A-III-4 Farebné oázy

Úlohu budeme samozrejme riešiť pomocou teórie grafov. Na začiatku máme graf tvorený n izolovanými vrcholmi (oázami), každý inej farby. V takomto grafe máme efektívne simulovať dva typy operácií: prefarbenie vrcholov z jednej farby na inú a pridanie všetkých možných hrán medzi vrcholmi dvoch farieb.
Obmedzenia na veľkosť vstupu sú však také, že si nemôžeme dovoliť popísané operácie skutočne vykonávať. Ak napr. máme 500 000 bielych a 500 000 čiernych vrcholov a príde zmena druhého typu pre bielu a čiernu, mali by sme pridať až $500\,000^2$ nových hrán.

Zamyslime sa najskôr nad verziou úlohy, kedy najskôr len prefarbujeme a potom len spájame. Predstavme si, že sme nejak spracovali všetky prefarbovania a teraz pre každú oázu vieme, akú má farbu. Ako teraz šikovnejšie robiť spájanie?
Predstavme si, že okrem toho, že budeme mať jeden vrchol pre každú oázu, si spravíme aj jeden vrchol pre každú farbu. Každú oázu si potom spojíme s vrcholom pre jej farbu. Takýto prístup vyzerá nádejne: keď máme spojiť dve farby, stačilo by namiesto spájania vrcholov systémom „každý s každým“ pridať len jednu hranu, ktorou spojíme vrcholy predstavujúce celé farby.
Takáto konštrukcia má ešte nejaké muchy. Nechali sme napríklad nezodpovedanú otázku, aké dĺžky majú mať jednotlivé hrany. Táto otázka však bude dôležitá až o chvíľu, najskôr vyriešime iný, dôležitejší problém. Tým je skutočnosť, že akonáhle spojíme napr. všetky červené oázy s vrcholom predstavujúcim červenú farbu, bude sa dať po týchto hranách chodiť aj medzi ľubovoľnými dvoma červenými vrcholmi. To ale vo všeobecnosti nemáme vedieť spraviť.
Pomôžeme si pomerne klasickým trikom: niektoré hrany v našom novom grafe budú *orientované* (teda prechodné len jedným smerom) a pre každú farbu budeme mať nie jeden, ale až dva vrcholy. Tie budeme volať *výstupný a vstupný.*
Nové hrany v tomto grafe spravíme nasledovne:

- Z každej oázy i budeme mať orientovanú hranu dĺžky 0 do *výstupného* vrcholu pre jej farbu.
- Symetricky, zo *vstupného* vrchola pre farbu budeme mať orientovanú hranu dĺžky 0 do každej oázy tej farby.
- Ak máme hranou spojiť farby x a y, spravíme to tak, že pridáme orientovanú hranu dĺžky 1 z výstupného vrcholu x do vstupného y, a tiež naopak – teda z výstupného y do vstupného x.

V takto zostrojenom grafe potom už naozaj bude platiť, že každým pridaním novej hrany pridáme možnosť prejsť medzi dvoma farbami, pričom takýto prechod má celkovú dĺžku $0 + 1 + 0 = 1$.

Teraz si ešte ukážeme, ako toto riešenie zovšeobecniť a vyriešiť pôvodnú súťažnú úlohu. Na to potrebujeme ešte vymyslieť, ako efektívne vyriešiť prefarbovanie.
Na začiatku si vytvoríme výstupné a vstupné vrcholy pre všetky pôvodné farby a prepojíme ich hranami dĺžky 0 s oázami tak, ako sme si popísali vyššie.
Teraz si potrebujeme rozmyslieť, ako odsimulovať prefarbenie všetkých oáz farby x na farbu y. Opäť začneme nesprávnou myšlienkou, ukážeme si, prečo nefunguje a potom aj to, ako to opraviť.
Tu by sme mohli mať pokušenie len pridať orientovanú hranu dĺžky 0 z výstupného vrcholu farby x do výstupného vrcholu farby y (a symetricky pre vstupné vrcholy). Veď predsa ak mala doteraz oáza farbu x, tak sa z jej vrcholu vieme dostať do vrcholu pre farbu x, a ak odtiaľ dostaneme možnosť ísť do vrcholu pre farbu y, tak sa už doň vieme dostať zo všetkých oáz, ktoré majú teraz farbu y.
Prečo toto nefunguje? Hneď z dvoch dôvodov. Prvým je, že sme mohli *v skoršom kroku* spojiť hranou farbu y s nejakou inou farbou z. Oázy, ktoré vtedy ešte mali farbu x, tieto hrany nemajú mať. Ale ak im umožníme dostať sa do výstupného vrcholu pre farbu y, dostanú tým prístup práve k týmto už skôr pridaným hranám. No a druhým dôvodom je, že síce sme práve prestali mať použitú farbu x, ale v budúcnosti môžeme nejaké iné oázy prefarbiť na farbu x. Ak by sme potom tie znova pripojili k tomu istému vrcholu pre farbu x, spravilo by to ešte väčší chaos.
Opraviť túto myšlienku vieme nasledovne: namiesto prefarbovania x na y pridáme ďalšiu, úplne novú farbu y' a na tú prefarbíme aj x, aj y. Pridáme teda orientované hrany dĺžky 0 z výstupných vrcholov pre x aj y do

výstupného vrcholu pre y' a potom symetricky aj orientované hrany zo vstupného vrcholu pre y' do vstupných vrcholov pre pôvodné dve farby. Navyše si v pomocnom poli zapamätáme, že farbu x už momentálne nemáme (ak sa v budúcnosti objaví, je to nová farba) a že ak sa v budúcnosti na vstupe objaví farba y, tak namiesto nej máme použiť túto novú farbu y'.

Takto už dostaneme graf, v ktorom sú vzdialenosti medzi oázami rovnaké ako v grafe, ktorý by sme dostali priamočiarou simuláciou postupu zo zadania.
Presnejšie, ak sme v priamočiarej simulácii niekedy pridali hranu e z oázy x do oázy y pridanú v okamihu, kedy x mala farbu x' a y mala farbu y', v našom grafe sa z x vieme dostať do y cestou dĺžky 1 tak, že najskôr orientovanými hranami dĺžky 0 prejdeme cez všetky (výstupné vrcholy pre) farby, ktoré mal vrchol x od začiatku do okamihu, kedy bola pridaná hrana e, potom prejdeme orientovanou hranou do (vstupného) vrcholu pre farbu, ktorú vtedy mal vrchol y, a odtiaľ sa vrátime cez (vstupné vrcholy pre) skoršie farby ktoré mal y späť až do samotného vrcholu y.
A aj naopak, ak existuje ľubovoľná cesta z oázy x do oázy y v našom grafe, tak musí mať vyššie popísaný tvar a z toho vieme odvodiť, že (aj kedy) sme v priamočiarej simulácii tieto dve oázy naozaj prepojili.

Je zjavné, že po inicializácii v čase $O(n)$ vieme každú z q zmien realizovať v konštantnom čase. Ako sme si práve zdôvodnili, na konci dostaneme graf s $O(n+q)$ vrcholmi a $O(n+q)$ hranami, ktorý má správne vzdialenosti. Odpoveď, ktorú máme vypísať, preto vieme zistiť jednoducho tak, že v našom šikovnejšie postavenom grafe nájdeme najkratšiu cestu z vrcholu u do vrcholu v.
Keďže všetky hrany majú dĺžky 0 alebo 1, môžeme na to stále použiť prehľadávanie do šírky (BFS), len s drobnou úpravou: keď nájdeme novú lepšiu vzdialenosť do vrcholu v a prišli sme doň hranou dĺžky 0, zaradíme ho nie na koniec ale *na začiatok* fronty vrcholov čakajúcich na spracovanie. (Podobne ako pri klasickom BFS takto dosiahneme, že neustále platí invariant, že vo fronte čakajú na spracovanie najskôr vrcholy ktoré sú od štartu vo vzdialenosti d a až za nimi možno sú vrcholy, do ktorých je najlepšia zatiaľ nájdená vzdialenosť $d+1$.)
Takto teda dostávame riešenie s celkovou časovou aj pamäťovou zložitosťou $O(n+q)$.

Listing programu (C++)

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

struct hrana { int dest, len, nxt; };
ostream& operator<< (ostream& out, const hrana &e) { return out << "(" << e.dest << "@" << e.len << ")"; }

int n, q, u, v, f;
vector<hrana> G;
vector<int> prva_hrana;
vector<int> realna_farba(n);

void pridaj_hranu(int src, int dest, int len) {
    int id = G.size();
    G.push_back( {dest, len, prva_hrana[src]} );
    prva_hrana[src] = id;
}

void prefarbi(int x, int y) {
    int rx = realna_farba[x], ry = realna_farba[y];
    if (rx == -1) return;

    int rz = f++;
    realna_farba[x] = -1;
    realna_farba[y] = rz;

    pridaj_hranu(n+2*rx, n+2*rz, 0);
    pridaj_hranu(n+2*rz+1, n+2*rx+1, 0);

    if (ry != -1) {
        pridaj_hranu(n+2*ry, n+2*rz, 0);
        pridaj_hranu(n+2*rz+1, n+2*ry+1, 0);
    }
}

void spoj(int x, int y) {
    int rx = realna_farba[x], ry = realna_farba[y];
    if (rx == -1 || ry == -1) return;
    pridaj_hranu(n+2*rx, n+2*ry+1, 1);
    pridaj_hranu(n+2*ry, n+2*rx+1, 1);
}
```

```
int bfs() {
    int nn = n + 2*f;
    vector<int> vzdialenost(nn, nn+47);
    vzdialenost[u] = 0;
    deque<int> Q;
    Q.push_back(u);
    while (!Q.empty()) {
        int kde = Q.front();
        Q.pop_front();
        int eid = prva_hrana[kde];
        while (eid != -1) {
            hrana e = G[eid];
            eid = e.nxt;
            int kam = e.dest, vzd = vzdialenost[kde] + e.len;
            if (vzdialenost[kam] <= vzd) continue;
            vzdialenost[kam] = vzd;
            if (e.len) Q.push_back(kam); else Q.push_front(kam);
        }
    }
    return vzdialenost[v];
}

int main() {
    ios_base::sync_with_stdio(false); cin.tie(NULL); cout.tie(NULL);
    cin >> n >> q >> u >> v;
    G.reserve(2*n + 4*q + 7);
    prva_hrana.resize(3*n + 2*q + 7, -1);
    f = n;
    realna_farba.resize(n);
    for (int i=0; i<n; ++i) {
        realna_farba[i] = i;
        pridaj_hranu(i, n+2*i, 0);
        pridaj_hranu(n+2*i+1, i, 0);
    }
    for (int qi=0; qi<q; ++qi) {
        string cmd;
        int x, y;
        cin >> cmd >> x >> y;
        if (cmd == "p") prefarbi(x,y); else spoj(x,y);
    }
    cout << bfs() << endl;
}
```

A-III-5 Jednosmerná jednokoľajka

Začneme niekoľkými pozorovaniami.

- Keď sa dobehnú nejaké vlaky, idú spolu rýchlosťou prvého = najpomalšieho z nich. Žiadna skupina vlakov teda nikdy nejde pomalšie ako osobák.
- Osobák je najpomalší, a teda nikdy žiaden iný vlak nedobehne. Preto platí, že s akým meškaním ho vypravíme, s takým príde do cieľa.
- Expres je najrýchlejší. Ten naopak nikdy nebude obmedzovať iné vlaky: keď máme ľubovoľný grafikon a vynecháme z neho jeden expres, ostatné vlaky by stále do cieľa dorazili v presne rovnakých časoch ako keď tam aj ten expres bol. To ale znamená, že expres nikdy nemá zmysel zdržiavať v Jablonici – ostatné vlaky neovplyvní a jemu samotnému to zjavne nemá ako pomôcť. Môžeme ho teda vždy okamžite vypraviť.
- V sade vstupov, v ktorej nemáme žiadne expresy, môžeme tú istú úvahu spraviť pre rýchliky. Všetky rýchliky teda môžeme rovno vypraviť, ostáva nám len rozhodnúť, koľko pozdržať ktorý osobák.
- Vlaky rovnakej rýchlosti stačí vždy vypraviť v ich pôvodnom poradí – teda nikdy sa nám napr. neoplatí vypraviť rýchlik A, ktorý mal odísť v čase 10, až po rýchliku B, ktorý mal odísť v čase 17. Totiž ak by sme rýchlik A namiesto toho vypravili tesne pred rýchlikom B, príde A do cieľa v rovnakom alebo lepšom čase ako predtým (teraz je to naraz s B, predtým to bolo buď naraz alebo neskôr), B príde do cieľa v presne rovnakom čase ako predtým a žiaden z ostatných vlakov to neovplyvní negatívne – vlaky A a B pôjdu spolu presne rovnako ako predtým šiel B, a vlakom vypraveným po B teraz len z trate zmizol vlak A, ktorý ich predtým mohol spomaliť.
- Každý vlak, ktorý vypravíme neskôr ako v pôvodnom čase odchodu, môžeme vypraviť bezprostredne po inom vlaku. Totiž ak máme ľubovoľný plán, kedy vypraviť vlaky, a vieme niektorý vlak vypraviť skôr *bez toho, aby sme zmenili relatívne poradie vlakov* (t.j. bez toho, aby tento vlak predbehol nejaký iný), tak

zjavne nič nepokazíme, keď tak spravíme – na vlaky pred ním to vplyv nemá, posúvaný vlak určite nepríde do cieľa neskôr ako v pôvodnom scenári a vlakom vypraveným po ňom to môže tiež len prospieť.

- Ak ideme na trať pustiť vlak nejakého typu a v Jablonici nám vlakov toho typu čaká viac, nič nepokazíme, ak ich naraz pustíme všetky.
- Ak by sme tesne po sebe chceli pustiť na trať pomalší a po ňom rýchlejší vlak, nič nepokazíme, ak ich namiesto toho pustíme v opačnom poradí – keď ich poradie vymeníme, pomalší vlak príde do cieľa rovnako a rýchlejší možno skôr ale určite nie neskôr ako v pôvodnom scenári. Preto ak ideme naraz na trať pustiť veľa vlakov rôznych typov, nič nepokazíme, keď ich pustíme v poradí od najrýchlejšieho po najpomalší.
- Kombináciou predchádzajúcich dvoch úvah dostávame, že ak ideme na trať pustiť vlak nejakého typu X, tak môžeme postupne od najrýchlejšieho po najpomalší pustiť všetky v Jablonici čakajúce vlaky, ktoré sú typu X alebo rýchlejšie.

Na záver týchto úvah ešte pre istotu zdôrazníme, že vo všetkých predchádzajúcich odsekoch slová ako „nič nepokazíme, keď“ formálne znamenajú, že hocijaké riešenie vieme upraviť na aspoň rovnako dobré s práve skúmanou vlastnosťou. No a z tohto vyplýva, že aj medzi optimálnymi riešeniami vždy musí existovať také s práve skúmanou vlastnosťou. A akonáhle vieme toto, stačí nám nájsť najlepšie riešenie spomedzi tých s uvedenou vlastnosťou a budeme mať zaručené, že ide zároveň o najlepšie riešenie celej pôvodnej úlohy.

Riešenie v exponenciálnom čase

Pozrime sa, čo sa môže stať, ak budeme pri vypravovaní vlakov dodržiavať pravidlá, ktoré sme si odvodili vyššie. Expresy vždy vypravíme, akonáhle to ide. Z ostatných dvoch typov môžeme mať v ľubovoľnej chvíli v Jablonici niekoľko kusov, ktoré už mali odísť, ale ešte sme ich nevypravili. No a situácia v stanici sa môže zmeniť len v okamihoch, kedy bol práve pristavený nový vlak. V každej takej chvíli sa môžeme rozhodnúť, či vypravíme len čakajúce expresy, alebo aj rýchliky, alebo úplne všetky čakajúce vlaky.
Pre každý z nanajvýš n časov pristavenia vlaku máme teda tri možnosti, ktoré vlaky pustiť. Keď sa pre každý z časov rozhodneme, ktorá z týchto troch možností v ňom nastáva, je už presne určené, ktorý vlak kedy príde do cieľa, a teda aj ich celkové meškanie. Súťažnú úlohu teda vieme vyriešiť vyskúšaním všetkých $O(3^n)$ možností a vybraním najlepšieho riešenia.

Riešenie dynamickým programovaním

Časovú zložitosť vyššie uvedeného riešenia vieme vylepšiť na polynomiálnu pomocou dynamického programovania. Každý zaujímavý okamih počas postupného vypravovania vlakov vyššie popísaným spôsobom môžeme popísať troma číslami: prvé bude udávať počet vlakov, ktoré už mohli z Jablonice odísť, druhé bude posledný čas, kedy sme na trať pustili rýchliky a tretie posledný čas, kedy sme pustili osobáky.
Takýchto okamihov (ktoré budeme nazývať *stavy*), je teda len polynomiálne veľa: aj vlakov aj možných časov odchodu je len $O(n)$, dokopy teda máme $O(n^3)$ stavov.

Pozor, toto nie je ekvivalentné s pamätaním si čísel udávajúcich *koľko* rýchlikov a osobákov sme už pustili na trať. Z posledných časov odchodu vieme tieto počty jednoznačne určiť, keďže vieme, že sme vtedy vypravili všetky čakajúce vlaky daného typu, ale naopak to neplatí – rovnakému počtu vypravených vlakov môžu zodpovedať rôzne časy ich vypravenia na trať, tomu môžu zodpovedať rôzne časy príchodu posledného z nich do cieľa, a to môže mať vplyv na to, koľko budú v cieli meškať vlaky, ktoré sme ešte nespracovali.
Všimnite si tiež, že z popisu konkrétneho stavu vieme jednoznačne určiť aj to, kedy príde do cieľa posledný z doteraz vypravených vlakov. A teda keď sa v budúcnosti rozhodneme vypraviť nejaké ďalšie vlaky, budeme vedieť jednoznačne určiť, kedy tie prídu do cieľa, a teda aj to, koľko ktorý z nich v cieli mešká.

Našu úlohu teraz vieme vyriešiť tak, že budeme postupne po jednom spracúvať vlaky (v poradí, v akom sú pristavené v Jablonici) a naraz simulovať všetky možné potenciálne-optimálne spôsoby vypravovania vlakov. Po každom spracovanom vlaku si vypočítame množinu všetkých stavov, v ktorých sa simulácia môže nachádzať.
Kľúčové pozorovanie pre efektívnosť tohto algoritmu je, že ak sa kedykoľvek vieme rôznymi spôsobmi dostať do toho istého stavu, stačí ďalej pokračovať s tým spôsobom, pri ktorom sme doteraz vyrobili najmenej meškania.

Keď takto spracujeme všetky vlaky a zistíme informácie o všetkých dosiahnuteľných stavoch, budeme mať vyriešenú našu súťažnú úlohu – jej riešením bude najmenšie celkové meškanie spočítané pre niektorý zo stavov, v ktorých sme už vypravili všetky vlaky všetkých typov.

Na začiatku sme v jedinom možnom stave. Ten si vieme popísať tak, že sme rýchliky aj osobáky naposledy vypravili v čase $-\infty$ a budeme si pamätať, že pre tento stav majú doteraz vypravené vlaky (ktorých je nula) celkové meškanie 0.
Predstavme si teraz, že sme už spracovali prvých x vlakov a vieme, v ktorých stavoch po nich vieme byť a o každom aj to, akým najlepším spôsobom ho vieme dosiahnuť. Ako spracujeme nasledujúci vlak? Tak, že z každého dosiahnuteľného stavu skúsime všetkými možnými spôsobmi pokračovať.
Pribudol nám vlak. Ak je to expres, rovno ho vypravíme. Potom sa rozhodneme, či vypraviť aj všetky čakajúce rýchliky. A ak sme vypravili aj tie, následne sa rozhodneme, či vypraviť aj všetky čakajúce osobáky. Takto dostaneme tri možné nové stavy. Ku každému z nich vieme aj spočítať celkové meškanie doteraz vypravených vlakov – zoberieme to z minulého stavu a pripočítame k nemu meškanie vlakov, ktoré sme práve vypravili. Tento posledný krok si rozpíšeme detailnejšie.
V prvom rade si treba rozmyslieť, že o každom z vypravených vlakov vieme povedať, kedy príde do Brezovej. Tento čas je rovný maximu z času, kedy tam prišiel predchádzajúci vlak (ak ho cestou dobehneme) a súčtu aktuálneho času a času, ktorý daný typ vlaku potrebuje na cestu (teda času, kedy by sme tam prišli ak nijaký iný vlak cestou nedobehneme). No a keď vieme, kedy tam vlak bude a tiež vieme, kedy tam mal byť (to je priamo dané údajmi na vstupe), dostaneme jeho meškanie ako rozdiel týchto dvoch časov.
No a v druhom rade je tu otázka, ako toto meškanie spočítať efektívne. Dalo by sa síce vypravované vlaky spracúvať po jednom, ale vieme to robiť aj šikovnejšie.
Jeden možný elegantný spôsob je taký, že ku každému stavu si budeme pamätať nielen optimálny súčet meškania už vypravených vlakov, ale taktiež si budeme priebežne udržiavať aj sadu údajov odvoditeľných zo samotných parametrov stavu. Pre každý typ vlaku si budeme pamätať dve čísla: koľko vlakov toho typu čaká na vypravenie a aký je súčet časov, kedy by mali byť v Brezovej bez meškania.
Keď nám pribudne nový vlak, tieto čísla pre jeho typ vieme upraviť v konštantnom čase. A keď sa rozhodneme vypraviť všetky vlaky konkrétneho typu, vieme, že všetky prídu do cieľa v tom istom čase. Keď ten určíme a vynásobíme ich počtom, dostaneme súčet ich reálnych časov príchodu do Brezovej. No a keď od toho odčítame nami šikovne spočítaný súčet ich plánovaných časov príchodu, dostaneme súčet ich meškaní – čiže presne to, čo sme chceli vedieť.
Takéto riešenie teda každý dosiahnuteľný stav raz spracuje, a to v konštantnom čase. Jeho celková časová zložitosť je preto $O(n^3)$. Pamäťová zložitosť vie byť $O(n^2)$: keď z množiny stavov dosiahnuteľných po spracovaní x vlakov vypočítame novú množinu stavov po spracovaní $x + 1$ vlakov, môžeme tú prvú množinu zabudnúť a ďalej pokračovať len s tou druhou.

Listing programu (C++)

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

const int MAXN = 501;
const long long NEKONECNO = 1LL << 62;
const map<string,int> TYP_VLAKU = { {"Os",0}, {"R",1}, {"Ex",2} };

struct stav {
    long long min_doteraz;
    int posledny_prichod;
    int pocet_cakajucich[3];
    long long sucet_prichodov[3];
    stav() { min_doteraz = NEKONECNO; }
};

stav best[MAXN+1][MAXN+1], new_best[MAXN+1][MAXN+1];

int main() {
    int N;
    vector<int> T(3);
    cin >> N >> T[0] >> T[1] >> T[2];

    vector<int> casy_odchodu(N+1); // casy_odchodu[0] je zarazka: fiktivny cas davno pred startom kedy vsetko odislo

    best[0][0].min_doteraz = best[0][0].posledny_prichod = 0;
```

```
    // postupne spracuvame vlaky
    for (int n=1; n<=N; ++n) {
        string styp;
        cin >> casy_odchodu[n] >> styp;
        int typ = TYP_VLAKU.at(styp);

        for (int last0=0; last0<=n; ++last0) for (int last1=last0; last1<=n; ++last1)
            new_best[last0][last1].min_doteraz = NEKONECNO;

        // postupne prechadzame cez vsetky dosiahnutelne stavy, kazdemu pridame aktualny vlak
        int pc[3], idx[3];
        long long sp[3];
        for (int last0=0; last0<n; ++last0) for (int last1=last0; last1<n; ++last1) {
            stav teraz = best[last0][last1];
            if (teraz.min_doteraz == NEKONECNO) continue; // nedosiahnutelny stav

            for (int t=0; t<3; ++t) { pc[t] = teraz.pocet_cakajucich[t]; sp[t] = teraz.sucet_prichodov[t]; }
            pc[typ] += 1;
            sp[typ] += casy_odchodu[n] + T[typ];

            // vyskusame vsetky tri moznosti, ze ako vela toho vypravit
            long long nove_meskanie = teraz.min_doteraz;
            int novy_prichod = teraz.posledny_prichod;
            idx[0] = last0; idx[1] = last1;

            for (int t=2; t>=0; --t) {
                novy_prichod = max( novy_prichod, casy_odchodu[n]+T[t] );
                nove_meskanie += 1LL * pc[t] * novy_prichod - sp[t];
                idx[t] = n;
                pc[t] = 0;
                sp[t] = 0;
                stav &novy = new_best[idx[0]][idx[1]];
                novy.min_doteraz = min( novy.min_doteraz, nove_meskanie );
                novy.posledny_prichod = novy_prichod;
                for (int t=0; t<3; ++t) { novy.pocet_cakajucich[t] = pc[t]; novy.sucet_prichodov[t] = sp[t]; }
            }
        }
        for (int last0=0; last0<=n; ++last0) for (int last1=last0; last1<=n; ++last1)
            best[last0][last1] = new_best[last0][last1];
    }

    long long odpoved = NEKONECNO;
    for (int last0=0; last0<=N; ++last0) for (int last1=last0; last1<=N; ++last1) {
        stav teraz = best[last0][last1];
        if (accumulate(teraz.pocet_cakajucich, teraz.pocet_cakajucich+3, 0) == 0)
            odpoved = min( odpoved, teraz.min_doteraz );
    }
    cout << odpoved << endl;
}
```

A-III-6 Triedička III

Začneme tým, že si prejdeme myšlienky riešení pre jednotlivé sady. Následne si ukážeme pár trikov pre príjemnejšiu implementáciu generátora programov a na záver si ukážeme konkrétne riešenia využívajúce tieto triky.

Lineárne riešenie

Pomocou $O(n)$ krokov vieme súťažnú úlohu vyriešiť pomerne priamočiaro. Napr. môžeme n-krát spraviť fázu, v ktorej každé jadro pošle cyklicky doľava ID aj tajomstvo. Takto postupne každé jadro uvidí všetky ID a všetky tajomstvá, no a keď uvidí ID, ktoré sleduje, môže si zapamätať jeho tajomstvo.

Všetci sledujú jedného

Jadro, ktoré sleduje samé seba, oznámi svoje tajomstvo jadru 0 (podobne, ako vieme do jadra 0 dostať minimum). Potom túto hodnotu z jadra 0 rozkopírujeme do všetkých jadier.

Každý sleduje iného, rôzna parita

Povieme si, že jadrá s párnym ID budú *vysielať* a jadrá s nepárnym ID *prijímať*. V prvom kroku usporiadame jadrá tak, aby sme mali najskôr všetky vysielajúce a potom všetky prijímajúce. Teraz si každé jadro vypočíta svoj kľúč: vysielajúce jadro bude mať ako kľúč svoje ID, prijímajúce jadro bude mať ako kľúč ID, ktoré sleduje.

Keď teraz jadrá usporiadame podľa tohto kľúča, vznikne nám $n/2$ dvojíc, v ktorých má každé prijímajúce jadro naľavo od seba jadro, ktoré sleduje. Všetky prijímajúce jadrá sa teda vedia naraz dozvedieť hľadané tajomstvá. No a následne si to celé ešte raz zopakujeme, len si jadrá vymenia roly – tie, ktoré vysielali budú teraz prijímať a naopak. Túto podúlohu sme teda dokonca zvládli vyriešiť v konštantnom čase.

Každý sleduje iného – prvé riešenie

Všimnime si, že v predchádzajúcom riešení sme vďaka dodatočnej podmienke o rôznej parite i a s_i vedeli zaručiť, že keď jadro i prijíma, jadro s_i vysiela, a teda vytvoria dvojicu.
Čo presne sa stane, keď to isté riešenie použijeme, ak nemáme zaručenú rôznu paritu? Pre jadro x, ktoré náhodou má rôznu paritu s jadrom s_x, toto riešenie stále zafunguje: keď x prijíma, obaja si spočítajú kľúč s_x, a teda po usporiadaní skončia vedľa seba a x sa dozvie správne tajomstvo. Ostatné jadrá vo všeobecnosti skončia vedľa niekoho iného ako chceli. Tieto jadrá sa želané tajomstvo nedozvedia. Všimnime si ale, že aspoň vieme spoznať, že táto situácia nastala. Keď sa prijímajúce jadro pozerá doľava, vie si totiž nielen pozrieť tajomstvo, ale aj skontrolovať, či naozaj má naľavo od seba jadro, ktoré sleduje.
Namiesto parity (teda skutočnosti, že sa i a s_i líšia v poslednom bite) by sme vedeli použiť ľubovoľný iný bit. Teda napr. namiesto parity rozdelíme jadrá podľa toho, či má ich ID nastavený tretí bit alebo nie. Keď následne použijeme ten istý postup, zase nám vzniknú nejaké dvojice, v ktorých sa prijímajúce jadro dozvie tajomstvo – tie, v ktorých sa i a s_i vo zvolenom bite líšia.

Túto podúlohu vieme teda vyriešit tak, že riešenie predchádzajúcej podúlohy spustíme $(\log n)$-krát: raz pre každý bit. Ak sa i líši od s_i, určite existuje aspoň jeden bit, v ktorom sa líšia, no a v príslušnom kole keď i bude prijímať, s_i bude vysielať a teda sa i dozvie tajomstvo.
Ostáva už len ošetriť prípad $i = s_i$. Ten je ale triviálny: ak vrchol sleduje sám seba, hľadané tajomstvo už pozná.

Každý sleduje iného – druhé riešenie

Ešte v pôvodnom poradí si jadrá rozdeľme do dvojíc: 0 s 1, 2 s 3, atď. V každej dvojici si jadrá navzájom oznámia svoje hodnoty s_i aj t_i.
Pozrime sa na ľubovoľné jadro i. Ak ním sledované jadro s_i je v tej istej dvojici, už jeho tajomstvo pozná. A ak nie, tak vďaka triku, ktorý sme práve spravili, už existuje jadro *opačnej parity ako* i, ktoré pozná tajomstvo, ktoré i chce zistiť.
Distribúciu tajomstiev medzi dvojicami teda teraz vieme spraviť tak, že jadrá s párnym ID budú každé za svoju dvojicu vysielať a jadrá s nepárnym ID za svoju dvojicu prijímať. Potom si to vymenia: nepárne vysielajú, párne prijímajú. No a na záver si už len vo dvojici oznámia, čo sa dozvedeli.
Vysielanie jedným smerom sa dá spraviť napr. na štyri kroky, aby sme mali každú kombináciu „za koho z mojej dvojice vysielam“ a „za koho z mojej dvojice prijímam“.
Toto alternatívne riešenie je ešte efektívnejšie ako predchádzajúce – beží dokonca v konštantnom čase.

Všeobecné riešenie

Ak môže veľa jadier sledovať jedno, tak nás ani trik z predchádzajúcej podúlohy nespasí. Ak sa budeme snažiť vyrobiť podobné riešenie s vysielajúcimi a prijímajúcimi jadrami, budú nám vznikať situácie, kedy sa napravo od vysielajúceho jadra nazbiera celý rad jadier, ktoré chcú poznať jeho tajomstvo. A rozhodne nie je zjavné, ako dosiahnuť, aby sa toto tajomstvo dozvedeli všetky.
Spravme to postupne. Začneme tým, že si jadrá usporiadame podľa toho, koho sledujú. V každej takto získanej skupine posledné jadro (teda to s najväčším ID) bude aktívne a ostatné jadrá budú pasívne. Teraz najskôr dosiahneme, aby sa aktívne jadrá dozvedeli tajomstvá, ktoré hľadajú, a potom sa zamyslíme nad tým, ako toto tajomstvo následne rozkopírovať zvyšku skupiny.
Všimnime si, že priamo z definície platí, že žiadne dve aktívne jadrá nesledujú to isté. Preto môžeme použiť ľubovoľné z riešení predchádzajúcej podúlohy a dosiahneme tak, že všetky aktívne jadrá už budú vedieť hľadané tajomstvá. (Jediná potrebná úprava je, že prijímajúce pasívne jadrá pri usporadúvaní upraceme nabok, aby nezavadzali – napr. tak, že im dáme kľúč -1 alebo $n + 47$.)

Teraz už ostáva len rozkopírovať získanú informácie – naraz z každého aktívneho jadra do pasívnych jadier, ktoré sledujú to isté jadro ako ono.
Keď už aktívne jadrá vedia svoje odpovede, znova si všetky jadrá usporiadame (najskôr podľa ID a potom podľa s_i), čím dostaneme súvislé bloky jadier, ktoré majú rovnaké s_i. Na konci každého bloku je aktívne jadro, ktoré preň pozná odpoveď. Teraz môžeme použiť ako podprogram riešenie podúlohy C z krajského kola a očíslovať si jadrá v tomto novom poradí. Akonáhle už máme toto nové očíslovanie, rozkopírovať si údaje v rámci každého bloku už vieme ľahšie.
Alternatívne môžeme dostať stručnejšie riešenie, ak len priamo upravíme tú istú techniku na vyriešenie tejto úlohy. Tento jednoduchší algoritmus si ľahko popíšeme pomocou rekurzie:

1. Usporiadame nevyradené jadrá podľa aktuálneho ID a potom podľa s_i. Dostaneme súvislé bloky, v ktorých chceme kopírovať. (Na začiatku sú aktuálne ID rovné skutočným ID a nik ešte nie je vyradený.)
2. Všade, kde máme v jednom bloku dvojicu susedov s aktuálnymi ID $2k$ a $2k+1$, ľavého z nich vyradíme.
3. Všetky aktuálne ID vydelíme dvomi.
4. Rekurzívne vyriešime rozkopírovanie informácií medzi jadrami, ktoré ostali nevyradené.
5. Vrátime späť zmeny, ktoré sme spravili v krokoch 3 a 2, následne si jadrá opäť zoradíme ako v kroku 1.
6. Jadrá, ktoré práve prestali byť vyradené, si od svojho pravého suseda skopírujú jeho tajomstvo.

Potrebná hĺbka rekurzie je zjavne $\log n$, potom nám už z každého úseku ostane len jeho aktívne jadro, a to už odpoveď pozná. Následne počas vynárania z rekurzie zjavne vždy platí, že keď nejaké jadro j prestane byť vyradené, jeho pravý sused je v tom istom bloku a už pozná jeho odpoveď, a teda sa ju aj jadro j dozvie.
No a rekurziu môžeme skutočne použiť na vyriešenie našej úlohy, ak chceme – pri implementácii generátora! Ale v podstate rovnako dobre vieme vyššie popísaný program vygenerovať aj čisto pomocou cyklov – najskôr $(\log n)$-krát zopakujeme kroky 1-3 a potom rovnako veľakrát kroky 5-6.

Šikovná implementácia generátora

Generátor programov pre triedičku sa samozrejme dá naprogramovať tak, že napíšeme ručne kostru programu pre triedičku, tú potom vložíme do programu ako pole reťazcov a potom program obsah tohto poľa pomocou vhodných cyklov vypíše a prípadne upraví (napr. priebežne mení hodnoty niektorých konštánt). Vieme to ale spraviť aj lepšie. Ideálne by bolo vedieť písať programy pre triedičku priamo v „normálnom“ programovacom jazyku a nejak dosiahnuť, aby z toho potom vznikol skutočný program pre triedičku.
Prvý krok týmto smerom je napísať si napr. funkciu `inst` (inštrukcia), ktorá vypíše svoje parametre na štandardný výstup (t.j. pridá inštrukciu do vypisovaného programu) a následne ako návratovú hodnotu vráti číslo práve vypísanej inštrukcie. Toto číslo potom môžeme priamo použiť ako parameter pre ďalšiu inštrukciu, ktorá s výsledkom tej predchádzajúcej ďalej pracuje.
Pri programovaní triedičky sa nám často stáva, že jeden logický krok, ako napr. „podľa parity ID zober buď svoj vstup alebo vstup ľavého suseda“ musíme rozpísať ako veľa elementárnych krokov. Vyššie popísaný trik nám umožní všetky tieto kroky spojiť do menej lepšie čitateľných inštrukcií.
Môžeme začať tým, čo chceme spraviť: if. Ten vyzerá nasledovne: `inst("if", ?, ?, ?)`. If má dostať tri parametre. Najskôr vypočítame prvý: paritu ID. Tú vypočítame tak, že si zistíme ID a potom spravíme operáciu „modulo 2“. Zatiaľ teda máme nasledovné:

```
parita = inst("%", inst("id"), inst("const", 2))
inst("if", parita, ?, ?)
```

Následne potom môžeme za ďalší otáznik doplniť, ako vypočítať hodnotu, ktorú chceme dostať, ak je podmienka splnená – teda napr. tento otáznik nahradíme `inst("input", 1)`. A tak ďalej.
Už sme videli, že návratovú hodnotu celého riadku si môžeme priradiť v našom programe do premennej a tej hodnotu neskôr použiť v ďalších príkazoch. Tiež ale môžeme v rámci parametrov `inst` použiť aj symbolické meno (ako prvý parameter dáme napr. `"tmp:"`) a neskôr použiť to.

Ešte šikovnejšia implementácia generátora

Moderné programovacie jazyky nám umožňujú spraviť ešte viac: definovať si vlastnú triedu, ktorá bude predstavovať výsledok postupnosti inštrukcií. Pre takúto triedu si následne vieme definovať operátory, ktoré budú

realizovať aritmetické a logické operácie, a ďalšie funkcie pracujúce s ňou a zodpovedajúce ostatným inštrukciám triedičky. Každý operátor aj každá funkcia vypíše na výstup príslušnú inštrukciu pre triedičku a následne vráti číslo práve vypísanej inštrukcie, aby sme s jej výstupom mohli pracovať ďalej. Toto nám umožní ešte stručnejšie a prehľadnejšie zapisovať výpočty na triedičke.
Ukážeme si jednu možnú implementáciu takéhoto generátora v Pythone.

```
# globalna premenna s cislom naposledy pouzitej instrukcie
posledna_instr = 0

# pomocna funkcia pre ciastocne dosadenie konstant za prve parametre funkcie
# vyskusaj si napr. co spravi "helloprint = dosad(print, 'Hello,')" a potom "helloprint('Bob')"
def dosad(funkcia, *konstanty):
    # definujeme novu funkciu, ktora zavola povodnu ale s dosadenymi konstantami za prve parametre
    def nova_funkcia(*parametre): return funkcia(*konstanty, *parametre)
    # vratime tuto funkciu ako vystup
    return nova_funkcia

# pomocna funkcia pre vypisanie instrukcie a vratenie jej cisla
def instrukcia(prikaz, *argumenty):
    # ak su nejake argumenty cisla a nie sme 'input' ani 'const', spravime z nich konstanty
    if prikaz not in ['const', 'input']:
        argumenty = [ x if isinstance(x, Vysledok) else instrukcia('const',x) for x in argumenty ]
    # zvysime pocitadlo instrukcii, vypiseme ju
    global posledna_instr
    posledna_instr += 1
    print(prikaz, *argumenty)
    return Vysledok(posledna_instr)

# trieda predstavujuca vysledok instrukcie
class Vysledok:
    # ked ju vyrobime, povieme jej, ake ma cislo a ona si ho zapamata
    def __init__(self, cislo): self.cislo = cislo
    # ked tento vysledok neskor pouzivame ako argument, vypise sa zapamatane cislo
    def __str__(self): return str(self.cislo)

# triede Vysledok pridame "magicke operatory" pre matematicke a logicke operacie
# (tu pridavame len priblizne tie, co pouzijeme, dalo by sa ich mat aj viac)
operatory = ['__add__', '__sub__', '__mul__', '__truediv__', '__mod__', '__and__', '__or__', '__eq__', '__lt__']
symboly = ['+', '-', '*', '/', '%', '&', '|', '=', '<']
for op, sym in zip(operatory, symboly): setattr( Vysledok, op, dosad(instrukcia, sym) )

# pre ostatne instrukcie si vyrobime samostatne funkcie; inac pomenujeme tie co sa biju s keywordmi v Pythone
ine_instrukcie = [ 'const', 'copy', 'sort', 'left', 'right' ]
for cmd in ine_instrukcie: globals()[cmd] = dosad(instrukcia, cmd)
idcko = dosad(instrukcia, 'id')
vstup = dosad(instrukcia, 'input')
lnot  = dosad(instrukcia, '!')
ak    = dosad(instrukcia, 'if')
```

Implementácie niektorých generátorov

Ukážeme si teraz, ako pomocou vyššie implementovaného generátora naprogramujeme riešenia niektorých podúloh. Vo všetkých programoch sme si už zo vstupu načítali n (počet jadier, pre ktoré generujeme program) a z neho vypočítali $\log_2 n$ (napr. `logn = n.bit_length() - 1`).
Lineárne riešenie je naozaj na pár riadkov:

```
sledujem, tajomstvo = vstup(1), vstup(2)
vidim_id, vidim_t, vystup = idcko(), copy(tajomstvo), const(0)
for kolo in range(n):
    vidim_id, vidim_t = right(vidim_id), right(vidim_t)
    vystup = ak( vidim_id == sledujem, vidim_t, vystup )
```

Pre názornosť, keď tento program spustíme na vstupe „`4 - -`“, dostaneme toto:

```
input 1
input 2
id
copy 2
const 0
right 3
right 4
= 6 1
if 8 7 5
right 6
right 7
= 10 1
if 12 11 9
right 10
right 11
= 14 1
if 16 15 13
```

```
right 14
right 15
= 18 1
if 20 19 17
```

Riešenie podúlohy, kde každý sleduje iného a sledovaný má vždy opačnú paritu ako sledujúci:

```
moje_id, sledujem, tajomstvo, vystup = idcko(), vstup(1), vstup(2), const(0)
for parita in range(2):
    prijimam = (moje_id + parita) % 2
    sort( prijimam )
    kluc = ak( prijimam, sledujem, moje_id )
    sort( kluc )
    vystup = ak( prijimam, left(tajomstvo), vystup )
```

A vylepšenie na riešenie podúlohy, kde už nemusia byť opačné parity:

```
moje_id, sledujem, tajomstvo = idcko(), vstup(1), vstup(2)
vystup = ak( moje_id == sledujem, tajomstvo, 0 )
moc2 = const(1)
for bit in range(logn):
    for parita in range(2):
        prijimam = (moje_id / moc2 + parita) % 2
        sort( prijimam )
        kluc = ak( prijimam, sledujem, moje_id )
        sort( kluc )
        vystup = ak( prijimam & ( left(moje_id) == sledujem ), left(tajomstvo), vystup )
    moc2 *= 2
copy(vystup)
```

Na záver jedna možná implementácia plného riešenia:

```
def kopiruj(n):
    global akt_id, sledujem, vyradene, vystup
    if n == 1: return

    kluc = ak( vyradene, -1, akt_id )
    sort(kluc)
    sort(sledujem)
    vyradit = lnot(vyradene) & ((akt_id % 2) == 0) & (right(sledujem) == sledujem) & (right(akt_id) == akt_id + 1)
    vyradene = vyradene | vyradit
    save_id = copy(akt_id)
    akt_id /= 2

    kopiruj(n//2) # rekurzivne volanie

    akt_id = copy(save_id)
    vyradene = ak( vyradit, 0, vyradene )
    sort(kluc)
    sort(sledujem)
    vystup = ak( vyradit, right(vystup), vystup )

# v prvom kroku zistime, ktore jadra su aktivne
moje_id, sledujem, tajomstvo = idcko(), vstup(1), vstup(2)
sort(moje_id)
sort(sledujem)
aktivne = lnot(sledujem == right(sledujem)) | ((sledujem == right(sledujem)) & (moje_id > right(moje_id)))

# aktivne jadra zistia hladane tajomstva
vystup = ak( aktivne & (moje_id == sledujem), tajomstvo, 0 )
moc2 = const(1)
for bit in range(logn):
    for parita in range(2):
        prijimam = (moje_id / moc2 + parita) % 2
        sort( prijimam )
        kluc = ak( prijimam, -1, moje_id )
        kluc = ak( prijimam & aktivne, sledujem, kluc )
        sort( kluc )
        vystup = ak( aktivne & prijimam & ( left(moje_id) == sledujem ), left(tajomstvo), vystup )
    moc2 *= 2

# zvysnym jadram ich rozkopirujeme pomocou rekurzie
vyradene, akt_id = const(0), copy(moje_id)
kopiruj(n)
```

ŠTYRIDSIATY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Forišek
Recenzia: Michal Anderle, Tomáš Belan, Paulína Smolárová
Slovenská komisia Olympiády v informatike
Vydal: NIVAM – Národný inštitút vzdelávania a mládeže, Bratislava 2025