Riešenia kategórie A

A-I-1 Špióni

Vo všeobecnosti je nájdenie *najväčšej kliky* v grafe NP-úplný problém. To okrem iného znamená, že zrejme neexistuje algoritmus, ktorý to zvládne v polynomiálnom čase. Základom riešenia našej úlohy bude teda analýza toho, čím je naša sieť špiónov špeciálna a ako nám to pomôže nájsť lepšie riešenie.

Pripomeňme si podmienku, ktorú podľa zadania naša sieť spĺňa: Pre ľubovoľné prirodzené číslo k platí: ak ľubovoľným spôsobom vyberieme k špiónov, bude medzi nimi najviac $3k$ dvojíc špiónov, ktorí spolu komunikujú.

Z tejto podmienky priamo vyplýva, že dokopy v celej sieti máme nanajvýš $3n$ dvojíc komunikujúcich špiónov (stačí dosadiť $k = n$). Vieme toho však zistiť omnoho viac.

V prvom rade vieme dokázať, že v našej sieti nemôže existovať klika tvorená ôsmimi (a teda ani viac) špiónmi. Prečo? Keď máme osem špiónov, komunikuje spolu nanajvýš $3 \times 8 = 24$ dvojíc. Ale osem špiónov, to je 28 dvojíc, a teda niektorá dvojica špiónov určite spolu nekomunikuje – preto nemôže ísť o kliku.

V tomto okamihu už teda vieme navrhnúť prvý algoritmus s polynomiálnou časovou zložitosťou: Vyskúšame všetky množiny tvorené 1 až 7 špiónmi a o každej overíme, či tvorí kliku. Tento algoritmus má časovú zložitosť $O(n^7)$ a ak ho nejak nezlepšíme, bude použiteľný len pre veľmi malé vstupy.

Lepšie riešenie

Až 7 bodov sa dalo získať za riešenie, ktoré zvládne vyriešiť vstupy, v ktorých platí $1 \leq n \leq 1000$ a zároveň celkový počet klík špiónov (všetkých, nie len maximálnych) neprevýši 64 000.

Na takéto riešenie nám stačí vylepšiť to predchádzajúce tak, aby sme zbytočne neskúšali možnosti, ktoré nevedú k riešeniu. Presnejšie, ak už o nejakej množine špiónov zistíme, že kliku netvorí, vieme, že nemá zmysel skúšať ani žiadnu jej nadmnožinu. Ak teda použijeme *backtracking* (prehľadávanie s návratom), pri ktorom postupne prezeráme všetky množiny špiónov, ktoré ešte tvoria kliku, vieme dostať dostatočne dobré riešenie.

Kvadratické riešenie

Prekvapivo, existujú aj zaručene efektívne riešenia našej úlohy.

Kľúčové pozorovanie sme tu už uviedli, ale teraz ho uvedieme v trochu inej podobe: V ľubovoľnej skupine k špiónov (či už je to klika alebo nie) musí existovať špión, ktorý komunikuje s nanajvýš šiestimi inými. (Ak by to neplatilo, mali by sme v rámci skupiny viac ako $6k/2 = 3k$ dvojíc komunikujúcich špiónov. Alebo v grafovej terminológii: ak máme nanajvýš $3k$ hrán, tak je súčet stupňov vrcholov nanajvýš $6k$, a teda niektorý vrchol má stupeň nanajvýš 6.)

Vyberme si ľubovoľného takéhoto špióna. Sú dve možnosti: buď je súčasťou najväčšej kliky, alebo nie.

Ak je súčasťou najväčšej kliky, tvoria ju on + niektorí spomedzi špiónov, s ktorými komunikuje. Keďže tých je nanajvýš 6, máme nanajvýš $2^6 = 64$ skupín špiónov, ktoré treba skontrolovať. A keďže každú z nich tvorí nanajvýš 7 špiónov, celú túto kontrolu vieme spraviť v konštantnom čase.

A potom nám už ostane len druhý prípad: nájsť najväčšiu kliku, do ktorej náš špión nepatrí. Na to stačí nášho špióna zo siete odstrániť. Dostaneme tým sieť tvorenú $n-1$ špiónmi. Zjavne aj táto sieť spĺňa podmienku zo zadania, a teda pre ňu môžeme celý postup zopakovať – opäť nájdeme špióna, ktorý má nanajvýš 6 známych, a tak ďalej. Takto pokračujeme, až kým sa nám všetci špióni neminú.

Ako je to s časovou zložitosťou tohto algoritmu?

V prvom rade sme vás trochu zavádzali tvrdením, že pre daných (nanajvýš) 7 špiónov vieme v konštantnom čase overiť, či tvoria kliku. Na to potrebujeme pre každú dvojicu špiónov zistiť, či spolu komunikujú. A toto nevieme triviálne spraviť v konštantnom čase. Zatiaľ sa teda uspokojíme s pomalým priamočiarym riešením: vždy, keď potrebujeme vedieť, či spolu špióni x a y komunikujú, prejdeme zoznam všetkých špiónov, s ktorými komunikuje x, a overíme, či je v ňom y. Takto vieme ľubovoľnú potenciálnu kliku overiť v čase $\Theta(n)$.

Priamočiara implementácia celého algoritmu má potom časovú zložitosť $\Theta(n^2)$. Totiž máme n iterácií (jednu pre každého špióna) a pri každej potrebujeme:

– najskôr čas $\Theta(n)$ na to, aby sme si našli vhodného špióna, ktorý pozná nanajvýš 6 iných,
– potom opäť čas $\Theta(n)$ na to, aby sme skontrolovali, ktoré podmnožiny jeho známych tvoria kliky,
– nakoniec čas $\Theta(n)$ na to, aby sme spracovaného špióna z grafu odstránili.

Riešenie vzorové – takmer optimálne

Ak chceme lepšie riešenie, potrebujeme všetky tri vyššie popísané kroky robiť efektívnejšie.

Efektívnejšie hľadať ďalšieho špióna na spracovanie je ľahké. Špiónov, ktorých už môžeme spracovať (t.j. tých, čo majú 6 alebo menej známych) si budeme udržiavať vo fronte. No a vždy, keď nejakého špióna spracujeme a odstránime, skontrolujeme každého jeho známeho, či mu práve neklesol počet známych na 6. Ak áno, pridáme ho do fronty. Takto dosiahneme, že vhodného špióna na spracovanie vždy nájdeme v konštantnom čase.

Zvyšné dva kroky vieme zefektívniť použitím lepších dátových štruktúr. Môžeme si napríklad pre každého špióna jeho známych pamätať ako usporiadanú množinu (vyvažovaný binárny strom, `set` v C++), alebo ako neusporiadanú množinu (hešovacia tabuľka, `unordered_set` v C++). V prvom prípade dostaneme riešenie so zaručenou časovou zložitosťou $O(n \log n)$, v druhom prípade dostaneme *očakávanú* časovú zložitosť $O(n)$.

Toto sú riešenia, aké je vhodné písať na praktickej súťaži – využívajú knižničné dátové štruktúry, vďaka čomu sú stručné a prehľadné, a zároveň sú dostatočne efektívne na zisk plného počtu bodov.

Implementačný detail: prezeranie podmnožín

V nasledujúcom listingu programu si všimnite elegantný spôsob, akým prechádzame všetky podmnožiny známych práve spracúvaného špióna.

Ak máme z-prvkovú množinu, tá má 2^z rôznych podmnožín. Každú podmnožinu P vieme popísať nejakým z-bitovým reťazcom: pre každý prvok pôvodnej množiny napíšeme 0, ak do P nepatrí, alebo 1, ak do P patrí. Všetky podmnožiny takto priamo zodpovedajú číslam od 0 po $2^z - 1$. Keď teda chceme postupne každú podmnožinu prezrieť a spracovať, stačí nám prejsť v cykle (premennou `subset`) cez všetky tieto čísla. Keď už máme v premennej `subset` číslo konkrétnej podmnožiny, jej prvky ľahko nájdeme pomocou bitových operácií.

Listing programu (C++)

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_set>
using namespace std;

int N, M;
vector< unordered_set<int> > G; // G[x] je množina špiónov, ktorí komunikujú so špiónom x
vector<int> najvacsia_klika;

bool je_klika( const vector<int> &spioni ) {
    // overíme, či daná skupina špiónov tvorí kliku -- teda či každá dvojica spolu komunikuje
    for (int x : spioni) for (int y : spioni) if (x<y && G[x].count(y)==0) return false;
    return true;
}

int main() {
    // načítame vstup a prečíslujeme špiónov na 0 až n-1
    cin >> N >> M;
    G.resize(N);
    for (int m=0; m<M; ++m) {
        int x,y; cin >> x >> y; --x; --y;
        G[x].insert(y); G[y].insert(x);
    }

    // do fronty na spracovanie dáme špiónov, ktorí už teraz majú nanajvýš 6 známych
    queue<int> Q;
    for (int n=0; n<N; ++n) if (G[n].size()<=6) Q.push(n);

    // postupne po jednom spracúvame špiónov
    while (!Q.empty()) {
        int kto = Q.front(); Q.pop();
        vector<int> kandidati( G[kto].begin(), G[kto].end() ); // všetci známi špióna "kto"

        // prejdeme všetkých 2^Z podmnožín množiny kandidátov, o každej vyskúšame, či je to klika
        int Z = kandidati.size();
        for (int subset=0; subset<(1<<Z); ++subset) {
            vector<int> mozna_klika(1,kto);
            for (int z=0; z<Z; ++z) if (subset & 1<<z) mozna_klika.push_back( kandidati[z] );
            if (mozna_klika.size() > najvacsia_klika.size() && je_klika( mozna_klika ))
```

```
                najvacsia_klika = mozna_klika;
        }

        // odstránime práve spracovaného špióna
        // ak tým vznikli noví špióni s nanajvýš 6 známymi, pridáme ich do fronty na spracovanie
        for (int x : kandidati) {
            G[x].erase( kto );
            if (G[x].size()==6) Q.push(x);
        }
    }
    // vypíšeme prvky najväčšej nájdenej kliky
    for (int x : najvacsia_klika) cout << (x+1) << endl;
}
```

Optimálne riešenie

Z teoretického hľadiska je zaujímavá aj otázka, či existuje riešenie tejto súťažnej úlohy, ktoré má aj v najhoršom možnom prípade zaručenú lineárnu časovú zložitosť. Odpoveď na túto otázku je kladná.

Pomôžeme si drobným trikom. Rozdelíme riešenie na dve fázy. V prvej fáze zostrojíme poradie, v ktorom budeme špiónov spracúvať. Zároveň si pre každého špióna zapamätáme len tých jeho (nanajvýš 6) známych, ktorých ešte má v okamihu, keď ho spracúvame. Všimnime si, že takto sme zredukovali množstvo zapamätanej informácie na polovicu: ak je na vstupe dané, že špióni x a y spolu komunikujú, budeme si to pamätať len u toho špióna, ktorého skôr spracujeme.

Táto šikovnejšia reprezentácia vstupu nám potom ponúka presne to, čo potrebujeme: vieme pomocou nej v zaručene konštantnom čase overiť, či spolu nejakí dvaja špióni p a q komunikujú. Na to stačí najskôr prejsť zoznam pamätaný pre špióna p (najviac 6 špiónov) a overiť, či je v ňom q, a následne prejsť zoznam pre špióna q (opäť najviac 6 špiónov) a overiť, či je v ňom p. (Alternatívne, ak si pre každého špióna pamätáme poradie, v ktorom má byť spracovaný, stačí prezrieť len jeden správny zoznam.) Nasleduje implementácia tohto riešenia.

Listing programu (C++)

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

int N, M;
vector< vector<int> > G; // G[x] je zoznam špiónov, ktorí komunikujú so špiónom x
vector< vector<int> > H; // H vznikne z G redukciou popísanou vo vzorovom riešení
vector<int> najvacsia_klika;

bool komunikuju( int x, int y ) {
    // na nanajvýš 12 krokov overíme, či špióni x a y spolu komunikujú
    for (int z : H[x]) if (z==y) return true;
    for (int z : H[y]) if (z==x) return true;
    return false;
}

bool je_klika( const vector<int> &spioni ) {
    // overíme, či daná skupina špiónov tvorí kliku -- teda či každá dvojica spolu komunikuje
    for (int x : spioni) for (int y : spioni) if (x<y && !komunikuju(x,y)) return false;
    return true;
}

int main() {
    // načítame vstup a prečíslujeme špiónov na 0 až n-1
    cin >> N >> M;
    G.resize(N); H.resize(N);
    for (int m=0; m<M; ++m) {
        int x,y; cin >> x >> y; --x; --y;
        G[x].push_back(y); G[y].push_back(x);
    }

    // do fronty na spracovanie dáme špiónov, ktorí už teraz majú nanajvýš 6 známych
    vector<bool> spracovany(N,false);
    vector<int> poradie;
    vector<int> stupen(N); for (int n=0; n<N; ++n) stupen[n] = G[n].size();
    queue<int> Q;
    for (int n=0; n<N; ++n) if (stupen[n]<=6) Q.push(n);

    // postupne po jednom spracúvame špiónov a vypĺňame pole H
    while (!Q.empty()) {
        int kto = Q.front(); Q.pop();
        poradie.push_back(kto);
        for (int x : G[kto]) if (!spracovany[x]) {
            H[kto].push_back(x);
```

```
            --stupen[x];
            if (stupen[x]==6) Q.push(x);
        }
        spracovany[kto] = true;
    }

    // ešte raz v tom istom poradí spracúvame špiónov a hľadáme kliky
    for (int kto : poradie) {
        // prejdeme všetkých 2^Z podmnožín množiny kandidátov, o každej vyskúšame, či je to klika
        int Z = H[kto].size();
        for (int subset=0; subset<(1<<Z); ++subset) {
            vector<int> mozna_klika(1,kto);
            for (int z=0; z<Z; ++z) if (subset & 1<<z) mozna_klika.push_back( H[kto][z] );
            if (mozna_klika.size() > najvacsia_klika.size() && je_klika( mozna_klika ))
                najvacsia_klika = mozna_klika;
        }
    }

    // vypíšeme prvky najväčšej nájdenej kliky
    for (int x : najvacsia_klika) cout << (x+1) << endl;
}
```

A-I-2 Rozvod elektriny

Našou úlohou je rozdeliť zadanú sieť na súvislé neprekrývajúce sa oblasti. V každej oblasti sa má nachádzať práve jedna elektráreň, ktorá bude zásobovať elektrinou všetky mestá v tejto oblasti. Výkon elektrárne nesmie byť menší ako súčet spotrieb miest, ktoré zásobuje. Navyše nás obmedzuje kapacita vedení medzi jednotlivými lokalitami.

Zatiaľ to vyzerá na poriadne zložitú úlohu; našťastie však sieť elektrických vedení tvorí strom. Za jeho koreň vyhlásme vrchol číslo 1, čiže krajské mesto. Strom zavesme za koreň – získame tak možnosť používať intuitívne pojmy nahor (smerom ku koreňu) a nadol. Z každého vrcholu v (okrem koreňa) vedie práve jedna hrana smerom nahor; nazvime túto hranu otcovská a jej cieľový vrchol nazvime otcom vrcholu v. Ostatných susedov vrcholu v budeme volať synovia. Vrcholy rôzne od koreňa, ktoré nemajú synov, sa nazývajú listy.

Myšlienka vzorového riešenia

Situácia vo vyšších úrovniach stromu je celkom neprehľadná – nevieme na prvý pohľad určiť, ku ktorej elektrárni treba napojiť konkrétny vrchol. Začnime preto od listov, kde máme menší počet možností:

Vezmime si ľubovoľný list ℓ, v ktorom je nejaké mesto; jeho spotrebu označme d_ℓ. Nech ho pripojíme ku ktorejkoľvek elektrárni, musíme využiť jeho otcovskú hranu. To znamená, že kapacita otcovskej hrany nesmie byť menšia ako d_ℓ, inak riešenie úlohy neexistuje.

Ak je otcom listu ℓ nejaká elektráreň p s výkonom d_p, nemáme inú možnosť ako napojiť ℓ na p (cez p by totiž neprešla elektrina zo žiadnej inej elektrárne). V prípade, že $d_\ell > d_p$, výkon elektrárne nestačí na zásobenie mesta, teda riešenie neexistuje. Inak môžeme odstrániť zo stromu vrchol ℓ a znížiť výkon elektrárne p na $d_p - d_\ell$. Dostaneme tak ekvivalentnú úlohu na $n-1$ vrcholoch, ktorá je riešiteľná práve vtedy, ak je riešiteľná pôvodná úloha.

Ak je otcom listu ℓ nejaké mesto p so spotrebou d_p, potom mesto ℓ musí byť napojené na tú istú elektráreň ako p. Ekvivalentnú menšiu úlohu teda dostaneme tak, že odstránime zo stromu list ℓ a zvýšime spotrebu mesta p na $d_p + d_\ell$.

Vezmime si ľubovoľný list ℓ, v ktorom je nejaká elektráreň a ktorého otcom je tiež elektráreň. Energiu z ℓ cez otca preniesť nemôžeme, preto jednoducho odstránime zo stromu list ℓ a budeme riešiť menšiu ekvivalentnú úlohu.

Ak sa už nedá použiť žiadne z predchádzajúcich zjednodušení úlohy, v každom liste máme nejakú elektráreň, ktorej otcom je nejaké mesto. Vezmime si také mesto p, ktorého všetci synovia $\ell_1, \ell_2, \ldots, \ell_k$ sú listy (v popísanom prípade musí aspoň jedno také mesto existovať).

Množstvo energie, ktorým môže elektráreň ℓ_i zásobovať mesto p, je obmedzené nielen jej výkonom, ale aj kapacitou vedenia, ktoré ju spája s p. Pre každú z elektrární ℓ_1, ℓ_2, …, ℓ_k si preto vypočítame množstvo prenositeľnej energie. Zrejme nič nestratíme, ak budeme ďalej brať do úvahy len najväčšiu z týchto hodnôt; označme si ju d_ℓ.

V prípade, že je p koreňom, nemáme inú možnosť, ako napojiť ho na elektráreň v niektorom z jeho synov. Porovnáme preto d_ℓ so spotrebou d_p – riešenie existuje práve vtedy, keď maximálne množstvo prenositeľnej energie zo synovskej elektrárne postačuje na pokrytie spotreby mesta p.

Aj vtedy, keď má mesto p otca, budeme sa snažiť pripojiť p k elektrárni v niektorom z jeho synov. Táto voľba nám, intuitívne povedané, ponechá viac energie vo vyšších častiach stromu. Teda ak $d_\ell \geq d_p$, potom pripojíme p k synovskej elektrárni s najväčšou hodnotou prenositeľnej energie. Listy $\ell_1, \ell_2, \ldots, \ell_k$ odstránime zo stromu, mesto p nahradíme elektrárňou s výkonom $d_\ell - d_p$ (množstvo energie nevyužitej v p) a ďalej budeme riešiť menšiu ekvivalentnú úlohu.

Zostáva rozobrať posledný prípad: $d_\ell < d_p$. Vtedy musí byť mesto p pripojené k elektrárni cez otcovskú hranu. Odstránime listy $\ell_1, \ell_2, \ldots, \ell_k$ a riešime zjednodušenú úlohu.

Postupným odoberaním vrcholov podľa predchádzajúceho rozboru prípadov časom dostaneme strom pozostávajúci z koreňa a niekoľkých listov s elektrárňami. Takúto situáciu sme tiež rozobrali.

Efektívna implementácia

V prvom rade potrebujeme vedieť rýchlo hľadať listy stromu. Našťastie sú lokality na vstupe očíslované v poradí, v akom boli pripájané k súvislej sieti. To znamená, že vrchol číslo n je listom; keď ho odstránime zo stromu, vrchol číslo $n-1$ bude listom; a tak ďalej.

Vrcholy teda budeme spracúvať od konca. Pre každé mesto v si budeme okrem informácií zo vstupu pamätať ešte hodnotu `ponuka[`v`]`, čo bude najväčšie množstvo energie prenositeľnej z jeho synov, ktorých sme už spracovali. K tomu navyše prislúcha hodnota `od_koho[`v`]` – číslo elektrárne, od ktorej pochádza najlepšia ponuka.

Ak je práve spracúvaným listom elektráreň, skúsime zlepšiť ponuku jej otcovi (samozrejme len v prípade, ak to je mesto).

V prípade, že spracúvame mesto, skúsime využiť najlepšiu ponuku a posunúť ju ďalej otcovi zmenšenú o spotrebovanú energiu. Ak ponuka nestačí na pokrytie spotreby mesta, skúsime sa napojiť cez otcovskú hranu. Ak je otcom elektráreň, znížime jej výkon o spotrebu mesta; ak je otcom mesto, zvýšime jeho spotrebu.

Po spracovaní všetkých vrcholov potrebujeme vypísať čísla elektrární, ktoré sme priradili jednotlivým lokalitám. Na to si postupne vypĺňame pole `riesenie`. Väčšinou vieme určiť správnu hodnotu už pri spracovaní vrcholu. Jediný prípad, keď tomu tak nie je, nastáva pri zlyhaní ponuky a spoľahnutí sa na otcovskú hranu, ktorá vedie do iného mesta. Vtedy totiž vieme len to, že vrcholu priradíme rovnakú elektráreň ako neskôr jeho otcovi. Preto nakoniec ešte raz prejdeme poľom `riesenie` odpredu a podopĺňame chýbajúce hodnoty.

Keďže každý vrchol spracujeme v konštantnom čase, celková časová zložitosť je $O(n)$.

Listing programu (C++)

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

int n;
vector<bool> je_mesto;
vector<int> otec;          // otec vrcholu v zakorenenom strome
vector<int> kapacita;      // kapacita vedenia do otca
vector<long long> watty;   // výkon elektrárne / spotreba mesta
vector<int> ponuka;        // najväčšie množstvo energie prenositeľnej od synov...
vector<int> od_koho;       // ... a príslušná elektráreň
vector<int> riesenie;      // priradenie elektrární mestám

bool vyries() {
    for (int i = n - 1; i >= 1; --i)
        if (je_mesto[i]) {
            if (ponuka[i] >= watty[i]) {
                riesenie[i] = od_koho[i];
                if (je_mesto[otec[i]]) {
                    int moja_ponuka = min(kapacita[i], (int) (ponuka[i] - watty[i]));
                    if (moja_ponuka > ponuka[otec[i]]) {
                        ponuka[otec[i]] = moja_ponuka;
                        od_koho[otec[i]] = od_koho[i];
                    }
                }
            } else {
                if (kapacita[i] < watty[i])
```

```
                return false;
            if (je_mesto[otec[i]]) {
                watty[otec[i]] += watty[i];
            } else {
                if (watty[otec[i]] < watty[i])
                    return false;
                riesenie[i] = otec[i];
                watty[otec[i]] -= watty[i];
            }
        }
    } else {
        riesenie[i] = i;
        if (je_mesto[otec[i]]) {
            int moja_ponuka = min(kapacita[i], (int) watty[i]);
            if (moja_ponuka > ponuka[otec[i]]) {
                ponuka[otec[i]] = moja_ponuka;
                od_koho[otec[i]] = i;
            }
        }
    }
    if (ponuka[0] < watty[0])
        return false;
    riesenie[0] = od_koho[0];
    for (int i = 0; i < n; ++i)
        if (riesenie[i] == -1)
            riesenie[i] = riesenie[otec[i]];
    return true;
}

int main() {
    scanf("%d", &n);
    je_mesto.resize(n); otec.resize(n); kapacita.resize(n); watty.resize(n);
    ponuka.resize(n, -1); od_koho.resize(n); riesenie.resize(n, -1);
    je_mesto[0] = true; otec[0] = -1; kapacita[0] = -1;
    scanf("%lld", &watty[0]);
    for (int i = 1; i < n; ++i) {
        char typ;
        scanf("␣%c%d%d%lld", &typ, &otec[i], &kapacita[i], &watty[i]);
        je_mesto[i] = (typ == 'M');
        --otec[i];
    }
    if (vyries()) {
        for (int i = 0; i < n; ++i)
            printf("%d%c", riesenie[i] + 1, (i == n - 1) ? '\n' : '␣');
    } else {
        printf("nemozne\n");
    }
}
```

A-I-3 Húsenice

Škorec Ignác si musí zvoliť správny smer, ktorým z hniezda vyletí. Na výber má samozrejme nekonečne veľa možností, takže neprichádza do úvahy riešenie „vyskúšame všetky možnosti a vyberieme najlepšiu z nich“. Intuitívne sa nám však zdá, že „zmysluplných“ možností na výber až tak veľa nebude. Zväčša bude zrejme jedno, či Ignác poletí pod azimutom 47.004 alebo 47.005. Ako ale toto pozorovanie sformulovať exaktne?

Skúšanie len rozumných možností

Predstavme si, že si Ignác zvolil nejaký smer. Nech je tento smer taký, že nepoletí ponad koniec žiadnej z húseníc. Ignácovu trasu si teda môžeme predstaviť ako polpriamku pretínajúcu vnútro niektorých úsečiek. Túto polpriamku teraz môžeme do ľubovoľnej strany *otáčať* (teda spojito meniť smer, ktorým vedie), až kým „nenarazíme“ na koniec niektorej z húseníc. Je zjavné, že počas otáčania sa nijak nezmení množina húseníc, ktoré Ignác vyzbiera. Zmena môže nastať v poslednom okamihu – ale ak nastane, ide o zmenu k lepšiemu. Môže sa totiž stať, že pri našom postupnom menení smeru narazíme na koniec *novej* húsenice, ktorú doteraz Ignác uloviť nevedel.

Čo sme práve ukázali? Nech by si Ignác zvolil akýkoľvek smer letu, my mu ho vieme vždy upraviť tak, aby zozbieral aspoň toľko isto húseníc, a zároveň letel ponad koniec jednej z nich. Inými slovami, medzi optimálnymi riešeniami (ktorých môže byť opäť aj nekonečne veľa) určite existuje také, pri ktorom Ignác letí ponad koniec jednej z húseníc.

Takto už ale dostávame prvé funkčné riešenie. Máme n húseníc, každá má dva konce, dokopy teda potrebujeme skontrolovať $2n$ polpriamok. Pre každú z nich spočítame, koľko húseníc by Ignác vyzbieral, a vyberieme

najlepšiu možnosť. Pre konkrétnu polpriamku a konkrétnu húsenicu vieme v konštantnom čase zistiť, či sa pretínajú. (Toto podrobnejšie vysvetlíme nižšie.) Konkrétnu polpriamku teda vieme skontrolovať v čase $\Theta(n)$ a tým pádom je celková časová zložitosť tohto riešenia $\Theta(n^2)$.

Za zapamätanie stojí hlavná myšlienka vyššie popísaného riešenia: nájdeme jednoduché kritérium, ako spomedzi množstva možností vybrať nejakú malú zaujímavú sadu, ktorá určite obsahuje (aspoň jednu) optimálnu možnosť. Podobná technika sa dá pri návrhu efektívnych algoritmov (obzvlášť v oblasti výpočtovej geometrie) použiť veľmi často.

Skúste si napríklad podobnou technikou vyriešiť nasledovnú úlohu: Máme obdĺžnikovú dosku tvorenú bodmi na súradniciach $(0, 0)$ až (x_0, y_0). V doske je $n \leq 50$ bodových dier na súradniciach (x_1, y_1) až (x_n, y_n). Chceme z dosky vyrezať čo najväčší (obsahom) obdĺžnik, ktorý bude mať strany rovnobežné so súradnicovými osami a nebude vo vnútri obsahovať žiadnu z dier.

Priesečník úsečky a polpriamky

Predstavme si, že sme pre Ignáca zvolili nejakú polpriamku a že sme si vybrali jednu konkrétnu húsenicu, o ktorej chceme zistiť, či ju Ignác uloví. Predĺžme Ignácovu trasu aj húsenicu na priamky. Môžu nastať dva prípady: buď sú tieto dve priamky rovnobežné, alebo sú rôznobežné. Ak sú rovnobežné, podmienka v zadaní nám garantuje, že nie sú totožné, a teda nemajú žiaden spoločný bod. Ak sú rôznobežné, majú práve jeden priesečník.

Oboje vieme vypočítať napríklad riešením sústavy vhodných rovníc. Nech je Ignácova priamka určená bodom (x_1, y_1) a nech má húsenica konce (x_2, y_2) a (x_3, y_3). Potom Ignácovu priamku tvoria body tvaru (sx_1, sy_1) pre $s \in \mathbb{R}$ a húsenicinu priamku zase body tvaru $\Big(x_2 + t(x_3 - x_2),\ y_2 + t(y_3 - y_2)\Big)$ pre $t \in \mathbb{R}$. (Dokonca vieme aj presnejšie povedať, že Ignácovu polpriamku tvoria práve body s $0 \leq s$ a húsenicinu úsečku body, pre ktoré platí $0 \leq t \leq 1$.)

Priesečník našich dvoch priamok teda spočítame tak, že hľadáme dvojicu s, t, pre ktorú dosadením s do Ignácovho vzorca dostaneme ten istý bod ako dosadením t do vzorca pre húsenicu. Takto dostávame dve lineárne rovnice – jednu pre x-ovú súradnicu, druhú pre y-ovú:

$$sx_1 = x_2 + t(x_3 - x_2)$$
$$sy_1 = y_2 + t(y_3 - y_2)$$

Ak sú priamky rovnobežné a nie sú totožné, táto sústava nemá riešenie. Ak sú rôznobežné, existuje práve jedno riešenie. Keď toto riešenie nájdeme, skontrolujeme ešte, či daný priesečník priamok leží aj na Ignácovej polpriamke, aj na húsenicinej úsečke – teda či s a t spĺňajú vyššie uvedené nerovnosti.

Existuje aj elegantnejšie riešenie, ktoré sa zaobíde bez presného výpočtu súradníc priesečníka. Stačí namiesto bodu (x_1, y_1) zobrať nejaký bod (cx_1, cy_1), ktorý už je dostatočne ďaleko, a potom pomocou vektorových súčinov overiť, či sa pretínajú úsečky $(0, 0) - (cx_1, cy_1)$ a $(x_2, y_2) - (x_3, y_3)$.

Efektívnejšie riešenie

Vyššie popísané kvadratické riešenie je použiteľné, ak by húseníc bolo pár tisíc. Pre väčšie počty húseníc budeme potrebovať efektívnejšie riešenie. Vylepšíme preto naše kvadratické riešenie pomocou ďalšej štandardnej techniky vo výpočtovej geometrii: *zametania*. Základná myšlienka bude jednoduchá – namiesto toho, aby sme každú zaujímavú polpriamku vyhodnocovali úplne odznova, budeme simulovať postupné otáčanie polpriamky a pri tom si udržiavať množinu úsečiek, ktoré v danej chvíli pretína.

Začať môžeme napríklad tým, že Ignácovu polpriamku umiestnime na kladnú poloos osi x, prejdeme všetky húsenice a zistíme, ktoré teraz pretína.

Následne začneme našu polpriamku otáčať proti smeru hodinových ručičiek. Samozrejme, nie spojito, len budeme postupne prechádzať cez všetky situácie, ktoré by spracúvalo aj predchádzajúce, pomalšie riešenie. Aby sme to vedeli robiť efektívne, usporiadame si všetky konce húseníc do poradia, v ktorom cez ne bude naša polpriamka prechádzať. (Odborne sa tomu hovorí, že konce húseníc *usporiadame polárne*. Ide vlastne o

usporiadanie, pri ktorom je kľúčom orientovaný uhol od kladnej poloosi osi x po polpriamku prechádzajúcu cez daný bod.)

Ako teraz vyzerá spracovanie novej polohy polpriamky? Ak ide o „začiatok“ húsenice (teda ak ide o húsenicu, ktorú doteraz naša polpriamka nepretínala), zvýšime si počítadlo húseníc, inak ho znížime. Teda každú novú polpriamku spracujeme v konštantnom čase. Celkom slušné zlepšenie, nie?

Jediné, na čo si treba dať pozor, sú situácie, kedy naraz nastane viacero *udalostí* – teda ak v tom istom okamihu viaceré húsenice „začnú“ a viaceré iné „skončia“. Takéto situácie ale ošetríme ľahko. Stačí pri spracovaní uprednostniť začiatky nových húseníc a až po nich spracovať konce.

Najpomalšou časťou tohto riešenia je samotné triedenie: $2n$ koncov húseníc vieme usporiadať v čase $\Theta(n \log n)$. Zvyšok algoritmu má potom lineárnu časovú zložitosť.

Implementačné detaily

Geometrické úlohy zvádzajú k použitiu reálnych čísel. Počítač však skutočné reálne čísla používať nevie, vie s nimi narábať len približne. A pri tom vznikajú zaokrúhľovacie chyby, s ktorými potom musíme v riešení počítať. (Nikdy by sme napríklad nemali testovať presnú rovnosť dvoch „počítačových reálnych“ čísel. Namiesto toho treba za rovné považovať napr. hocijaké dve hodnoty, ktoré sa od seba líšia menej ako vhodná tolerancia.)

Ak sa to dá, omnoho lepšie je napísať program tak, aby sme si vystačili len s celými číslami. Potom máme istotu, že všetky výpočty s nimi budú presné. Takto vieme riešiť aj našu súťažnú úlohu. Myšlienku len načrtneme, detaily si čitateľ isto ľahko nájde v listingu programu.

V prvej fáze riešenia potrebujeme zistiť, ktoré úsečky pretínajú kladnú poloos osi x, čo ide ľahko. V druhej fáze riešenia potrebujeme polárne usporiadať konce úsečiek. Dva konkrétne body porovnáme tak, že najskôr porovnáme polrovinu, v ktorej ležia. Ak to nerozhodlo, tak sa pozrieme na znamienko vektorového súčinu vektorov, ktoré zodpovedajú našim bodom. A ak ani toto nerozhodlo (znamienko vyšlo 0), tak „začiatky“ majú byť skôr ako „konce“.

Listing programu (C++)

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
struct point { int x,y,priorita; };
int ZACIATOK=0, KONIEC=1;

bool pretina_xplus(const point &A, const point &B) {
    if ( (A.y>0 && B.y>0) || (A.y<0 && B.y<0) ) return false; // oba sú v tej istej polrovine
    if (A.y==0) return (A.x>=0); if (B.y==0) return (B.x>=0); // jeden je na osi x
    // inak stačí zistiť, či je správny uhol medzi nimi < 180
    if (A.y > 0) return (B.x*A.y - A.x*B.y > 0); else return (B.x*A.y - A.x*B.y < 0);
}

bool skor_polarne(const point &A, const point &B) {
    int hA = (A.y>0 || (A.y==0 && A.x>0)), hB = (B.y>0 || (B.y==0 && B.x>0)); // polroviny
    if (hA != hB) return (hA > hB); // ak sú v rôznych, vieme odpovedať
    int vp = A.x*B.y - B.x*A.y; // inak zistíme vektorový súčin
    if (vp != 0) return (vp > 0); // ak vieme, rozhodneme podľa neho
    return A.priorita < B.priorita; // inak podľa toho, čo je začiatok a čo koniec
}

int main() {
    int N; cin >> N;
    vector<point> body; // pole obsahujúce všetky konce úsečiek, ktoré treba spracovať
    int pretatych_teraz=0;

    for (int n=0; n<N; ++n) {
        // načítame ďalšiu úsečku
        point A, B; cin >> A.x >> A.y >> B.x >> B.y;
        // vymeníme jej konce tak, aby bol uhol A0B menej ako 180 stupňov
        if (A.x*B.y - B.x*A.y < 0) swap(A,B);
        A.priorita=ZACIATOK; B.priorita=KONIEC;
        // zistíme, či pretína poloos x+, a podľa toho vložíme body na spracovanie
        if (pretina_xplus(A,B)) {
            ++pretatych_teraz;
            if (A.y!=0) body.push_back(A); // ak začína na x+, už sme tento bod akože spracovali
            body.push_back(B);
        } else {
            body.push_back(A);
            body.push_back(B);
        }
```

```
    }
    int pretatych_max = pretatych_teraz;
    point optimalny_smer {1,0,false};

    // polárne usporiadame body
    sort( body.begin(), body.end(), skor_polarne );

    // postupne prechádzame a spracúvame všetky body, potom vypíšeme výsledok
    for (point bod : body) {
        if (bod.priorita==0) ++pretatych_teraz; else --pretatych_teraz;
        if (pretatych_teraz > pretatych_max) {
            pretatych_max = pretatych_teraz;
            optimalny_smer = bod;
        }
    }
    cout << optimalny_smer.x << " " << optimalny_smer.y << "\n";
}
```

A-I-4 Log-space výpočty

Riešenia jednotlivých podúloh si ukážeme v trochu inom poradí – na koniec si necháme jedno možné riešenie podúlohy B, ktoré využíva podúlohu C.

Podúloha A – usporiadanie prvkov

Zjavne nemôžeme použiť žiaden z klasických efektívnych algoritmov – tie totiž skoro všetky prvky v poli preusporadúvajú, čo my spraviť nevieme. Na druhej strane, nik nás nenúti mať časovú zložitosť $O(n \log n)$. A toto plne využijeme: naše riešenie bude mať kvadratickú časovú zložitosť.

Najskôr uvažujme trochu jednoduchšie zadanie: nech sú všetky prvky poľa $A[1..n]$, ktoré ideme usporiadať, navzájom rôzne. Pozrime sa na konkrétny prvok $A[i]$. Potrebujeme ho umiestniť niekam do výstupného poľa B, ale kam? Odpoveď je jednoduchá: prejdeme pole A a zistíme, koľko z jeho prvkov je menších ako $A[i]$. Práve tie budú v poli B naľavo od neho, správny index kam umiestniť $A[i]$ je teda o jedno väčší od ich počtu.

Ako toto riešenie upraviť pre prípad, keď pole A môže obsahovať viackrát tú istú hodnotu? U prvkov s rovnakou hodnotou zachovávame ich poradie z poľa A. Keď teda spracúvame prvok $A[i]$ a zaujíma nás, koľko prvkov bude v poli B naľavo od neho, zarátame aj prvky s rovnakou hodnotou na pozíciách menších ako i.

Listing programu (Pascal)

```
var n : integer;                        { vstup: dĺžka postupnosti }
    A : array [1..n] of integer;        { vstup: postupnosť čísel }
    B : array [1..n] of integer;        { výstup: usporiadaná postupnosť }
    i, j, vlavo : integer;

begin
    for i := 1 to n do begin
        vlavo := 0;
        for j := 1 to n do begin
            if (A[j] < A[i]) then inc(vlavo);
            if (A[j] = A[i]) and (j<i) then inc(vlavo);
        end;
        B[ vlavo+1 ] := A[i];
    end;
end.
```

Podúloha B – overenie permutácie

Aj táto podúloha má ľahké riešenie v kvadratickom čase. Opäť, jednoduchšie by sa riešila, ak by všetky prvky v poli A boli navzájom rôzne. Vtedy by nám stačilo pre každý prvok poľa A prejsť pole B a overiť, či sa v ňom tento prvok tiež nachádza.

Ako si teraz poradíme s poľami, v ktorých sa prvky môžu opakovať? Vyššie uvedený algoritmus môžeme zovšeobecniť nasledovne: Postupne pre každý prvok poľa A zistíme, koľkokrát sa vyskytuje v poli A, koľkokrát v poli B, a tieto dve hodnoty porovnáme. Ak niekedy nájdeme rozdiel, vieme, že sa naše dve polia líšia. A naopak, ak má každý prvok poľa A rovnaký počet výskytov aj v poli B, musí už pole B byť nutne preusporiadaním poľa A. (Uvedomte si tiež, že B má rovnakú dĺžku ako A, a teda nemôže už obsahovať nič navyše.)

Listing programu (Pascal)

```
var n : integer;                        { vstup: dĺžka postupnosti }
    A : array [1..n] of integer;        { vstup: prvá postupnosť čísel }
    B : array [1..n] of integer;        { vstup: druhá postupnosť čísel }
    i, j, pocetA, pocetB : integer;

begin
    for i := 1 to n do begin
        pocetA := 0;
        for j := 1 to n do if (A[i] = A[j]) then inc(pocetA);
        pocetB := 0;
        for j := 1 to n do if (A[i] = B[j]) then inc(pocetB);
        if (pocetA <> pocetB) then begin
            writeln('nie');
            halt;
        end;
    end;
    writeln('ano');
end.
```

Podúloha C – ako sa zbaviť pomocného poľa

Podľa zadania máme dva log-space programy $\mathcal{F}$ a $\mathcal{G}$. Program $\mathcal{F}$ dostane vstup v poli $A[1..n]$ a vyrobí z neho výstupné pole $B[1..n]$. Program $\mathcal{G}$ dostane na vstupe pole B, ktoré vyrobil program $\mathcal{F}$, a svoj výstup zapíše do výstupného poľa $C[1..n]$.

Ako ich skombinovať do nového programu $\mathcal{H}$, ktorý vyrobí priamo z poľa A pole C? Odpoveď opäť nie je príliš zložitá: Program $\mathcal{H}$ bude robiť presne to isté ako program $\mathcal{G}$ – až na situácie, v ktorých sa program $\mathcal{G}$ snaží čítať z poľa B. Toto pole samozrejme náš program $\mathcal{H}$ nemá k dispozícii. Poznáme ale program $\mathcal{F}$, ktorý ho celé vie vypočítať!

A teda kedykoľvek, keď program $\mathcal{H}$ potrebuje prečítať nejaké políčko $B[i]$, spustíme ako podprogram znova a znova od začiatku program $\mathcal{F}$. Počas behu $\mathcal{F}$ ignorujeme všetky jeho zápisy do poľa B (ktoré nemáme) okrem jediného – zápisu na to políčko $B[i]$, ktoré nás práve zaujíma. Túto hodnotu si uložíme do pomocnej premennej. Keď podprogram $\mathcal{F}$ dobehne, program $\mathcal{H}$ pokračuje v simulácii programu $\mathcal{G}$, pričom ako hodnotu $B[i]$ použije hodnotu, ktorú sme si zistili a uložili do pomocnej premennej.

Potrebujeme sa ešte zamyslieť nad pamäťovou zložitosťou. Ako je to s ňou? Simulujeme program $\mathcal{G}$, ktorý bol log-space. A počas jeho behu občas spustíme ako podprogram program $\mathcal{F}$, ktorý je tiež log-space, a na uloženie jedného políčka jeho výstupu použijeme jednu pomocnú premennú. V súčte sú teda celkové pamäťové nároky dostatočne malé.

Všimnite si tiež, že by takáto konštrukcia nefungovala, ak by si mohli log-space programy prepisovať vstupné pole (potom by sme nevedeli viackrát spustiť $\mathcal{F}$) alebo ak by mohli čítať výstupné pole (potom by sme počas simulácie $\mathcal{F}$ nemohli zahadzovať tie prvky poľa B, ktoré nás práve nezaujímajú).

Ešte raz podúloha B – overenie permutácie

Podúlohu B sme už síce vyriešili, bude však poučné vyriešiť ju ešte raz. V podúlohe A sme napísali program, ktorý dané pole usporiada. Ak by sme mali pomocné polia, vedeli by sme podúlohu B vyriešiť ľahko: usporiadame pole A, usporiadame pole B a výsledky porovnáme.

Keďže pomocné polia nemáme, nevieme to spraviť takto priamočiaro. Vieme však použiť techniku z riešenia podúlohy C. Riešenie podúlohy A upravíme na funkciu, ktorá namiesto vyplnenia celého výstupného poľa len vráti jeho jeden konkrétny prvok. Takáto funkcia môže vyzerať napr. nasledovne:

Listing programu (Pascal)

```
var n : integer;                        { vstup: dĺžka postupnosti }
    A : array [1..n] of integer;        { vstup: postupnosť čísel }
    B : array [1..n] of integer;        { výstup: usporiadaná postupnosť }
    i, j, vlavo : integer;

function q_ty_najmensi_prvok_A ( q : integer ) : integer;
begin
    for i := 1 to n do begin
        vlavo := 0;
```

```
        for j := 1 to n do begin
            if (A[j] < A[i]) then inc(vlavo);
            if (A[j] = A[i]) and (j<i) then inc(vlavo);
        end;
        { tu bolo povodne toto: "B[ vlavo+1 ] := A[i];" }
        { namiesto toho tu vacsinu zapisov odignorujeme, len jeden pouzijeme }
        if (vlavo+1 = q) then q_ty_najmensi_prvok_A := A[i];
    end;
end.
```

Analogickú funkciu si môžeme napísať pre usporiadanie poľa B. No a na overenie, či pole B vzniklo preusporiadaním poľa A, nám teraz stačí pre každé i od 1 po n overiť, či sa i-ty najmenší prvok poľa A rovná i-temu najmenšiemu prvku poľa B.

Za domácu úlohu si rozmyslite, že takto naozaj dostaneme log-space program, a odhadnite jeho pamäťovú aj časovú zložitosť.

DVADSIATY ÔSMY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Forišek, Peter Fulla
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2012