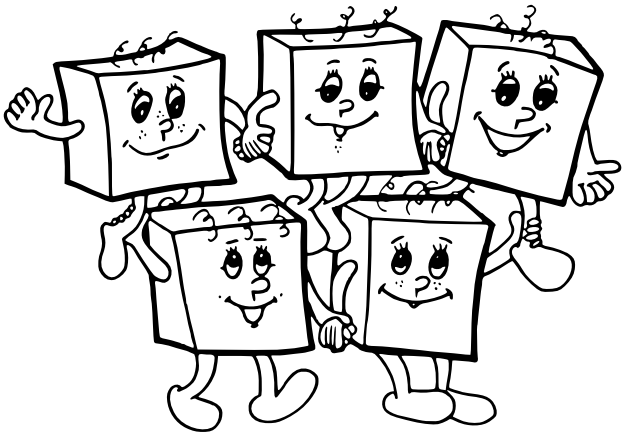
OLYMPIÁDA V INFORMATIKE

NA STREDNÝCH ŠKOLÁCH

http://oi.sk/



dvadsiaty šiesty ročník
školský rok 2010/11

riešenia celoštátneho kola
kategória A

1. súťažný deň

A-III-1 Romantické básničky

Vyriešime najskôr jednoduchšiu úlohu: ku zadanému slovu nájsť v lexikóne **ľubovoľné** z tých, ktoré sa s ním najviac rýmujú.

Jedno možné efektívne riešenie je použiť dátovú štruktúru nazývanú písmenkový strom (po anglicky *trie*). Písmenkový strom je zakorenený strom, v ktorom každý vrchol má najviac 26 synov, a hrany do synov sú označené rôznymi písmenkami (od `a` po `z`). Každá cesta z koreňa nadol zodpovedá slovu, ktoré si „prečítame“ na hranách, po ktorých ideme.

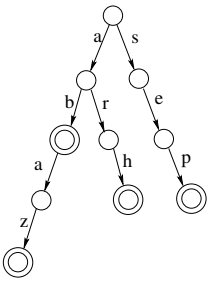
Písmenkový strom sa dá použiť na uloženie množiny slov. Jednoducho vytvoríme všetky vrcholy, ktoré treba na to, aby sme si mohli na ceste z koreňa dole „prečítať“ každé z našich slov, a následne označíme tie vrcholy, kde niektoré z uložených slov končí. (Napríklad si do toho vrcholu napíšeme poradové číslo slova, ktoré sa tam končí.)

Ako pomocou písmenkového stromu vyriešiť našu zjednodušenú úlohu? Úplne jednoducho. Na začiatku zoberieme všetky slová v lexikóne a vložíme ich do písmenkového stromu – lenže nie normálne, ale odzadu, teda začínajúc ich posledným písmenom.

Príklad takéhoto stromu si môžete pozrieť na obrázku 1. Dvojitým krúžkom sú znázornené vrcholy, kde niektoré zo slov končí. Z každého vrcholu dodola vedie v skutočnosti až 26 hrán – tie, ktoré nevedú nikam (NULL pointre), sme pre prehľadnosť nekreslili.

Keď nám teraz príde ľubovoľná otázka, môžeme aj tú čítať odzadu a zároveň putovať dodola po písmenkovom strome. Napríklad majme lexikón z obrázku 1 a predstavme si, že nám prišla otázka `kobra`. Čítajúc písmená `a` a `r` sa dostaneme do jedného z vrcholov v našom písmenkovom strome. Odtiaľ však už ďalej dodola hrana na písmeno `b` nevedie, vieme teda, že lepší rým ako dve písmená nevyrobíme.

Ako teraz nájsť jedno z najlepšie sa rýmujúcich slov? Na to sa stačí z vrcholu, kde práve sme, ľubovoľnou cestou pohnúť dodola, a to až kým nenájdeme vrchol, kde nejaké slovo končí (respektíve začína). Toto sa nám



Obr. 1: Písmenkový strom, do ktorého sme odzadu vložili slová `ba`, `hra`, `pes` a `zaba`.

skôr či neskôr musí stať – najneskôr vtedy, keď prídeme do listu, keďže každý list nutne zodpovedá nejakému slovu.

V našom príklade by sme mali jedinú možnosť: pohnúť sa dodola na písmeno `h`, čím sme úspešne zistili, že najlepším rýmom ku slovu `kobra` je slovo `hra`.

V zadaní súťažnej úlohy ale bola ešte jedna požiadavka: ak je viac možných odpovedí, musíme nájsť tú lexikograficky najmenšiu (čítané samozrejme odpredu). Vyššie popísané riešenie teda musíme upraviť tak, aby vedelo splniť aj túto požiadavku.

V prvom rade si rozmyslite, že samotný písmenkový strom, ktorý sme si zostrojili, nám nestačí – nevieme z neho lokálne zistiť, ktoré slovo si vybrať, ak ich máme na výber viac. Napríklad si predstavte, že máme v lexikóne slová `zaba` a `huba` a hľadáme najlepší rým k slovu `hudba`. Pri hľadaní nájdeme vrchol zodpovedajúci ich spoločnému suffixu `ba`. Z neho dodola ale vedie viac možností – hrana na písmeno `a` aj hrana na písmeno `u`. A správna odpoveď `huba` sa v tomto prípade skrýva pod písmenom `u` – to ale nemáme ako v danej chvíli zistiť, lebo prvé písmená slov `zaba` a `huba` sú ukryté niekde hlbšie v strome.

Keďže primárnym kritériom hodnotenia riešenia je rýchlosť odpovedania na otázky, potrebujeme si hľadané odpovede nejak predpočítať. Presnejšie, v každom vrchole nášho písmenkového stromu si budeme pamätať odpoveď – číslo slova z lexikónu, ktoré máme vypísať na výstup, ak sme pri hľadaní rýmu skončili v danom vrchole.

Ako tieto hodnoty spočítať efektívne? Pomohlo by nám, keby slová v lexikóne boli usporiadané podľa abecedy. Potom totiž stačí v tomto poradí vkladať ich reverzy do písmenkového stromu a v každom vrchole si zapamätať číslo slova, pri spracúvaní ktorého sme dotyčný vrchol vytvorili.

No a ako najefektívnejšie usporiadať lexikón? Jednoducho: využijeme na to opäť písmenkový strom. (Spravíme si inú premennú, ale použijeme tú istú implementáciu písmenkového stromu.) Tentokrát doň najskôr klasicky (odpredu) vložíme všetky slová z lexikónu, a následne tento strom prehľadáme do hĺbky. Synov každého vrcholu budeme spracúvať v abecednom poradí. Poradie, v ktorom stretneme vrcholy zodpovedajúce koncom slov, takto bude zjavne zodpovedať ich abecednému poradiu.

Na usporiadanie lexikónu pomocou písmenkového stromu potrebujeme čas priamo úmerný súčtu dĺžok jeho slov. Taktiež vytvorenie písmenkového stromu reverzov slov z lexikónu vieme potom spraviť v rádovo rovnakom čase. Následne spracovanie každej otázky prebehne priamočiaro: načítame otázku, v čase najviac lineárnom od jej dĺžky nájdeme vrchol zodpovedajúci najlepšiemu rýmu, z toho vrcholu v konštantnom čase vieme číslo slova, ktoré je správnou odpoveďou, a to len vypíšeme.

Aj fáza predspracovania, aj spracovanie jednej otázky, má teda optimálnu časovú zložitosť. (Lepšie to nejde, lebo musíme aspoň čítať vstup a vypisovať výstup.)

Listing programu:

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
```

```
using namespace std;

const int MAX_V = 1000047;

int n,m;

vector <string> lexikon;

// Vrati reverz stringu
string reversed(string s) {
        reverse(s.begin(),s.end());
        return s;
}

struct node {
        int sons[26], id; // id = Ktoremu slovu patrim?
        node();
};

node::node() {
        id=-1;
        for (int i=0;i<26;++i) sons[i]=-1;
}

struct trie {
        node a[MAX_V]; // Miesto pointrov pouzijeme pole
        int root,num;
        trie();
        void insert(const string &s, int n, bool vsetky);
        void lexicographic(int pos);
        int find(const string &s);
};

trie sorter, suffix; // Jeden strom na triedenie odpredu, druhy na reverzy

trie::trie() {
        root=0;
        num=1;
}

// Vlozi do trie retazec s
// ak vsetky==false, vklada klasicky: posledny vrchol oznaci n-kom
// ak vsetky==true, oznaci n-kom vsetky novovytvorene vrcholy
void trie::insert(const string &s, int n, bool vsetky=false) {
        int node=root;
        if (vsetky && a[node].id==-1) a[node].id=n;
        for (unsigned i=0;i<s.size();++i) {
                if (a[node].sons[s[i] - 'a']==-1) a[node].sons[s[i] - 'a'] = num++;
                node=a[node].sons[s[i] - 'a'];
                if (vsetky && a[node].id==-1) a[node].id=n;
        }
        if (!vsetky) a[node].id=n;
}

// Prechadza strom v lexikografickom poradi (prehladava ho do hlbky)
// Vzdy ked najde slovo, vlozi jeho reverz do stromu reverzov
void trie::lexicographic(int pos=0) {
        if (a[pos].id!=-1) { // Ak vo vrchole konci slovo
                suffix.insert( reversed(lexikon[a[pos].id]), a[pos].id, true );
        }
        for (int i=0;i<26;i++) { // Rekurzivne prehladaj
                if (a[pos].sons[i]!=-1)
                lexicographic(a[pos].sons[i]);
        }
}

// Najde najdlhsi prefix a vrati ID slova, ktoremu patri
int trie::find(const string &s) {
        int node=root;
        for (unsigned i=0;i<s.size();++i) {
                int next = a[node].sons[ s[i]-'a' ];
                if (next==-1) break; else node=next;
        }
        return a[node].id;
}
```

```
int main() {
        string s;
        cin >> n;
        for (int i=0;i<n;++i) {
                cin >> s;
                lexikon.push_back(s);
                sorter.insert( lexikon[i], i );
        }
        sorter.lexicographic();
        cin >> m;
        for (int i=0;i<m;++i) {
                cin >> s;
                cout << lexikon[ suffix.find(reversed(s)) ] << endl;
        }
}
```

A-III-2 Úrad

Štruktúra úradnikov tvorí strom, v ktorom sú úradníci vrcholy. Úlohy, ktoré vykonávajú úradnici budeme niekedy označovať ako farby vrcholov.

Pomalšie riešenia

Najjednoduchšie riešenie je pustiť z každého vrcholu prehľadávanie smerom nadol, ktoré hľadá vrcholy farby 1. Pokiaľ narazíme na vrchol rovnakej farby, tak prehľadávanie zastavíme. Pokiaľ narazíme na vrchol farby 1, tak vieme, že daný úradník niečo robí. Jedno prehľadávanie nám trvá $O(n)$ času, takže máme celkový čas $O(n^2)$.

Tento horný odhad časovej zložitosti je síce správny, ale nie tesný. Všimnime si ľubovoľnú konkrétnu farbu f. Z každého vrcholu tejto farby raz spustíme prehľadávanie. To sa ale zastaví, keď nájde pod sebou iné vrcholy tej istej farby, pod nimi už teda neprehľadáva. Keď teda uvažujeme dokopy všetky prehľadávania z vrcholov farby f, vieme, že prezrú dokopy každú hranu stromu najviac raz. Preto majú všetky tieto prehľadávania spolu časovú zložitosť $O(n)$. A keďže máme m farieb, je celková časová zložitosť $O(mn)$.

Iné riešenie s rovnakou časovou zložitosťou: Budeme priamo riešiť každú farbu jedným prechodom. Pre každú farbu $f \in \{2, 3, \ldots, m\}$ vykonáme nasledovný proces: Zmeníme strom na orientovaný graf, ktorého hrany smerujú smerom nahor. Do grafu pridáme nový „štartovný“ vrchol X, z neho budú smerovať všetky hrany do vrcholov s farbou 1. Zároveň pri spracúvaní farby f neuvažujeme hrany vedúce dohora z vrcholov farby f. Teraz už len spustíme z vrcholu X prehľadávanie. Ak narazíme na vrchol farby f, tak vieme, že tento úradník niečo robí. Takto nájdeme všetkých úradnikov farby f, čo niečo robia. Jedno prehľadávanie opäť trvá čas $O(n)$. Celkový čas máme teda $O(mn)$.

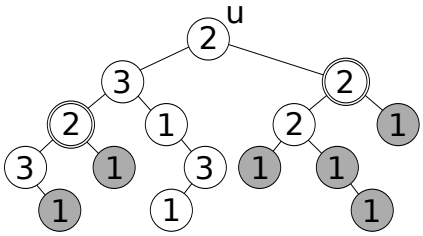
Vzorové riešenie

Medzi všetkými (aj nie priamymi) podriadenými úradníka u je niekoľko podriadených pre styk s verejnosťou. Ich počet označíme $P(u)$. Všetky hodnoty $P(u)$ vieme ľahko spočítať v čase $O(n)$ pomocou jedného prehľadávania do hĺbky. Stačí si uvedomiť, že $P(u)$ vieme spočítať ako súčet všetkých $P(x)$, kde x je priamym podriadeným u, a ešte $+1$ ak samotný u robí styk s verejnosťou.

Predstavme si teraz, že z úradníka u sme spustili smerom nadol prehľadávanie z prvého vyššie popísaného pomalšieho riešenia – zisťujeme teda, odkiaľ všadiaľ ku nám môžu prísť klienti. Zamestnancov pre styk s verejnosťou, ktorých toto prehľadávanie nájde, budeme volať *nepokrytými* a ich počet označíme $N(u)$. Zjavne nám stačí pre každé u spočítať $N(u)$: totiž u nič nerobí práve vtedy, keď je táto hodnota nulová.

Okrem zamestnancov pre styk s verejnosťou toto prehľadávanie nájde aj niekoľko iných zamestnancov, podriadených úradníka u, ktorí majú rovnakú farbu ako on. Týchto budeme volať jeho *poskokmi*. Poskok úradníka u je teda každý úradník v, ktorý má rovnakú farbu ako u a zároveň platí, že u je jeho najbližší nadriadený s touto farbou.

Na obrázku 2 je príklad časti hierarchie. Čísla vo vrcholoch sú farby, teda typy úloh, ktoré jednotliví úradníci robia. Úradník u je úplne hore (v koreni stromu). Dvojitým krúžkom sú vyznačení jeho poskokovia. Sivou farbou sú vyznačení tí úradníci pre styk s verejnosťou, ktorých poskokovia pokryli.



Obr. 2: Ukážka podstromu pre nejakého úradníka u.

Hodnotu $N(u)$ teraz vieme spočítať tak, že od hodnoty $P(u)$ odpočítame tých úradníkov, ktorých pokryjú poskokovia vrcholu u – teda odpočítame všetky $P(v)$, kde v je poskokom u. (Všimnite si, napr. pomocou obrázku 2, že toto môžeme spraviť, lebo množiny pracovníkov pokrytých rôznymi poskokmi úradníka u sú disjunktné. Formálny dôkaz by vyzeral takto: Uvažujme ľubovoľného úradníka x spomedzi $P(u)$ úradníkov pre styk s verejnosťou, ktorí sú podriadenými u. Na ceste medzi ním a u leží najviac jeden poskok u – viac ako jeden tam byť nemôže, lebo potom by ten z nich, ktorý je ďalej od u, nebol poskokom.)

Potrebujeme teda efektívne nájsť pre každého úradníka všetkých jeho poskokov. Všimnime si, že každý úradník iný ako minister je poskokom práve jedného iného. Preto nám pre každý vrchol stačí nájsť toho jeho nadriadeného, komu je poskokom. Ak to zvládneme v lineárnom čase pre celý strom, máme lineárne riešenie úlohy.

Najbližšieho nadriadeného rovnakej farby vieme pre všetky vrcholy nájsť takto: Najprv si naicializujeme m zásobníkov, jeden pre každú farbu. Teraz začneme strom prehľadávať do hĺbky. Keď prídeme do vrcholu v s farbou f, tak sa pozrieme na vrch zásobníku pre farbu f. Vrchol, ktorý je tam umiestnený, je ten, ktorému je aktuálny vrchol v poskokom. Toto si zapamätáme. Následne vložíme do zásobníku pre farbu f vrchol v a pokračujeme prehľadávaním jeho podstromu. A keď toto prehľadávanie skončí a z vrchola v odchádzame, tak ho zase vyberieme z vrchu príslušného zásobníku.

Celkovo teda vykonáme nasledovné:

1. Vypočítame prehľadávaním do hĺbky hodnoty $P(u)$ – počet podriadených pre styk s verejnosťou.
2. Vypočítame prehľadávaním do hĺbky pomocou m zásobníkov pre každého úradníka jeho poskokov.
3. Pre každého úradníka u vypočítame jeho hodnotu $N(u)$.
4. Úradníkov s $N(u) = 0$ vypíšeme na výstup.

Každý z týchto krokov nám zaberie $O(n)$ času a spotrebujeme $O(n)$ pamäte. V programe prvé dva kroky robíme súčasne počas jedného prehľadávania. Treba ešte dodať, že ministra je potrebné ošetriť samostatne – ten nič nerobí len vtedy, ak by nič nerobil v každej možnej farbe.

Listing programu:

```
#include <cstdio>
#include <vector>
#include <stack>
using namespace std;

struct Vrchol {
    vector<int> next;
    int color, seen, boss;
};

int N, M;
Vrchol v[100000];
stack<int> color_stack[100000];

// dfs(x) prehlada podstrom pod vrcholom x,
//   vrati pocet uplne vsetkych 1 v tomto podstrome (vratane pripadne vrcholu x)
//   a tiez ku kazdemu vrcholu najde najblizsieho predka rovnakej farby
int dfs(int x) {
    v[x].seen = (v[x].color==1 ? 1 : 0); // 1 ak my sme farby 1, 0 inak
```

```
    // Ak sme v koreni, vlozime ho do vsetkych zasobnikov, inak len do spravneho
    //   a tiez si zapamatame najblizsieho predka rovnakej farby ako mame my
    if (x == 0) {
        for (int i = 0; i <= M; i++) color_stack[i].push(0);
    } else {
        v[x].boss = color_stack[ v[x].color ].top();
        color_stack[ v[x].color ].push(x);
    }

    // Prehladame strom pod sebou
    for (unsigned i = 0; i < v[x].next.size(); i++) v[x].seen += dfs(v[x].next[i]);

    // Upraceme spravny zasobnik a vratime spocitany pocet jedniciek
    color_stack[v[x].color].pop();
    return v[x].seen;
}

int main() {
    scanf("%d%d", &N, &M);
    for (int i = 1; i < N; i++) {
        v[i].seen = 0;
        int p;
        scanf("%d", &p);
        v[p-1].next.push_back(i);
    }
    v[0].color = 0;
    for (int i = 1; i < N; i++) scanf("%d", &v[i].color);
    dfs(0);

    // v premennej seen v kazdom vrchole mame teraz pocet vsetkych jednotiek
    // pod nim, odratame tie pokryte vrcholmi, ktorym je on boss
    for (int i=1; i<N; i++) v[ v[i].boss ].seen -= v[i].seen;

    // vypiseme vsetkych, ktori nemaju pod sebou nepokryte jednotky
    for (int i=1; i<N; i++) if (v[i].seen==0) printf("%d\n",i+1);
}
```

A-III-3 Grafový počítač pomáha krtkovi

Triedenie

Našim cieľom je nájsť algoritmus, ktorý pomocou grafového počítača usporiada čísla efektívnejšie ako to vieme bez neho. Určite nebude existovať riešenie v lepšom ako lineárnom čase, keďže zadaných n čísel musíme aspoň načítať a vypísať. Ak teda nájdeme riešenie v čase $\Theta(n)$, budeme mať istotu, že je optimálne.

Ako na to? Pohľad do tabuľky operácií, ktoré vieme na grafovom počítači robiť, nám ponúka jedinú z nich, ktorá prihliada na váhy hrán: `Find`. Ak má `Find` viac možností, ktorý podgraf vybrať, máme zaručené, že vždy vyberie ten s najmenšou váhou. Na tejto operácii postavíme naše riešenie.

Začneme s prázdnym grafom. Postupne v ňom vyrobíme n hrán, ktorých váhy budú triedené čísla. A potom dokola n-krát pomocou `Find` nájdeme najľahšiu z hrán, jej váhu vypíšeme na výstup a dotyčnú hranu z grafu odstránime. Takéto riešenie bude zjavne správne a bude mať lineárnu časovú zložitosť.

Zopár detailov k implementácii: Graf, ktorý v prvej fáze vyrobíme, bude mať tvar hviezdy – i-temu číslu na vstupe bude zodpovedať hrana medzi vrcholmi s id 1 a $i+1$. Každý vrchol navyše dostane značku rovnú svojmu id. Toto potrebujeme na to, aby sme v druhej fáze vedeli, ktorú hranu vlastne vymazať.

V druhej fáze budeme pomocou `Find` hľadať najlacnejší podgraf tvorený jednou hranou. Tej prvý vrchol musí mať značku 1, druhý môže byť ľubovoľný. Keď tento podgraf nájdeme, cena jeho hrany je číslo, ktoré vypíšeme na výstup, a značka druhého vrcholu nám hovorí, ktorú hranu zmazať v pôvodnom grafe.

Listing programu:

```
procedure triedenie;
var n, i, x : Integer;
    G, H, hrana : Graph;
begin
   read(n);
   G := EmptyG; AddV(G,1);
```

```
   { nacitavame cisla a pridavame hrany }
   for i:=1 to n do begin read(x); AddV(G,i+1); AddE(G,1,i+1,x); end;

   { dokola hladame a vypisujeme najlacnejsiu hranu }
   hrana := EmptyG; AddV(hrana,1); AddV(hrana,undef); AddE(hrana,1,2,undef);
   for i:=1 to n do begin
      H := Find(G,hrana,IM_value,IM_any);
      writeln( GetE(H,1,2) );
      DelE(G,1,GetV(H,2));
   end;
end;
```

Krtkove brlôžky

Našou úlohou je nájsť súvislý podgraf s najmenším súčtom dĺžok hrán. Keďže sú dĺžky hrán kladné, hľadaný podgraf určite nebude obsahovať žiaden cyklus. Totiž keby nejaký cyklus obsahoval, môžeme jeho ľubovoľnú hranu zahodiť. Tým dostaneme podgraf, ktorý bude lacnejší (lebo sme zahodili hranu) a zároveň naďalej súvislý (lebo zvyšok cyklu naďalej spája vrcholy, ktoré spájala zahodená hrana).

Z toho plynie záver, že hľadaný najlacnejší podgraf je vždy strom obsahujúci všetkých n vrcholov. Takýto strom sa nazýva *kostra grafu*. Má vždy presne $n-1$ hrán. (Predstavme si krtka ako postupne čistí chodbičky. Na začiatku má n izolovaných brlôžkov – teda n komponentov súvislosti. Vždy, keď očistí chodbičku, spojí tým dva komponenty do jedného. Takže po očistení $n-1$ chodbičiek už bude celá nora tvoriť jeden komponent súvislosti a krtko môže ísť oddychovať.)

Náš cieľ teda môžeme sformulovať nasledovne: v danom grafe potrebujeme nájsť jeho *najlacnejšiu (najľahšiu) kostru*. Algoritmus, ktorý bude vzorovým riešením, si najskôr popíšeme všeobecne, až potom ukážeme, ako ho efektívne implementovať na grafovom počítači. (Pôjde o algoritmus, ktorý sa dá efektívne, hoci s horšou časovou zložitosťou, implementovať aj na počítači klasickom.)

Lema (pomocné tvrdenie): Nech G je súvislý ohodnotený graf, ktorého množina vrcholov je V a ktorého hrany majú navzájom rôzne ceny. Rozdeľme vrcholy G do dvoch disjunktných množín A a B. Potom najlacnejšia hrana medzi A a B musí byť súčasťou najlacnejšej kostry.

Dôkaz: Sporom. Najlacnejšiu hranu, ktorej jeden koniec je v množine A a druhý v množine B, nazvime h. Majme teda najlacnejšiu kostru, ktorá hranu h neobsahuje. Pridajme hranu h do nej. Tým nám určite vznikol cyklus: tvorí ho pridaná hrana h a cesta po pôvodnej kostre medzi koncovými vrcholmi hrany h. Jeden koniec tejto cesty leží v množine A, druhý v množine B, preto musí na tejto ceste existovať aspoň jedna hrana h', ktorá vedie medzi množinami A a B.

No a čo sa stane, keď teraz z nášho podgrafu túto hranu h' zahodíme? Výsledný graf bude naďalej súvislý, lebo sme zahodili hranu z cyklu. Bude to teda opäť nejaká kostra. Lenže hrana h bola najlacnejšia zo všetkých hrán vedúcich medzi A do B. Špeciálne teda je h lacnejšia aj ako h'. Ale to znamená, že nová kostra je lacnejšia ako tá pôvodná, a to je spor s predpokladom, že pôvodná kostra bola najlacnejšia.

Práve dokázaná lema nám umožní zostrojiť najlacnejšiu kostru postupne – v každom kroku k nej pridáme jednu novú hranu. Tento algoritmus ako prvý objavil český matematik Vojtěch Jarník, nezávisle na ňom neskôr Robert Prim, po nich sa nazýva Jarníkov-Primov algoritmus. Vyzerá nasledovne: Vrcholy grafu budeme mať rozdelené do dvoch skupín: už spojené (množina A) a ešte nespojené (množina B). Na začiatku je v množine A jeden ľubovoľný vrchol, všetky ostatné sú v množine B. V každom kroku teraz nájdeme najlacnejšiu hranu medzi A a B. Tú pridáme do zostrojovanej kostry a jej doteraz nepripojený koniec presunieme z B do A. Takto zjavne platí, že po každom kroku máme množinu A celú spojenú hranami, ktoré sme dovtedy vybrali. A vďaka dokázanej leme vieme, že každá z hrán, ktoré sme vybrali, musí v najlacnejšej kostre ležať. Keď teda po $n-1$ krokoch tento proces skončí, všetky vrcholy sú v množine A a vybrané hrany nutne tvoria najlacnejšiu kostru.

Všimnite si, že Jarníkov-Primov algoritmus zodpovedá „pažravému“ rozhodovaniu sa krtka. Ten začne v brlôžku, v ktorom sa prebudil, a v každom kroku si vyberie na prečistenie najkratšiu z chodbičiek, ku ktorým sa vie dostať (a ktoré nie je zbytočné čistiť). Tým postupne rozširuje sieť brlôžkov, do ktorých sa už dá dostať.

Priamočiara implementácia tohto algoritmu na klasickom počítači by mala časovú zložitosť $\Theta(n^3)$. Totiž $(n-1)$-krát hľadáme novú hranu do kostry. Na jej nájdenie vždy potrebujeme prezrieť všetky hrany medzi aktuálnymi množinami A a B, a tých môže byť až rádovo n^2.

Časová zložitosť sa dá zlepšiť na $\Theta(n^2)$ tak, že si budeme pre každý vrchol v B pamätať najkratšiu hranu, ktorá z neho vedie do nejakého vrcholu v A. Vždy, keď presúvame nejaký vrchol v z B do A, tak si tieto hodnoty v čase $O(n)$ prepočítame.

Pre riedke grafy existuje aj implementácia s časovou zložitosťou $\Theta(m \log n)$, kde m je počet hrán grafu. (Úprava vyzerá podobne ako u Dijkstrovho algoritmu: vrcholy množiny B máme uložené vo vhodnej prioritnej fronte tak, aby sme vedeli efektívnejšie nájsť ten, ktorý je momentálne k množine A najbližšie.)

Ako Jarníkov-Primov algoritmus efektívne implementovať na grafovom počítači? Naše riešenie bude mať časovú zložitosť $\Theta(n)$. Použijeme postup veľmi podobný riešeniu prvej podúlohy: novú hranu, ktorú treba pridať do kostry, nájdeme vhodným volaním `Find` v konštantnom čase.

Potrebujeme ešte vyriešiť jeden technický detail: Keď nájdeme novú hranu kostry, potrebujeme vedieť zistiť nie len jej cenu, ale aj čísla vrcholov, ktoré spája. Na to potrebujeme použiť značky. Lenže zároveň potrebujeme nejak vedieť špecifikovať množinu hrán, spomedzi ktorých najlacnejšiu hľadáme.

Existuje viacero možností, ako toto dosiahnuť, my si ukážeme jednu z nich: pridáme do grafu dva nové vrcholy a a b, pričom po každom kole algoritmu bude platiť, že vrchol a je spojený s vrcholmi v množine A a vrchol b s vrcholmi v množine B. Všetky tieto nové hrany budú mať váhu 0. Potom najlacnejšiu hranu z A do B nájdeme tak, že budeme hľadať najlacnejšiu cestu tvaru $a - x - y - b$. Keď ju nájdeme, pozrieme sa na cenu hrany $x - y$ a na značky týchto dvoch vrcholov. Takúto hranu pridáme do výstupu. (Všimnite si, že vyrobíme výstup v presne takej forme ako je v zadaní – teda vrcholy vo výstupe budú bez značiek a vybraté hrany budú mať každá svoju pôvodnú váhu.)

Listing programu:

```
function kostra(G : Graph) : Graph;
var n, i : Integer;
    vystup, cesta, vyber : Graph;
begin
   { oznacime povodne vrcholy cislami }
   n := CountV(G);
   for i:=1 to n do SetV(G,i,i);

   { pridame nove vrcholy predstavujuce komponenty }
   AddV(G,n+1); AddV(G,n+2);
   AddE(G,1,n+1,0); for i:=2 to n do AddE(G,i,n+2,0);

   { vyrobime si cestu s tromi hranami, ktoru budeme vyhladavat v G }
   cesta := EmptyG;
   AddV(cesta,n+1); AddV(cesta,undef); AddV(cesta,undef); AddV(cesta,n+2);
   AddE(cesta,1,2,undef); AddE(cesta,2,3,undef); AddE(cesta,3,4,undef);

   { vyrobime si vystupny graf s n izolovanymi vrcholmi }
   vystup := EmptyG;
   for i:=1 to n do AddV(vystup,undef);

   { hladame a pridavame hrany }
   for i:=1 to n-1 do begin
      { najdeme hranu a pridame ju do vystupu }
      vyber := Find( G, cesta, IM_value, IM_any );
      AddE( vystup, GetV(vyber,2), GetV(vyber,3), GetE(vyber,2,3) );
      { upravime mnoziny spojenych a nespojenych vrcholov }
      DelE( G, n+2, GetV(vyber,3) );
      AddE( G, n+1, GetV(vyber,3), 0 );
   end;
   kostra := vystup;
end;
```

SLOVENSKÁ KOMISIA OLYMPIÁDY V INFORMATIKE

DVADSIATY ŠIESTY ROČNÍK OLYMPIÁDY V INFORMATIKE

Zodpovedný redaktor: Michal Forišek
Sadzba programom LaTeX