P-III-4 Hyperjezdec

Nikoho zřejmě nepřekvapí, že i třetí z úloh o zobecnění šachového jezdce se bude řešit dynamickým programováním. Je jen potřeba správně vyřešit detaily.

Je jednoduché si rozmyslet, že hyperjezdec je „opak“ šachové dámy: může jít právě tam, kam dáma nemůže. (Dáma může právě na políčka, které mají buď jednu souřadnici stejnou, anebo mají stejný součet resp. rozdíl souřadnic jako políčko, na kterém právě stojí.) Pokud tedy chceme šikovně zjistit celkový počet způsobů, jimiž se hyperjezdec může dostat na nějaké políčko, stačí nám sečíst možnosti pro celou šachovnici a od toho odečíst možnosti pro dámu.

Přesněji: Budeme počítat hodnoty $P[i,j,k]$: počet způsobů, kterými se lze dostat na políčko (j,k) s tím, že jsme doteď navštívili ve správném pořadí prvních i písmen slova w.

Některé tyto hodnoty jsou zjevné. Označme jako $S[j,k]$ písmeno na políčku (j,k). Pokud $S[j,k] \neq w[i]$, tak jistě $P[i,j,k] = 0$. Neboli po přeskákání prvních i písmen slova w můžeme skončit jedině na těch políčcích, která obsahují i-té písmeno slova w.

Také je zjevné, že $P[1,j,k] = 1$ právě tehdy, když $S[j,k] = w[1]$, a $P[1,j,k] = 0$ jinak. Jediný způsob, jak navštívit první písmeno slova w, je totiž na něm začít.

Vždy, když spočítáme všechny hodnoty $P[i,\star,\star]$ pro nějaké konkrétní i, vezmeme tabulku těchto hodnot a vhodně si ji zpracujeme, abychom efektivněji mohli počítat hodnoty $P[i+1,\star,\star]$. Spočítáme si následující údaje: Celkový počet možností pro každý řádek, pro každý sloupec, pro každou úhlopříčku (v obou směrech) a také celkový počet možností za celou šachovnici.

Konkrétní hodnotu $P[i+1,j,k]$ pro j,k takové, že $S[j,k] = w[i+1]$ potom spočítáme tak, že vezmeme celkový počet možností, jak udělat i kroků a skončit kdekoliv na šachovnici, a od toho odečteme ty možnosti, kde skončíme na políčku, odkud hyperjezdec neumí skočit na (j,k). Odečteme tedy možnosti v řádku j, ve sloupci k, na úhlopříčce se součtem souřadnic $j+k$ a na úhlopříčce s rozdílem souřadnic $j-k$. Tím jsme již téměř dostali správný výsledek, až na samotné políčko (j,k). To započíst nechceme, jelikož hyperjezdec nesmí zůstat na místě. Momentálně jsme ho jednou k odpovědi přičetli (v rámci celkového součtu) a čtyřikrát odečetli. Je proto potřeba k výsledku ještě třikrát přičíst $P[i,j,k]$.

Pro každé i toto řešení nejdříve stráví $\mathcal{O}(rs)$ času předvýpočtem pomocných součtů a následně v konstantním čase vypočítá každou z rs hodnot $P[i+1,\star,\star]$. Dohromady má tedy toto řešení časovou složitost $\mathcal{O}(nrs)$. Paměťové složitosti $\mathcal{O}(rs)$ lze dosáhnout tak, že si uvědomíme, že v každém kroku nám stačí pamatovat si pouze již spočítanou tabulku $P[i,\star,\star]$, aktuálně počítanou tabulku $P[i+1,\star,\star]$ a dalších $\mathcal{O}(rs)$ pomocných součtů.

```
#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;
}
```

P-III-5 Kvádr

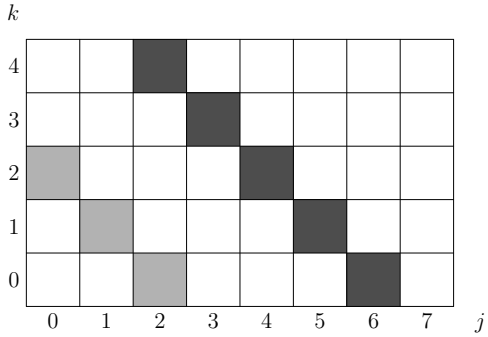
Tato úloha je o tom, jak moc zvládneme zoptimalizovat algoritmus ze zadání. Ten zřejmě opravdu projde všechna políčka daného trojrozměrného pole, a to ve velmi specifickém pořadí. Políčka na výstupu jsou uspořádaná primárně podle součtu souřadnic a potom podle první, druhé a nakonec třetí souřadnice.

Zadaný algoritmus má časovou složitost $\mathcal{O}((a+b+c)abc)$. Pár bodů lze získat už tak, že upravíme hranice cyklů, abychom pro každé s procházeli jen ta políčka, jejichž souřadnice mají součet s. Takto zlepšíme časovou složitost na $\mathcal{O}(abc)$ (a pro každou otázku zoptimalizovaný algoritmus odsimulujeme pro daný počet kroků bez toho, abychom něco skutečně vypisovali).

Jak takovou úpravu udělat? Ukážeme si to pro proměnnou i, zbylé dvě úpravy jsou podobné. Jako horní hranici pro i si vezmeme $\min(s, a-1)$, protože nemá smysl zkoušet $i > s$. Proměnné j a k mohou mít součet od 0 do $b+c-2$. Pokud je s větší než $b+c-2$, nemá smysl zkoušet $i = 0$. Obecně je nejmenší i, které už má smysl zkoušet, rovno $\max(0, s-(b+c-2))$.

Řezy obdélníka

Představme si, že už jsme zvolili konkrétní hodnoty s a i a že nás zajímá, kolik existuje vyhovujících dvojic (j,k), kde j a k jsou z povoleného rozsahu a platí $s = i+j+k$ (čili $j+k = s-i$). Všechny možné dvojice (j,k) můžeme znázornit jako obdélník o rozměrech $b \times c$. Konkrétní součet potom určuje konkrétní úhlopříčku tohoto obdélníka. Na obrázku níže světle šedá úhlopříčka odpovídá součtu $j+k = 2$, zatímco tmavě šedá představuje součet $j+k = 6$.



Pro každou konkrétní úhlopříčku jednoduše v konstantním čase spočítáme její délku. V konstantnm čase umíme dokonce i o libovolném bloku vedle sebe ležících úhlopříček říci, kolik mají dohromady políček, jde jen o součty aritmetických a konstantních posloupností. Např. na obrázku mají úhlopříčky, na nichž $2 \le j+k \le 6$ dohromady $3+4+5+5+5$ políček.

Vyzbrojeni tímto pozorováním můžeme získat další body za efektivní řešení dvojrozměrného případu ($a = 1$). Pro každou otázku nejprve efektivně zjistíme, na které úhlopříčce hledané políčko leží, a potom už jednoduše v konstantním čase určíme jeho souřadnice.

Zjistit správnou úhlopříčku lze dokonce v konstantním čase, pokud si na to odvodíme vzorec, ale pro mnohé může být pohodlnější určit ji binárním vyhledáváním. Pro každé s výše uvedeným způsobem můžeme určit, kolik celkem políček navštívíme pro součty od 1 do s. Stačí tedy najít největší s, pro nějž je tento součet menší než příslušná otázka n_q. Potom víme, že ta následující úhlopříčka je ta správná, a víme i to, kolikáté z jejích políček chceme.

Řezy kvádru

Toto řešení lze zobecnit i do tří rozměrů, jen místo řezu obdélníku přímkou nyní dostaneme řez kvádru rovinou. Pro konkrétní s se na všechny trojice souřadnic (i, j, k) s $i + j + k = s$ můžeme dívat jako na jednu takovou rovinu.

I obsah tohoto řezu lze vyjádřit vzorcem, resp. jednoduchým výpočtem v konstantním čase, například následovně:

- Trojúhelníková čísla jsou součty od 1 do n. Platí $T(n) = 1 + 2 + \cdots + n = n(n + 1)/2$.
- Všechny trojice s $i + j + k = s$ a $i, j, k \geq 0$ tvoří trojúhelník, dohromady jich je $T(s + 1)$.
- Pro malé hodnoty s jsou všechna tato políčka v našem kvádru.
- Od jisté hranice už dostaneme nějaké trojice mimo kvádr. Například jakmile $s \geq a$, budou některé trojice (i, j, k) mimo kvádr proto, že mají příliš velké i.

  Všechny trojice s příliš velkým i opět tvoří trojúhelník – konkrétně jde o body $(i' + a, j, k)$, kde $i', j, k \geq 0$ a $i' + j + k = s - a$. Takovýchto trojic je tedy $T(s - a + 1)$, podobně pro j a k.
- Od další hranice se pak ještě stane to, že některé špatné trojice odečteme dvakrát. Například jakmile $s \geq a + b$, budeme mít špatné trojice, v nichž jsou příliš vysoké hodnoty i a j. Ty jsme jednou započetli v $T(s + 1)$ a potom dvakrát odečetli v $T(s - a + 1)$ a $T(s - b + 1)$. Nyní za ně tedy ještě potřebujeme jednou přičíst jejich počet, což je $T(s - a - b + 1)$.

Funkce na výpočet obsahu řezu tedy bude vypadat takto:

```
long long obsah_rezu(long long a, long long b, long long c, long long s) {
    // Kolik má kvádr políček (i,j,k) takových, že i+j+k = s?

    // Započítáme všechny trojice s i,j,k >= 0.
    long long odpoved = (s+1)*(s+2)/2;
    // Odečteme všechny, kde je nějaká souřadnice příliš veliká.
    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;
    // Dvakrát jsme odečetli ty, kde jsou dvě souřadnice příliš veliké.
    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;
}
```

Když se nyní podíváme na vzorec, který jsme dostali, vidíme, že výsledný vztah je vždy kvadratickou funkcí proměnné s. A existuje jen malý konečný počet míst, kde se koeficienty této kvadratické funkce změní (je to tehdy, když řez prochází některým z vrcholů kvádru). I v trojrozměrném případě tedy umíme v konstantním čase vypočítat celkový počet políček v libovolném souvislém bloku řezů, potřebujeme na to jen umět sečíst kvadratickou posloupnost (což si ukážeme níže).

Jakmile toto dokážeme, můžeme už libovolnou otázku zodpovědět v logaritmickém čase: Prvním binárním vyhledáváním najdeme řez, v němž hledané políčko leží, druhým binárním vyhledáváním určíme hodnotu i a potom už v konstantním čase umíme dopočítat správné j a dostat jednoznačně určené k.

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

typedef long long llong;

llong obsah_rezu(llong a, llong b, llong c, llong s); // Viz výše

vector<llong> hranice_2d, hranice_3d;

llong obsah_nekolika_rezu(llong a, llong b, llong c, llong s) {
    // Sčítá obsah_rezu pro 0..s-1
    llong odpoved = 0;
    for (unsigned i = 0; i + 1 < hranice_3d.size(); ++i) {
        llong lo = hranice_3d[i], hi = min(s, hranice_3d[i+1]);
        if (hi - lo <= 3) {
            for (llong j = lo; j < hi; ++j) odpoved += obsah_rezu(a,b,c,j);
        } else {
            llong n = hi - lo;
            llong a0 = obsah_rezu(a,b,c,lo);
            llong a1 = obsah_rezu(a,b,c,lo+1);
            llong 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;
}

llong obsah_uhlopricky(llong b, llong c, llong s) {
    // Kolik má obdélník políček (j,k) takových, že j+k = s?
    llong odpoved = s + 1;
    if (s >= b) odpoved -= (s - b + 1);
    if (s >= c) odpoved -= (s - c + 1);
    return odpoved;
}

llong obsah_nekolika_uhlopricek(llong b, llong c, llong s) {
    // Sčítá obsah_uhlopricky pro 0..s-1
    llong odpoved = 0;
    for (unsigned i = 0; i + 1 < hranice_2d.size(); ++i) {
        llong lo = hranice_2d[i], hi = min(s, hranice_2d[i+1]);
        if (hi - lo <= 3) {
            for (llong j = lo; j < hi; ++j) odpoved += obsah_uhlopricky(b,c,j);
        } else {
            llong n = hi - lo;
```

```
            llong a0 = obsah_uhlopricky(b,c,lo);
            llong a1 = obsah_uhlopricky(b,c,hi-1);
            odpoved += (a0 + a1) * n / 2;
        }
        if (hi == s) break;
    }
    return odpoved;
}

llong obsah_useku_uhlopricek(llong b, llong c, llong lo, llong hi) {
    return obsah_nekolika_uhlopricek(b,c,hi) - obsah_nekolika_uhlopricek(b,c,lo);
}

void vyres_rez(llong a, llong b, llong c, llong s, llong n) {
    llong ilo = max(0LL, s-b-c+2), ihi = min(a-1, s);
    llong lo = ilo, hi = ihi+1;
    while (hi - lo > 1) {
        llong med = (lo + hi) / 2;
        // Chceme i z intervalu [ilo, med)
        // Tato i odpovídají součtům [s-med+1, s-ilo+1)
        llong cur = obsah_useku_uhlopricek(b, c, s-med+1, s-ilo+1);
        if (cur < n) lo = med; else hi = med;
    }
    n -= obsah_useku_uhlopricek(b, c, s-lo+1, s-ilo+1);
    llong i = lo;
    llong jmin = max(0LL, s-i-c+1);
    llong j = jmin + n - 1;
    llong k = s - i - j;
    cout << i << " " << j << " " << k << "\n";
}

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

    set<llong> 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<llong>(pomocne_hranice_3d.begin(), pomocne_hranice_3d.end());

    set<llong> pomocne_hranice_2d =
        { 0, b-1, c-1, b+c-2, b+c-1 };
    hranice_2d = vector<llong>(pomocne_hranice_2d.begin(), pomocne_hranice_2d.end());

    while (q--) {
        llong n; cin >> n;
        // Najdi, ve kterém jsme řezu
        llong lo = 0, hi = a+b+c-2;
        while (hi - lo > 1) {
            llong med = (hi + lo) / 2;
            if (obsah_nekolika_rezu(a,b,c,med) < n) lo = med; else hi = med;
        }
        n -= obsah_nekolika_rezu(a,b,c,lo);
        // Najdi políčko v rámci toho řezu
        vyres_rez(a,b,c,lo,n);
    }
}
```

Alternativní řešení

Namísto výše popsaných binárních vyhledávání můžeme řešení efektivně implementovat i tak, že si všechny otázky uspořádáme. Potom jednou provedeme efektivní simulaci algoritmu, během které přeskakujeme všechny bloky řezů, jež neobsahují žádnou otázku. Pro takovéto bloky si jen výše odvozenými vzorci spočítáme, kolik políček dohromady obsahují, a nemusíme řešit, v jakém pořadí je navštíví. V rámci řezů, které nějaká hledaná políčka obsahují, potom uděláme totéž s bloky úhlopříček.

Součet kvadratické posloupnosti

Posloupnost x je kvadratická posloupnost, pokud $x_i = ai^2 + bi + c$, kde a, b, c jsou nějaké konstanty. Chceme-li sečíst jejích prvních pár členů, dostaneme výraz

$$\sum_{i=1}^{n} x_i = a\sum_{i=1}^{n} i^2 + b\sum_{i=1}^{n} i + nc.$$

Víme, že $\sum_{i=1}^{n} i = \frac{n(n-1)}{2}$, takže nám zbývá odvodit vzorec pro součet $\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2$.

Roznásobme si výraz $(k-1)^3 = k^3 - 3k^2 + 3k - 1$. Úpravou dostaneme $k^3 - (k-1)^3 = 3k^2 - 3k + 1$. Sečtěme nyní tuto rovnost pro k od 1 do n. Dostaneme:

$$(n^3 - (n-1)^3) + ((n-1)^3 - (n-2)^3) + \cdots + (1^3 - 0^3) = 3\sum_{k=1}^{n} k^2 - 3\sum_{k=1}^{n} k + n.$$

Všimněte si, že na levé straně se všechny členy kromě $n^3 - 0^3$ odečtou. Na pravé straně můžeme použít známý vzorec $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$, a tím dostaneme:

$$n^3 = 3\sum_{k=1}^{n} k^2 - 3\frac{n(n+1)}{2} + n.$$

Nakonec můžeme vyjádřit $\sum_{k=1}^{n} k^2$ a dostaneme kýžený vzoreček:

$$\sum_{k=1}^{n} k^2 = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6} = \frac{n(n+1)(2n+1)}{6}.$$

Pro vyřešení úlohy nebylo potřeba si tento vzoreček pamatovat ani ho umět odvodit. Stačí vědět, že $\sum_{k=1}^{n} k^2$ je polynom třetího stupně v n (stejně jako $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$ je polynom druhého stupně a $\sum_{k=1}^{n} 1 = n$ je polynom prvního stupně), tedy platí $\sum_{k=1}^{n} k^2 = P(n) = an^3 + bn^2 + cn + d$ pro nějaká (zatím neznámá) čísla a, b, c, d.

Zřejmě pro $n = 0$ má součet vyjít nulový, tj. $P(0) = 0$, čili $d = 0$. Je velice jednoduché ručně spočítat součet první jedné (1), dvou (5) a tří (14) druhých mocnin, tedy víme, že $P(1) = 1$, $P(2) = 5$ a $P(3) = 14$. Dosazením do P dostanete soustavu tří lineárních rovnic o třech neznámých, kterou není těžké vyřešit a spočítat koeficienty P.

P-III-6 Firma

Začneme nějakými základními pozorováními, z nichž potom složíme první funkční řešení. Tomu následně postupně vylepšíme složitost.

Při popisu řešení budeme místo časové složitosti počítat počet dotazů, které Adamovi položíte. Jelikož ale zbytek řešení bude vždy jednoduchý, tak bude platit, že jde zároveň i o odhad časové složitosti.

Základní pozorování

Firma hloubky k má $n = 1+2+2^2+\cdots+2^k = 2^{k+1}-1$ zaměstnanců. Firma je úplný binární strom s ředitelkou v kořeni a se stážisty v jednotlivých listech. Jelikož je to strom, má přesně $n-1$ hran (dvojic přímý nadřízený – přímý podřízený).

Hrany ve firmě jsou sice orientované (záleží na tom, kdo je nadřízeným koho), my je ale budeme objevovat jako neorientované, tj. budeme vědět, že nějaká dvojice v hierarchii sousedí, obecně ale nebudeme vědět, kdo z nich dvou je nadřízený.

Předpokládejme, že nyní známe všechny neorientované hrany. Jak potom vyřešit úlohu? Ředitelku nalezneme lehce, jelikož jako jediná má stupeň dva (stážisté mají stupeň 1 a všichni ostatní stupeň 3). Jakmile jsme nalezli ředitelku, stačí z ní celou firmu prohledat (je jedno, zda do šířky či do hloubky) a hrany zorientovat.

Nalezení hran na kubický počet dotazů

Všimněte si, že vrcholy x a y jsou spojené hranou právě tehdy, když neexistuje nikdo, skrz nějž by chodily zprávy mezi x a y. Pro každou dvojici vrcholů tedy můžeme ověřit, zda je mezi nimi hrana, tak, že postupně vyzkoušíme všechny další vrcholy z a vždy se zeptáme, zda z leží mezi x a y. Takto najdeme všechny hranu na $\mathcal{O}(n^3)$ dotazů.

Nalezení souseda na lineární počet dotazů

Když při dotazu, zda z leží mezi x a y, dostaneme odpověď „ano“, neznamená to jen, že x a y nejsou spojené hranou. Dozvíme se tím i to, že z má k x blíže než y. Mějme nyní nějaký konkrétní vrchol v. Ukážeme si, jak pomocí tohoto pozorování nalézt nějakého jeho souseda na $\mathcal{O}(n)$ dotazů.

Začněme s libovolným jiným vrcholem jako kandidátem. Postupně budeme procházet všechny další vrcholy a pro každý takový vrchol x se zeptáme, zda x leží mezi v a aktuálním kandidátem. Pokud leží, stane se x novým kandidátem. Ve chvíli, kdy skončíme (na každý vrchol jsme se již podívali), musí aktuální kandidát být sousedem vrcholu v. To si nyní dokážeme:

Jakmile jsme zvolili prvního kandidáta, máme jednoznačně určenou cestu z něj do vrcholu v. Všimněte si, že všechny vrcholy, které během hledání budeme mít jako kandidáty, leží na této cestě, a ve chvíli, kdy jsme nalezli nového kandidáta, se posuneme po této cestě blíže k x. Označme jako s její poslední vrchol před v – zřejmě s je soused v. Ve chvíli, kdy jsme kontrolovali s, jsme nutně museli dostat odpověď „ano“. Tehdy se s stalo kandidátem a muselo jím zůstat až do konce, jelikož mezi v a s už žádný jiný vrchol neleží.

Nalezení všech sousedů na lineární počet dotazů

Každý z našich vrcholů má nejvýše tři sousedy. Pokud tedy chceme najít všechny sousedy vrcholu v, mohlo by stačit výše popsaný postup nejvýše třikrát zopakovat. Musíme si však dát pozor, abychom v pozdějších kolech nenašli znovu některého z již nalezených sousedů. Jak to udělat?

Kdybychom vrchol v ze stromu odstranili, rozpade se zbytek na tolik komponent, kolik má v sousedů. Při hledání souseda platí, že najdeme vždy toho, z jehož komponenty zvolíme prvního kandidáta.

Představme si, že jsme právě našli nějakého souseda s vrcholu v. Nyní se pro každý další vrchol x zeptáme, zda s leží mezi x a v. Kladnou odpověď dostaneme právě pro ty vrcholy, které leží ve stejné komponentě jako s. Takto jsme na lineární počet dotazů našli celou komponentu obsahující s. Zbylé sousedy v už pak budeme hledat jen mezi zbylými vrcholy.

Jelikož má v nejvýše tři sousedy, stačí výše popsaný postup zopakovat nejvýše třikrát. Pokaždé si zvolíme kandidáta z komponenty, již jsme ještě nezpracovali, na méně než n otázek najdeme souseda vrcholu v a na dalších méně než n otázek najdeme celou komponentu, v níž tento soused leží.

Dohromady tedy položíme méně než $6n$ otázek (ve skutečnosti jde ještě ušetřit, například nemusíme hledat vrcholy ve stejné komponentně jako případný třetí soused, budou to všechny zbývající vrcholy), získáme tak sousedy vrcholu v a pro každého z nich i seznam vrcholů tvořících jeho komponentu.

Toto nám dává kvadratické řešení: Pro každý vrchol zvlášť na lineární počet dotazů najdeme všechny jeho hrany a následně použijeme stejný postup jako ve výše popsaném řešení s kubickým počtem dotazů.

Efektivní hledání kořene

Nyní si ukážeme, jak efektivně najít ředitelku celé firmy. Začněme s libovolným vrcholem a spusťme pro něj výše popsanou funkci, která najde jeho sousedy. Pokud jsme se dozvěděli, že má právě dva sousedy, měli jsme štěstí a jsme hotovi. V opačném případě si vybereme toho souseda, který má největší komponentu, přesuneme se do něj a celý postup zopakujeme.

Je jednoduché nahlédnout, že pokud aktuální vrchol není kořenem stromu, tak komponenta s kořenem obsahuje více než polovinu celkového počtu vrcholů. Proto bez ohledu na to, kde začneme, povede výše popsaný postup k tomu, že v každém kroku se ve stromě pohneme o krok blíže ke kořeni.

Nejhorší, co se nám mohlo stát, je začít v listu, tedy v některém ze stážistů. Tehdy budeme potřebovat k kroků na to, abychom se dostali k ředitelce. Jelikož k je přibližně $\log_2(n)$, potřebujeme $\mathcal{O}(\log n)$ kroků, každý z nichž umíme provést s lineárním počtem dotazů. Proto tímto postupem najdeme ředitelku na $\mathcal{O}(n \log n)$ dotazů.

Vzorové řešení

Právě popsaným postupem na $\mathcal{O}(n \log n)$ dotazů najdeme kořen celého stromu a rekurzivně budeme zpracovávat jeho podstromy a rekonstruovat hrany stromu.

Naše rekurzivní funkce dostane dva parametry: Kořen podstromu, který chceme zpracovat, a seznam všech vrcholů v tomto podstromu.

Implementace této funkce bude jednoduchá: Opět použijeme naši funkci na nalezení všech sousedů aktuálního kořenu, tentokrát se však budeme ptát jen na vrcholy patřící do jeho podstromu. Pro každého z nich se rekurzivně zavoláme na něj a na jeho komponentu. Stejně jako například u algoritmu MergeSort pro třídění lze spočítat, že celé zpracování stromu bude dohromady vyžadovat $\mathcal{O}(n \log n)$ dotazů (zpracování stromu s n vrcholy vyžaduje $\mathcal{O}(n)$ dotazů a nejvýše dvě rekurzivní volání na stromy se zhruba $n/2$ vrcholy). Dostáváme tak řešení s celkovou časovou složitostí $\mathcal{O}(n \log n)$.

```
#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> prirazeny;

void najdi_sousedy(int vrchol, vector<int> okoli, vector<int> &sousede,
                   vector< vector<int> > &komponenty) {
    sousede.clear();
    komponenty.clear();
    int P = okoli.size();
    int pocet_prirazenych = 0;
    for (int x : okoli) {
        prirazeny[x] = false;
        if (x == vrchol) { prirazeny[x] = true; ++pocet_prirazenych; }
    }
    while (pocet_prirazenych < P) {
        int kandidat = -1;
        for (int n = 0; n < P; ++n) {
            int curr = okoli[n];
            if (prirazeny[curr]) continue;
            if (kandidat == -1) {
                kandidat = curr;
                continue;
            }
            if (otazka(kandidat, curr, vrchol)) kandidat = curr;
        }
        sousede.push_back(kandidat);
        prirazeny[kandidat] = true;
        ++pocet_prirazenych;
        // Zde by to šlo ještě zoptimalizovat: Máme-li již třetího souseda,
        // jeho komponentu tvoří všichni nepřiřazení.
        vector<int> komponenta;
        for (int n=0; n<P; ++n) {
            int curr = okoli[n];
            if (prirazeny[curr]) continue;
            if (otazka(vrchol,kandidat,curr)) {
```

```
                komponenta.push_back(curr);
                prirazeny[curr] = true;
                ++pocet_prirazenych;
            }
        }
        komponenty.push_back(komponenta);
    }
}

int main() {
    int K, L;
    cin >> K >> L;

    int N = (1 << (K+1)) - 1;
    prirazeny.resize(N+1);
    vector<int> vsichni(N);
    iota( vsichni.begin(), vsichni.end(), 1 );

    int where = 1;
    vector<int> sousede;
    vector<vector<int> > komponenty;

    while (true) {
        najdi_sousedy(where, vsichni, sousede, komponenty);
        if (sousede.size() == 2) break;
        int i = 0;
        for (int j=  0; j < int(sousede.size()); ++j)
            if (komponenty[j].size() > komponenty[i].size())
                i = j;
        where = sousede[i];
    }
    vector< vector<int> > odpovedi(N+1);
    odpovedi[where] = sousede;

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