A-III-4 Hyperjazdec

Nikoho zrejme neprekvapí, že aj tretia z úloh o zovšeobecnení šachového jazdca sa bude riešiť dynamickým programovaním. Treba len správne zvládnuť detaily.
Ľahko sa presvedčíme, že hyperjazdec je akoby opakom šachovej dámy: môže ísť všade, kam dáma ísť nevie. Ak teda chceme šikovne zistiť celkový počet spôsobov, ktorými sa vie hyperjazdec dostať na nejaké políčko, stačí nám sčítať možnosti pre celú šachovnicu a od toho odčítať možnosti pre dámu.
Ešte raz a poriadnejšie. Budeme počítať hodnoty $P[i, j, k]$: počet spôsobov, ktorými sa vieme dostať na políčko (j, k) s tým, že sme zatiaľ navštívili v správnom poradí prvých i písmen slova w.
Niektoré tieto hodnoty sú zjavné. Označme $S[j, k]$ písmeno na políčku (j, k) na šachovnici. Ak $S[j, k] \neq w[i]$, tak zjavne $P[i, j, k] = 0$. Slovne: po preskákaní prvých i písmen slova w vieme byť na políčku len vtedy, ak obsahuje i-te písmeno slova w.
Tiež je zjavné, že ak $S[j, k] = w[1]$, tak $P[1, j, k] = 1$: jediný spôsob, ako navštíviť prvé písmeno slova w, je začať na ňom.

Vždy, keď vypočítame všetky hodnoty $P[i, \star, \star]$ pre konkrétne i, zoberieme tabuľku týchto hodnôt a vhodne si ich spracujeme, aby sme vedeli efektívnejšie počítať hodnoty $P[i + 1, \star, \star]$. Presnejšie, spočítame si nasledovné údaje: celkový počet možností pre každý riadok, pre každý stĺpec, pre každú uhlopriečku (v jednom aj druhom smere) a na záver ešte celkový počet možností za celú šachovnicu.
Konkrétnu hodnotu $P[i + 1, j, k]$ kde $S[j, k] = w[i + 1]$ potom vypočítame tak, že zoberieme celkový počet možností ako spraviť i krokov a skončiť kdekoľvek na šachovnici, a od toho odpočítame tie možnosti, kde skončíme na políčku, odkiaľ hyperjazdec nevie potiahnuť na (j, k). Odpočítame teda všetky možnosti v riadku j, v stĺpci k, na uhlopriečke so súčtom súradníc $j + k$ a na uhlopriečke s rozdielom súradníc $j - k$. Takto už sme takmer dostali správny výsledok, až na samotné políčko (j, k). Toto políčko nechceme zarátať, keďže hyperjazdec nesmie zostať na mieste. Aktuálne sme ho raz pričítali k odpovedi (v rámci celkového súčtu) a potom sme ho od odpovede štyrikrát odčítali. Momentálne je teda počet možností pre toto políčko od správnej odpovede trikrát odčítaný, a teda ho ešte k nej treba trikrát pričítať.
Pre každé i toto riešenie najskôr strávi $O(rs)$ času predpočítaním pomocných súčtov a následne v konštantnom čase vypočíta každú z hodnôt v P. Dokopy má teda toto riešenie časovú zložitosť $O(nrs)$.

Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;

const int MOD = 1000000007;

int main() {
    int R, C;
    cin >> R >> C;
    string W;
    cin >> W;
    vector<string> board(R);
    for (int r=0; r<R; ++r) cin >> board[r];

    vector< vector<long long> > ways(R, vector<long long>(C,0));
    for (int r=0; r<R; ++r) for (int c=0; c<C; ++c) if (board[r][c] == W[0]) ways[r][c] = 1;

    for (int n=1; n<int(W.size()); ++n) {
        long long old_total = 0;
        vector<long long> columns(C,0), rows(R,0), plus(R+C-1,0), minus(R+C-1,0);
        for (int r=0; r<R; ++r) for (int c=0; c<C; ++c) {
            old_total      += ways[r][c];
            columns[c]     += ways[r][c];
            rows[r]        += ways[r][c];
            plus[r+c]      += ways[r][c];
            minus[r+C-1-c] += ways[r][c];
        }

        vector< vector<long long> > updated_ways(R, vector<long long>(C,0));
        for (int r=0; r<R; ++r) for (int c=0; c<C; ++c) if (board[r][c] == W[n]) {
            updated_ways[r][c]  = old_total;
            updated_ways[r][c] -= rows[r];
            updated_ways[r][c] -= columns[c];
            updated_ways[r][c] -= plus[r+c];
            updated_ways[r][c] -= minus[r+C-1-c];
            updated_ways[r][c] += 3 * ways[r][c];
            updated_ways[r][c] %= MOD;
```

```
            updated_ways[r][c] += MOD;
            updated_ways[r][c] %= MOD;
        }
        ways = updated_ways;
    }

    long long answer = 0;
    for (int r=0; r<R; ++r) for (int c=0; c<C; ++c) answer += ways[r][c];
    answer %= MOD;
    cout << answer << endl;
}
```

A-III-5 Kváder

Celé riešenie tejto úlohy je o tom, ako veľmi zvládneme zoptimalizovať algoritmus zo zadania. Ten zjavne naozaj iteruje cez všetky políčka daného trojrozmerného poľa, a to vo veľmi špecifickom poradí: políčka na výstupe sú usporiadané primárne podľa súčtu ich súradníc a sekundárne potom podľa prvej, druhej a následne tretej súradnice.

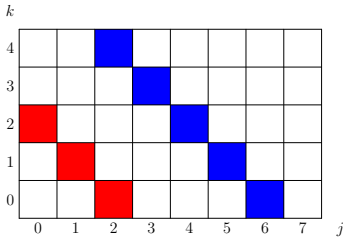
Zadaný algoritmus má časovú zložitosť $O((a+b+c)abc)$. Pár bodov vieme získať už za to, že upravíme hranice cyklov tak, aby sme pre každé s naozaj prechádzali len tie políčka, ktorých súradnice majú súčet s. Takto zlepšíme časovú zložitosť na $O(abc)$.

Ako takúto úpravu spraviť? Ukážeme si to pre premennú i, zvyšné dve úpravy sú podobné. Ako hornú hranicu pre i zoberieme $\min(s, a-1)$, keďže nemá zmysel skúšať $i > s$. Premenné j a k vedia mať súčet od 0 po $b+c-2$. Ak je s väčšie ako $b+c-2$, nemá už zmysel skúšať $i = 0$. Vo všeobecnosti najmenšie i, ktoré má zmysel skúšať, je teda $\max(0, s-(b+c-2))$.

Rezy obdĺžnika

Pozrime sa teraz na dvojrozmernú verziu našej úlohy. Predstavme si, že sme už zvolili konkrétne hodnoty s a i. Zaujíma nás teraz, koľko existuje vyhovujúcich dvojíc (j, k). Ide o dvojice, kde j a k sú z povoleného rozsahu a platí $s = i + j + k$, čiže $j + k = s - i$.

Všetky možné dvojice (j, k) si vieme znázorniť ako obdĺžnik rozmerov $b \times c$. Konkrétny súčet potom určuje konkrétnu uhlopriečku tohto obdĺžnika. Na obrázku nižšie červená uhlopriečka zodpovedá súčtu $j + k = 2$, zatiaľ čo modrá predstavuje súčet $j + k = 6$.



Ku každej konkrétnej uhlopriečke ľahko v konštantnom čase spočítame jej dĺžku. V konštantnom čase vieme dokonca o ľubovoľnom bloku vedľa seba ležiacich uhlopriečok povedať, koľko majú dokopy políčok. Vo všeobecnosti ide len o súčty aritmetických a konštantných postupností. Napr. na obrázku majú uhlopriečky, na ktorých $2 \leq j + k \leq 6$, dokopy 3+4+5+5+5 políčok.

Vyzbrojení týmto pozorovaním vieme dostať ďalšie body za efektívne riešenie dvojrozmerného prípadu. Pre každú otázku najskôr efektívne určíme, na ktorej uhlopriečke hľadané políčko leží, a potom už ľahko v konštantnom čase určíme jeho súradnice.

Zistiť správnu uhlopriečku sa dá v konštantnom čase, ak sa nám chce odvodiť si na to vzorec, ale pre mnohých môže byť pohodlnejšie určiť ju binárnym vyhľadávaním. Pre každé s vieme vyššie spomínaným spôsobom určiť, koľko políčok dokopy navštívime pre súčty od 1 po s. Stačí teda nájsť najväčšie s, pre ktoré je tento súčet menší ako príslušná otázka n_q. Potom vieme, že tá nasledujúca uhlopriečka je tá správna, a aj to, koľké z jej políčok chceme.

Rezy kvádra

V princípe rovnaké riešenie vieme zovšeobecniť aj do troch rozmerov. Len namiesto rezu obdĺžnika priamkou teraz dostaneme rez kvádra rovinou: pre konkrétne s sa na všetky trojice súradníc (i, j, k) s $i+j+k = s$ môžeme dívať ako na jednu takúto rovinu.
Aj obsah takéhoto rezu vieme vyjadriť vzorcom, resp. jednoduchým výpočtom v konštantnom čase, napríklad nasledovne:

- Trojuholníkové čísla sú súčty od 1 po n. Platí $T(n) = 1 + 2 + \cdots + n = n(n+1)/2$.
- Všetky trojice s $i + j + k = s$ a $i, j, k \geq 0$ tvoria trojuholník, dokopy ich je $T(s+1)$.
- Pre malé hodnoty s sú všetky tieto políčka v našom kvádri.
- Od istej hranice už dostaneme nejaké trojice mimo kvádra. Napr. akonáhle $s \geq a$, budeme mať nejaké trojice (i, j, k), ktoré sú mimo kvôli tomu, že už majú priveľké i.

  Všetky trojice s priveľkým i opäť tvoria trojuholník – ide o všetky body $(i' + a, j, k)$ kde $i', j, k \geq 0$ a $i' + j + k = s - a$.

  Takýchto trojíc je teda $T(s - a + 1)$. Podobne pre j a k.
- No a od ďalšej hranice sa potom ešte stane to, že niektoré zlé trojice sme odpočítali dvakrát. Napr. akonáhle $s \geq a + b$, budeme mať nejaké zlé trojice, v ktorých sú i aj j priveľké. Tie sme raz zarátali v $T(s+1)$ a potom dvakrát odrátali v $T(s - a + 1)$ a $T(s - b + 1)$. Teraz teda za ne ešte potrebujeme raz prirátať ich počet: $T(s - a - b + 1)$.

Listing programu (C++)

```
long long obsah_rezu(long long a, long long b, long long c, long long s) {
    // kolko ma kvader policok (i,j,k) takych ze i+j+k = s?
    // zaratame vsetky trojice s i,j,k >= 0
    long long odpoved = (s+1)*(s+2)/2;
    // odratame vsetky kde je nejaka suradnica privelka
    if (s >= a) odpoved -= (s-a+1)*(s-a+2)/2;
    if (s >= b) odpoved -= (s-b+1)*(s-b+2)/2;
    if (s >= c) odpoved -= (s-c+1)*(s-c+2)/2;
    // dvakrat sme odratali tie, kde su dve privelke
    if (s >= a+b) odpoved += (s-a-b+1)*(s-a-b+2)/2;
    if (s >= a+c) odpoved += (s-a-c+1)*(s-a-c+2)/2;
    if (s >= b+c) odpoved += (s-b-c+1)*(s-b-c+2)/2;
    return odpoved;
}
```

Keď sa teraz pozrieme na vzorec, ktorý sme dostali, vidíme, že výsledný vzťah je vždy kvadratickou funkciou premennej s. A existuje len malý konečný počet miest, kde sa koeficienty tejto kvadratickej funkcie zmenia – je to vtedy, keď rez prechádza niektorým z vrcholov kvádra. Aj v trojrozmernom prípade teda vieme v konštantnom čase vypočítať celkový počet políčok v ľubovoľnom súvislom bloku rezov. Potrebujeme na to len vedieť sčítať kvadratickú postupnosť.
No a akonáhle toto zvládneme, už vieme ľubovoľnú otázku zodpovedať v logaritmickom čase: prvým binárnym vyhľadávaním nájdeme rez, v ktorom hľadané políčko leží, druhým binárnym vyhľadávaním určíme hodnotu i a potom už v konštantnom čase vieme dopočítať správne j a dostať jednoznačne určené k.

Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;

long long obsah_rezu(long long a, long long b, long long c, long long s); // vid vyssie

vector<long long> hranice_2d, hranice_3d;

long long obsah_vela_rezov(long long a, long long b, long long c, long long s) {
    // scita obsah_rezu pre 0..s-1
    long long odpoved = 0;
    for (unsigned i=0; i+1<hranice_3d.size(); ++i) {
        long long lo = hranice_3d[i], hi = min(s, hranice_3d[i+1]);
        if (hi - lo <= 3) {
            for (long long j=lo; j<hi; ++j) odpoved += obsah_rezu(a,b,c,j);
        } else {
            long long n = hi - lo;
```

```
            long long a0 = obsah_rezu(a,b,c,lo);
            long long a1 = obsah_rezu(a,b,c,lo+1);
            long long a2 = obsah_rezu(a,b,c,lo+2);
            odpoved += a0*n + (a1-a0)*n*(n-1)/2 + (a2-2*a1+a0)*((n*(n-1)*(n-2))/6);
        }
        if (hi == s) break;
    }
    return odpoved;
}

long long obsah_uhlopriecky(long long b, long long c, long long s) {
    // kolko ma obdlznik policok (j,k) takych ze j+k = s?
    long long odpoved = s + 1;
    if (s >= b) odpoved -= (s - b + 1);
    if (s >= c) odpoved -= (s - c + 1);
    return odpoved;
}

long long obsah_vela_uhlopriecok(long long b, long long c, long long s) {
    // scita obsah_uhlopriecky pre 0..s-1
    long long odpoved = 0;
    for (unsigned i=0; i+1<hranice_2d.size(); ++i) {
        long long lo = hranice_2d[i], hi = min(s, hranice_2d[i+1]);
        if (hi - lo <= 3) {
            for (long long j=lo; j<hi; ++j) odpoved += obsah_uhlopriecky(b,c,j);
        } else {
            long long n = hi - lo;
            long long a0 = obsah_uhlopriecky(b,c,lo);
            long long a1 = obsah_uhlopriecky(b,c,hi-1);
            odpoved += (a0 + a1) * n / 2;
        }
        if (hi == s) break;
    }
    return odpoved;
}

long long obsah_useku_uhlopriecok(long long b, long long c, long long lo, long long hi) {
    return obsah_vela_uhlopriecok(b,c,hi) - obsah_vela_uhlopriecok(b,c,lo);
}

void vyries_rez(long long a, long long b, long long c, long long s, long long n) {
    long long ilo = max(0LL, s-b-c+2), ihi = min(a-1, s);
    long long lo = ilo, hi = ihi+1;
    while (hi - lo > 1) {
        long long med = (lo + hi) / 2;
        // chceme i z intervalu [ilo, med)
        // tieto i zodpovedaju suctom [s-med+1, s-ilo+1)
        long long cur = obsah_useku_uhlopriecok(b, c, s-med+1, s-ilo+1);
        if (cur < n) lo = med; else hi = med;
    }
    n -= obsah_useku_uhlopriecok(b, c, s-lo+1, s-ilo+1);
    long long i = lo;
    long long jmin = max(0LL, s-i-c+1);
    long long j = jmin + n - 1;
    long long k = s - i - j;
    cout << i << "␣" << j << "␣" << k << "\n";
}

int main() {
    long long a, b, c, q; cin >> a >> b >> c >> q;

    set<long long> pomocne_hranice_3d = {0,a-1,b-1,c-1,a+b-2,a+c-2,b+c-2,a+b+c-3,a+b+c-2};
    hranice_3d = vector<long long>(pomocne_hranice_3d.begin(), pomocne_hranice_3d.end());
    set<long long> pomocne_hranice_2d = {0,b-1,c-1,b+c-2,b+c-1};
    hranice_2d = vector<long long>(pomocne_hranice_2d.begin(), pomocne_hranice_2d.end());

    while (q--) {
        long long n; cin >> n;
        // najdi, v ktorom reze sme
        long long lo = 0, hi = a+b+c-2;
        while (hi - lo > 1) {
            long long med = (hi + lo) / 2;
            if (obsah_vela_rezov(a,b,c,med) < n) lo = med; else hi = med;
        }
        n -= obsah_vela_rezov(a,b,c,lo);
        // najdi policko v ramci toho rezu
        vyries_rez(a,b,c,lo,n);
    }
}
```

Alternatívne riešenie

Namiesto vyššie popísaných binárnych vyhľadávaní vieme efektívne riešenie implementovať aj tak, že si všetky otázky usporiadame. Potom raz spravíme efektívnu simuláciu, počas ktorej preskakujeme všetky bloky rezov, ktoré neobsahujú žiadnu otázku. Pre takéto bloky si len vyššie popísanými vzorcami zistíme, koľko políčok

dokopy obsahujú – nemusíme už riešiť, v akom poradí ich navštívia. V rámci rezov, ktoré obsahujú niektoré hľadané políčka, potom robíme to isté s blokmi uhlopriečok.

Súčet kvadratickej postupnosti

Majme kvadratickú postupnosť x, kde $x_i = ai^2 + bi + c$.
Pozrime sa na rozdiely medzi jej členmi. Nech $y_i = x_{i+1} - x_i$. Potom môžeme počítať:

$$y_i = \big(a(i+1)^2 + b(i+1) + c\big) - \big(ai^2 + bi + c\big) = 2ai + a + b$$

Preto rozdiely kvadratickej postupnosti vždy tvoria lineárnu postupnosť. No a podobne môžeme zdôvodniť aj ešte zjavnejšie pozorovanie: diferencie pre lineárnu postupnosť sú konštantné
Príklad: v prvom riadku je kvadratická postupnosť $3i^2 + i + 7$, v druhom sú jej diferencie a v treťom diferencie jej diferencii.

```
7    11    21    37    59    87   121   161   207   259
   4    10    16    22    28    34    40    46    52
     6     6     6     6     6     6     6     6
```

A toto isté pozorovanie môžeme spraviť aj opačným smerom: prefixové súčty kvadratickej postupnosti tvoria vždy kubickú postupnosť. Presnejšie, ak označíme s_i súčet prvých i prvkov postupnosti x, tak určite existujú práve jedny konkrétne konštanty a', b', c', d' také, že pre všetky i platí $s_i = a'i^3 + b'i^2 + c'i + d'$.
Tieto nové konštanty vieme pre ľubovoľnú kvadratickú postupnosť nájsť vyriešením vhodnej sústavy lineárnych rovníc. Chceme, aby sme dostali správne hodnoty s_0 až s_3. Každá z týchto požiadaviek nám dá jednu takúto rovnicu. Túto sústavu rovníc môžeme nechať riešiť náš program, ale keďže koeficienty sa nemenia, môžeme ju vyriešiť už „na papieri“. Máme nasledovné rovnice:

$$\begin{aligned} 0 &= s_0 &&= 0a' + 0b' + 0c' + d' \\ x_0 &= s_1 &&= 1a' + 1b' + 1c' + d' \\ x_0 + x_1 &= s_2 &&= 8a' + 4b' + 2c' + d' \\ x_0 + x_1 + x_2 &= s_3 &&= 27a' + 9b' + 3c' + d' \end{aligned}$$

Z prvej rovnice rovno vidíme, že $d' = 0$. Od tretej rovnice odčítame dvojnásobok druhej a od štvrtej odčítame trojnásobok druhej, aby sme sa v nich zbavili c'. Dostaneme nové rovnice $6a' + 2b' = x_1 - x_0$ a $24a' + 6b' = x_2 + x_1 - 2x_0$. Následne odčítame trojnásobok tretej rovnice od štvrtej, tým dostaneme zo štvrtej rovnice samostatnú rovnicu pre a'. Vypočítané a' dosadíme do tretej, z nej vypočítané b' do druhej a dopočítaním c' máme kompletný vzorec.
Tento vzorec vieme pekne zapísať pomocou x_0, $y_0 = x_1 - x_0$ a $z_0 = y_1 - y_0$ nasledovne:

$$s_n \;=\; x_0 + \cdots + x_{n-1} \;=\; nx_0 + \frac{n(n-1)}{2} y_0 + \frac{n(n-1)(n-2)}{6} z_0$$

resp. po dosadení za y_0 a z_0 dostávame:

$$s_n \;=\; nx_0 + \frac{n(n-1)}{2}(x_1 - x_0) + \frac{n(n-1)(n-2)}{6}(x_2 - 2x_1 + x_0)$$

Toto je teda všeobecný vzorec pre súčet prvých n členov kvadratickej postupnosti s danou dĺžkou a danými prvými troma členmi.

A-III-6 Firma

Začneme nejakými základnými pozorovaniami, na základe ktorých spravíme prvé funkčné riešenie. Tomu potom postupne vylepšíme časovú zložitosť.

Malo by byť zjavné, že časovú zložitosť ľubovoľného riešenia vieme zdola ohraničiť počtom otázok, ktoré toto riešenie položí. U všetkých nižšie uvedených riešení bude platiť, že ich vieme implementovať tak, aby sme si toto nepokazili – teda asymptotická časová zložitosť každého riešenia bude rovnaká ako asymptotický počet otázok, ktoré položí.

Základné pozorovania

Firma hĺbky k má $n = 1 + 2 + 2^2 + \cdots + 2^k = 2^{k+1} - 1$ zamestnancov. Firma je úplný binárny strom so šéfkou v koreni a stážistami v jednotlivých listoch. Keďže je to strom, má presne $n - 1$ hrán = vzťahov kde jeden človek je priamym nadriadeným iného.
Hrany vo firme sú síce orientované (záleží na tom, kto je koho nadriadeným), my ich ale budeme objavovať ako neorientované. Budeme teda vedieť, že nejaká dvojica v hierarchii firmy susedí, vo všeobecnosti však nebudeme vedieť, kto z nich je koho šéfom.
Ako by sa súťažná úloha riešila, ak by sme poznali všetky neorientované hrany?
Šéfku by sme našli ľahko: stážisti majú stupeň 1, šéfka má stupeň 2 a všetky ostatné vrcholy majú stupeň 3.
Akonáhle máme nájdenú šéfku, stačí z nej celú firmu prehľadať (je jedno, či do šírky alebo hĺbky).

Nájdenie hrán v kubickom čase

Hrubou silou vieme všetky hrany nájsť nasledovným spôsobom: Vrcholy x a y sú spojené hranou práve vtedy, ak neexistuje nik, cez koho by chodili správy medzi x a y. Pre každú dvojicu vrcholov teda môžeme overiť, či je medzi nimi hrana, tak, že postupne vyskúšame každý iný vrchol ako z a vždy sa opýtame, či z leží medzi x a y. Takto zostrojíme množinu všetkých hrán v čase $O(n^3)$.

Nájdenie suseda v lineárnom čase

Ak pri kontrole, či z leží medzi x a y, narazíme na odpoveď „áno“, neznamená to len to, že x a y nie sú spojené hranou. Znamená to aj to, že z leží bližšie ku x ako y.
Majme teraz konkrétny vrchol v. Ukážeme si, ako v lineárnom čase nájsť nejakého jeho suseda.
Zoberme si ako kandidáta ľubovoľný iný vrchol. Teraz postupne prejdeme cez všetky ostatné vrcholy. Pre každý takýto vrchol x položíme otázku, či x leží medzi v a aktuálnym kandidátom. Ak neleží, nič sa nedeje. Ak leží, novým kandidátom na suseda vrcholu v je odteraz x.
Keď tento proces skončí (na každý vrchol sme sa raz pozreli), aktuálny kandidát musí byť susedom vrcholu v.
Dôkaz: Akonáhle sme si zvolili prvého kandidáta, je jednoznačne určená cesta z neho do vrcholu v. Nech s je predposledný vrchol na tejto ceste – sused vrcholu v. Potom vyššie popísaný proces musí skončiť tým, že kandidátom je vrchol s. Totiž vždy, keď sa zmení kandidát, posunieme sa po tejto ceste niekam bližšie ku vrcholu v, no a v okamihu, keď sa pozrieme na vrchol s, ten sa stane kandidátom. A tým už nutne zostane až do konca, keďže medzi s a v už žiadny iný vrchol neleží.

Nájdenie všetkých susedov v lineárnom čase

Každý z našich vrcholov má nanajvýš troch susedov. Ak teda chceme nájsť všetkých susedov vrcholu v, mohlo by stačiť vyššie popísaný postup nanajvýš trikrát zopakovať.
Kde však môže nastať problém? Ak v neskoršom kole zle zvolíme kandidáta, ktorým začať, môžeme skončiť na tom istom susedovi, ktorého sme našli už skôr. Ako tomu predísť?
Ak by sme vrchol v zo stromu odstránili, rozpadne sa nám zvyšok na toľko komponentov, koľko má v susedov. Pri hľadaní suseda platí, že nájdeme vždy toho, v koho komponente zvolíme prvého kandidáta.
Predstavme si, že sme práve našli nejakého suseda s vrcholu v. Teraz sa pre každý iný vrchol x opýtame, či s leží medzi x a v. Kladnú odpoveď dostaneme práve pre všetky vrcholy, ktoré ležia v rovnakom komponente ako s. Takto sme teda v lineárnom čase našli celý komponent obsahujúci s. Zvyšných susedov v budeme hľadať už len medzi zvyšnými vrcholmi.

Keďže každý v má nanajvýš troch susedov, stačí vyššie popísaný postup nanajvýš trikrát zopakovať. Zakaždým si zvolíme kandidáta medzi ešte nespracovanými vrcholmi, na menej ako n otázok nájdeme suseda vrcholu v a na ďalších menej ako n otázok nájdeme celý komponent, v ktorom tento sused leží.

Dokopy teda položíme menej ako $6n$ otázok. Výstupom je zoznam susedov vrcholu v a pre každého z nich aj zoznam vrcholov tvoriacich jeho komponent.
(Počet otázok vieme zlepšiť na menej ako $5n$: ak má v troch susedov, netreba už hľadať komponent pre tretieho z nich – musia ho zjavne tvoriť všetky zvyšné vrcholy.)

Dostávame teda kvadratické riešenie: pre každý vrchol zvlášť v lineárnom čase nájdeme všetky hrany z neho a následne použijeme rovnaký postup ako vo vyššie popísanom riešení, ktoré bežalo v kubickom čase.

Efektívne nájdenie koreňa

Ukážeme si teraz, ako vieme efektívne nájsť šéfku celej firmy.
Zoberme si ľubovoľný vrchol v našom strome a spustime preň vyššie popísanú funkciu. Ak sme sa dozvedeli, že má práve dvoch susedov, mali sme šťastie a sme hotoví. V opačnom prípade si vyberieme toho suseda, ktorý má najväčší komponent, presunieme sa doň a celý postup zopakujeme.
Malo by byť zjavné, že ak aktuálny vrchol v nie je koreňom stromu, tak komponent, v ktorom je koreň stromu, obsahuje viac než polovicu celkového počtu vrcholov. Preto bez ohľadu na to, kde začneme, povedie vyššie popísaný postup k tomu, že v každom kroku sa v strome pohneme o krok bližšie ku koreňu.
Najhoršie, čo sa nám mohlo stať, je začať v liste – teda niektorom zo stážistov vo firme. Vtedy budeme potrebovať až k krokov na to, aby sme sa dostali ku šéfke.
Keďže $k \approx \log_2 n$, potrebujeme $O(\log n)$ krokov. A keďže každý z nich vieme spraviť v lineárnom čase, týmto postupom vieme nájsť šéfku firmy v čase $O(n \log n)$.

Všeobecnejšie hľadanie susedov

Aby sme vedeli efektívne implementovať zvyšok riešenia, trochu si ešte upravíme funkciu na hľadanie susedov vrcholu. Do tejto funkcie budeme ako parameter odovzdávať nielen vrchol, ktorého susedov hľadáme, ale tiež aj množinu vrcholov, v ktorej ich hľadáme. Táto množina bude zaručene tvorená niekoľkými komponentmi: pre každého suseda v nej budú do nej patriť aj všetky vrcholy ležiace za ním.
Jediná zmena bude v tom, že ako kandidátov budeme postupne prechádzať len vrcholy z danej množiny. Časová zložitosť potom bude pri dobrej implementácii lineárna len od veľkosti prezeranej množiny (a nie celého stromu).

Efektívne zostrojenie hierarchie od šéfky

Vyššie sme si popísali, ako v čase $O(n \log n)$ nájdeme koreň celého stromu. Teraz budeme postupne rekurzívne spracúvať jeho podstromy a rekonštruovať hrany stromu.
Naša rekurzívna funkcia dostane dva parametre: koreň podstromu, ktorý chceme spracovať, a zoznam všetkých vrcholov v tomto podstrome.
Implementácia tejto funkcie bude jednoduchá. Použijeme našu vylepšenú funkciu na nájdenie všetkých susedov aktuálneho koreňa, ktorí ležia v aktuálnom podstrome. Pre každého z nich sa rekurzívne zavoláme na neho a za ním ležiaci komponent súvislosti – čiže jeho menší podstrom.
Rovnako ako napríklad pri triedení MergeSort vieme zdôvodniť, že celé spracovanie stromu bude dokopy vyžadovať $O(n \log n)$ krokov výpočtu. (Spracovanie stromu s n vrcholmi si vyžaduje $O(n)$ krokov výpočtu a navyše dve rekurzívne volania na stromy so zhruba $n/2$ vrcholmi.)
Dostávame teda riešenie s celkovou časovou zložitosťou $O(n \log n)$.

Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;

bool otazka(int a, int b, int c) {
    cout << 0 << "␣" << a << "␣" << b << "␣" << c << endl << flush;
    int ans;
    cin >> ans;
    return ans > 0;
}

vector<bool> priradeny;
```

```
void najdi_susedov(int vrchol, vector<int> okolie, vector<int> &susedia, vector< vector<int> > &komponenty) {
    // najdeme vsetkych susedov daneho vrcholu v danej podmnozine grafu
    susedia.clear();
    komponenty.clear();
    int P = okolie.size();
    int pocet_priradenych = 0;
    for (int x : okolie) {
        priradeny[x] = false;
        if (x == vrchol) { priradeny[x] = true; ++pocet_priradenych; }
    }
    while (pocet_priradenych < P) {
        int kandidat = -1;
        for (int n=0; n<P; ++n) {
            int curr = okolie[n];
            if (priradeny[curr]) continue;
            if (kandidat == -1) {
                kandidat = curr;
                continue;
            }
            if (otazka(kandidat,curr,vrchol)) kandidat = curr;
        }
        susedia.push_back(kandidat);
        priradeny[kandidat] = true;
        ++pocet_priradenych;
        // tu sa este da optimalizovat: ak uz mame tretieho suseda,
        // vsetci nepriradeni su jeho komponent
        vector<int> komponent;
        for (int n=0; n<P; ++n) {
            int curr = okolie[n];
            if (priradeny[curr]) continue;
            if (otazka(vrchol,kandidat,curr)) {
                komponent.push_back(curr);
                priradeny[curr] = true;
                ++pocet_priradenych;
            }
        }
        komponenty.push_back(komponent);
    }
}

int main() {
    int K, L;
    cin >> K >> L;
    int N = (1 << (K+1)) - 1;
    priradeny.resize(N+1);
    vector<int> vsetci(N);
    iota( vsetci.begin(), vsetci.end(), 1 );
    int where = 1;
    vector<int> susedia;
    vector<vector<int> > komponenty;
    while (true) {
        najdi_susedov(where, vsetci, susedia, komponenty);
        if (susedia.size() == 2) break;
        int i = 0;
        for (int j=0; j<int(susedia.size()); ++j) if (komponenty[j].size() > komponenty[i].size()) i = j;
        where = susedia[i];
    }
    vector< vector<int> > odpovede(N+1);
    odpovede[where] = susedia;

    queue<int> korene;
    queue<vector<int> > podstromy;
    for (int i=0; i<int(susedia.size()); ++i) {
        korene.push(susedia[i]);
        podstromy.push(komponenty[i]);
    }
    while (!korene.empty()) {
        int koren = korene.front(); korene.pop();
        vector<int> podstrom = podstromy.front(); podstromy.pop();
        najdi_susedov(koren, podstrom, susedia, komponenty);
        odpovede[koren] = susedia;
        for (int i=0; i<int(susedia.size()); ++i) {
            korene.push(susedia[i]);
            podstromy.push(komponenty[i]);
        }
    }
    cout << 1 << endl << flush;
    for (int n=1; n<=N; ++n) {
        cout << odpovede[n].size();
        for (int x : odpovede[n]) cout << "␣" << x;
        cout << endl << flush;
    }
}
```

TRIDSIATY SIEDMY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2022