A-II-1 Hrozienka nemajú mať nožičky!

Existuje veľa rôznych pomalých riešení, ktoré postupne iterujú cez možné štvorce a testujú, či majú dostatočne málo pavúkov. To úplne najpriamočiarejšie („vyskúšaj postupne úplne všetky štvorce a každý skontroluj tak, že prejdeš všetky jeho políčka a spočítaš pavúky“) má časovú zložitosť $O(n^5)$.

Keď si hrozienka predstavíme ako nuly a pavúky ako jednotky, hľadáme najväčší štvorec so súčtom nanajvýš p. Kľúčom ku všetkým efektívnym riešeniam sú *dvojrozmerné prefixové súčty* týchto hodnôt. Totiž, keď si v čase $O(n^2)$ predpočítame pre každé (r, s) súčet v prvých r riadkoch a s stĺpcoch tabuľky na vstupe, vieme potom v konštantnom čase vypočítať súčet ľubovoľného obdĺžnika – a teda aj ľubovoľného štvorca. O ľubovoľnom štvorcovom kuse koláča budeme takto vedieť v konštantnom čase povedať, či je dobrý alebo zlý.
Detailnejší popis prefixových súčtov nájdete tu: https://www.ksp.sk/kucharka/2d_prefixove_sumy/.
Vyššie popísané riešenie hrubou silou vieme teraz priamočiaro zlepšiť na časovú zložitosť $O(n^3)$: stále prejdeme všetky možné štvorce, ale teraz už vieme každý z nich otestovať v konštantnom čase.
(Štvorcov je rádovo n^3, lebo máme n^2 možností, kde bude pravý dolný roh, a pre každý takto zvolený roh máme nanajvýš n možností pre dĺžku strany. Tým už je celý štvorec jednoznačne určený.)

Ešte lepšie riešenie vieme dostať tak, že si budeme šikovnejšie voliť, na ktoré štvorce sa pozrieť a na ktoré nie. Môžeme si napríklad všimnúť nasledovnú vlastnosť: keď si zvolíme pravý dolný roh štvorca a teraz postupne zväčšujeme jeho veľkosť od 1 po n, odpoveď na otázku, či je aktuálny štvorec dobrý, bude *monotónna*. Akonáhle prvýkrát narazíme na štvorec, ktorý už má priveľa pavúkov, vieme, že aj všetky väčšie štvorce s týmto istým pravým dolným rohom budú zlé – totiž každý väčší štvorec tiež bude obsahovať všetky tieto pavúky (a možno aj nejaké ďalšie).
To ale znamená, že pre každý pravý dolný roh môžeme na určenie optimálnej dĺžky strany použiť *binárne vyhľadávanie*. Takto pre konkrétny roh nájdeme optimálne riešenie pomocou $O(\log n)$ otázok, čím dostávame celkovú časovú zložitosť $O(n^2 \log n)$.

Zložitejšie vzorové riešenie

Vráťme sa späť ku kubickému riešeniu, v ktorom sme pre každý roh postupne vyskúšali všetky možné dĺžky strany. Urobíme v ňom dve drobné zlepšenia, ktoré ale dokopy dramaticky zlepšia jeho časovú zložitosť.
Predstavme si, že už sme spracovali nejaké rohy a že najväčší doteraz nájdený dobrý štvorec mal veľkosť $x = 5$. Keď ideme skúšať nasledujúci roh, zjavne nemá zmysel skúšať preň dĺžku strany 1 až 5: takto veľké dobré štvorce sme už videli. Začneme preto skúšať až od $x + 1$.
No a nebudeme skúšať všetky väčšie dĺžky strany – ako sme si už uvedomili v predchádzajúcom riešení, akonáhle narazíme na zlý štvorec, vieme, že aj všetky väčšie už budú zlé, môžeme teda prestať skúšať aktuálny pravý dolný roh a posunúť sa na ďalší.
S týmito dvoma vylepšeniami dostaneme riešenie, ktorého časová zložitosť závisí od n už nie kubicky ale len kvadraticky. Môžeme si totiž všimnúť, že po každom teste štvorca buď zväčšíme x (našli sme nové riešenie lepšie ako všetky doteraz), alebo sa posunieme na ďalší pravý dolný roh. No a za celý beh riešenia sa dokopy n^2-krát posunieme na ďalší roh a nanajvýš n-krát zväčšíme x.

Stručnejšie vzorové riešenie

V doteraz popísaných riešeniach nezáležalo na presnom poradí, v ktorom sme skúšali možnosti pre pravý dolný roh hľadaného štvorca. Teraz si ukážeme, že keď to budeme robiť tým úplne najviac priamočiarym spôsobom, vieme implementáciu predchádzajúceho riešenia ešte výrazne zjednodušiť.
V pseudokóde bude celé výsledné riešenie veľmi stručné:

```
x = 0
pre každý riadok r od 0 po n-1:
    pre každý stĺpec s od 0 po n-1:
        ak je štvorec s pravým dolným rohom (r,s) a stranou x+1 dobrý:
            x = x+1
```

Tvrdíme teda, že pre každý roh nám stačí vyskúšať len jednu dĺžku strany: o jedna väčšiu ako doterajšie optimum.
Správnosť algoritmu dokážeme sporom. Nech existuje vstup, na ktorom náš algoritmus nenájde optimálne riešenie. Toto zjavne musí nastať tak, že pre nejaký roh je aj štvorec veľkosti $x + 2$ dobrý, ale my sa naň nepozrieme. Nech roh (r, s) je prvý roh počas behu programu, v ktorom toto nastalo. Potom by ale nutne musel byť dobrý aj štvorec s rohom $(r - 1, s - 1)$ a stranou $x + 1$, keďže tento je podmnožinou nášho aktuálneho štvorca veľkosti $x + 2$. Lenže roh $(r - 1, s - 1)$ sme už spracovali skôr a vieme, že každý skôr spracovaný roh mal optimálny štvorec veľkosti nanajvýš rovnej súčasnému x, čo je hľadaný spor.
Tento algoritmus má zjavne časovú zložitosť kvadratickú od n.

Listing programu (C++)

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

int main() {
    // nacitame vstup
    int N, P;
    cin >> N >> P;
    vector< vector<int> > kolac(N, vector<int>(N));
    for (int r=0; r<N; ++r) {
        string S;
        cin >> S;
        for (int s=0; s<N; ++s) kolac[r][s] = int( S[s] == 'P' );
    }

    // spocitame prefixove sucty
    vector< vector<int> > psum(N+1, vector<int>(N+1, 0));
    for (int r=0; r<N; ++r) for (int s=0; s<N; ++s)
        psum[r+1][s+1] = kolac[r][s] + psum[r+1][s] + psum[r][s+1] - psum[r][s];

    // postupne prejdeme vsetky rohy
    int x = 0;
    for (int r=1; r<=N; ++r) for (int s=1; s<=N; ++s) {
        if (min(r,s) < x+1) continue; // skusany stvorec trci von zo vstupu
        int pavukov = psum[r][s] - psum[r-x-1][s] - psum[r][s-x-1] + psum[r-x-1][s-x-1];
        if (pavukov <= P) ++x;
    }
    cout << x << endl;
}
```

A-II-2 Recyklujeme

Štruktúra tohto vzorového riešenia: Najskôr si ukážeme, ako sa dá dostať k efektívnemu (a následne aj k optimálnemu) riešeniu úlohy bez trikov, len postupnou analýzou problému. Potom si ukážeme šikovnejšie riešenie, ktoré je jednoduchšie na implementáciu, ale vyžaduje šikovnejší pohľad na úlohu. No a na záver si ešte ukážeme protipríklad na jednu prirodzenú ale nesprávnu pažravú stratégiu.

Postupná analýza problému

Ak by sme mali len samé plasty, je riešenie jednoduché: robot postupne zájde ku každej fľaši, zodvihne ju a prinesie ju nazad. Zjavne vôbec nezáleží na poradí, v ktorom bude fľaše zbierať, celková vzdialenosť, ktorú prejde, a teda aj celkový čas budú vždy rovnaké. Vo zvyšku riešenia budeme riešiť ostatné situácie – teda predpokladať, že existuje aspoň jedna plechovka.

Ak máme aj nejaké plechovky, jedno možné (pozor, nie nutne optimálne) riešenie vyzerá nasledovne: postupne po jednej vyzbierame všetky fľaše, potom od koša na fľaše prejdeme k nejakej plechovke, zodvihneme ju, odnesieme ju do koša na hliník, a potom postupne chodíme odtiaľ pozbierať ostatné plechovky.
Rovnako ako vyššie, ani u tohto typu riešenia zjavne nezáleží ani na poradí zberu fliaš na začiatku, ani na poradí zberu plechoviek na konci, celkový čas je vždy rovný súčtu časov potrebných na zdvihnutie a vyhodenie jednotlivých odpadkov. Teda, až na jednu výnimku – jedno rozhodnutie, kde vieme ovplyvniť celkový čas. Tým je voľba plechovky, ktorú zdvihneme ako prvú (cestou od jedného koša k druhému).
Akú celkovú vzdialenosť prejde robot pri takomto riešení? Pre každú fľašu aj každú plechovku inú od prvej prejde

dvakrát jej vzdialenosť od príslušného koša. Ale pre prvú plechovku namiesto dvojnásobku jej vzdialenosti od koša na hliník prejde robot raz jej vzdialenosť od koša na plasty a raz jej vzdialenosť od koša na hliník.
Ak by sme chceli nájsť najlepšie riešenie *tohto konkrétneho typu*, mali by sme vybrať tú plechovku, pre ktorú najviac ušetríme – teda tú plechovku, pre ktorú je rozdiel (jej vzdialenosť od koša na plasty) mínus (jej vzdialenosť od koša na hliník) najmenší možný.

Pre poriadok si teraz ešte explicitne uveďme jedno pozorovanie: Stačí hľadať riešenie, v ktorom prázdny robot vždy ide priamo k nejakému odpadu a robot nesúci odpad ide priamo k správnemu košu.
Toto priamočiaro vyplýva z trojuholníkovej nerovnosti. Napr. nemá zmysel uvažovať riešenie, v ktorom robot niekedy „naprázdno“ prejde od koša na plasty ku košu na hliník, a potom odtiaľ k nejakej plechovke, lebo keby namiesto toho šiel priamo k tej plechovke, dostane sa do tej istej situácie, a to buď v tom istom čase, alebo ešte skôr.
(A dokonca by vyššie uvedené tvrdenie platilo aj vtedy, ak by sme robotovi povolili nesený odpad položiť hocikde na lúke a neskôr sa poň vrátiť. Aj pri takejto všeobecnejšej verzii našej úlohy vieme zdôvodniť, že keď už raz odpad nesieme, nič nepokazíme, keď ho zoberieme priamo do správneho koša. Formálny dôkaz sa dá spraviť napríklad tak, že sa pozrieme na posledný odpad prinesený do koša a ukážeme, že pohyby robota počas riešenia vieme bez zmeny celkového času preusporiadať tak, aby na konci riešenia prišiel po tento odpad a potom ho odniesol celú cestu až do koša.)

Vráťme sa späť k vyššie uvedenému riešeniu. Je vždy optimálne najskôr pozbierať všetky fľaše a potom všetky plechovky? Alebo sa nám niekedy oplatí robiť to na striedačku?
Pozrime sa napríklad na nasledovnú situáciu: Všetko je na jednej priamke. Na súradnici 0 je kôš na plasty, na súradnici 1 je n fliaš, na súradnici 10 je kôš na hliník a na súradnici 9 je n plechoviek. Tu je naozaj optimálne najskôr pozbierať všetky fľaše (každú vyhodíme za 2 sekundy) a až potom odísť postupne vyhadzovať plechovky. Tu teda je optimálne riešiť každý typ odpadu zvlášť.
Lenže všetko sa zmení, ak presunieme našich n fliaš na súradnicu 9 a naopak n plechoviek dáme na súradnicu 1. Ak by sme aj teraz zbierali najskôr fľaše a potom plechovky, tak na každú fľašu aj na každú plechovku okrem prvej budeme potrebovať 18 sekúnd. Omnoho lepšie je tentokrát chodiť medzi košmi tam a späť, pričom cestou jedným smerom vždy vezmeme plechovku a druhým smerom fľašu. Takto na každý odpad pripadne len 10 sekúnd.
Ak chceme nájsť optimálne riešenie, potrebujeme teda nejak vedieť zistiť, kedy sa oplatí ostať pri jednom koši a kedy naopak prechádzať medzi košmi tam a späť.

Budeme hovoriť, že pri zbere odpadkov nastala *zmena*, ak ideme zdvihnúť odpad opačného typu ako ten, ktorý sme práve vyhodili.
Na začiatku tohto vzorového riešenia sme si ukázali, ako nájsť optimálne riešenie s práve jednou zmenou. Skúsme teraz túto úvahu zovšeobecniť.
Pozrime sa, ako vyzerá ľubovoľné riešenie s práve dvoma zmenami:

1. Najskôr zbierame nejaké fľaše a nosíme ich do koša.
2. Potom prejdeme k druhému košu, cestou k nemu zoberieme a vyhodíme jednu plechovku.
3. Potom zbierame nejaké plechovky (pričom ich už postupne vyzbierame všetky).
4. Potom prejdeme späť k prvému košu, cestou k nemu zoberieme a vyhodíme jednu fľašu.
5. Dozbierame všetky fľaše, ktoré sme nevyzbierali v bode 1.

Je zjavné, že na celkový čas riešenia nemá vplyv to, ktoré fľaše vyzbierame v kroku 1 a ktoré až v kroku 5 – môžeme teda bez ujmy na všeobecnosti predpokladať, že si všetky necháme až na krok 5.
Celkový čas takéhoto riešenia je jednoznačne určený dvoma voľbami: tým, ktorú plechovku zobrať v kroku 2, a tým, ktorú fľašu zobrať v kroku 4.
No a obe tieto rozhodnutia vieme spraviť pažravo: najlepšiu plechovku vyberieme podľa presne toho istého kritéria ako pri jednej zmene, a najlepšiu fľašu zase presne naopak.
No a teraz by už mohlo byť jasné, ako túto úvahu zovšeobecniť ďalej. Najlepšie riešenie s práve z zmenami vyzerá nasledovne: Počas riešenia presne z-krát prejdeme k opačnému košu a cestou zoberieme a vyhodíme

jeden odpad príslušného typu. Keď sme poslednýkrát pri niektorom koši, dozbierame všetky ostatné odpadky, ktoré doň patria.
No a vybrať, ktoré konkrétne odpadky sa oplatí zbierať počas jednotlivých zmien, vieme naďalej pažravo. Ak napríklad 3× budeme robiť zmenu od plastov k hliníku, tak vieme, že najviac ušetríme vtedy, keď pri nich zoberieme prvé tri plechovky v poradí podľa toho, koľko pri ktorej ušetríme.

Zhrnutie vzorového riešenia

Vzorové riešenie teda bude vyzerať nasledovne:
Pre každú fľašu si vypočítame, koľko času ušetríme, ak ju zoberieme vyhodiť počas zmeny – teda čas číselne rovný rozdielu jej vzdialeností od oboch košov. Všetky fľaše si usporiadame podľa tohto kľúča od najlepšej po najhoršiu.
(Všimnite si, že pre niektoré fľaše môže byť táto hodnota aj záporná – teda vyhodiť ju počas zmeny trvá dlhšie ako mimo nej. Tieto fľaše skončia na konci nášho usporiadaného zoznamu.)
Pre plechovky potom spravíme presne to isté, len rozdiel počítame s opačným znamienkom.
Následne postupne vyskúšame všetky prípustné možnosti pre z, teda počet zmien počas riešenia. Pre každé z vyššie popísanou úvahou nájdeme čas optimálneho riešenia s príslušným počtom zmien. No a spomedzi týchto kandidátov už len stačí vybrať najlepšieho – tým musí byť hľadané globálne optimum.

Na výrobu usporiadaných zoznamov fliaš a plechoviek potrebujeme čas $(f \log f + p \log p)$. Zvyšok riešenia už vieme implementovať v lineárnom čase, ktorý je oproti tomu zanedbateľný. (Najľahším spôsobom, ako to spraviť, je postupne zväčšovať z a striedavo pridávať ďalšiu plechovku a ďalšiu fľašu medzi tie, ktoré sú vyhodené počas zmien.)

Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;
typedef struct { int x, y; } point;

double vzdialenost(const point &A, const point &B) { return hypot(B.x-A.x, B.y-A.y); }

int main() {
    cout << fixed << setprecision(10);

    point kos_flase, kos_plech;
    cin >> kos_flase.x >> kos_flase.y >> kos_plech.x >> kos_plech.y;

    int f; cin >> f;
    vector<point> flase(f);
    for (int i=0; i<f; ++i) cin >> flase[i].x >> flase[i].y;
    int p; cin >> p;
    vector<point> plech(p);
    for (int i=0; i<p; ++i) cin >> plech[i].x >> plech[i].y;

    double odpoved = 0;
    for (const auto &f : flase) odpoved += 2*vzdialenost(f, kos_flase);
    if (p == 0) { cout << odpoved << endl; return 0; }
    for (const auto &p : plech) odpoved += 2*vzdialenost(p, kos_plech);

    vector< vector<double> > zlepsi(2, vector<double>(1, -1e9));
    for (const auto &f : flase) zlepsi[0].push_back( vzdialenost(f, kos_flase) - vzdialenost(f, kos_plech) );
    for (const auto &p : plech) zlepsi[1].push_back( vzdialenost(p, kos_plech) - vzdialenost(p, kos_flase) );
    for (int i=0; i<2; ++i) sort( zlepsi[i].begin(), zlepsi[i].end() );

    odpoved -= zlepsi[1].back();
    zlepsi[1].pop_back();
    int kde = 0;
    while (true) {
        if (zlepsi[kde].back() <= 0 && zlepsi[kde].back() + zlepsi[!kde].back() <= 0) break;
        odpoved -= zlepsi[kde].back();
        zlepsi[kde].pop_back();
        kde = !kde;
    }
    cout << odpoved << endl;
}
```

Myšlienková chyba a ako ju opraviť

Na jednom mieste sa v našom vzorovom riešení dá ľahko spraviť logická chyba. Zjavne, kým pri ďalšej zmene

dostávame kladnú úsporu tak sa ju oplatí pridať. Mohlo by sa preto zdať, že optimálny počet zmien vieme určiť pažravo: pri skúšaní všetkých možností môžeme prestať akonáhle už pridanie ďalšej zmeny nezlepší celkový čas riešenia. Lenže toto nie je úplne pravda. Niekedy sa totiž oplatí spraviť aj zmenu, pri ktorej si pohoršíme, lebo nám to následne umožní spraviť ďalšiu zmenu *idúc opačným smerom*, pri ktorej znova ušetríme, a to viac ako sme v predchádzajúcom kroku stratili.
Táto úvaha však nie je úplne zlá, lebo práve sme si popísali *jediný* protipríklad. Akonáhle v nejakom okamihu ani nasledujúca zmena ani nasledujúce dve zmeny dokopy nezlepšia celkový čas, môžeme prestať – zmeny po nich nasledujúce už nebudú lepšie od týchto, a teda sme práve našli optimálne riešenie.

O krok ďalej za vzorové riešenie

Vzorové riešenie, ktoré stačilo na plný počet bodov, v skutočnosti nie je úplne optimálne. Súťažnú úlohu vieme riešiť ešte o chlp efektívnejšie – v lineárnom čase, teda aj bez toho, aby sme museli odpadky usporiadať. Hlavná myšlienka riešenia je, že šikovne binárne vyhľadáme miesto, po ktoré sa oplatí pridávať zmeny.
Využijeme na to funkciu kth-element: keď si zvolíme jeden konkrétny index k do poľa, v čase lineárnom od jeho dĺžky ho vieme preusporiadať tak, aby na indexe k skončil ten prvok, ktorý tam je v usporiadanom poradí. A navyše vieme tiež dosiahnuť, aby naľavo od neho boli (ešte neusporiadané) práve tie prvky, ktoré by boli naľavo od neho v usporiadanom poli. Celé toto je zovšeobecnením algoritmu na nájdenie mediánu poľa v zaručene lineárnom čase.

Ak fliaš a plechoviek nie je zhruba rovnako veľa, tie nadbytočné z typu, ktorého je priveľa, vieme nájsť a zahodiť jedným volaním kth-element. (Zahodíme samozrejme tie, ktoré najmenej zlepšia čas riešenia pri zmene.)
Dvoma postupnými volaniami kth-element (prvé na celé pole, druhé na pol poľa) si pole s fľašami v lineárnom čase preusporiadame tak, aby jeho stredné dva prvky boli na správnom mieste. To isté spravíme aj s plechovkami. V strede oboch polí sme takto dostali štyri prvky, ktoré by vzorové riešenie postupne tesne po sebe pridávalo ako ďalšie zmeny. Ľahko skontrolujeme, či tieto zmeny ešte zlepšujú riešenie, a teda či sa pažravá verzia vzorového riešenia dostane až sem. Ak nie, môžeme zahodiť pravé polovice oboch polí. Ak áno, môžeme zarátať všetky zmeny zodpovedajúce ľavým poloviciam polí (všimnite si, že ich nepotrebujeme usporiadať, nezáleží totiž na ich poradí) a potom ich zahodiť a pokračovať ďalej len s pravými polovicami polí.
Bez ohľadu na to, ako dopadol test v strede, vieme, že sme strávili čas lineárny od aktuálnej dĺžky polí a potom sme ju zmenšili na polovicu. Celková časová zložitosť toho, keď budeme opakovať celý postup až kým nenájdeme presnú hranicu, po ktorú sa robiť zmeny oplatí, bude teda priamo úmerná $n + (n/2) + (n/4) + \cdots = 2n$, čiže lineárna.
Pri implementácii tohto riešenia vieme šikovne znížiť počet prípadov, ktoré treba rozobrať, tak, že naprogramujeme hľadanie najlepšieho *párneho* počtu zmien a potom ho spustíme dvakrát. Pred druhým spustením ručne spravíme najlepšiu prvú zmenu a odstránime ju z ich zoznamu, čím nám následné zavolanie tej istej funkcie nájde najlepší nepárny počet zmien.

Iné, šikovnejšie vzorové riešenie

Namiesto postupných optimalizácií jednotlivých rozhodnutí sa pozrime na celý proces naraz.
Prvé pozorovanie je, že každý z odpadkov má jednoznačne určené, kam patrí. Preto celková vzdialenosť *od odpadkov ku košom* je v každom riešení rovnaká – každý kus odpadu zanesieme, kam patrí. Optimálne je teda to riešenie, ktoré má najmenšiu celkovú vzdialenosť zvyšku – teda chceme dokopy prejsť čo najmenej idúc *od košov ku odpadkom*.

Keďže máme f fliaš, tak počas ľubovoľného riešenia sa nám musí presne f-krát stať, že sa s nejakou fľašou vrátime ku košu na ne. No a keďže máme p plechoviek, tak sa počas ľubovoľného riešenia p-krát dostaneme ku košu na plechovky.
To ale znamená, že v úplne každom riešení budeme $(f + 1)$-krát stáť pri koši na fľaše (plus jedna, lebo sme tam aj na začiatku) a p-krát stáť pri koši na plechovky. A s výnimkou úplného konca celého zberu sa v každej z týchto situácií vyberieme po nejaký ďalší kus odpadu.
Stačí nám teraz rozobrať dve možnosti: buď naše riešenie skončí pri koši na fľaše, alebo pri koši na plechovky. Obe možnosti sa budú riešiť rovnako. Nech trebárs skončíme pri koši na plechovky. Potom vieme, že počas

ľubovoľného riešenia tohto typu pôjdeme presne $(f + 1)$-krát od koša na fľaše ku nejakému odpadu a presne $(p - 1)$-krát pôjdeme od koša na plechovky ku nejakému odpadu.
No a keď si uvedomíme toto, už je jasné, ako toto spraviť najefektívnejšie. Keby sme si všetky odpadky usporiadali podľa toho, o koľko sú bližšie ku košu na plechovky ako ku košu na fľaše (teda podľa rozdielu týchto dvoch vzdialeností), tak je zjavne optimálne zobrať najlepších $p-1$ z nich, k tým ísť od koša na plechovky a ku zvyšku ísť od koša na fľaše. Takto zjavne dostaneme najmenší možný súčet vzdialeností od košov ku odpadkom.
Hotovo? Takmer, až na jeden detail: ešte potrebujeme zabezpečiť, aby sme nájdený scenár vedeli aj fyzicky zrealizovať – teda zabezpečiť, že existuje poradie, v ktorom vieme fyzicky vykonať nájdenú množinu presunov. To ide úplne vždy (rozmyslite si prečo), až na jeden jediný prípad: ak nám vyjde, že optimálne je ísť od koša na fľaše ku každej fľaši a od koša na plechovky ku každej plechovke. Pre toto riešenie by sme totiž vyzbierali všetky fľaše a následne by sme smutne pozerali, že nevieme ísť od koša na plechovky ku plechovke, lebo sme sa k nemu nikdy nedostali. Tento jeden špeciálny prípad ktorý nie je platným riešením treba teda ešte ošetriť a namiesto neho sa pozrieť na druhú najlepšiu možnosť – namiesto najhoršej z plechoviek zoberieme medzi odpadky zbierané od druhého koša najlepšiu z fliaš.

Toto riešenie vieme ľahko vylepšiť tak, aby nepoužívalo triedenie a bežalo v lineárnom čase. Správny počet najlepších odpadkov totiž vieme nájsť vhodným zavolaním už vyššie spomenutej funkcie kth-element.

Nesprávne pažravé stratégie

Aj keď je úloha riešiteľná pažravo, zďaleka to neznamená, že *každá* pažravá stratégia povedie ku optimálnemu riešeniu. Je preto užitočné zvyknúť si namiesto tipovania rozmýšľať okrem stratégie rovno aj nad dôkazom jej správnosti.

Intuitívne ale nesprávne riešenie: Keď som pri niektorom koši a rozhodujem sa, ktorý odpad zobrať ako nasledujúci, vyberiem si ten, ktorý je najbližšie k mojej aktuálnej polohe.
Toto je úplne ukážková pažravá stratégia: výberom zabezpečíme, že *tento konkrétny presun* ku odpadu bude najkratší možný. Ale tým sa nesmieme nechať uspokojiť a musíme si položiť otázky: „No dobre, ale naozaj vedie takáto postupnosť akcií k optimálnemu *súčtu* vzdialeností? Nemôže sa mi stať, že ak začnem iným prvým krokom, neskôr budem vedieť spraviť iné lepšie a dostanem lepší celkový súčet? Ak nie, prečo nie?“
A ak sa nám tento dôkaz nijak nedarí spraviť, je to možno preto, že dokazované tvrdenie ani neplatí. A tak je tomu aj v tomto prípade.
Uvažujme opäť všetko na priamke. Kôš na fľaše nech je na súradnici 0, kôš na plechovky na súradnici 10, a okrem toho ešte máme plechovku na súradnici 1 a fľašu na súradnici 2. Vyššie popísané pažravé riešenie pôjde po plechovku, s tou do jej koša, odtiaľ po fľašu a tú naspäť – dokopy teda prejde 20. Optimálne je zjavne ísť po fľašu, priniesť ju späť, a až potom ísť vyhodiť plechovku – takto prejdeme len 14.

A-II-3 Decentralizácia

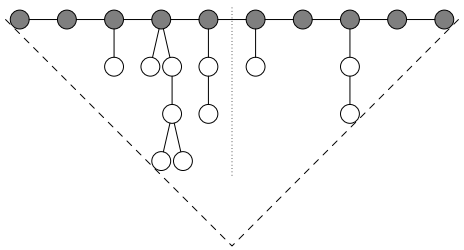
Preložme si zadanie do jazyka teórie grafov. Trasy karaván predstavujú hrany stromu. Priemer stromu sme si definovali ako dĺžku najdlhšej cesty v ňom. Odstránením vrcholu zo stromu vznikne les – teda niekoľko menších disjunktných stromov. Rozsah lesa sme si definovali ako najväčší z priemerov stromov, ktoré ho tvoria. Našou úlohou je nájsť taký vrchol, ktorého odstránením vznikne les s najmenším možným rozsahom.

Riešenie hrubou silou

Vzdialenosti z jedného vrcholu do všetkých ostatných vieme na strome spočítať v lineárnom čase ľubovoľným prehľadávaním (napr. do šírky alebo do hĺbky). Ak to postupne spravíme pre každý vrchol, dostaneme v kvadratickom čase celú tabuľku vzdialeností medzi vrcholmi. Súťažnú úlohu vieme následne vyriešiť v kubickom čase: Postupne vyskúšame všetky možnosti pre hlavnú oázu. Pre každú z nich rozdelíme vstupný strom na okresy, a potom každému okresu zistíme priemer v čase kvadratickom od jeho počtu vrcholov.

Priemer stromu v lineárnom čase – trikové riešenie

Označme si x ľubovoľný vrchol stromu. Spočítame vzdialenosti z x a označme y vrchol, ktorý je najďalej od x. Rovnako nájdime vrchol z, ktorý je najďalej od y. Potom dvojica (y, z) je vždy najvzdialenejšou dvojicou vrcholov v celom strome – ich vzdialenosť je teda hľadaným priemerom.
Pomocou tohto algoritmu vieme teda priemer ľubovoľného stromu vypočítať v čase priamo úmernom jeho počtu vrcholov, a teda súťažnú úlohu vyriešiť s časovou zložitosťou $O(n^2)$.
Hlavnú myšlienku dôkazu správnosti tohto trikového algoritmu si znázorníme graficky. Predstavme si, že niekto nakreslil celý strom tak, že jeho najdlhšia cesta (resp. ľubovoľná jedna z nich, ak ich je viac) vedie vodorovne a všetky ostatné hrany stromu vedú dodola.



Keďže vodorovná cesta je najdlhšia, všetky ostatné vrcholy musia ležať v trojuholníku naznačenom čiarkovane – keby nejaký vrchol bol hlbšie, cesta z neho dohora a potom doprava alebo doľava by bola dlhšia od tej vodorovnej.
Potom ale ak ako x zvolíme ľubovoľný vrchol v ľavej polovici obrázku, bude y vrchol vpravo hore a z vľavo hore. Pre x v pravej polovici dostaneme to isté symetricky. Preto náš algoritmus naozaj vždy nájde najdlhšiu cestu v celom strome.
(Technický detail: Ak v pôvodnom strome existuje viac ako jedna najdlhšia cesta, môžeme mať aj viac možností, ktoré vrcholy dostaneme ako y a z ku konkrétnemu x. Ľahko však nahliadneme, že v takýchto prípadoch síce dostaneme inú cestu $y - z$ ako tú, čo sme si nakreslili vodorovne, bude však mať rovnakú dĺžku.)

Priemer stromu v lineárnom čase – dynamické programovanie

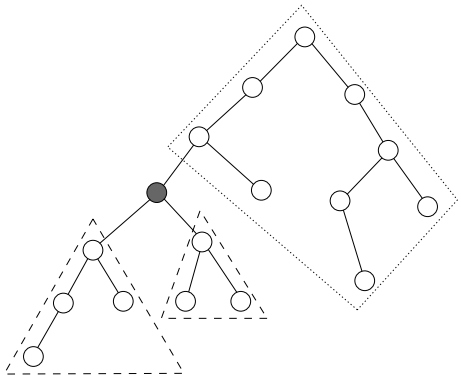
Ak chceme súťažnú úlohu vyriešiť v lepšom ako kvadratickom čase, už si nemôžeme dovoliť začať tým, že skúšame všetky možnosti pre hlavnú oázu. Budeme si musieť najskôr predpočítať nejaké štrukturálne údaje o strome, aby sme potom po každom výbere hlavnej oázy vedeli efektívne povedať, aké priemery majú okresy, ktoré vzniknú.
Na to sa nám oplatí sa najskôr zamyslieť nad iným algoritmom, ktorý nám tiež umožní zistiť priemer stromu v lineárnom čase. Výhodou tohto druhého prístupu bude, že pri ňom nielen nájdeme celkovo najdlhšiu cestu, ale tiež sa toho viac dozvieme o podstromoch nášho stromu.
Začneme tým, že si náš strom zakoreníme – zvolíme si ľubovoľný jeho vrchol ako koreň a zavesíme ho zaň. Všetky hrany takto získali orientáciu: dohora je ku koreňu, dodola je od koreňa. Najbližší vrchol smerom dohora voláme otec a najbližšie vrcholy smerom dodola sú synovia vrcholu.
Predstavme si, že v zakorenenom strome ideme po úplne ľubovoľnej ceste. (Pripomíname, že na ceste sa žiaden vrchol neopakuje.) Tvrdíme, že to nutne vyzerá tak, že najskôr prejdeme po nejakých (možno nula) hranách smerom hore a potom po zvyšných (opäť možno nula) hranách smerom dole. Prečo? Lebo akonáhle sa prvýkrát pohneme hranou dodola, už nikdy sa nemôžeme pohnúť dohora – znamenalo by to návrat po hrane, ktorou sme práve prišli.
Každú cestu $a - b$ po zakorenenom strome môžeme teda jednoznačne rozdeliť na dve časti: $a - c$ idúca dohora a $c - b$ idúca dodola. (Všimnite si, že vrchol c, kde sa „otáčame“, je najbližším spoločným predkom vrcholov a a b.) Inými slovami, každá cesta má práve jeden najvyšší vrchol c a vieme ju vyrobiť tak, že zlepíme dve (možno prázdne) cesty vedúce z c dodola. (Pričom ak sú obe neprázdne, tak musia začínať krokom do rôznych vrcholov.)

Tieto myšlienky nás už pomerne ľahko privedú k algoritmu na nájdenie najdlhšej cesty v strome. Začínajúc z koreňa prehľadáme celý strom do hĺbky. Keď opúšťame vrchol i (a teda už sme spracovali všetky vrcholy pod ním), vypočítame preň dva údaje: dĺžku d_i najdlhšej cesty začínajúcej vo vrchole i a idúcej dodola; dĺžku p_i najdlhšej cesty, ktorá *prechádza* vrcholom i a tento vrchol je jej najvyšší bod.

Pre každého syna j vrcholu i vieme nájsť najdlhšiu cestu, ktorá ide z i cez j dodola. Táto pozostáva z hrany z i do j a následne najdlhšej cesty z j dodola – ktorej dĺžku d_j sme už spočítali. Pre každé j takto dostávame jedného kandidáta na najdlhšiu cestu z i dodola. Hodnota d_i je zjavne rovná najväčšiemu z týchto kandidátov a hodnota p_i je súčtom najväčších dvoch. (Rozmyslite si špeciálne prípady, keď má vrchol synov 0 alebo 1.)
Hodnoty d_i a p_i vieme pre každý konkrétny vrchol i spočítať v čase priamo úmernom počtu hrán, ktoré z neho vedú dodola. Strom má dokopy len $n-1$ hrán, a preto ho celý dokopy spracujeme v lineárnom čase. Najväčšia z hodnôt p je potom hľadanou odpoveďou.

Príprava na vzorové riešenia

Do vyššie popísaného algoritmu vieme ešte ľahko doplniť výpočet jednej ďalšej hodnoty m_i: dĺžky najdlhšej cesty, ktorá je celá v podstrome s koreňom i. Túto v každom vrchole spočítame ako poslednú – platí, že m_i je maximom z hodnoty p_i a hodnôt m_j v synoch vrcholu i. Túto hodnotu budeme využívať v oboch vzorových riešeniach.
Ukážme si na obrázku, čo presne sa stane, keď nejaký vrchol i zo stromu odstránime: Pre každého syna vrcholu i dostaneme jeden okres tvorený celým podstromom, ktorého koreňom je daný syn. A navyše (ak odstránený vrchol i nie je koreňom celého stromu) dostaneme jeden *špeciálny* okres, ktorý obsahuje otca vrcholu i. V tomto okrese leží vždy aj koreň celého stromu. Na obrázku nižšie je špeciálny okres ohraničený bodkovanou čiarou a ostatné okresy čiarkovanou.



My potrebujeme pre každý vrchol zistiť priemery ním určených okresov. Okresy obsahujúce jeho synov vieme ľahko spracovať: v každom synovi j nám jeho hodnota m_j priamo udáva priemer jeho okresu. Už teda ostáva len jediné: efektívne spočítať pre každý vrchol (okrem koreňa) priemer jeho špeciálneho okresu.
Vo zvyšku tohto textu si postupne popíšeme dve lineárne riešenia súťažnej úlohy. To prvé bude využívať jedno pozorovanie navyše, vďaka ktorému bude mať stručnejšiu a ľahšiu implementáciu. To druhé bude zase používať všeobecnejšiu techniku.

Vzorové riešenie s ľahšou implementáciou

Začneme tým, že si ľubovoľným spôsobom (vyššie sme si popísali dva možné) nájdeme jednu konkrétnu najdlhšiu cestu $y-z$ v našom strome. Teraz spravíme nasledovné pozorovanie: Ak bude hlavná oáza kdekoľvek mimo cesty $y-z$, bude celá táto cesta ležať v jednom z okresov. Následne ale bude priemer tohto okresu rovný dĺžke cesty $y-z$, a teda bude mať rovnakú hodnotu aj rozsah celého príslušného rozdelenia. Potrebujeme už preto vyskúšať ako hlavnú oázu len tie, ktoré ležia na ceste $y-z$.
Všimnime si, čo sa stane, keď spustíme vyššie popísané prehľadávanie do hĺbky, pričom začneme z vrcholu y. Pre každý vrchol cesty $y-z$ si vypočítame priemery všetkých jeho okresov okrem špeciálneho – a ten vždy zodpovedá hrane dohora, čiže hrane ležiacej na ceste $y-z$.
Teraz spustíme ten istý algoritmus, ale z vrcholu z. Všetky hrany cesty $y-z$ sú pri tomto druhom prechode orientované opačným smerom, a teda pre vrcholy na tejto ceste platí, že každý ich okres, ktorý bol v prvom prehľadávaní špeciálny, je v tomto druhom obyčajný. To ale znamená, že v tomto prechode sa pre každý vrchol cesty $y-z$ dozvieme priemer toho okresu, ktorý nám po prvom prechode chýbal. Tým už máme všetko potrebné na spočítanie rozsahu rozdelenia Absurdistanu pre každý vrchol cesty $y-z$.

To isté inými slovami: Majme na ceste z vrcholu y do vrcholu z tri po sebe idúce vrcholy a, b, c. Potom pre vrchol b:

- Priemer okresu, do ktorého sa z b ide hranou do c, dostaneme ako hodnotu m_c počas prehľadávania z vrcholu y.
- Priemer okresu, do ktorého sa z b ide hranou do a, dostaneme ako hodnotu m_a počas prehľadávania z vrcholu z.
- Priemer okresu, do ktorého sa z b ide hranou do iného vrcholu x, dostaneme ako hodnotu m_x počas oboch prehľadávaní – v oboch z nich bude x synom b.

Dokopy len konštantne veľa ráz prejdeme zadaný strom, a keďže každý prechod vieme spraviť v lineárnom čase, má toto riešenie celkovú časovú zložitosť $O(n)$.

Listing programu (C++)

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

struct hrana { int ciel, dlzka; };
int N;
vector< vector<hrana> > graf;
vector<int> D, P, M, H, O;

void load() {
    cin >> N;
    graf.clear(); graf.resize(N);
    for (int m=0; m<N-1; ++m) {
        int x, y, d;
        cin >> x >> y >> d;
        graf[x].push_back( {y,d} );
        graf[y].push_back( {x,d} );
    }
}

void init_dfs() {
    for (auto pole : {&D,&P,&M,&H,&O}) { pole->clear(); pole->resize(N,0); }
}

void dfs(int v, int otec=-1) {
    O[v] = otec;
    vector<int> dodola;
    for (auto h : graf[v]) if (h.ciel != otec) {
        H[h.ciel] = H[v] + h.dlzka;
        dfs(h.ciel, v);
        dodola.push_back( D[h.ciel] + h.dlzka );
    }
    int deti = dodola.size();
    if (deti == 1) D[v] = P[v] = dodola[0];
    if (deti >= 2) {
        nth_element(dodola.begin(), dodola.begin()+(deti-2), dodola.end());
        D[v] = dodola[deti-1];
        P[v] = dodola[deti-1] + dodola[deti-2];
    }
    M[v] = P[v];
    for (auto h : graf[v]) if (h.ciel != otec) M[v] = max( M[v], M[h.ciel] );
}

int main() {
    load();
    // zacneme hocikde, najdeme y = najvzdialenejsi vrchol
    init_dfs();
    dfs(0);
    int y = max_element( H.begin(), H.end() ) - H.begin();
    // zacneme z y, pre kazdy vrchol spocitame rozsah jeho obycajnych okresov
    init_dfs();
    dfs(y);
    vector<int> rozsah(N,0);
    for (int i=0; i<N; ++i) for (auto h : graf[i]) if (h.ciel != O[i]) rozsah[i] = max( rozsah[i], M[h.ciel] );
    // najdeme z = najvzdialenejsi vrchol od y, oznacime si cestu y-z
    int z = max_element( H.begin(), H.end() ) - H.begin();
    int odpoved = H[z];
    vector<bool> na_ceste(N, false);
    for (int kde=z; kde!=-1; kde=O[kde]) na_ceste[kde] = true;
    // zacneme zo z, potom opat pozrieme na obycajne okresy
    init_dfs();
    dfs(z);
    for (int i=0; i<N; ++i) for (auto h : graf[i]) if (h.ciel != O[i]) rozsah[i] = max( rozsah[i], M[h.ciel] );
    for (int i=0; i<N; ++i) if (na_ceste[i]) odpoved = min( odpoved, rozsah[i] );
    cout << odpoved << endl;
}
```

Alternatívne vzorové riešenie – všeobecnejšia technika

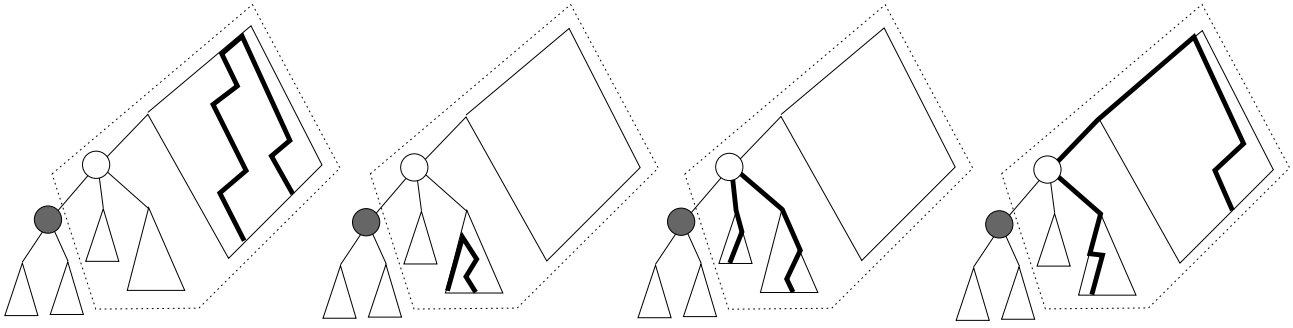
V tejto časti si ukážeme, že vieme v lineárnom čase spočítať priemery úplne všetkých možných okresov. Toto riešenie si ukážeme preto, že jeho technika sa následne dá zovšeobecniť na riešenie iných podobných úloh.
Základ tejto techniky je, že špeciálny okres pre daný vrchol w sa skladá zo špeciálneho okresu pre jeho otca (nazvime ho v), samotného vrcholu v a potom obyčajných okresov s koreňmi v ostatných synoch vrcholu v (všetkých okrem w samotného). Tento nový špeciálny okres teda vznikne tak, že vrcholom v spojíme dokopy niekoľko okresov, ktoré sme už mohli mať skôr spracované. Potrebujeme si preto rozmyslieť, čo všetko o nich potrebujeme mať predpočítané, aby sme vedeli toto spojenie spraviť efektívne.

Toto riešenie budeme implementovať v dvoch prechodoch. V prvom si strom zakoreníme v ľubovoľnom vrchole a spustíme z neho vyššie popísané prehľadávanie do hĺbky. To nám vypočíta hodnoty d, p a m. Teraz teda pre každý vrchol poznáme priemery jeho obyčajných okresov a tiež nejaké informácie o tvare jeho vlastného podstromu.
V druhom prechode si potom budeme pre každý vrchol v počítať dva nové údaje. Prvým bude ten, čo chceme: priemer jemu zodpovedajúceho špeciálneho okresu. Druhým bude pomocný údaj: dĺžka h_v najdlhšej cesty, ktorá začína v tomto vrchole a prvý krok má smerom dohora.
Strom budeme prehľadávať z toho istého koreňa ako v prvom prechode. Samotný koreň spracujeme ľahko: Nemá špeciálny okres ani cestu začínajúcu krokom hore, obe zodpovedajúce hodnoty mu preto ešte pred prehľadávaním inicializujeme na $-\infty$. Následne celý strom prehľadáme, *ale tentokrát do šírky*. Vždy, keď prídeme do nejakého vrcholu, spracujeme *naraz* všetkých jeho synov, vypočítame pre nich všetko potrebné a potom ich zaradíme do fronty na spracovanie. Takto dosiahneme, aby sme aj v tomto prechode každý vrchol spracovali v čase priamo úmernom počtu jeho synov.

Majme v strome nejaký vrchol v, ktorý ideme práve spracovať. Nech w je niektorý syn vrcholu v. Ako sme už uviedli vyššie, špeciálny okres pre w je tvorený vrcholom v, všetkými vrcholmi, do ktorých sa z v ide dohora (čiže špeciálnym okresom pre vrchol v) a všetkými vrcholmi, do ktorých sa z v ide cez synov iných ako w.
Pre najdlhšiu cestu v špeciálnom okrese pre vrchol w preto máme štyri možnosti:

1. buď celá leží v pôvodnom špeciálnom okrese pre v
2. alebo celá leží v podstrome niektorého iného syna vrcholu v
3. alebo má najvyšší bod vo v a ide dodola cez iné vrcholy ako w
4. alebo prechádza cez v dohora.

Všetky štyri typy sú znázornené (zľava doprava) na obrázku nižšie. Sivý vrchol je w, jeho špeciálny okres je ohraničený bodkovane. Biely vrchol je v. Príklad najdlhšej cesty je hrubou čiarou.



Teraz vieme postupne vykonať nasledovné výpočty:

1. Najdlhšia cesta prvého typu je najdlhšou cestou v špeciálnom okrese pre v.
2. Najdlhšia cesta druhého typu v podstrome pod konkrétnym synom x vrcholu v má dĺžku m_x.

Pre konkrétneho syna w vrcholu v teda platí, že najdlhšia cesta tohto typu v špeciálnom okrese pre w má dĺžku rovnú maximu z hodnôt m v ostatných synoch vrchola v. Toto vieme efektívne spočítať tak, že nájdeme dvoch synov v, ktorí majú najväčšiu hodnotu m. Potom pre každého syna vieme nájsť najlepšieho iného tak, že prezrieme len týchto dvoch kandidátov.

3. Pre každého syna w vrcholu v sme si už v prvom prechode vypočítali dĺžku d_w najdlhšej cesty z neho dodola. Teraz teda vieme pre každé w v konštantnom čase spočítať dĺžku najdlhšej cesty z v cez w dodola. Z týchto hodnôt si stačí ponechať *tri* najväčšie (a ku každej si pamätať jej prislúchajuce w). Potom vieme pre každé w nájsť v konštantnom čase dve najdlhšie cesty ktoré idú dodola cez synov rôznych od w aj seba navzájom. Tieto dokopy tvoria najdlhšiu cestu tretieho typu.

4. Najdlhšiu cestu štvrtého typu vieme rozdeliť na dve časti: cestu idúcu z v dohora a dodola. Dĺžku najdlhšej cesty z v dohora sme si už predpočítali v skoršej iterácii tohto postupu. No a pre každé w dĺžku najdlhšej cesty dodola ale nie cez w určíme v konštantnom čase rovnako ako v predchádzajúcom kroku.

5. Na záver nám ešte ostáva si aj pre w dopočítať dĺžku najdlhšej cesty začínajúc z neho dohora. Tá začína hranou z w do v a následne máme dve možnosti: buď pokračuje najdlhšou možnou cestou dohora alebo najdlhšou možnou cestou dodola cez iného syna ako w. Obe tieto možnosti už poznáme: sú to práve tie dve cesty, z ktorých sme zložili odpoveď v predchádzajúcom odseku.

Ako sme sľubovali, ukázali sme, ako konkrétny vrchol spracovať v čase lineárnom od jeho počtu synov. Dokopy teda aj tento druhý prechod stromom pobeží v lineárnom čase. Po jeho skončení už v každom vrchole vieme zistiť jemu zodpovedajúci rozsah a teda vybrať najlepšiu hlavnú oázu.

A-II-4 Triedička II

Podúloha A (3 body): si priemerný?

Nech s je súčet n-prvkovej postupnosti na vstupe. Potom každý prvok x chceme porovnať s jej priemerom, teda s hodnotou s/n. To je to isté, ako porovnať hodnotu nx s hodnotou s. Takto preformulovanú operáciu navyše vieme pohodlne spraviť v celých číslach.
V domácom kole sme si už napísali program, ktorý v jadre 0 zistí hodnotu súčtu všetkých vstupov. Môžeme teda začať tým, že tento program spustíme. Zaň pridáme druhú fázu, v ktorej vypočítaný súčet rozkopírujeme z jadra 0 do všetkých ostatných. No a na záver si už len každé jadro zvlášť vynásobí svoj prvok počtom jadier a výsledok porovná so súčtom celej postupnosti.

Asi o polovicu stručnejšie riešenie dostaneme, keď namiesto dvoch fáz spravíme len jednu, počas ktorej si naraz všetky prvky počítajú súčet čoraz dlhších úsekov, až nakoniec dostanú súčet celého poľa. Ukážeme si jeden z mnohých podobných spôsobov, ako presne toto dosiahnuť.
Predpokladajme, že máme pole rozdelené na úseky dĺžky 2^k, pričom každý prvok pozná súčet svojho úseku. (Napr. pre $n = 16$ a $k = 2$ by sme mali pole tvaru `AAAABBBBCCCCDDDD` – každé písmeno predstavuje jeden prvok, prvky s rovnakým písmenom tvoria jeden úsek a pamätajú si jeho súčet.)
Teraz preusporiadame pole tak, že prvky z prvej polovice dáme na párne indexy a prvky z druhej polovice na nepárne. (V našom príklade dostaneme pole `ACACACACBDBDBDBD`.) Každý prvok teraz sčíta svoj doterajší súčet so súčtom správneho suseda (podľa parity nového indexu). Tým dostaneme nové pole, ktoré má rovnakú vlastnosť ako to, s ktorým sme začínali, len úseky už majú dĺžku 2^{k+1}.
Pre vstup s 2^{20} prvkami teda stačí začať zo vstupného poľa (každý prvok má svoj vlastný súčet, čo zodpovedá úseku dĺžky 2^k pre $k = 0$) a 20-krát zopakovať vyššie popísaný postup.

Ako program si ukážeme tento druhý prístup, keďže riešenie podúlohy B používa podobnú techniku ako to, ktoré si potrebuje rozkopírovať spočítaný súčet.

```
cislo:       id
sucet:       input 1
dva:         const 2
n:           const 1048576   # = 2^20
n_pol:       / n dva

################################################ blok, ktorý 20x zopakujeme
parita:      % cislo dva
lavy_sucet:  left sucet
pravy_sucet: right sucet
sused_sucet: if parita lavy_sucet pravy_sucet
sucet:       + sucet sused_sucet
som_velky:   >= cislo n_pol
cislo:       % cislo n_pol
cislo:       * cislo dva
cislo:       + cislo som_velky
             sort cislo
################################################ koniec bloku

vstup:       input 1
nkrat_vstup: * vstup n
nadpriemer:  > nkrat_vstup sucet
podpriemer:  < nkrat_vstup sucet
odpoved:     - nadpriemer podpriemer        # (1-0) pre nadpriemerné, (0-1) pre podpriemerné
```

Podúloha B (3 body): prefixové súčty

Už v riešeniach domáceho kola sme si načrtli, že pomocou triedičky vieme efektívne riešiť úlohy, ktoré na klasickom počítači vieme riešiť pomocou intervalového stromu. (Popis intervalových stromov: https://www.ksp.sk/kucharka/intervalovy_strom/.)
Ako vieme spočítať prefixové súčty na intervalovom strome? Zdola hore si spravíme súčtový intervalový strom: v každom vnútornom vrchole si spočítame súčet hodnôt v jeho dvoch synoch. Následne chceme, aby každý vrchol zistil súčet všetkých prvkov naľavo od jeho intervalu. Toto vieme spraviť tak, že prejdeme strom zhora dole. V koreni je táto hodnota rovná nule. Keď sme už spracovali nejaký vrchol v a vieme, že súčet prvkov naľavo od neho je s, tak pre ľavého syna vrcholu v je tento súčet tiež s a pre pravého syna potrebujeme ku s pridať súčet intervalu zodpovedajúceho ľavému synovi.
Tento algoritmus vieme implementovať aj na triedičke. Pri implementácii tohto algoritmu na triedičke si potrebujeme poradiť ešte s jedným technickým detailom: zatiaľ čo v intervalovom strome máme vstup v listoch a vnútorné vrcholy vytvoríme nové, na triedičke nevieme pridávať nové jadrá.
Na to si ale stačí uvedomiť, že si vieme v jednom jadre pamätať viac hodnôt – každé jadro si predsa pamätá celú svoju históriu výpočtov a ku každej z týchto hodnôt sa vieme vrátiť. Vieme si preto dokopy v našich jadrách vypočítať a zapamätať celý intervalový strom. Na spodnej úrovni použijeme všetky jadrá, na ďalšej len každé druhé, a tak ďalej – z každej dvojice jadier vždy necháme len to pravé.
Aby sme vedeli pristupovať ku všetkým vrstvám intervalového stromu, program pre triedičku vygenerujeme tak, že inštrukcie dostanú do názvu číslo vrstvy stromu, ktorú počítajú.
Takto si vyrobíme súčtový intervalový strom:

```
cislo0:      id
sucet0:      input 1
dva:         const 2
n:           const 1048576   # = 2^20

################################################ prvý blok, ktorý 20x zopakujeme
```

```
som_zivy0:    < cislo0 n
som_pravy0:   % cislo0 dva
som_pravy0:   & som_zivy0 som_pravy0
lavy_sucet0:  left sucet0
novy_sucet0:  + lavy_sucet0 sucet0
sucet1:       if som_pravy0 novy_sucet0 sucet0
cislo1:       / cislo0 dva
cislo1:       if som_pravy0 cislo1 n
              sort cislo1
################################################ koniec prvého bloku
```

V prvej kópii bloku sú ešte všetky jadrá „živé“. Podľa parity čísla ich rozdelíme na ľavé a pravé. Pravé jadro k svojmu súčtu pripočíta súčet zo svojho ľavého suseda, čím dostaneme nový súčet (`sucet1`) zodpovedajúci vrcholu vo vyššej vrstve intervalového stromu. Následne vrcholy prečíslujeme: všetky ľavé dostanú číslo n, čím ich prestaneme používať na ďalších vrstvách, a všetky pravé si zmenšia číslo na polovicu.
Vyznačený blok ešte 19× zopakujeme, v každej kópii však inkrementujeme všetky čísla v názvoch inštrukcií – teda po prvom bloku bude nasledovať inštrukcia `som_zivy1`, a tak ďalej.
Keď už máme strom hotový, spočítame si pre každý vrchol intervalového stromu hodnotu `prev` s vyššie popísaným významom: súčet prvkov ostro naľavo. Toto spravíme opäť pomocou 20 blokov inštrukcií, len teraz sa budeme v číslovaní vracať od 19 naspäť k nule – strom prechádzame zhora dole. Všimnite si, že pri výpočte používame hodnoty `lavy_sucet` vypočítané počas prvého prechodu.

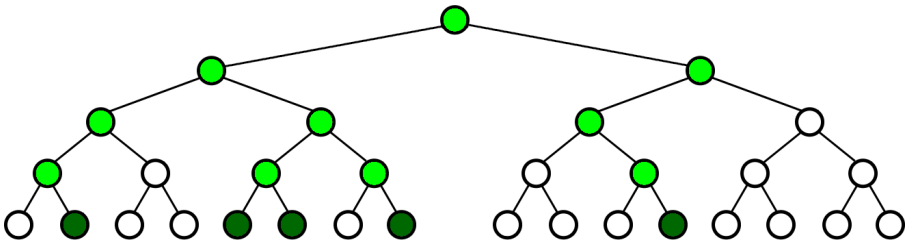
```
prev20:       const 0

################################################ druhý blok, ktorý 20x zopakujeme
              sort cislo19
pravy_prev19: right prev20
prev19:       if som_pravy19 prev20 pravy_prev19
novy_prev19:  + lavy_sucet19 prev19
prev19:       if som_pravy19 novy_prev19 prev19
################################################ koniec druhého bloku

sucet:        + sucet0 prev0
```

Podúloha C (4 body): koľký som v poradí

Zdalo by sa, že stačí jadrá usporiadať podľa ich vstupu a potom každému priradiť index, na ktorom sa ocitlo. Lenže túto operáciu v našom modeli robiť nevieme a ukáže sa, že vypočítať tieto indexy je prekvapivo netriviálne. Využijeme na to pozorovanie, že index prvku v usporiadanom poli je rovný počtu iných prvkov, ktoré sú od neho menšie. Všetky hodnoty na vstupe sú navzájom rôzne nezáporné celé čísla menšie ako 2^{30}. Predstavme si preto pole dĺžky 2^{30}, v ktorom sú samé nuly. Následne pre každé číslo x v našej postupnosti zmeníme nulu na indexe x na jednotku. Všimnite si, že prefixové súčty takto zostrojenej postupnosti nesú informáciu, ktorú potrebujeme: pre každý prvok x, ktorý sme dostali na vstupe, platí, súčet prvkov tohto imaginárneho poľa na indexoch menších ako x je rovný počtu menších prvkov na našom vstupe – a teda hľadanej odpovedi pre jadro, v ktorom je hodnota x.
Ak by sme mali 2^{30} jadier, mohli by sme priamo použiť riešenie podúlohy B. My ich však máme menej – len $n = 2^{20}$. Čo s tým?
Predstavme si, že sme si nad našim poľom dĺžky 2^{30} postavili intervalový strom ako v podúlohe B. Teraz pre každú jednotku v poli ofarbíme na zeleno list, ktorý ju obsahuje, a potom aj jeho otca, otca jeho otca, a tak ďalej až po koreň. Teraz platí, že každý vrchol, ktorý nie je zelený, má pod sebou len samé nuly. Takéto vrcholy ale vôbec nepotrebujeme pri výpočte prefixových súčtov spracovať. Cestou dohora vieme, že súčet celého podstromu pod daným vrcholom je 0, a cestou dodola nepotrebujeme dopočítať, akému prefixovému súčtu zodpovedajú vrcholy v tomto podstrome, keďže ich na nič nepoužijeme.



Intervalový strom. Tmavozelené listy obsahujú jednotky, biele listy nuly.
Vo zvyšku stromu sú zelenou vnútorné vrcholy, ktoré stačí dopočítať.

Stačilo by teda dopočítať, čo sa deje v zelených vrcholoch nášho intervalového stromu. No a na to už zjavne máme jadier dosť – na začiatku každé jadro zodpovedá svojmu vlastnému zelenému listu, a ako ideme hore intervalovým stromom, počet zelených vrcholov nemá ako stúpať, vždy len ostane rovnaký (ak majú každé dva zelené vrcholy iného otca) alebo sa zmenší.
Výpočet teraz bude prebiehať rovnako ako v podúlohe B, len si v každom jadre budeme pamätať číslo jemu zodpovedajúceho zeleného vrcholu a budeme robiť kontroly navyše: keď sa pozrieme na jadro naľavo alebo napravo, skontrolujeme si aj jeho zelené číslo. Ak je naozaj naším susedom v intervalovom strome, spracujeme jeho hodnotu, ak nie, vieme, že v intervalovom strome susedíme s bielym vrcholom, ktorého hodnota je nula.

Pozrime sa na výsledný program pre triedičku. Začneme prvou fázou: postavením súčtového intervalového stromu.
Jadrá predstavujúce zelené vrcholy si zistia, či sú ľavé alebo pravé v aktuálnej dvojici (`som_lavy`), pozrú sa na číslo svojho príslušného suseda (`prave_cislo`) a z toho zistia, či je ten ich dvojičkou (`zije_pravy`) alebo je niekde ďalej v strome. Podľa toho si do premennej vypočítame súčet našej dvojičky (`pravy_sucet`): ak ju máme, je to jej súčet, ak nie, je to nula. Na záver si ešte dáme pozor na to, že ak máme dva zelené vrcholy tvoriace dvojičku (`zijeme_obaja`), tak v ďalšej úrovni z nich bude len jeden zelený vrchol – ľavému z nich vtedy nastavíme číslo na 2^{30} a prestaneme ho používať.
Toto celé zopakujeme 30-krát, pričom si vo všetkých názvoch inštrukcií postupne zvyšujeme čísla – v prvom bloku vypočítame `som_zivy0`, v druhom `som_zivy1`, atď.

```
cislo0:         input 1
                sort cislo0
sucet0:         const 1
nula:           const 0
jedna:          const 1
dva:            const 2
maxval:         const 1073741824

################################################## prvý blok, ktorý 30x zopakujeme
som_zivy0:      < cislo0 maxval
som_pravy0:     % cislo0 dva
som_pravy0:     & som_pravy0 som_zivy0
som_lavy0:      ! som_pravy0
som_lavy0:      & som_lavy0 som_zivy0

lave_cislo0:    left cislo0
lave_treba0:    - cislo0 jedna
zije_lavy0:     = lave_cislo0 lave_treba0
prave_cislo0:   right cislo0
prave_treba0:   + cislo0 jedna
zije_pravy0:    = prave_cislo0 prave_treba0
```

```
lavy_sucet0:   left sucet0
lavy_sucet0:   if som_pravy0 lavy_sucet0 nula
lavy_sucet0:   if zije_lavy0 lavy_sucet0 nula
pravy_sucet0:  right sucet0
pravy_sucet0:  if som_lavy0 pravy_sucet0 nula
pravy_sucet0:  if zije_pravy0 pravy_sucet0 nula
novy_sucet0:   + lavy_sucet0 pravy_sucet0
novy_sucet0:   + novy_sucet0 sucet0
sucet1:        if som_zivy0 novy_sucet0 sucet0

cislo1:        / cislo0 dva
cislo1:        if som_zivy0 cislo1 maxval
zijeme_obaja0: & som_lavy0 zije_pravy0
cislo1:        if zijeme_obaja0 maxval cislo1
               sort cislo1
################################################ koniec prvého bloku
```

No a už ostáva len druhá fáza. V pamäti jadier už máme obsah všetkých zelených vrcholov nášho intervalového stromu, teraz už len potrebujeme prejsť zhora dole a poposielať si hodnotu `prev`, aby sme dostali prefixové súčty.
Toto spravíme nasledovne: Každý vrchol intervalového stromu si zistí hodnotu `prev` zo svojho rodiča. Ten je skoro vždy zapísaný v tom istom jadre o úroveň vyššie. Jediná výnimka: ak sa vo vyššej úrovni stromu moje jadro už nepoužíva (o úroveň vyššie `som_zivy` je nula), som v tejto úrovni ľavý z dvojičky zelených vrcholov, a teda údaje o mojom rodičovi sú v mojom pravom susedovi (`pravy_prev`). Ak som ľavý syn, takto získanú hodnotu rovno použijem, ak som pravý, tak k nej pripočítam súčet z ľavého syna (ten už poznám v `lavy_sucet`).

```
som_zivy30:    < cislo30 maxval
prev30:        copy nula

################################################ druhý blok, ktorý 30x zopakujeme
               sort cislo29
pravy_prev29:  right prev30
prev29:        if som_zivy30 prev30 pravy_prev29
novy_prev29:   + lavy_sucet29 prev29
prev29:        if som_pravy29 novy_prev29 prev29
################################################ koniec druhého bloku

vystup:        copy prev0
```

ŠTYRIDSIATY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: NIVAM – Národný inštitút vzdelávania a mládeže, Bratislava 2024