Riešenia kategórie A

A-II-1 Výkrm

Z celého kráľovstva sa začali zbiehať odvážni junáci a junáčky, ktorí chceli kráľovskému páru pomôcť. Prvý odvážlivec predstavil kráľovnej Totňi nasledovné riešenie: Každý návrh skontrolujeme osobitne. Pre každé jedlo v ňom skúsime vybrať vhodné mesto, v ktorom ho má Hurim zjesť. Kde by mal zjesť prvé jedlo z plánu? Ak má na výber viacero miest, kde ho podávajú, zjavne *nič nepokazíme*, ak mu ho dáme v meste s najmenším číslom – ostane nám tak pri plnení zvyšku plánu na výber najviac iných miest. A v tejto úvahe môžeme pokračovať aj ďalej. Vždy, keď chceme Hurimovi dať nejaké ďalšie jedlo, ideme z mesta, kde Hurim naposledy jedol, ďalej, až kým nestretneme *prvé* také, kde sa toto konkrétne jedlo podáva. Ak sa takto dostaneme až na koniec kráľovstva a práve hľadané jedlo už nikde nepodávali, návrh vyhlásime za nezrealizovateľný.

Je zjavné, že ak sa nám podarí nájsť ku každému jedlu mesto, kde ho Hurim zje, môžeme oprávnene prehlásiť návrh za zrealizovateľný. Kráľovná Totňa však mala ešte isté pochybnosti, či sa niekedy nemôže stať, že by sme dostali dobrý rozvrh, ale prehlásili ho za nerealizovateľný. A tak odvážlivec pod hrozbou sťatia hlavy musel dokázať kráľovnej, že pre každý zrealizovateľný plán náš algoritmus naozaj jednu možnú realizáciu nájde.

Myšlienku dôkazu sme si naznačili už vyššie. Sporom. Nech teda existujú nejaké spôsoby realizácie, ale náš algoritmus žiaden nenájde. Zo všetkých realizácií si vyberieme „najmenšiu“, teda tú, ktorá má na začiatku čo najviac krokov spoločných s čiastočným plánom, ktorý našiel náš algoritmus. V nejakom kroku sa potom ale líšia – náš algoritmus chcel vybrať nejaké mesto m_1, zatiaľ čo zvolená realizácia použije iné mesto m_2. Zjavne musí platiť $m_1 < m_2$, lebo m_1 je prvé mesto, ktoré sa v danej situácii dalo použiť. To ale znamená, že v nami zvolenej realizácii môžeme *namiesto* mesta m_2 použiť mesto m_1 – a to je spor s tým, že sme na začiatku vybrali realizáciu, ktorá sa najdlhšie zhodovala s našim algoritmom.

(Alebo iný pohľad na vyššie uvedený dôkaz: ak máme ľubovoľnú korektnú realizáciu, vieme ju prerobiť na tú, ktorú nájde náš algoritmus – a to tak, že postupne ideme po jednotlivých jedlách plánu a zakaždým skontrolujeme, či nemôžeme zjedenie daného jedla posunúť do skoršieho mesta.)

A tak náš prvý odvážlivec našiel funkčný algoritmus, pracujúci v čase $O(k \cdot (n + \ell))$: pre každý z k plánov postupne prechádza cez všetkých n miest a odškrtáva si, ktoré z ℓ jedál daného plánu už Hurim zjedol. A keďže zjavne má zmysel uvažovať len situácie, kde $\ell \leq n$, môžeme tento odhad zjednodušiť na $O(kn)$.

Vám takýto algoritmus mohol pomôcť k piatim bodom, kráľovnej však nestačil, lebo bol príliš pomalý a kým by sa dočkala výsledku, bolo by neskoro.

Našťastie sa našiel niekto ďalší, kto prišiel na šikovné vylepšenie: Pre každé jedlo vytvoríme zoznam miest, v ktorých je toto jedlo špecialita. Potom keď kontrolujeme konkrétny plán, nemusíme ísť zaradom po celom kráľovstve. Keď vieme, aké jedlo má Hurim zjesť ako ďalšie v poradí, môžeme si v zozname jeho výskytov nájsť prvé vhodné mesto (t.j., mesto s číslom väčším ako to, kde práve sme) pomocou binárneho vyhľadávania.

Keďže miest je dokopy n, každý zoznam výskytov má dĺžku najviac n, a teda čas potrebný na nájdenie nasledujúceho výskytu práve hľadaného jedla vieme zhora odhadnúť ako $O(\log n)$.

Celkovo sa takto časová zložitosť riešenia zlepší na $O(n + k\ell \log n)$: najskôr načítame celý vstup, a potom pre každý z k plánov a každé z ℓ jedál v ňom použijeme jedno binárne vyhľadávanie na nájdenie správneho nasledujúceho mesta (ak existuje). Za takéto riešenie sa dalo získať až deväť bodov. Kráľovnej už stačilo, my sa s ním však neuspokojíme a ukážeme si, ako sa zbaviť ešte aj toho logaritmu.

Jeden možný spôsob, ako sa logaritmu zbaviť, poddaní vymysleli rýchlo, ale fungoval len pre malé m: namiesto binárneho vyhľadávania si jednoducho pre každú dvojicu (mesto,jedlo) predpočítame, v ktorom najbližšom meste s väčším číslom toto jedlo podávajú. Pre konkrétne jedlo toto predpočítanie vieme spraviť v čase $O(n)$ prechodom cez všetky mestá (od konca), dokopy má teda predpočítanie časovú aj pamäťovú zložitosť $O(mn)$. Následne už vieme každú položku každého plánu spracovať v konštantnom čase, celková časová zložitosť je teda $O(mn + k\ell)$. Za takéto riešenie sa dalo získať sedem bodov.

Nakoniec predsa len prišiel aj poddaný, ktorý dokázal nájsť optimálne riešenie. Pochopil, že pre dosiahnutie úspechu musíme kontrolovať všetky plány naraz. Aby sme to vedeli robiť efektívne, budeme si v plánoch udržiavať poriadok, a preto budú roztriedené do m šuflíkov, podľa toho, ktorým jedlom začínajú. Počas kontroly

budeme z plánov odtrhávať začiatok, a následne ich preskupovať do nových šuflíkov podľa toho, čo nové sa na ich začiatku ocitlo.

Po úvodnom rozdelení do šuflíkov sa pozrieme na zoznam miest, a budeme dokola opakovať nasledovné: Nech sa v práve spracúvanom meste kráľovstva varí jedlo x. Toto jedlo Hurim zje práve vtedy, keď je na začiatku jeho plánu. Preto zoberieme všetky plány, ktoré začínajú číslom x (máme ich v x-tom šuflíku), z každého odtrhneme začiatočnú položku, ktorú sme práve splnili, a rozhádžeme ich do správnych nových šuflíkov podľa novej začiatočnej položky.

Plány, ktoré sme celé poodtŕhali, sú realizovateľné, ostatné nie. Dôkaz vyplýva z toho, že vlastne len naraz pre všetky plány robíme jednoduchý pažravý algoritmus, ktorého správnosť sme si dokázali na začiatku tohto vzorového riešenia.

Čo sa týka samotnej implementácie, jedna možnosť je, že šuflíky, zoznam miest, a plány budú obyčajné polia. A ku každému poľu budeme mať ešte premennú, ukazujúcu na jeho aktuálny „začiatok“, teda miesto, kde začína ešte nespracovaná časť. V šuflíku si samozrejme pamätáme len *indexy* plánov, ktoré práve obsahuje. Vďaka tomu vieme robiť všetky potrebné operácie v konštantnom čase.

Oveľa pohodlnejšie a rovnako rýchle bude použiť na implementáciu šuflíkov, zoznamu miest a plánov fronty. Zjednoduší sa nám tým odtrhávanie začiatku.

V oboch prípadoch si však treba dávať pozor na jednu vec: Pokiaľ presúvame plán zo šuflíka späť do toho istého šuflíka, nechceme, aby sa spracoval znova. (To je ekvivalentné s tým, že nedovoľujeme dve zastávky v tom istom meste.)

Celková zložitosť algoritmu bude $O(n + k\ell)$ pretože každá položka bude odtrhnutá práve raz a na jej odtrhnutie vynaložím konštantný čas. Položiek je n na zozname miest a $k\ell$ v plánoch. Táto časová zložitosť je zjavne optimálna, keďže toľko času potrebujeme už len na samotné načítanie vstupu. Pamäťová zložitosť je tiež $O(n + k\ell)$, pretože si všetky plány aj zoznam miest potrebujeme naraz pamätať.

Listing programu (C++)

```
// Fruit of Light; Apple Strawberry
#include<cstdio>
#include<algorithm>
#include<vector>
#include<queue>
using namespace std;
typedef queue<int> fronta;

int n,m,k,l,a;
fronta mesta;
vector<fronta> sufliky, plany;

int main(){
    // NAČÍTAME VSTUP, VYTVORÍME FRONTY
    scanf("%d%d%d%d",&n,&m,&k,&l);
    for(int i = 0; i<n; ++i) { scanf("%d",&a); mesta.push(a); }
    plany.resize(k);
    sufliky.resize(m+1);
    for(int i = 0; i<k; ++i){
        for (int j = 0; j<l; ++j) { scanf("%d",&a); plany[i].push(a); }
        sufliky[plany[i].front()].push(i);
    }

    // SKONTROLUJEME PLÁNY
    while(!mesta.empty()){
        int x = mesta.front();
        mesta.pop();
        int pocet = sufliky[x].size();
        while (pocet--){
            int i = sufliky[x].front();
            sufliky[x].pop();
            plany[i].pop();
            if (!plany[i].empty()) sufliky[plany[i].front()].push(i);
        }
    }

    //VYPÍŠEME REALIZOVATEĽNÉ PLÁNY
    for(int i = 0; i<k; ++i) printf("%s\n",(plany[i].empty())?"ano":"nie");
}
```

A-II-2 Vyvážené jablone

V celom vzorovom riešení používame takú orientáciu jablone ako v zadaní. Koreň je teda úplne na spodku a z každého uzlu idú dve vetvy *dohora*. Podstrom určený uzlom u bude tá časť jablone, do ktorej z koreňa musíme ísť cez uzol u. Aj samotný uzol u patrí do tohto podstromu.

Pri riešení súťažnej úlohy je najjednoduchšie postupovať od listov ku koreňu. Samotný list je vždy vyvážený. Uzol, z ktorého idú dohora vetvy do dvoch listov, vyvážime veľmi jednoducho – odtrhneme jablká z ťažšieho listu tak, aby sme počet jabĺk v oboch listoch vyrovnali.

Predstavme si teraz, že ideme vyvažovať uzol u, pričom predpokladáme, že všetky uzly jeho podstromu okrem samotného u sú už vyvážené. Pri vyvažovaní u už vo všeobecnosti nevieme jablká trhať po jednom, tak ako to bolo pri listoch, pretože potrebujeme udržať vyvážené oba jeho podstromy. Potrebujeme nájsť najväčší počet jabĺk, ktorý vieme dosiahnuť aj v jednom, aj v druhom podstrome.

Definujme si *výšku uzla* ako (maximálnu) výšku jablone nad daným uzlom, teda smerom k listom. (Každý list teda má výšku 0.) Dokážeme si nasledovné pozorovanie: ak má vrchol výšku h a jeho podstrom je vyvážený, tak z toho podstromu vieme bez porušenia vyváženosti trhať práve a len po 2^h jablkách. Toto dokážeme matematickou indukciou podľa h.

1° Vrchol s výškou 0 je list, z neho vieme jablká trhať po jednom.

2° Nech je celý podstrom nad uzlom v vyvážený a nech má uzol v výšku $h_v > 1$. Z neho idú dohora vetvy do dvoch vrcholov l a r, ktoré majú výšky h_l a h_r. Bez ujmy na všeobecnosti, nech $h_l \geq h_r$, a teda $h_v = h_l + 1$. Keďže v je vyvážený, podstromy s koreňmi l a r majú rovnako veľa jabĺk. Čiže ak nechceme pokaziť vyváženosť, musíme z nich trhať rovnako veľa. Zároveň ale musíme na každej strane jablká trhať tak, aby sme nepokazili vyváženosť daného podstromu. Z podstromu l vieme trhať po 2^{h_l} jablkách, z podstromu r po 2^{h_r}. Keďže $2^{h_r} \mid 2^{h_l}$, všetky počty, ktoré vieme odtŕhať aj v jednom, aj v druhom podstrome, sú tvaru $k \cdot 2^{h_l}$ pre $k \geq 0$. A teda z celého podstromu s koreňom v vieme trhať práve po $2 \cdot 2^{h_l} = 2^{h_v}$ jablkách.

Keď teda máme vyvážený strom s koreňom v, ktorý momentálne obsahuje j jabĺk, vieme, že všetky prípustné počty jabĺk v tomto strome sú práve tvaru $j - k \cdot 2^{h_v}$ pre $k \geq 0$ (pričom samozrejme nemôžeme ísť do mínusu). Na tomto vieme založiť riešenie s lineárnou časovou zložitosťou: postupne od listov ku koreňu spracúvame celý strom, pričom vždy, keď spájame dva podstromy, nájdeme najväčší počet jabĺk dosiahnuteľný na oboch stranách. Pri tom pomôže uvedomiť si a dokázať, že j je vždy násobkom 2^{h_v}. Pri implementácii takéhoto riešenia si ale treba dať pozor na to, že mocniny dvoch veľmi rýchlo rastú, a tak nám môže pre hlboké stromy pretiecť nejaká premenná.

My si ale ukážeme ešte trochu viac o vyvážených jabloniach a to nám potom umožní pohodlnejšiu implementáciu. Nové, silnejšie pozorovanie vyzerá nasledovne: Pre každú vyváženú jabloň platí: ak je na nej dokopy j jabĺk, tak pre každé i platí, že v každom liste, ktorý je vo vzdialenosti i od koreňa, je presne $j/2^i$ jabĺk.

Prečo je to tak? Predstavte si, že všetky jablká z celej vyváženej jablone presunieme do koreňa. Ich počet označíme j. Následne ich necháme, nech sa všetky naraz začnú presúvať do listov, do ktorých patria. V každom uzle, vrátane koreňa, sa jablká rozdelia na dve rovnako veľké polovice. Teda keď sú práve vo vzdialenosti i od koreňa, v každom vrchole ich je $j/2^i$. V tých vrcholoch, ktoré sú listami, tieto jablká zostanú, tie v uzloch sa opäť rozdelia na polovice a putujú ďalej.

Inými slovami: Nech h je výška celej našej jablone. Z práve dokázaného pozorovania vieme, že v každom liste, ktorý je v tejto výške, bude po vyvážení rovnaký počet jabĺk, označme ho x. Toto jediné x popisuje celú vyváženú jabloň – každý list vo vzdialenosti i od koreňa bude mať $x \cdot 2^{h-i}$ jabĺk, a dokopy na celej jabloni bude presne $x \cdot 2^h$ jabĺk.

Hľadáme teda najväčšiu hodnotu našej neznámej x, ktorú vieme dosiahnuť pre dané súčasné rozloženie jabĺk. Pre každý list dostávame jedno obmedzenie hodnoty x zhora: ak je daný list vo vzdialenosti d_i od koreňa, mal by mať $x \cdot 2^{h-d_i}$ jabĺk. No keďže ich má v súčasnosti j_i a môžeme jablká len odoberať, nie pridávať, musí platiť $x \cdot 2^{h-d_i} \leq j_i$, a teda $x \leq \lfloor j_i / 2^{h-d_i} \rfloor$.

Stačí teda raz prejsť celým stromom, pre každý list zistiť jeho vzdialenosť od koreňa, z toho spočítať najväčšie prípustné x, a z toho rovno vieme výsledný počet jabĺk po vyvážení, a teda aj počet potrebných odtrhnutí.

Aj toto riešenie má časovú zložitosť lineárnu od počtu vrcholov jablone, vieme ho však implementovať výrazne stručnejšie a navyše nám pri našej implementácii nehrozí pretečenie premenných.

Listing programu (C++)

```
#include <algorithm>
#include <iostream>
#include <numeric>
#include <vector>
using namespace std;

vector< pair<int, int> > dohora;     // vstup: pre každý uzol: kam z neho vedú dohora vetvy
vector<long long> pocet_jablk;       // vstup: pre každý list: koľko je v ňom jabĺk?
vector<bool> je_list;                // vstup: pre každý uzol: je to list?
vector<int> vzdialenosti;            // sem si vyplníme pre každý uzol, ako je ďaleko od koreňa

void dfs(int kde, int vzdialenost) { // parametre: v akom vrchole som a ako je ďaleko od koreňa
    vzdialenosti[kde] = vzdialenost;
    if (!je_list[kde]) {
        dfs( dohora[kde].first,  vzdialenost+1 );
        dfs( dohora[kde].second, vzdialenost+1 );
    }
}

int main() {
    int N; cin >> N;

    dohora.resize(N+1); pocet_jablk.resize(N+1); je_list.resize(N+1,false);

    for (int n=1; n<=N; ++n) {
        string typ; cin >> typ;
        if (typ=="U") {
            cin >> dohora[n].first >> dohora[n].second;
        } else {
            je_list[n] = true;
            cin >> pocet_jablk[n];
        }
    }

    vzdialenosti.resize(N+1);
    dfs(1,0);

    int h = *max_element( vzdialenosti.begin(), vzdialenosti.end() );
    long long x = *max_element( pocet_jablk.begin(), pocet_jablk.end() );
    for (int n=1; n<=N; ++n) if (je_list[n]) x = min(x, pocet_jablk[n] >> (h-vzdialenosti[n]) );

    long long zostane = x;
    if (zostane > 0) zostane <<= h; // toto nemôže pretiecť, lebo ak x>0, h je nutne malé
    cout << (accumulate(pocet_jablk.begin(),pocet_jablk.end(),0LL) - zostane) << "\n";
}
```

A-II-3 Zaokrúhľovanie

Táto úloha patrila medzi ťažšie z tohto kola Olympiády. Najskôr si ukážeme zopár dôležitých pozorovaní, následne si našu úlohu prevedieme na úplne iná vyzerajúcu úlohu, a tú nakoniec vyriešime.

Dôležité pozorovania

Prvé, čo si musíme uvedomiť, je, že celé časti našich čísiel sú úplne zbytočné. Ak máme niekde v mriežke necelé číslo x, môžeme ho pokojne nahradiť číslom $x - \lfloor x \rfloor$. Ešte stále dostaneme platnú tabuľku (súčet riadku aj stĺpca sme zmenili o celé číslo) a na riešení to nič nezmení – to zaokrúhlenie, ktoré fungovalo predtým, bude nutne fungovať aj teraz a naopak. Preto nám stačí vedieť riešiť úlohu, v ktorej sú všetky čísla na vstupe z intervalu $(0, 1)$.

Zoberme si teraz nejaký riadok a povedzme si, že jeho súčet je k. Keďže naše čísla ležia v intervale $(0, 1)$ pri zaokrúhľovaní sa rozhodujeme vždy medzi číslom 0 a 1. A ak máme zachovať súčet riadku k, znamená to, že práve k čísiel v tomto riadku musíme zmeniť na 1. Súčet každého riadku/stĺpca nám teda určuje, koľko čísiel v ňom máme zmeniť na 1.

Prevod na inú úlohu

To čo sme zistili je síce pekné, ale uvažovať o tom v takomto tvare je veľmi obtiažne. Bolo by lepšie si to

previesť na inú úlohu, o ktorej by sa nám uvažovalo ľahšie. K tomu nám môže pomôcť, že keď zmeníme nejaké číslo v mriežke na 1, ovplyvní to jeden riadok a jeden stĺpec. Hľadáme teda niečo čo dáva do súvisu dve množiny vecí.

Vhodná štruktúra je bipartitný graf. Budeme mať dve množiny vrcholov R a S, kde každý vrchol v R predstavuje jeden riadok a vrchol v S stĺpec. Hrana medzi dvoma vrcholmi r_i a s_j znamená, že sme číslo v i-tom riadku a j-tom stĺpci zaokrúhlili na 1. Zo vstupu vieme počet stĺpcov aj riadkov. A vieme z neho pre každý riadok aj stĺpec zistiť, koľko čísiel v ňom sa má zaokrúhliť na 1 – teda počet hrán, ktoré musia vychádzať z konkrétneho vrcholu v našom grafe.

Dostávame teda úlohu z teórie grafov. Máme zostrojiť bipartitný graf, pričom každý vrchol má vopred určený počet hrán, ktoré z neho majú vychádzať – stupeň daného vrchola.

(Odbočka: Uvedomte si, že táto nová úloha je o niečo všeobecnejšia. Totiž existujú niektoré jej zadania, ktoré nezodpovedajú žiadnej matici reálnych čísel. To nám ale nevadí – ak túto novú úlohu vyriešime, vyriešime tým aj našu pôvodnú.)

Problémom je, že nemôžeme mať násobné hrany, teda nemôže byť medzi dvoma vrcholmi viac ako jedna hrana. Ak by sme teda priraďovali hrany len tak ako príde, mohlo by sa nám stať, že síce riešenie existuje, ale my ho nenájdeme – na konci nám zostanú nejaké vrcholy, ktoré ešte nemajú dostatočný stupeň, no zároveň už nemôžeme pridať žiadnu hranu, lebo všetky hrany, ktoré by sme potrebovali pridať, už v grafe máme.

Potrebujeme nájsť nejaký spôsob zostrojovania grafu, ktoré niečo takéto vylúči – teda ktorý nám zaručí, že ak graf s danými stupňami vrcholov existuje, tak jeden taký nájdeme.

Pažravý algoritmus

Vyslovíme teraz tvrdenie, ktoré nám pomôže nájsť efektívne riešenie našej novej úlohy.

Tvrdíme: Ak existuje bipartitný graf B s danými stupňami vrcholov, existuje aj B', ktorý má rovnaké stupne vrcholov a navyše platí, že konkrétny vrchol r z množiny R je spojený s $\deg(r)$ vrcholmi z S, ktoré majú najväčší stupeň.[1]

Dôkaz: Majme teda nejaký bipartitný graf B a v ňom vrchol r. Ukážeme, že graf B vždy vieme postupne prerobiť na požadovaný graf B'

Ak je r pripojený ku $\deg(r)$ vrcholom s najvyšším stupňom, je všetko v poriadku a $B' = B$. Ak nie, tak existuje vrchol s_1 v množine S, ktorý je spojený s r, ale nepatrí medzi $\deg(r)$ najväčších. A tiež existuje vrchol s_2 z množiny S, ktorý leží medzi $\deg(r)$ najväčšími, ale nie je spojený s r. Keďže $\deg(s_2) > \deg(s_1)$, tak existuje vrchol r' z množiny R, ktorý je spojený s s_2, ale nie s s_1. Ak teraz nahradíme hrany $r-s_1$ a $r'-s_2$ za hrany $r-s_2$ a $r'-s_1$, máme stále korektný bipartitný graf, opravili sme si však jednu zlú hranu z vrcholu r. Opakovaním tohto postupu prerobíme v konečnom počte krokov pôvodný graf B na graf, v ktorom je r spojený s $\deg(r)$ vrcholmi s najväčším stupňom, q.e.d.

Toto nám dáva postup ako budovať daný graf: Zoberieme ľubovoľný $r \in R$ a nájdeme mu zodpovedajúce vrcholy v S. Teraz môžeme na vrchol r zabudnúť a vrcholom z S, ktoré sme s ním spojili, znížiť stupeň o 1. Tým dostávame ten istý problém, len na menšom grafe. Tento postup teda opakujeme, až kým buď nespracujeme celé R, alebo nenarazíme na spor.

Našu pôvodnú úlohu teda vyriešime nasledovne: Pre každý riadok a stĺpec zistíme počet jednotiek, ktoré v ňom musíme pridať – teda stupeň zodpovedajúceho vrcholu v našom grafe. Následne budeme postupne vypĺňať jednotlivé riadky. (Keďže nezáleží na poradí, v akom to robíme, budeme ich jednoducho vypĺňať v poradí, v akom sú na vstupe.) Nech teraz spracúvame riadok, v ktorom má byť k jednotiek. Na to nám stačí nájsť k stĺpcov, v ktorých chýba najviac jednotiek, a vyplniť ich tam.

Jediná otvorená otázka je, ako budem vyberať k stĺpcov s najväčším stupňom. Použitím haldy alebo nejakého klasického efektívneho algoritmu (napr. MergeSortu alebo knižničnej funkcie `sort`) na triedenie dostanem pre tabuľku rozmerov $r \times s$ časovú zložitosť $O(rs \log s)$. Uvedomme si však, že čísla, ktoré triedim, sú z rozsahu 0 až r. Preto môžem použiť CountSort, alebo iný triediaci algoritmus s lineárnou časovou zložitosťou. Takto dostaneme časovú zložitosť $O(rs)$. Toto je zároveň aj optimum, keďže musíme načítať vstup.

[1] Značenie $\deg(r)$ predstavuje stupeň vrchola v.

Listing programu (C++)

```
#include<cstdio>
#include<algorithm>
#include<vector>
#include <cmath>
using namespace std;

int main(){
    int r,s;
    scanf("%d %d ",&r,&s);
    double A[r][s];
    for(int i=0; i<r; i++)
        for(int j=0; j<s; j++)
        {
            double p;
            scanf("%lf ",&p);
            A[i][j]=p-floor(p);
        }
    int R[r],S[s];
    char V[r][s];
    for(int i=0; i<r; i++)
    {
        double p=0;
        for(int j=0; j<s; j++) p+=A[i][j];
        R[i]=ceil(p);
    }
    for(int i=0; i<s; i++)
    {
        double p=0;
        for(int j=0; j<s; j++) p+=A[j][i];
        S[i]=ceil(p);
    }
    for(int i=0; i<r; i++)
        for(int j=0; j<s; j++) V[i][j]='D';
    vector<vector<int> > C;
    for(int i=0; i<r; i++)
    {
        C.clear();
        C.resize(s+1);
        for(int j=0; j<s; j++) C[S[j]].push_back(j);
        int p=0;
        int kde=s;
        while(p!=R[i])
        {
            for(int k=0; k<C[kde].size(); k++)
            {
                if(p==R[i]) break;
                V[i][C[kde][k]]='H';
                p++; S[C[kde][k]]--;
            }
            kde--;
        }
    }
    for(int i=0; i<r; i++)
    {
        for(int j=0; j<s; j++) printf("%c",V[i][j]);
        printf("\n");
    }
}
```

Alternatívne riešenie

Súťažná úloha sa dala (síce pomalšie, ale ešte stále v polynomiálnom čase) vyriešiť aj bez transformácie na úlohu o zostrojení bipartitného grafu. Toto pomalšie riešenie uvádzame aj z toho dôvodu, že z neho vyplynie, že naša úloha vždy má riešenie – teda že vždy existuje aspoň jeden vhodný spôsob zaokrúhľovania.

V tomto riešení budeme zaokrúhľovať čísla postupne. Vždy si nájdeme nejakú množinu čísel, ktoré ešte nie sú celé, a vhodne ju upravíme. Vhodná úprava bude mať dve vlastnosti: nezmení súčet v žiadnom riadku ani stĺpci, a navyše po nej bude aspoň jedno zo zmenených čísel celé.

Najjednoduchšia úprava bude vyzerať nasledovne: Vyberieme si dva riadky a dva stĺpce také, že všetky štyri čísla, ktoré ležia v tých riadkoch a zároveň v tých stĺpcoch sú necelé. Teraz môžeme dve z nich („ľavé horné“ a „pravé dolné“) o nejakú hodnotu δ zvýšiť a zároveň zvyšné dve o tú istú hodnotu δ znížiť. Tým sa zjavne nezmení súčet žiadneho riadku ani stĺpca. No a keď hodnotu δ zvolíme vhodne, niektoré z nich (aspoň jedno) stúpne na 1 alebo klesne na 0, zatiaľ čo ostatné (najviac tri) ostanú v intervale $(0, 1)$.

Napríklad keby sme niekde v tabuľke videli čísla:

```
.... 0.47 .... 0.21 ....
.... .... .... .... ....
.... 0.17 .... 0.62 ....
```

tak si zvolíme $\delta = 0.17$, čím dostaneme:

```
.... 0.64 .... 0.04 ....
.... .... .... .... ....
.... 0.00 .... 0.79 ....
```

a úspešne sme tak zvýšili počet celých čísel.

Samozrejme, časom sa nám takéto štvorice čísel minú – v každej už bude aspoň jedno celé číslo. Vtedy budeme musieť začať hľadať väčšie skupiny čísel, ktoré zmeníme.

Ako ich hľadať? Ak ešte nie sme hotoví, je v našej tabuľke aspoň jedno necelé číslo. Jedno také si vyberieme. Riadok, v ktorom je, má celočíselný súčet, preto v ňom musí byť aspoň jedno ďalšie necelé číslo. Jedno také si nájdeme a pozrieme sa pre zmenu na jeho stĺpec. Aj v tom musí niekde byť ešte aspoň jedno iné necelé číslo. A takto pokračujeme ďalej, striedavo hľadajúc ďalšie číslo v riadku a v stĺpci.

A keďže pokračovať môžeme do nekonečna, časom sa nám nejaký riadok alebo stĺpec zopakuje. V tom okamihu sme našli cyklus: nejakú postupnosť k riadkov a k stĺpcov, ktorá nám určuje $2k$ necelých čísel – v každom riadku aj stĺpci dve z nich. No a s nimi spravíme presne to isté: na striedačku ich budeme zväčšovať a zmenšovať o tú istú hodnotu δ.

Práve sme teda ukázali, že vždy, keď ešte máme v našej tabuľke nejaké necelé čísla, vieme ju upraviť tak, aby sa aspoň jedno z nich zmenilo na celé. Opakovaním vyššie popísaného postupu teda časom musíme dostať tabuľku, v ktorej už sú všetky čísla celé.

Hľadanie jednej vhodnej skupiny políčok vieme v tabuľke rozmerov $r \times s$ spraviť v čase $O(rs)$ – lebo pred nájdením cyklu prejdeme každý riadok a každý stĺpec najviac raz. Jej úpravu potom spravíme v zanedbateľnom čase $O(\min(r, s))$. No a keďže máme tabuľku s rs políčkami a v každej iterácii zmeníme na celé aspoň jedno z nich, bude nám určite stačiť rs iterácií. Celkovú časovú zložitosť tohto riešenia teda vieme zhora odhadnúť ako $O(r^2 s^2)$.

A-I-4 Log-space výpočty

Podúloha A – Násobenie veľkých čísel

Násobiť veľké čísla sme sa učili už na základnej škole. Čísla α a β si jednoducho napíšeme pod seba, potom postupne násobíme α všetkými ciframi β (na to stačí poznať malú násobilku), pričom pri každej ďalšej cifre sa posunieme o jeden stĺpec doľava. Nakoniec všetky medzivýsledky sčítame. Napríklad pre $\alpha = 137$ a $\beta = 123$ vyzerá násobenie takto:

$$\begin{array}{cccccc} & & & 1 & 3 & 7 \\ & & \times & 1 & 2 & 3 \\ \hline & & & 4 & 1 & 1 \\ & & 2 & 7 & 4 & \\ & 1 & 3 & 7 & & \\ \hline & 1 & 6 & 8 & 5 & 1 \end{array}$$

Tento algoritmus beží v čase $O(n^2)$ a dá sa jednoducho implementovať s pomocným poľom. Pri log-space výpočtoch si toto nemôžeme dovoliť a musíme sa zamyslieť nad implementáciou *bez* pomocného poľa.

Výsledok budeme počítať cifru po cifre, stĺpec po stĺpci. Pre jednoduchosť zatiaľ zabudnime na prenosy do

vyšších rádov. Potom posledná cifra $C[0]$ je zrejme $A[0] * B[0]$.

$$\begin{aligned} C[1] &= A[0] * B[1] + A[1] * B[0] \\ C[2] &= A[0] * B[2] + A[1] * B[1] + A[2] * B[0] \\ C[3] &= A[0] * B[3] + A[1] * B[2] + A[2] * B[1] + A[3] * B[0] \\ &\vdots \end{aligned}$$

Vo všeobecnosti je $C[i]$ súčet nejakých $A[j] * B[k]$, pričom $j + k = i$. V skutočnosti k tomuto môže ešte pribudnúť prenos z nižšieho rádu a ak dostaneme výsledok > 9, $C[i]$ bude iba posledná cifra tohto súčtu. Teda

$$\begin{aligned} \text{súčet} &\leftarrow \text{prenos} + \sum_{j+k=i} A[j] * B[k] \\ C[i] &\leftarrow \text{súčet mod } 10 \\ \text{prenos} &\leftarrow \lfloor \text{súčet}/10 \rfloor \end{aligned}$$

Takto stačí použiť len zopár pomocných premenných. Navyše súčet bude nadobúdať len hodnoty od 0 po $90n$ (môžeme mať najviac n-krát $9 * 9 = 81n$ plus prenos $9n$ z nižšieho rádu), takže program je naozaj log-space.

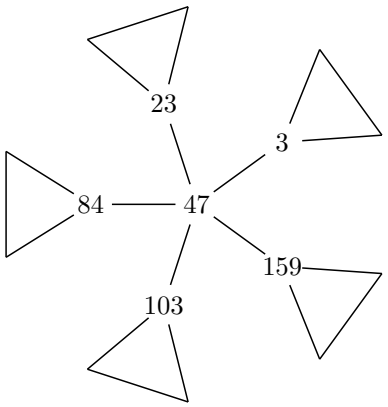
Listing programu (Pascal)

```
var n : integer;                        { vstup: dĺžka čísel }
    A, B : array [0..n-1] of integer;   { vstup: alfa, beta }
    C : array [0..2*n-1] of integer;    { výstup: gama = alfa * beta }
    i, j, p : integer;

begin
    p := 0;
    for i := 0 to 2*n-1 do begin
        for j := max(0,i+1-n) to min(i,n-1) do
            p := p + A[j] * B[i - j];
        C[i] := p mod 10; { posledná cifra je C[i] }
        p := p div 10;    { zvyšok sa prenesie do vyššieho rádu }
    end;
end.
```

Podúloha B – Prehľadávanie lesu

Tradične by sme túto úlohu riešili prehľadávaním do hĺbky alebo do šírky. Začneme z jedného vrcholu, prehľadáme celý jeho komponent a ak pritom natrafíme na druhý vrchol, sú spolu, inak nie. Obe riešenia však používajú veľa pamäte – pre každý vrchol si pamätáme, či sme ho už navštívili alebo nie a navyše máme zásobník alebo frontu vrcholov, ktoré sa chystáme navštíviť. V log-space programoch si však môžeme pamätať informácie iba o zopár (konštantne veľa) vrcholoch.



Využijeme, že zadaný graf je les. Predstavme si, že stojíme vo vrchole 47 na obrázku vyššie. Okolo nás je 5 susedov a keby sme vrchol 47 odstránili, graf by sa rozpadol na 5 podstromov (zakreslených len schématicky ako

trojuholníky). Nemáme dosť pamäte na to, aby sme si pamätali, v ktorých podstromoch sme už boli a v ktorých nie, ale môžeme si susedov pekne zoradiť podľa identifikátorov. Budeme postupovať pomocou jednoduchého pravidla, pričom si budeme pamätať dve veci: v ktorom vrchole sme a z ktorého vrcholu práve prichádzame. Pokračovať budeme vždy cyklicky tým nasledujúcim vrcholom v utriedenom poradí.

Napríklad, ak sme do 47 prišli z vrcholu 84, ďalej budeme pokračovať vo vrchole 103 (pričom si pamätáme, že sme tam prišli zo 47). Keď sa prehľadávanie z vrcholu 103 vráti (t.j. sme v 47 a prišli sme zo 103), nasledujúci vrchol bude 159. Keď sa vrátime zo 159, ideme do 3, atď.

Treba si rozmyslieť ešte tri veci: 1. Že takéto prehľadávanie funguje a naozaj prejdeme celý strom. To sa dá dokázať napríklad indukciou: Ak vôjdeme do listu, tak z neho po tej istej hrane výjdeme aj naspäť (lebo inú hranu nemáme). Ak stojíme vo vrchole w a jeho susedia sú $w_1, w_2, \ldots, w_k$ zoradení od najmenšieho po najväčšieho, pričom do w sme sa dostali z w_j, potom budeme pokračovať vo w_{j+1}. Z indukčného predpokladu sa do w z w_{j+1} vrátime a budeme pokračovať vo w_{j+2}, atď., až kým cyklicky neobehneme všetkých susedov a vrátime sa do w_j.

Celý algoritmus si môžeme predstaviť aj tak, že vrcholy grafu sú miestnosti a hrany sú chodby. My sa pravou rukou držíme jednej steny a ideme len popri nej. Ak je daný graf strom, potom ho takýmto spôsobom celý prejdeme. (Rozmyslite si, že pre všeobecné grafy to nefunguje.)

2. Kedy máme zastať? Jedna možnosť je zapamätať si, v ktorom vrchole sme začínali a ktorá bola prvá hrana, ktorou sme sa vydali (to sú len 2 čísla). Ak sa dostaneme na začiatok a budeme sa chcieť vydať toutou hranou, môžeme skončiť. Druhá, lenivá možnosť je spraviť jednoducho $2m$ krokov – totiž komponent môže mať najviac m hrán a keď každú prejdeme dvakrát (tam a späť), dostaneme sa na začiatok.

3. Dá sa tento algoritmus implementovať ako log-space program? Dá. Pamätáme si iba v ktorom vrchole sme a odkiaľ sme prišli. V danom vrchol vždy prejdeme celý zoznam hrán a nájdeme nasledujúceho suseda (to je najmenší väčší alebo, ak väčší sused neexistuje, tak úplne najmenší sused; toto je implementované vo funkcii `dalsia_hrana`).

Listing programu (Pascal)

```
var n, m, u, v : integer;               { vstup: počet vrcholov a hrán, štart a cieľ }
    A, B : array [1..m] of integer;     { vstup: hrany grafu }
    spolu : integer;                    { výstup: sú u a v v jednom komponente súvislosti? }
    w, z, i, tmp : integer; { som vo vrchole w, prisiel som sem zo z }

function dalsia_hrana (w, z : integer) : integer;
var i, minn, nextn : integer;
begin
    minn := z; { sused s najmenším číslom }
    nextn := m+1; { sused s najmenším číslom väčším ako z }
    for i := 1 to m do begin
        if A[i] = w then begin
            if B[i] < minn then minn := B[i];
            if (B[i] > z) and (B[i] < nextn) then nextn := B[i];
        end;
        if B[i] = w then begin
            if A[i] < minn then minn := A[i];
            if (A[i] > z) and (A[i] < nextn) then nextn := A[i];
        end;
    end;
    if nextn <= m then dalsia_hrana := nextn
    else dalsia_hrana := minn; { neexistuje sused s väčším číslom, vrátime najmenšieho suseda }
end;

begin
    w := u
    z := -1;
    for i := 1 to 2*m + 47 do begin
        if w = v then begin
            spolu := 1; halt;
        end;
        tmp := dalsia_hrana (w, z);
        z := w;
        w := tmp;
    end;
    spolu := 0;
end.
```

Pre zaujímavosť na záver ešte poznamenajme, že naše riešenie dosť podstatne využíva fakt, že graf na vstupe je les a komponenty sú stromy. Log-space program pre všeobecné neorientované grafy existuje, avšak je oveľa oveľa komplikovanejší (nad rámec OI aj vysokoškolského učiva); pred necelými 9 rokmi ho vynašiel Omer

Reingold. Podobná otázka pre orientované grafy, teda či existuje algoritmus s malou pamäťovou zložitosťou, ktorý by rozhodol, či sa v danom grafe dá z vrcholu u dostať do vrcholu v, je dodnes otvorená. Poznáme iba algoritmus s pamäťovou zložitosťou $O((\log n)^2)$ a predpokladá sa, že žiadny log-space program ani neexistuje. Nikto to však zatiaľ nevie dokázať.

DVADSIATY ÔSMY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Ján Hozza, Michal Hozza, Kubo Kováč
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2012