A-III-1 Dobíjacie stanice

V celom vzorovom riešení budeme konzistentne používať nasledujúcu terminológiu: V súlade s teóriou grafov budeme namiesto osád a ciest medzi nimi hovoriť o vrcholoch a hranách zadaného stromu. Dobíjacím staniciam budeme stručne hovoriť *nabíjačky*. Množstvo energie, ktoré auto má na cestu ďalej, budeme volať *nabitie*. Nabitie auta nadobúda celočíselné hodnoty od 0 po k, pričom 0 je vybité auto, ktoré už nikam ísť nevie.

Prehľadávanie stavového priestoru

Riešenie za 4 body sa dalo založiť na prehľadávaní stavového priestoru. Stav auta je dvojica (aktuálny vrchol, aktuálne nabitie). Postavíme si nový graf, ktorého vrcholy sú tieto stavy a hrany predstavujú možné akcie. Pre každú pôvodnú hranu „viem ísť z x do y“ teda dostaneme k nových hrán „keď som v x s nabitím $z > 0$, viem ísť do y a tam mať nabitie $z - 1$“, a navyše nám pribudnú hrany „keď som v x a je tu nabíjačka, viem si zvýšiť nabitie na k“.
Takýto graf má zjavne $O(nk)$ vrcholov a $O(nk)$ hrán. Postupne pre každý stav tvaru „som v x s plne nabitým autom“ spustíme prehľadávanie, ktoré zistí, ktoré iné stavy (a tým pádom ktoré iné vrcholy) sú z tohto dosiahnuteľné. Takéto riešenie má celkovú časovú zložitosť $O(n^2k)$.

Ako ďaleko mám najbližšiu nabíjačku?

Pomerne ľahko si vieme spočítať, ako ďaleko máme z každého vrcholu k najbližšej nabíjačke. Toto vieme dokonca zistiť naraz pre všetky vrcholy v čase $O(n)$: stačí spustiť prehľadávanie do šírky, ktoré začne tým, že všetkým vrcholom s nabíjačkou nastaví vzdialenosť na 0 a zaradí ich do fronty na spracovanie. Vypočítané vzdialenosti budeme označovať b_i.
Teraz si rozmyslime, aký význam má hodnota $k - b_i$. Ide o najväčšie nabitie, ktoré v tomto vrchole vieme mať. Teda, s výnimkou úplného začiatku cesty. Presnejšia formulácia je, že ak sa kedykoľvek vo vrchole i ocitneme s nabitím menším ako $k - b_i$, nikdy v budúcnosti už v ňom nebudeme vedieť mať nabitie väčšie ako $k - b_i$.
No a keďže nám pri cestovaní po krajine vôbec nezáleží na dĺžke cesty, nič nepokazíme, keď si „odskočíme“ dobiť auto vždy, keď môžeme. Predstavme si, že sme práve prišli do vrcholu i a máme aktuálne nabitie j také, že $j < k - b_i$ ale $j \geq b_i$. Vždy, keď takáto situácia nastane, nič nepokazíme, ak sa prevezieme na najbližšiu nabíjačku (kam sa vieme dostať, keďže $j \geq b_i$), tam dobijeme auto a zase sa vrátime naspäť do i.
Zvoľme si teraz začiatočný vrchol a spustime z neho prehľadávanie pôvodného grafu. (Keďže ide o strom, je jedno, či prehľadávame do šírky, hĺbky, alebo ešte ináč.) Keď navštívime vrchol, pozrieme sa, či si vieme odskočiť dobiť batériu (pokojne aj do ešte nespracovaného vrcholu) a ak áno, tak to spravíme. Následne si pre práve spracovaný vrchol zapamätáme, aké nabitie práve máme, a s takým nabitím z neho potom pokračujeme ďalej.
Tvrdíme, že tento algoritmus navštívi práve tie vrcholy, ktoré sú autom dosiahnuteľné, a ku každému bude mať poznačené najväčšie nabitie, ktoré v ňom vieme mať, nech by sme zo zvoleného štartu chodili akokoľvek. Toto ľahko dokážeme napr. matematickou indukciou podľa vzdialenosti od štartu. Pre každý vrchol je jednoznačne určené, odkiaľ doň prvýkrát prídeme (sme na strome) a ak už vieme, že pre predošlý vrchol sme mali najlepšie možné nabitie, keď sme z neho odchádzali, vieme si zdôvodniť, že v oboch prípadoch (aj ak si v novom vrchole vieme odskočiť dobiť sa, aj ak už na to máme primalé nabitie) aj pre nový vrchol dostaneme najlepšiu možnú hodnotu.
Takto teda dostávame algoritmus za 6 bodov. Jeho najpomalšia časť je tá, ktorú sme práve popísali: z každého začiatočného vrcholu raz prehľadáme pôvodný strom. Toto nám bude trvať $O(n^2)$ krokov.

Počítame všetko naraz

Doterajšie riešenia všetky potrebovali spracúvať každý možný začiatok samostatne. Teraz si ukážeme riešenie, pri ktorom to už nebude potrebné: všetko potrebné spočítame pri jednom prechode daným stromom.
Strom budeme z jedného pevne zvoleného vrcholu (je jedno ktorého) prehľadávať do hĺbky. Pri tom je užitočná predstava, že sme si strom v dotyčnom vrchole *zakorenili* (akoby zavesili zaň). Tým je pre každý iný vrchol jednoznačne určený jeho *otec*: otcom v je prvý vrchol na ceste z v do koreňa stromu. Vrcholy, ktorých otcom je v, voláme jeho *synmi*. *Podstrom* s koreňom v tvoria okrem v aj jeho synovia, synovia jeho synov, atď. – teda práve tie vrcholy, z ktorých cesta do koreňa vedie cez v.

Značenie $m[u, v]$ budeme používať pre hodnoty, ktoré by vypočítalo vyššie uvedené kvadratické riešenie: maximálne množstvo nabitia, ktoré môžeme mať vo vrchole v, ak sme začínali z u.

Pomocné tvrdenie: Ak existuje nejaký spôsob, ako začať vo vrchole a s nabitím i_a a skončiť vo vrchole b s nabitím aspoň i_b, tak existuje aj spôsob, ako začať vo vrchole b s nabitím $k - i_b$ a skončiť vo vrchole a s nabitím aspoň $k - i_a$.
Dôkaz: Obrátime postupnosť krokov a rozmyslíme si, čo sa deje s nabitím na začiatku (pred prvým dobíjaním) a na konci (od posledného dobíjania neskôr).
Dôsledok: Ak sa dá začať v a s plne nabitým autom a nejak prísť do b, tak sa dá aj začať v b s plne nabitým autom a nejak prísť do a.

Ako sme už spomenuli, základom nášho algoritmu bude prehľadanie stromu do hĺbky. Počas neho si budeme pomocou dynamického programovania počítať vhodné údaje, z ktorých budeme vedieť vypočítať riešenie súťažnej úlohy.
Pre ľubovoľné dva vrcholy a, b platí, že jednoduchá cesta z a do b (taká, ktorá žiadnou hranou nejde dvakrát) ide v zakorenenom strome najskôr len dohora po nejaký vrchol c a potom zase len dodola. Vrchol c zvykneme nazývať najmenším spoločným predkom vrcholov a a b a značiť $lca(a, b)$. V našom algoritme budeme pri spracúvaní vrcholu c chcieť spočítať všetky dvojice (a, b) také, že $lca(a, b) = c$ a vieme prejsť elektromobilom medzi a a b. Takto zabezpečíme, že každú dvojicu zarátame práve raz.
Informácie o vrcholoch budeme postupne vypĺňať do dvojrozmerného poľa $C[1..n][0..k]$. Hodnoty v $C[v]$ popisujú podstrom s koreňom v. Ich význam je nasledovný: $C[v][x]$ je počet takých vrcholov u v podstrome s koreňom v, pre ktoré platí, že $m[u, v] = x$.

Spracovanie konkrétneho vrcholu v bude vyzerať nasledovne: Označme synov vrcholu v ako $s_1, \ldots, s_\ell$. Začneme tým, že sa rekurzívne zavoláme na každého z nich. Tým spočítame dvojice v podstrome daného syna a získame počty vrcholov v poli $C[s_i]$.
V poli $C[v]$ inicializujeme všetko na nulu a následne nastavíme $C[v][k]$ na 1 za samotný vrchol v. Teraz budeme postupne po jednom prechádzať synov a podľa ich polí $C[s_i]$ upravovať pole $C[v]$, detaily uvedieme nižšie. Keď spracúvam syna s_i, mám už v poli $C[v]$ údaje o prvých $i - 1$ synoch. Pomocou nich budeme vedieť spočítať počet hľadaných dvojíc takých, že práve jeden prvok z dvojice je v podstrome s koreňom s_i. (Ten druhý môže byť buď vo skôr spracovanom podstrome, alebo ním je samotný vrchol v.)
Detaily spracovania syna s_i:
Prvý krok bude, že zoberieme pole $C[s_i]$ a vyrobíme z neho pomocné pole D. Zatiaľ čo pole $C[s_i]$ malo vrcholy rozdelené podľa toho, s akým najväčším nabitím vieme z nich skončiť v s_i, v poli D chceme mať tie isté vrcholy prerozdelené podľa toho, s akým najväčším nabitím z nich vieme skončiť vo v.
Vrcholy zodpovedajúce $C[s_i][0]$ už môžeme odignorovať, z tých sa už nedalo ďalej pokračovať. Pre každé $j > 0$ sa nabitie j najskôr krokom z s_i do v o 1 zmenší, no a následne ho možno vieme upraviť tým, že si z v (rovnako ako v kvadratickom riešení) odskočíme dobiť auto a vrátime sa. Takto dostaneme novú hodnotu nabitia j'. Táto je rovnaká pre všetky vrcholy zodpovedajúce $C[s_i][j]$, takže jednoducho k $D[j']$ pripočítame $C[s_i][j]$.
Druhý krok bude, že spočítame všetky dvojice a, b, pre ktoré platí, že sa medzi nimi dá prejsť, $lca(a, b) = v$ a práve jeden z vrcholov a, b leží v podstrome s koreňom s_i.
Zoberme nejaké nabitie i. V aktuálnom podstrome existuje $D[i]$ vrcholov, z ktorých sa vieme dostať do v s nabitím i. Podľa pomocného tvrdenia vieme, že odtiaľto ďalej sa vieme dostať do ľubovoľného vrcholu, z ktorého vieme prísť do v s nabitím asoň $k - i$. Takýchto vrcholov máme teraz presne $C[v][k - i] + \cdots + C[v][k]$, pre toto konkrétne i teda dostávame $D[i] \cdot (C[v][k - i] + \cdots + C[v][k])$ neusporiadaných dvojíc. Toto chceme sčítať cez všetky i. Priamočiary postup by vyžadoval rádovo k^2 krokov, ale keď si všimneme, že stačí postupne zvyšovať i a hodnotu v zátvorke si držať v pomocnej premennej a tak ľahko sčítame všetky dvojice v čase $O(k)$.
Tretí krok je už ľahký: hodnoty poľa D pripočítame k príslušným hodnotám poľa $C[v]$ a tým sme spracovanie syna s_i skončili.

Ostáva nám jediné: odhad časovej zložitosti. Keďže strom má $n - 1$ hrán, majú všetky vrcholy dokopy $n - 1$ synov. Dokopy za celé riešenie n-krát inicializujeme vrchol, ktorý práve začíname spracúvať, a $(n - 1)$-krát

spracúvame nejakého syna nejakého vrcholu. Každá z týchto operácií trvá $O(k)$, celková časová zložitosť nášho riešenia je teda $O(nk)$.

Ešte lepšie riešenie

Na základe podobných myšlienok vieme súťažnú úlohu riešiť v časovej zložitosti $O(n \log n)$ a toto riešenie by fungovalo efektívne aj pre vstupy s veľkým k. Novou technikou, ktorú pri tom použijeme, je tzv. centroidová dekompozícia. V každom strome vieme v lineárnom čase nájsť centroid – taký vrchol, po ktorého odstránení bude mať každý komponent súvislosti nanajvýš polovicu pôvodného počtu vrcholov. Celé riešenie potom vyzerá nasledovne:

1. nájdi v strome jeden jeho centroid c
2. prehľadávaním z c v čase $O(n)$ spočítaj všetky dobré dvojice vrcholov, pre ktoré cesta vedie cez c
3. odstráň c, rozdeľ strom na samostatné komponenty a na každý sa rekurzívne zavolaj

Keďže pri každom rekurzívnom volaní zmenšíme veľkosť stromu aspoň dvakrát, bude hĺbka rekurzie nanajvýš $\log_2 n$. Následne vieme podobnou úvahou ako pri triedení MergeSort zdôvodniť, že celková časová zložitosť tohto algoritmu je tým pádom $O(n \log n)$. Detaily kroku 2 prenechávame na čitateľa, ktorému sa chce pokračovať za rámec tohto riešenia.

Listing programu (C++)

```
#include <iostream>
#include <vector>
#include <queue>

using namespace std;


int n, k;
vector<bool> stanice;
vector<vector<int>> e;
vector<int> d; // vzdialenost od najblizsej dobijacej stanice
vector<vector<int>> C;
long long res; // vysledok

void compute_d () { // BFS zo vsetkych dobijacich stanic naraz
    queue<int> q;
    d.resize(n, n + 1);
    for (int i = 0; i < n; ++i) if (stanice[i]) {
        q.push(i);
        d[i] = 0;
    }
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (auto v : e[u]) {
            if (d[v] == n + 1) {
                d[v] = d[u] + 1;
                q.push(v);
            }
        }
    }
}

// spracujeme vrchol v s rodicom parent
void solve_vertex (int v, int parent = -1) {
    // rekurzivne spracujeme potomkov
    for (auto u : e[v]) if (u != parent)
        solve_vertex(u, v);

    for (auto u : e[v]) if (u != parent) {
        // spracujeme syna u
        vector<int> Cx(k + 1, 0);
        for (int i = 1; i <= k; ++i) {
            int b = i - 1;
            if (b >= d[v]) b = max(b, k - d[v]);
            Cx[b] += C[u][i];
        }

        // zapocitame dvojice s jednym koncom v podstrome u
        long long pref_sum = 0;
        for (int i = 0; i <= k; ++i) {
            pref_sum += C[v][k - i];
```

```
            res += Cx[i] * pref_sum;
        }

        for (int i = 0; i <= k; ++i) {
            C[v][i] += Cx[i];
        }
    }

}

int main () {
    int s;

    cin >> n >> k >> s;

    stanice.resize(n, false);
    e.resize(n);

    int a, b;
    for (int i = 0; i < s; ++i) {
        cin >> a;
        stanice[a - 1] = true;
    }

    for (int i = 0; i < n - 1; ++i) {
        cin >> a >> b;
        --a; --b;
        e[a].push_back(b);
        e[b].push_back(a);
    }

    compute_d();

    C.resize(n);
    for (int i = 0; i < n; ++i) {
        C[i].resize(k + 1, 0);
        C[i][k] = 1;
    }

    res = 0;

    solve_vertex(0);

    cout << res * 2 << endl;
}
```

A-III-2 Základne na Marse

Pred čítaním tohto vzorového riešenia odporúčame čitateľovi oboznámiť sa s potrebnými základmi výpočtovej geometrie. V Kuchárke KSP `https://www.ksp.sk/kucharka/` nájdete články „Úvod do výpočtovej geometrie“, „Skalárny súčin a vektorový súčin“ a následne „Konvexný obal“.
Dva hlavné nástroje, ktoré budeme používať, budú test na vzájomnú polohu priamky a bodu a konvexný obal. Pre danú orientovanú priamku a bod vieme v konštantnom čase (napr. pomocou vektorového súčinu) vypočítať, na ktorej strane danej priamky daný bod leží. Konvexný obal danej konečnej množiny bodov je najmenší mnohouholník, ktorý ich všetky obsahuje. Ide vždy o konvexný mnohouholník, ktorého vrcholy sú v niektorých bodoch z dotyčnej množiny.

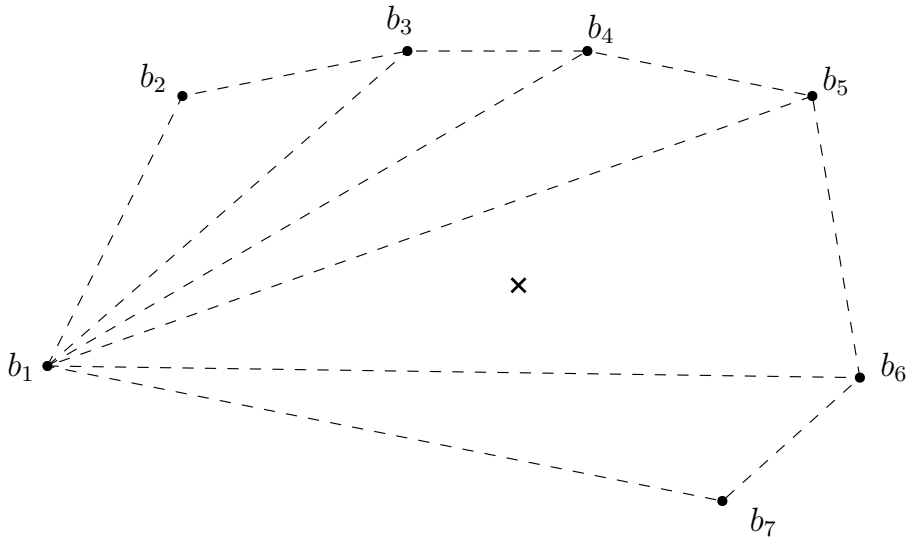
Riešenie hrubou silou

Začneme pozorovaním, že pre trojuholník ABC a bod D vieme v konštantnom čase overiť, či D leží vo vnútri ABC: stačí sa pozrieť na orientované priamky $\overrightarrow{AB}$, $\overrightarrow{BC}$ a $\overrightarrow{CA}$ a skontrolovať, že vo všetkých troch prípadoch leží D na tej istej strane od priamky. (Ak navyše napr. vieme, že body A, B, C ležia na obvode trojuholníka v poradí v smere ručičiek, môžeme explicitne kontrolovať, či D leží vo všetkých troch prípadoch napravo od príslušnej orientovanej priamky.)
Priamou aplikáciou tohto testu dostávame riešenie s časovou zložitosťou $O(v^3 z)$, za ktoré boli dva body.

Prečo konvexný obal?

Ak nejaká základňa leží v nejakom trojuholníku z vysielačov, tak zjavne leží aj v konvexnom obale všetkých vysielačov. Zaujímavé je, že platí aj opačná implikácia, a teda ekvivalencia: základňa leží v nejakom trojuholníku

práve vtedy, keď leží v konvexnom obale všetkých vysielačov.
Dôkaz: Vyberme si ľubovoľný vrchol konvexného obalu a spojme ho úsečkami so všetkými ostatnými vrcholmi. Konvexný obal s k vrcholmi sa nám týmto rozdelí na $k-2$ disjunktných trojuholníkov. Keďže vieme, že základňa leží vo vnútri konvexného obalu a že sú všetky body vo všeobecnej polohe, musí ležať vo vnútri práve jedného z týchto trojuholníkov.



Všimnite si, že z vyššie uvedeného dôkazu vyplýva aj silnejšie tvrdenie: Nech Z je nejaká základňa a V ľubovoľný vysielač, ktorý leží na konvexnom obale vysielačov. Ak existuje nejaký trojuholník obsahujúci Z, tak určite existuje nejaký trojuholník, ktorého jeden z vrcholov je V. (V dôkaze totiž stačí zvoliť V ako ten vrchol, z ktorého kreslíme úsečky.)
Toto pozorovanie nám *aj bez toho, aby sme museli zostrojiť konvexný obal* dáva pre našu úlohu algoritmus s časovou zložitosťou $O(v^2 z)$. Ak totiž napríklad za V zvolíme vysielač, ktorý má spomedzi všetkých najmenšiu x-ovú súradnicu, bude zjavné, že V musí ležať na konvexnom obale. Takže si stačí raz nájsť tento V a potom pre každú základňu otestovať všetky možné dvojice zvyšných vrcholov.

Efektívnejšie hľadanie trojuholníka

Tentokrát už naozaj začneme tým, že si explicitne zostrojíme konvexný obal množiny všetkých vysielačov. V kuchárke je popísaný v-podstate-optimálny algoritmus, ktorý toto zvládne v čase $O(v \log v)$.
Akonáhle efektívne zostrojíme konvexný obal, dostávame už zadarmo jedno možné 7-bodové riešenie: pre každú základňu Z a pevne zvolený vrchol V konvexného obalu stačí ako zvyšné dva body trojuholníka vyskúšať ostatné strany konvexného obalu. Takto dostávame riešenie s časovou zložitosťou $O(v \log v + vz)$.

Ide to však aj ešte šikovnejšie. Posledným krokom ku vzorovému riešeniu bude pozorovanie, že správny trojuholník vieme nájsť binárnym vyhľadávaním.
Očíslujme si vrcholy konvexného obalu tak ako na vyššie uvedenom obrázku: najľavejší (presnejšie vyšší z najľavejších, ak sú také dva) bude b_1 a ďalšie budú $b_2, \ldots, b_k$ postupne po obvode v smere ručičiek.
Každú základňu teraz môžeme spracovať nasledovne:

- Ak leží naľavo od priamky $\overrightarrow{b_1 b_2}$ alebo napravo od $\overrightarrow{b_1 b_k}$, zjavne je mimo celého konvexného obalu.
- Binárnym vyhľadávaním nájdeme najväčšie i také, že hľadaná základňa ešte leží napravo od priamky $\overrightarrow{b_1 b_i}$. (Na našom obrázku je $i = 5$.)
- Teraz už vieme, že naša základňa leží v uhle $b_i b_1 b_{i+1}$. Ešte skontrolujeme, či leží napravo od priamky $\overrightarrow{b_i b_{i+1}}$ a buď sme práve našli jeden vyhovujúci trojuholník, alebo zistili, že žiaden neexistuje.

Toto riešenie má časovú zložitosť $O((v+z)\log v)$.

Listing programu (C++)

```
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

typedef long long ll;

struct point {
    int id;
    ll x, y;

    bool operator== (const point& b) {
        return id == b.id && x == b.x && y == b.y;
    }

    bool operator< (const point& b) {
        if (x == b.x) return y < b.y;
        return x < b.x;
    }
};

int m, n;
vector<point> vysielace;
vector<point> obal;

// Vrati true, ak a, b, c idu v smere hodinovych ruciciek
// (t.j. ked stojime v bode a otoceni smerom na bod b, mame bod c niekde napravo)
bool clockwise (point a, point b, point c) {
    ll ux = c.x - a.x, uy = c.y - a.y;
    ll vx = b.x - a.x, vy = b.y - a.y;
    return ux*vy - uy*vx > 0;
}

// Najdeme vrcholy konvexneho obalu, usporiadane po obvode v smere ruciciek
vector<point> create_convex_hull (vector<point> input) {
    sort(input.begin(), input.end());
    vector<point> horna_polovica, dolna_polovica;

    for (auto p : input) {
        while (horna_polovica.size() >= 2 &&
            !clockwise(horna_polovica[horna_polovica.size()-2],
                       horna_polovica[horna_polovica.size()-1], p)) {
            horna_polovica.pop_back();
        }
        horna_polovica.push_back(p);
    }

    for (int i = input.size() - 1; i >= 0; --i) {
        auto p = input[i];
        while (dolna_polovica.size() >= 2 &&
            !clockwise(dolna_polovica[dolna_polovica.size()-2],
                       dolna_polovica[dolna_polovica.size()-1], p)) {
            dolna_polovica.pop_back();
        }
        dolna_polovica.push_back(p);
    }

    vector<point> output = horna_polovica;
    for (int i = 1; i + 1 < dolna_polovica.size(); ++i)
        output.push_back(dolna_polovica[i]);

    return output;
}

void vyries_zakladnu (point zakladna) {
    int l = 1, r = obal.size() - 1;
    while (r - l > 1) {
        int m = (l + r) / 2;
        if (clockwise(obal[0], obal[m], zakladna)) l = m;
        else r = m;
    }

    // Kedze sme okrajove pripady neosetrili na zaciatku, spravime teraz namiesto
    // toho kompletnu kontrolu, ci dana zakladna lezi v trojuholniku {obal[0], obal[l], obal[r]}.

    if (clockwise(obal[0], obal[l], zakladna) &&
        clockwise(obal[l], obal[r], zakladna) &&
        clockwise(obal[r], obal[0], zakladna))
        cout << obal[0].id << " " << obal[l].id << " " << obal[r].id << endl;
    else
        cout << "neexistuje" << endl;
}
```

```
int main () {
    cin >> m >> n;

    ll x, y;
    for (int i = 1; i <= m; ++i) {
        cin >> x >> y;
        vysielace.push_back(point{i, x, y});
    }

    obal = create_convex_hull(vysielace);

    for (int i = 0; i < n; ++i) {
        point zakladna;
        cin >> zakladna.x >> zakladna.y;
        vyries_zakladnu(zakladna);
    }
}
```

A-III-3 Hľadanie v externej pamäti

Binárne vyhľadávanie

Binárne vyhľadávanie spraví na poli dĺžky n približne $\log_2 n$ krokov. Interval, v ktorom hľadáme, sa v každom kroku zmenší na polovicu. Keď máme interval dĺžky d a pozrieme sa do jeho stredu, pozreli sme sa práve na prvok, ktorý leží vo vzdialenosti $d/2$ od oboch okrajov aktuálneho intervalu. A teda kým má tento interval dĺžku väčšiu ako $2b$, ležia každé dva prvky, na ktoré sa opýtame, vo vzdialenosti väčšej ako b – a teda každý v inom bloku. No a akonáhle má aktuálny interval dĺžku nanajvýš $2b$, leží celý v nanajvýš troch blokoch. Ak náš algoritmus v tejto chvíli všetky načíta do pamäte, bude mať zvyšok hľadania komunikačnú zložitosť konštantnú. Náš algoritmus teda bude fungovať v dvoch fázach. V prvej spraví toľko načítaní z disku, koľko krokov potrebuje binárne vyhľadávanie na zmenšenie intervalu z dĺžky n na dĺžku nanajvýš $2b$. Toto môžeme zhora odhadnúť počtom krokov, po ktorom bude dĺžka b, čiže $\log_2(n/b)$. V druhej fáze potom už spraví nanajvýš tri ďalšie načítania z disku. Komunikačná zložitosť je teda nanajvýš $\log_2(n/b) + 3$.

Binárne vyhľadávacie stromy

Z množiny, ktorú dostaneme na vstupe, by sme si vedeli vyrobiť napríklad binárny vyhľadávací strom a potom v množine vyhľadávať tak, že budeme hľadať v tomto strome. Keďže vopred poznáme celú množinu, ľahko vyrobíme takmer dokonale vyvážený strom: medián prvkov dáme do koreňa a následne tento postup rekurzívne zopakujeme v každom podstrome.
Takýto strom by sa následne dal na disk uložiť napríklad tak, ako zvykneme ukladať tradičnú binárnu haldu: postupne pre každú úroveň zapíšeme na disk všetky jej prvky zľava doprava.
Pomohli sme si tým ale? Ani nie. Vyhľadávanie v takomto strome má zjavne stále optimálnu časovú zložitosť $O(\log n)$, ale komunikačnú zložitosť dostaneme približne rovnakú ako v minulom riešení: zhruba $\log_2(n/b)$. (Tentokrát je lacných prvých zhruba $\log_2 b$ krokov, keďže prvých zhruba $\log_2 b$ úrovní stromu sa zmestí do prvého bloku disku. Potom už ale pre každú ďalšiu úroveň musíme robiť čítanie z disku.)
Aby sme dostali rádovo menšiu časovú zložitosť, potrebujeme z každého čítania z disku dostať viac informácie – ideálne toľko ako z toho prvého. Ako na to?

Vyhľadávacie stromy vyššieho stupňa

Použijeme vyhľadávacie stromy, v ktorých sa v každom vrchole rozhodujeme medzi viac ako dvoma možnosťami. (V literatúre sa takéto stromy zvyknú označovať B-stromy.) Vo všeobecnosti takýto strom vyzerá nasledovne: V každom vrchole je uložený nejaký (nie nutne rovnaký) počet kľúčov. Tieto sú usporiadané v ostro rastúcom poradí. Ak vrchol obsahuje k kľúčov s hodnotami $x_1 < \cdots < x_k$, tak má presne $k + 1$ podstromov. V prvom podstrome sú prvky s hodnotou menšou ako x_1, v druhom sú prvky s hodnotou medzi x_1 a x_2, a tak ďalej, až po posledný podstrom, v ktorom sú prvky s hodnotou väčšou ako x_k. (Niektoré podstromy môžu byť aj prázdne. Ak sú prázdne všetky, vrchol je listom stromu.)

Je zjavné, že takýto strom je zovšeobecnením klasického binárneho vyhľadávacieho stromu, ktorý má vo všetkých vrcholoch práve jeden kľúč. Vyhľadávanie naďalej funguje rovnako: keď prídeme do vrcholu, zistíme si, či je hľadaný prvok medzi kľúčmi, ktoré tu máme. Ak áno, sme hotoví, ak nie, zistíme, do ktorého podstromu hľadaný prvok patrí, a doň pokračujeme.
Kľúče uložené vo vrchole pritom nemusíme pozerať všetky – to by pre veľké k nebolo efektívne. Je ale zjavné, ako to robiť lepšie: keďže kľúče vo vrchole máme uložené v usporiadanom poli, môžeme na ňom použiť binárne vyhľadávanie. Takto na $\log_k$ krokov buď nájdeme správny prvok alebo správny podstrom.
Ako si zvoliť, aké veľké k chceme mať? Správna bude tá priamočiara možnosť: najväčšie, pre ktorú sa nám ešte celý vrchol zmestí do jedného bloku na disku. V našom riešení budeme pre jednoduchosť predpokladať, že okrem listov všade platí $k = b$: v každom vrchole je b kľúčov.
(Toto naozaj vieme dosiahnuť. Keďže náš strom len raz vytvárame a vieme ho mať uložený „ako haldu“, nepotrebujeme si vo vrchole ukladať ukazovatele na jeho podstromy. Keď sa podobné stromy používajú ako dynamické dátové štruktúry podporujúce aj vkladanie a výber prvkov, zmestí sa do bloku kľúčov menej. Asymptoticky to však nič nezmení, aj pre $b/2$ alebo $b/3$ kľúčov v každom vrchole by sme dostali rádovo rovnaké zložitosti.)
Takýto strom z našej množiny vytvoríme nasledovne: Vyberieme takých b prvkov, ktoré zvyšok množiny rozdelia čo najviac rovnomerne na $b + 1$ častí. (V usporiadanom poli by šlo zhruba o prvky na indexoch, ktoré sú násobkami $n/(b + 1)$.) Tieto prvky budú uložené v koreni. Pre každý podstrom následne jeho strom vyrobíme rekurzívne – teda zopakovaním tohto postupu pre jeho prvky.
Na každej úrovni nášho stromu spracúvame nové množiny, ktorých veľkosť je približne $(b + 1)$-krát menšia. To znamená, že po $\log_{b+1} n \approx \log_b n$ krokoch sa dostaneme do listu. Náš strom bude mať teda hĺbku približne $\log_b n$.
Akú má hľadanie v takomto strome časovú zložitosť? Postupne sa budeme pozerať na $\log_b n$ vrcholov a v každom z nich budeme v čase $\log_2 b$ binárne vyhľadávať v jeho kľúčoch. To nám dokopy dáva časovú zložitosť $\log_b n \cdot \log_2 b = \log_2 n$. Časová zložitosť je teda v poriadku.
No a komunikačná zložitosť hľadania je zjavne $\log_b n$.
(Praktické porovnanie: ak máme v poli $n = 10^9$ záznamov a veľkosť bloku $b = 100$, binárne vyhľadávanie z podúlohy A by potrebovalo čítať v najhoršom prípade až 27 blokov, zatiaľ čo novému riešeniu budú aj v najhoršom prípade stačiť štyri.)

Dôkaz optimálnosti

Uvažujme ľubovoľnú situáciu, kedy už máme nejakú konkrétnu množinu nejak spracovanú a následne nejak uloženú na disku.
Deterministický algoritmus sa môže rozhodovať len na základe prvkov množiny, ktoré mohol vidieť. Presnejšie, keď sa budeme pozerať na to, čo robí deterministický algoritmus pri hľadaní dvoch rôznych prvkov v tej istej množine, jediné miesto, kedy môže prestať robiť pre oba prvky to isté, je, keď spraví nejaké porovnanie hľadaného prvku s prvkom z množiny a to mu pre jeden prvok dá iný výsledok ako pre druhý.
Prvky množiny, ktoré mohol v nejakom okamihu počas svojho behu algoritmus vidieť, sú jednoznačne určené tým, ktoré bloky načítal do pamäte.
Predstavme si teda, že sme si vyskúšali náš algoritmus spustiť pre všetky možné prvky. Každý z nich sme hľadali (stále v tej istej množine) a počas toho sme sledovali, ktoré bloky z disku algoritmus číta. Všetky tieto priebehy si môžeme predstaviť ako jeden veľký strom: všetky spustenia programu začnú vždy rovnako a potom sa postupne rozvetvujú podľa toho, aké porovnávania prvkov robí a ako dopadajú.

Vo všetkých prípadoch teda náš neznámy algoritmus musí začať tým, že si niečo počíta a potom časom prvýkrát načíta z disku nejaký blok – pre všetky možné vstupy ten istý.
Následne môže hľadaný prvok ako chce porovnávať s prvkami v tomto bloku. To, čo sa stane ďalej, je nutne jednoznačne určené tým, ako dopadnú porovnávania hľadaného prvku s tými prvkami, ktoré prečítal z disku. Keďže sme z jedného bloku mohli získať nanajvýš b prvkov, je nanajvýš $2b + 1$ rôznych možností ako môžu porovnávania dopadnúť: b možností v ktorých je hľadaný prvok rovný jednému z prečítaných a $b + 1$ možností keď veľkosťou padne do niektorej medzery medzi nimi.
Všetky možné prvky y, ktoré môžeme hľadať, si teraz teda môžeme rozdeliť na (nanajvýš) $2b + 1$ kôpok podľa toho, ako pre ne dopadnú tieto porovnania. (Ak napríklad sú naše prvky mená a z prvého prečítaného bloku

sme dostali mená `Adam`, `Zuza` a `Miro`, tak hľadané mená `Fero` a `Jana` budú zatiaľ na tej istej kôpke „viac ako `Adam` a menej ako `Miro`“.)
Pre každú kôpku je jednoznačne určené, ako sa na nej náš algoritmus správa aj ďalej, a to až do okamihu, kým neprečíta z disku ďalší blok. Podľa informácie v ňom sa naša aktuálna kôpka vstupov znovu rozpadne na nanajvýš $2b+1$ menších. No a takto môžeme pokračovať aj ďalej, až kým algoritmus nezastane a nedá odpoveď.

Vráťme sa späť k predstave nášho stromu, v ktorom si znázorňujeme všetky možné priebehy hľadania pre nejakú konkrétnu množinu. Každý vrchol tohto stromu bude predstavovať jedno konkrétne čítanie nejakého bloku z disku. Ako sme si vyššie zdôvodnili, v každom vrchole sa tento strom rozvetví nanajvýš so stupňom $b+1$: tie vstupy, pre ktoré sa výpočet dostal do tohto vrcholu, v ňom prerozdelíme na $2b+1$ menších kôpok, ale pre b z nich tu výpočet skončí a len $b+1$ ich pokračuje ďalej.
(Je technicky možné, že nejaký hlúpy algoritmus síce už vedel zistiť, že hľadaný prvok v množine leží, napriek tomu ale pokračuje ďalej a bude robiť ďalšie čítania z disku. Takéto správanie však môže jeho komunikačnú zložitosť len zhoršiť, a teda sa ním nemusíme pri robení dolného odhadu zaoberať.)
Každá cesta z koreňa tohto stromu dodola zodpovedá jednému možnému priebehu hľadania. Dĺžka tejto cesty zodpovedá komunikačnej zložitosti. Zaujímať nás teda bude maximálna hĺbka tohto stromu: čím lepšiu komunikačnú zložitosť chceme mať (v najhoršom prípade), tým plytší strom musíme mať.
No a teraz už len vhodne odhadneme, aká táto hĺbka musí byť. Ak by sme mali strom, ktorý má hĺbku $h-1$ (čiže algoritmus, čo vždy prečíta nanajvýš h blokov z disku), bude mať tento strom dokopy nanajvýš $1+(b+1)+\cdots+(b+1)^{h-1} < (b+1)^h$ vrcholov. Pozrime sa teraz čisto na n prvkov, ktoré tvoria našu množinu M. Každý z nich keď hľadáme tak musíme niekedy nájsť. No ale v každom vrchole nájdeme nanajvýš b prvkov. Dokopy teda musí mať náš strom aspoň n/b vrcholov.
Z toho dostávame nutnú podmienku $(b+1)^h \geq n/b$, čo môžeme upraviť na slabšiu nutnú podmienku $(b+1)^h \geq n/(b+1)$ a tú prepísať do podoby $h+1 \geq \log_{b+1} n = c \log_b n$ pre vhodnú konštantu c. A z toho už vidíme, že ľubovoľný korektný algoritmus vyhľadávajúci v množine veľkosti n musí pre niektoré hľadané prvky z disku prečítať aspoň rádovo $\log_b n$ blokov, čo je presne to, čo sme chceli dokázať.

TRIDSIATY ŠIESTY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2021