operace kromě výpočtu hodnot d_i a z_i zaberou jen lineární čas. Ve zbytku řešení si ukážeme, jak výpočet hodnot d_i a z_i provést v čase $\mathcal{O}(N \log N)$. Tím získáme algoritmus s časovou složitostí $\mathcal{O}(N \log N)$. A jak na to? Použijeme upravené binární vyhledávací stromy. Pro každý uzel si kromě klíče, tj. hodnoty, podle které vyhledáváme (klíč tedy bude výška hory v_i), zapamatujeme i délku nejdelší rostoucí podposloupnosti, která v tomto vrcholu končí nebo začíná. Tj., při prvním výpočtu si budeme pamatovat hodnotu d_i a při druhém z_i. Navíc si ještě v každém uzlu budeme udržovat maximum z délek v podstromu tohoto uzlu.

Na začátku do stromu vložíme všechna v_i a délky nastavíme na 0. To provedeme například tak, že hodnoty v_i v čase $\mathcal{O}(N \log N)$ setřídíme, odstraníme opakované výskyty stejných čísel (různé kopce přece mohou být stejně vysoké) a pak strom rekurzivně sestavíme: rekurzivní procedura dostane úsek setříděného pole, vybere prostřední prvek (řekněme, že je to x-tý prvek pole), vytvoří pro něj uzel, jehož levý podstrom vznikne rekurzivním voláním na úsek od začátku po $(x-1)$-ní prvek a pravý podstrom voláním na úsek od $(x+1)$-ního prvku do konce. Všimněte si, že hloubka takto vytvořeného stromu bude řádově $\log_2 N$.

Nyní postupně počítáme hodnoty d_i. Nejprve zjistíme, jaká největší délka je ve stromu uložená u nějakého uzlu s hodnotou menší než v_i (pokud takovou nenajdeme, vrátíme 0). Tuto délku zvýšenou o 1 pak uložíme do d_i a do uzlu u s hodnotou v_i nastavíme délku d_i a upravíme maxima na cestě z u do kořene. Nyní bude platit, že ve stromě mají nenulovou délku cesty právě uzly s hodnotami $v_1, \ldots, v_i$, přičemž délka uložená v uzlu s hodnotou v je maximum z d_j pro taková j, že $v_j = v$ a $1 \leq j \leq i$ (speciálně tedy neplatí, že hodnota d_j pro $j < i$ je nutně rovna hodnotě d_i uložené v uzlu s hodnotou $v_i = v_j$). Výpočet hodnot z_i bude opět podobný, jen budeme postupovat od konce a hledat největší délku mezi uzly s hodnotami vyššími než v_i.

Zbývá tedy vyřešit, jak najdeme onu největší délku uloženou v uzlu s hodnotou menší než nějaké x. Pro její nalezení použijeme modifikovanou proceduru na vyhledání prvku v binárním vyhledávacím stromu. V průběhu hledání budeme největší délku počítat průběžně, na začátku ji nastavíme na 0. Rozlišíme tři případy, podle toho, zda je hodnota v uzlu menší než, větší než nebo rovna x. Je-li hodnota uzlu větší než x, pak jen pokračujeme v hledání v levém podstromu. Je-li menší, pak maximum porovnáme s délkou uloženou v uzlu a maximem délek uložených v levém podstromu (protože si pamatujeme maximum délek v podstromu, stačí přečíst

hodnotu z levého syna) a dále pokračujeme v pravém podstromu. Nu a pokud je hodnota rovna x, tak maximum porovnáme jen s uloženým maximem v levém synovi a skončíme. Poté, co dojdeme do uzlu s hodnotou x (takový tam určitě je), bude spočítané maximum buď 0 (to pokud neexistuje žádná uložená hodnota v_i menší než x) nebo právě požadovaná délka. Časová složitost této prohledávací operace je úměrná hloubce podstromu, tedy $\mathcal{O}(\log N)$.

Shrňme si nyní celý postup. Nejprve v čase $\mathcal{O}(N \log N)$ setřídíme výšky, pak z nich v lineárním čase sestavíme strom, v čase $\mathcal{O}(N \log N)$ spočteme hodnoty d_i a z_i (N-krát voláme dvě operace s logaritmickou složitostí), v lineárním čase pro každé k zjistíme, kolik existuje vrcholů, které jsou na k-tém místě v nějaké nejdelší rostoucí podposloupnosti, a nakonec, opět v lineárním čase, vypíšeme výsledek. Pokud se týká paměti, potřebujeme si uložit hodnoty v_i, d_i a z_i, pomocný vyhledávací strom použije $\mathcal{O}(N)$ paměti a k uložení počtů vrcholů na k-tém místě v nejdelší podposloupnosti nám též postačí lineární paměť. Časová složitost námi navrženého algoritmu je $\mathcal{O}(N \log N)$ a paměťová $\mathcal{O}(N)$.

```
program p31;

const maxN = 1000;

type PUzel = ^TUzel;
     TUzel = record
         hodnota: longint;
         levy, pravy: PUzel;
         delka, delkaVPodstromu: integer;
     end;
     TPoleVysek = array[1..maxN] of longint;

var N: integer;          { počet kopců }
    v: TPoleVysek;       { výšky kopců }
    strom: PUzel;        { pomocný strom}
    rostouciDo, rostouciZ: array[1..maxN] of integer;
    { délky nejdelších rostoucích posloupností končících (rostouciDo)
      a začínajících (rostouciZ) daným kopcem }
    maxDelka: integer;  { délka nejdelší rostoucí posloupnosti }

function max(a, b: integer): integer;
begin
    if a > b then max := a
    else max := b;
end;

procedure nacti; { načte hodnoty ze vstupu }
var i: integer;
begin
    readln(N);
    for i := 1 to N do
        read(v[i]);
end;
```

```
function postavStrom(var pole: TPoleVysek; zacatek, konec: integer): PUzel;
{ vyrobí z hodnot pole[zacatek...konec] vyvážený binární vyhledávací strom,
  za předpokladu, že hodnoty v poli ‘pole’ jsou setříděné. }
var stred: integer;
    uzel: PUzel;
begin
    if zacatek > konec then
        { interval zacatek-konec je prázdný, vrať prázdný strom }
        postavStrom := nil
    else begin
        stred := (zacatek + konec) div 2; { prostřední prvek }
        new(uzel);                        { vyrob uzel pro střed }
        uzel^.hodnota:= pole[stred];
        uzel^.delka := 0;
        uzel^.delkaVPodstromu := 0;
        { rekurzivně postav oba synovské podstromy }
        uzel^.levy := postavStrom(pole, zacatek, stred - 1);
        uzel^.pravy := postavStrom(pole, stred + 1, konec);
        postavStrom := uzel;              { vrať vytvořený uzel }
    end;
end;

procedure vymazStrom(uzel: PUzel); { nastaví všem uzlům nulové délky }
begin
    if uzel <> nil then begin
        uzel^.delka := 0;
        uzel^.delkaVPodstromu := 0;
        vymazStrom(uzel^.levy);
        vymazStrom(uzel^.pravy);
    end;
end;

function nejdelsiNizsi(uzel: PUzel; hodnota: longint): integer;
{ najde maximum z délek uložených u uzlů s hodnotou menší nez zadaný parametr }
var nejdelsi: integer;
begin
    nejdelsi := 0;
    while uzel <> nil do begin { hledáme uzel s hodnotou ‘hodnota’ }
        if uzel^.hodnota = hodnota then begin { podívej se do levého podstromu }
            if uzel^.levy <> nil then
                nejdelsi := max(nejdelsi, uzel^.levy^.delkaVPodstromu);
            uzel := nil; { konec cyklu }
        end else if uzel^.hodnota > hodnota then
            uzel := uzel^.levy { jen pokračujeme v hledání uzlu v levém podstromu }
        else begin {uzel^.hodnota < hodnota}
            { porovnej délky v levém podstromu i v uzlu samotném }
            nejdelsi := max(nejdelsi, uzel^.delka);
            if uzel^.levy <> nil then
                nejdelsi := max(nejdelsi, uzel^.levy^.delkaVPodstromu);
            { ... a pokračuj pravým podstromem }
            uzel := uzel^.pravy;
        end;
    end;
    nejdelsiNizsi := nejdelsi;
end;
```

```
function nejdelsiVyssi(uzel: PUzel; hodnota: longint): integer;
{ najde maximum z délek uložených u uzlů s hodnotou větší nez zadaný parametr }
var nejdelsi: integer;
begin
    nejdelsi := 0;
    while uzel <> nil do begin { hledáme uzel s hodnotou ‘hodnota’ }
        if uzel^.hodnota = hodnota then begin { podívej se do pravého podstromu }
            if uzel^.pravy <> nil then
                nejdelsi := max(nejdelsi, uzel^.pravy^.delkaVPodstromu);
            uzel := nil; {konec cyklu}
        end else if uzel^.hodnota < hodnota then
            uzel := uzel^.pravy { jen pokračujeme v hledání uzlu v pravém podstromu }
        else begin {uzel^.hodnota > hodnota}
            { porovnej délky v pravém podstromu i v uzlu samotném }
            nejdelsi := max(nejdelsi, uzel^.delka);
            if uzel^.pravy <> nil then
                nejdelsi := max(nejdelsi, uzel^.pravy^.delkaVPodstromu);
            { ... a pokračuj levým podstromem }
            uzel := uzel^.levy;
        end;
    end;
    nejdelsiVyssi := nejdelsi;
end;

procedure nastavDelku(uzel: PUzel; hodnota: longint; delka: integer);
{ najde ve stromě uzel s hodnotou ‘hodnota’ a nastaví mu zadanou délku }
{ také přepočítá příslušné hodnoty ‘delkaVPodstromu’ }
begin
    while uzel <> nil do begin { hledáme uzel s hodnotou ‘hodnota’ }
        if uzel^.hodnota = hodnota then begin { našli jsme }
            uzel^.delka := delka;
            uzel^.delkaVPodstromu := max(uzel^.delkaVPodstromu, delka);
            uzel := nil; { konec cyklu }
        end else begin
            { uprav maximum v podstromu }
            uzel^.delkaVPodstromu := max(uzel^.delkaVPodstromu, delka);
            { ... a vyber následující uzel }
            if uzel^.hodnota < hodnota then
                uzel := uzel^.pravy { hodnota je v pravém podstromu }
            else
                uzel := uzel^.levy; { hodnota je v levém podstromu }
        end;
    end;
end;

procedure zaradDolu(var pole: TPoleVysek; index, maxIndex:integer);
{ používáno procedurou heapSort. ‘Pole’ je reprezentace max-haldy v poli s délkou
  ‘maxIndex’; procedura zajistí, že prvek s indexem ‘index’ "probublá" dolů až na
  správné místo -- tj. buď bude listem, nebo jeho synové budou mít nižší hodnoty. }
var p: longint;
    j: integer;
begin
    while index * 2 <= maxIndex do begin
        { do ‘j’ dáme index syna s vyšší hodnotou }
        if index * 2 = maxIndex then
```

```
            j := index * 2 { jen jeden syn }
        else if pole[2*index] > pole[2*index + 1] then
            j := 2*index
        else
            j := 2*index + 1;
        { překontrolujeme podmínku }
        if pole[index] > pole[j] then
            index := maxIndex + 1 { podmínka splněna, ukončíme cyklus }
        else begin
            { podminka neplatí, vyměň hodnoty a pokračuj synem }
            p := pole[index]; pole[index]:= pole[j]; pole[j] :=p;
            index := j;
        end;
    end;
end;

procedure heapSort(var pole: TPoleVysek; delka: integer);
var i, pul: integer;
    p: longint;
begin
    { v poli si reprezentujeme haldu tak, že synové uzlu s indexem ‘i’ mají index 2*i
      a 2*i+1. Toto je obvyklá reprezentace, pokud první prvek pole má index 1. }
    { postavíme maximovou haldu }
    pul := delka div 2;
    for i := pul downto 1 do
        zaradDolu(pole, i, delka);
    { od konce tvořime setříděnou posloupnost }
    for i := delka downto 2 do begin
        { odstraníme maximum }
        p := pole[1]; pole[1] := pole[i]; pole[i] := p;
        { obnovíme haldu }
        zaradDolu(pole, 1, i - 1);
    end;
end;

procedure pripravaStromu; { z hodnot v poli ‘v’ vyrobí vyvážený strom }
var i: integer;
    setridenePole: TPoleVysek; { pomocné pole }
    pocetUnikatnich: integer;  { počet unikátních hodnot v pomocném poli }
begin
    { nejprve hodnoty překopírujeme do pomocného pole }
    for i:= 1 to N do
        setridenePole[i] := v[i];
    { pak je setřídíme }
    heapSort(setridenePole, N);
    { odstraníme duplicity v poli -- hodnoty v pomocném poli na indexech
      1..pocetUnikatnich budou všechny hodnoty z původního pole, ale budou
      navzájem různé a setříděné. To snadno uděláme kontrolou sousedních prvků. }
    if N = 0 then
        pocetUnikatnich := 0 { žádné hodnoty, žádné nejsou unikátní }
    else begin
        pocetUnikatnich := 1; { první hodnota v poli je určitě unikátní }
        for i := 2 to N do
            if setridenePole[i] = setridenePole[pocetUnikatnich] then begin
                { hodnota je stejná jako předchozí, nic neděláme }
```

```
            end else begin
                { nová hodnota -- přidáme na konec }
                pocetUnikatnich := pocetUnikatnich + 1;
                setridenePole[pocetUnikatnich] := setridenePole[i];
            end;
    end;
    { nakonec vyrobíme strom }
    strom := postavStrom(setridenePole, 1, pocetUnikatnich);
end;

procedure vypocetDelek;
var i: integer;
    vyska: longint;
    delka: integer;
begin
    maxDelka := 0;
    { nejprve počítáme délky posloupností končící daným kopcem }
    for i:= 1 to N do begin
        vyska := v[i];
        delka := nejdelsiNizsi(strom, vyska) + 1;
        nastavDelku(strom, vyska, delka);
        rostouciDo[i] := delka;
        if delka > maxDelka then { zároveň zjistíme maximum }
            maxDelka := delka;
    end;

    { nyní spočítáme délky posloupností začínajících daným kopcem }
    vymazStrom(strom);
    for i:= N downto 1 do begin
        vyska := v[i];
        delka := nejdelsiVyssi(strom, vyska) + 1;
        nastavDelku(strom, vyska, delka);
        rostouciZ[i] := delka;
    end;
end;

procedure vypis;
var i: integer;
    pocty: array[1..maxN] of integer;
    { pro každé 'i' obsahuje počet vrcholů, které mohou být na i-tém místě
      v nejdelší rostoucí podposloupnosti }
begin
    { vynulujeme pole }
    for i := 1 to N do
        pocty[i] := 0;
    { spočteme hodnoty pole pocty }
    for i := 1 to N do
        if rostouciDo[i] + rostouciZ[i] - 1 = maxDelka then
            { leží v nějaké nejdelší posloupnosti }
            pocty[rostouciDo[i]] := pocty[rostouciDo[i]] + 1;

    for i := 1 to N do begin
        if i > 1 then
            write(' ');
        if rostouciDo[i] + rostouciZ[i] - 1 = maxDelka then begin
```

```
            if pocty[rostouciDo[i]] = 1 then
                write('musím')
            else
                write('mohu')
        end else
            write('nemohu');
    end;
    writeln;
end;

begin
    nacti;
    pripravaStromu;
    vypocetDelek;
    vypis;
end.
```

P-III-2 Velechrám

Nejprve si úlohu převedeme do teorie grafů. Z velechrámu uděláme graf, jehož množinu vrcholů tvoří řádky a sloupce šachovnice. Hrana mezi i-tým řádkem a j-tým sloupcem vede právě tehdy, když na příslušném poli stojí sloup. Nyní místo sloupů barvíme hrany tohoto grafu.

Dvě hrany mají společný vrchol pravě tehdy, když odpovídají sloupům, které leží ve stejném řádku nebo sloupci. Různé barvy sloupů ve stejném řádku a sloupci znamenají různé barvy hran, které mají společný vrchol. Chceme tedy obarvit hrany zkonstruovaného grafu tak, aby u každého vrcholu byly barvy všech hran různé.

Všimněte si, že potřebujeme alespoň tolik barev, kolik je maximální stupeň grafu Δ, tj. největší počet hran vedoucí z některého z vrcholů grafu. Ukážeme, že nám jich zároveň právě tolik stačí. Navíc ukážeme, jak takové obarvení i rychle najít.

Pojďme rovnou na algoritmus. Začneme s grafem bez hran. Potom budeme hrany v libovolném pořadí po jedné přidávat.

Při přidávání hrany e mohou nastat dvě možnosti. Pokud se na sousedních hranách přidávané hrany e nevyskytuje ještě všech Δ barev, obarvíme e libovolnou ze zbývajících barev.

V opačném případě to bude složitější. Označme si jako C_e množinu barev přiřazených hranám, které mají s e alespoň jeden společný vrchol. Protože e má alespoň Δ sousedních hran, s oběma koncovými vrcholy e je incidentní alespoň jedna již obarvená hrana (Δ je maximální stupeň pomocného grafu). Označme C_1 a C_2 množinu barev sousedních hran incidentních s jednotlivými koncovými vrcholy hrany e. Vybereme dvojici barev $A \in C_1 \setminus C_2$ a $B \in C_2 \setminus C_1$. Takové barvy lze určitě vybrat, protože $C_1 \cup C_2 = C_e$ a zároveň $C_1 \neq C_e \neq C_2$.

Nyní přebarvíme některé hrany tak, abychom mohli e obarvit buď barvou A nebo barvou B. Zahoďme na chvíli všechny hrany, které nemají barvu A nebo B. Protože z jednoho vrcholu mohou vést pouze dvě hrany obarvené jednou z barev A a B, v grafu nám zbudou pouze cesty a kružnice. Navíc v koncových vrcholech hrany e začínají některé z cest, protože u obou koncových vrcholů e jedna z barev A

a B chybí. Pokud jsou tyto dvě cesty různé, prohodíme barvy A a B v jedné z těchto cest. Tím určitě neporušíme podmínku na různost barev u žádného vrcholu na cestě a navíc nyní chybí jedna z barev A a B mezi sousedy e. Obarvíme tedy touto barvou hranu e.

Může se nám stát, že koncové vrcholy e jsou spojeny cestou? Předpokládejme, že to tak je. Připomeňme si, že vrcholy pomocného grafu odpovídají řádkům a sloupcům šachovnice a hrany vždy vedou mezi jedním z řádků a jedním ze sloupců. Na cestě spojující koncové vrcholy hrany e se tedy pravidelně střídají vrcholy odpovídající řádkům a sloupcům šachovnice a tedy tato cesta obsahuje lichý počet hran. V takovém případě, ale její první a poslední hrana musí mít stejnou barvu (na cestě se pravidelně střídají hrany barvy A a hrany barvy B), což není možné.

Tím jsme ukázali, že vždy dokážeme přebarvit některé hrany tak, abychom mohli korektně obarvit nově přidanou hranu. Po přidání všech hran do pomocného grafu tak dostaneme korektní obarvení hran grafu.

Podívejme se na časovou a paměťovou složitost navrženého algoritmu. V naší implementaci algoritmu si potřebujeme pamatovat ke každému vrcholu v a barvě c, která hrana s barvou c je incidentní s vrcholem v. Samozřejmě si pamatujeme strukturu grafu a informace o jeho hranách a vrcholech. To nám dává paměťovou složitost $\mathcal{O}(N\Delta + M)$. Algoritmus přidává M hran a při každém přidání musí přebarvit až $\mathcal{O}(N)$ hran (nikdy nepřebarvujeme víc než dvě hrany u jednoho vrcholu). Celková časová složitost našeho algoritmu tedy je $\mathcal{O}(NM)$.

```
program p32;
const
  maxn  = 500;        { maximální rozměr šachovnice }
  maxv  = 2*maxn;     { max. počet vrcholů grafu }
  maxm  = maxn*maxn;  { max. počet hran grafu }
var
  n, m, k : Integer;  { počet vrcholů hran; max. stupeň }
  edge : array[1..maxm] of record             { jednotlivé hrany }
               color: Integer;                   { barva hrany }
               vert : array[1..2] of Integer;    { krajní vrcholy }
           end;
  inc  : array[1..maxv, 1..maxn] of Integer;  { inc[v,b] = hrana barvy 'b'
                                                u vrcholu 'v' }
  ec   : array[1..maxv] of Integer;           { stupně vrcholů }

procedure uncolor(e :Integer);                { odbarví hranu }
var v : Integer;
begin
   for v:= 1 to 2 do
      inc[edge[e].vert[v], edge[e].color] := 0;
end;

procedure color(e, c :Integer);               { obarví hranu }
var v : Integer;
begin
   edge[e].color := c;
   for v:= 1 to 2 do
```

```
      inc[edge[e].vert[v], c] := e;
end;

procedure recolor(v, remc, addc :Integer);    { přebarví cestu }
var e, other : Integer;
begin
  if inc[v][remc] > 0 then
  begin
     e := inc[v, remc];
     uncolor(e);

     other := edge[e].vert[1];
     if other = v then other := edge[e].vert[2];
     recolor(other, addc, remc);

     color(e, addc);
  end;
end;

var i, j, remc, addc, x, y : Integer;
    colored : Boolean;
begin
   Readln(n, m);                              { načteme vstup }
   for i:= 1 to m do
   begin
      Readln(x, y);
      edge[i].vert[1] := x;
      edge[i].vert[2] := y+n;
      ec[x] := ec[x]+1;
      ec[y+n] := ec[y+n]+1;
   end;

   k := 0;                                    { zjistíme max. stupeň }
   for i:= 1 to 2*n do
      if ec[i] > k then
         k := ec[i];

   for i:= 1 to m do                          { postupně přidáváme hrany }
   begin
      colored := false;
      for j:= 1 to k do                       { je ji možno obarvit rovnou? }
         if (inc[edge[i].vert[1], j] = 0) and (inc[edge[i].vert[2], j] = 0) then
         begin
            color(i, j);
            colored := true;
            break;
         end;

      if colored then continue;

      for j:= 1 to k do                       { hledáme volné barvy }
         if (inc[edge[i].vert[1], j] = 0) and (inc[edge[i].vert[2], j] > 0) then
         begin
            addc := j;
            break;
```

```
         end;
      for j:= 1 to k do
         if (inc[edge[i].vert[1], j] > 0) and (inc[edge[i].vert[2], j] = 0) then
         begin
            remc := j;
            break;
         end;

      recolor(edge[i].vert[1], remc, addc);   { přebarvíme cestu }
      color(i, remc);                         { a obarvíme novou hranu }
   end;

   Writeln(k);                                { už jen vypsat výstup }
   for i:= 1 to m do
      Writeln(edge[i].color);
end.
```

P-III-3 Překládací stroje

a) Překládací stroj bude pracovat na následující jednoduché myšlence: ke každé levé kulaté závorce obsahuje každý řetězec v množině A pravou závorku. Můžeme tedy levé kulaté závorky přeložit na řetězec a, pravé kulaté závorky na řetězec bb a na konec řetězce připsat libovolný počet znaků b. Aby se ve vytvořeném řetězci nepromíchaly znaky a a b mezi sebou, tj. nejdříve byly všechny znaky a a pak teprve znaky b, po nalezení prvního znaku, který není levou kulatou závorkou, budeme očekávat jen pravé kulaté závorky, tj. budeme překládat jen řetězce `(((···)))`. Řetězec `(((···)))` tvořený n dvojicemi závorek, se tedy přeloží na řetězec s n znaky a a alespoň $2n$ znaky b.

Formálně je tedy vytvořený překládací stroj pětice (K, Σ, P, q_0, F), kde $K = \{\textbf{začátek}, \textbf{konec}\}$, $\Sigma = \{\texttt{(}, \texttt{)}, \texttt{[}, \texttt{]}, \texttt{\{}, \texttt{\}}, a, b\}$, $q_0 = \textbf{začátek}$ a $F = \{\textbf{konec}\}$. Množina P pak obsahuje následující překladová pravidla:

- (**začátek**, `(`, a, **začátek**), tj., na začátku vypisujeme za levé kulaté závorky znaky a,
- (**začátek**, `)`, bb, **konec**) a (**konec**, `)`, bb, **konec**), tj. za pravé kulaté závorky vypisujeme řetězec bb,
- (**konec**, \$, b, **konec**), tj. na konec výstupního řetězce vypíšeme libovolný počet znaků b.

b) Do hledané množiny E prostě vhodně vložíme obě množiny A a D, a to tak, že řetězce začínající znakem a budou odpovídat řetězcům množiny A a řetězce začínající znakem b řetězcům množiny D. Řetězce množiny D lze do množiny E zakódovat jednoduše tak, že na jejich začátek připíšeme písmeno b, znaky a a b zdvojíme a znaky c nahradíme řetězcem ab. Řetězec $aabbcc \in D$ bude tedy odpovídat řetězci $baaaabbbbabab$.

Zakódování řetězců množiny A je složitější: každý ze znaků `(`, `)`, `[`, `]`, `{` a `}` nahradíme řetězcem aaa, aab, aba, abb, baa a bab, a navíc na začátek řetězce přidáme písmeno a. Tím jsme nadefinovali množinu E.

Samotné překládací stroje M_1 a M_2 je již snadné sestrojit. První stroj M_1 je pětice (K, Σ, P, q_0, F), kde $K = \{\textbf{začátek}, \textbf{stav}\}$, $\Sigma = \{\texttt{(}, \texttt{)}, \texttt{[}, \texttt{]}, \texttt{\{}, \texttt{\}}, a, b\}$, $q_0 =$ **začátek** a $F = \{\textbf{stav}\}$. Množina P pak obsahuje následující překladová pravidla:

- $(\textbf{začátek}, a, \varepsilon, \textbf{stav})$, nejdříve zkontrolujeme, že řetězec začíná znakem a,
- pak už jen překládáme řetězec pomocí pravidel
  $(\textbf{stav}, aaa, \texttt{(}, \textbf{stav})$, $(\textbf{stav}, aab, \texttt{)}, \textbf{stav})$, $(\textbf{stav}, aba, \texttt{[}, \textbf{stav})$,
  $(\textbf{stav}, abb, \texttt{]}, \textbf{stav})$, $(\textbf{stav}, baa, \texttt{\{}, \textbf{stav})$ a $(\textbf{stav}, bab, \texttt{\}}, \textbf{stav})$.

Druhý stroj M_2 je pětice (K, Σ, P, q_0, F), kde $K = \{\textbf{začátek}, \textbf{stav}\}$, $\Sigma = \{a, b, c\}$, $q_0 =$ **začátek** a $F = \{\textbf{stav}\}$. Množina P pak obsahuje následující překladová pravidla:

- $(\textbf{začátek}, b, \varepsilon, \textbf{stav})$, nejdříve zkontrolujeme, že řetězec začíná znakem b,
- pak už jen překládáme řetězec pomocí pravidel
  $(\textbf{stav}, aa, a, \textbf{stav})$, $(\textbf{stav}, bb, b, \textbf{stav})$ a $(\textbf{stav}, ab, c, \textbf{stav})$.

c) První myšlenka, která každého asi napadne, je přeložit levé závorky na znaky a a pravé závorky na znaky b. Takovéto řetězce určitě obsahují stejný počet znaků a a b, nicméně určitě takto nezískáme všechny řetězce množiny C (např. ty, které začínají znakem b). Zkusme tedy tuto myšlenku vylepšit a pro jeden typ závorek překládat levé závorky na a a pro jiný na b. Uvažujeme tedy následující překládací stroj P tvořený pěticí (K, Σ, P, q_0, F), kde $K = \{\textbf{stav}\}$, $\Sigma = \{\texttt{(}, \texttt{)}, \texttt{[}, \texttt{]}, \texttt{\{}, \texttt{\}}, a, b\}$, $q_0 =$ **stav**, $F = \{\textbf{stav}\}$ a množina P obsahuje následující čtyři překladová pravidla: $(\textbf{stav}, \texttt{(}, a, \textbf{stav})$, $(\textbf{stav}, \texttt{)}, b, \textbf{stav})$, $(\textbf{stav}, \texttt{[}, b, \textbf{stav})$ a $(\textbf{stav}, \texttt{]}, a, \textbf{stav})$.

Zbývá si rozmyslet, že množinu A přeloží stroj P na množinu C. Zjevně každé slovo, které stroj P vytvoří, náleží do množiny C. Nyní dokážeme indukcí, že stroj P vytvoří každé slovo $x_1x_2\ldots x_k \in C$ (číslo k je zjevně sudé). Pokud $k = 0$, je tvrzení jasné. Pokud $k = 2$, pak $x_1x_2 = ab$ nebo $x_1x_2 = ba$ a stroj P přeloží řetězec $y_1y_2 = \texttt{()}$ na ab a řetězec $y_1y_2 = \texttt{[]}$ na řetězec ba.

Předpokládejme tedy, že $k \geq 4$. Pokud $x_1 = a$ a $x_k = b$, pak podle indukčního předpokladu existuje řetězec $y_2\ldots y_{k-1}$, které se přeloží na řetězec $x_2\ldots x_{k-1}$. Řetězec $\texttt{(}y_2\ldots y_{k-1}\texttt{)}$ pak stroj P přeloží na $x_1x_2\ldots x_k$. Analogicky postupujeme, pokud $x_1 = b$ a $x_k = a$, s tím rozdílem, že místo kulatých závorek doplníme závorky hranaté.

Zaměřme se nyní na případ, kdy $x_1 = x_k = a$. Nechť $f(i)$ je rovno rozdílu počtu znaků a a počtu znaků b v řetězci $x_1\ldots x_i$. Zjevně platí $f(1) = 1$, $f(k-1) = -1$ a $f(k) = 0$. Protože se hodnoty $f(i)$ a $f(i+1)$ liší právě o 1 a $f(1) > 0 > f(k-1)$, musí existovat index $1 < \ell < k$, pro který je hodnota $f(\ell)$ nulová. Zjevně pak platí, že $x_1\ldots x_\ell \in C$ a $x_{\ell+1}\ldots x_k \in C$. Podle indukčního předpokladu pak existuje řetězec $y_1\ldots y_\ell \in A$, který stroj P přeloží na řetězec $x_1\ldots x_\ell$, a řetězec $y_{\ell+1}\ldots y_k \in A$, který se přeloží na řetězec $x_{\ell+1}\ldots x_k$. Řetězec $y_1\ldots y_k \in A$ je pak strojem P přeložen na řetězec $x_1x_2\ldots x_k$. Případ $x_1 = x_k = b$ lze rozebrat analogicky.