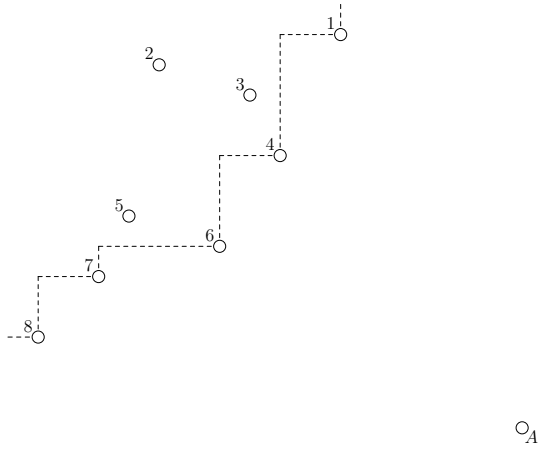
A-III-1 Ihrisko

Riešenie v kubickom čase je priamočiare: pre každú dvojicu bodov sa pozrieme na nimi určený obdĺžnik a pre každý zo zvyšných $n - 2$ bodov overíme, či v ňom leží. Spraviť niečo lepšie už ale vôbec nie je priamočiare.

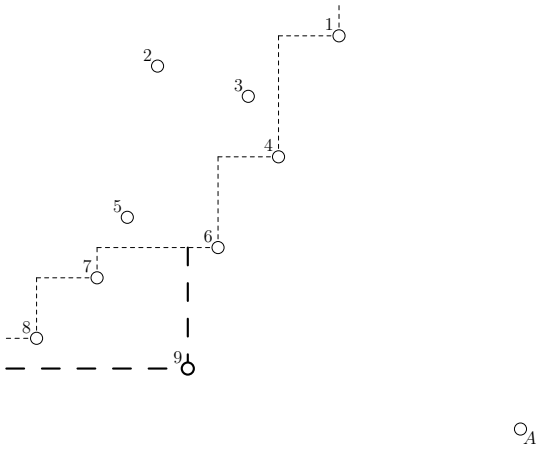
Riešenie v kvadratickom čase

Každý obdĺžnik má bod buď v ľavom dolnom alebo v pravom dolnom rohu. Nám stačí vedieť sčítať tie, ktoré majú bod v pravom dolnom rohu. Keď už takýto algoritmus nájdeme, stačí ho potom použiť dvakrát – raz ho použijeme priamo na zadaný vstup, potom rovinu preklopíme (zmeníme znamienko x-ovým súradniciam bodov) a potom použijeme náš algoritmus druhýkrát.
V tomto riešení budeme skúmať každý možný pravý dolný roh zvlášť. Máme teda konkrétny bod A a pýtame sa, koľko rôznych prázdnych obdĺžnikov na ňom má svoj pravý dolný roh. Ako ľavý horný roh pripadajú do úvahy samozrejme len body, ktoré vzhľadom na A ležia v druhom kvadrante – t.j. naľavo hore od neho.
Zoberme teda túto množinu bodov. Ak máme v tejto množine dva body X a Y také, že Y je v oboch súradniciach bližšie ku A ako X, tak budeme hovoriť, že Y zakrýva X. Napr. na obrázku 1 vidíme, že bod 4 zakrýva body 2 a 3. Ľavé horné rohy prázdnych obdĺžnikov sú potom zjavne práve tie body, ktoré nie sú zakryté žiadnym iným bodom.



Obr. 1: Príklad rozmiestnenia bodov vľavo hore od bodu A. Čiarkovaná čiara predstavuje hranicu oblasti, ktorú tieto body zakrývajú. Každý bod, ktorý leží na hranici, je ľavým horným rohom prázdneho obdĺžnika. A naopak, každý iný bod má medzi sebou a bodom A aspoň jeden z bodov hranice.

Body si môžeme usporiadať zhora dole a v tomto poradí ich spracovať. Priebežne si budeme pamätať množinu bodov, ktoré aktuálne určujú prázdne obdĺžniky. Tieto body budeme volať kandidáti. Vždy, keď spracúvame nový bod, spomedzi kandidátov odstránime tých, ktorých tento nový bod „zakryl“.
Množinu aktuálnych kandidátov si stačí pamätať v obyčajnom zásobníku, pričom na vrchu bude naposledy spracovaný z nich. Všimnite si, že keď ideme zásobníkom zdola hore (od najskôr vložených prvkov – na obrázku v poradí 1, 4, 6, 7, 8), obe ich súradnice klesajú.
Vždy, keď spracúvame nový bod, vieme, že má od všetkých skôr spracovaných menšiu y-ovú súradnicu. Spomedzi kandidátov teda zakryje všetkých, ktorí majú od neho väčšiu x-ovú súradnicu – no a tých máme práve na vrchu zásobníka, takže ich stačí postupne zo zásobníka vyberať a zahadzovať. Príklad takéhoto spracovania ďalšieho bodu je na obrázku 2.
Ak by sme toto spravili zvlášť pre každý bod A, dostali by sme riešenie s časovou zložitosťou $O(n^2 \log n)$: pre každý z n bodov prejdeme všetky ostatné, vyberieme spomedzi nich tie vľavo hore od neho, usporiadame ich a následne vyššie ich popísaným spôsobom zľava doprava spracujeme.



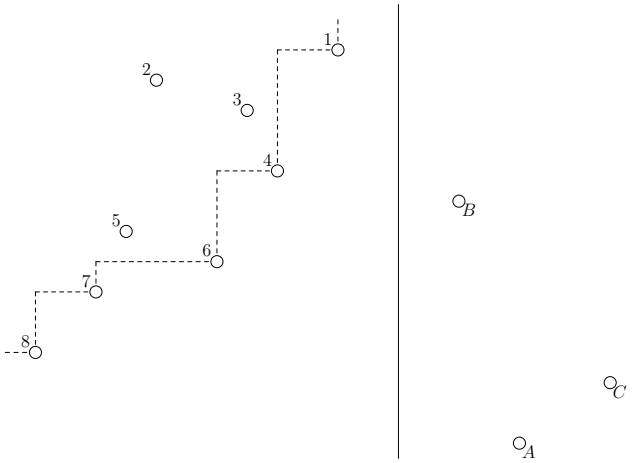
Obr. 2: Príklad spracovania ďalšieho bodu. Nový bod 9 leží nižšie ako všetci doterajší kandidáti. Po pridaní bodu 9 postupne zistíme, že zakryl kandidátov 8 a 7, tých teda zo zo zásobníka odstránime. Ďalší kandidát 6 už je viac vpravo ako bod 9, takže prestaneme odstraňovať. Na záver pridáme bod 9 na vrch zásobníka s kandidátmi.

Aj výber aj spracovanie vieme spraviť v lineárnom čase. (Pri spracúvaní si uvedomte, že každý bod raz pridáme medzi kandidátov a nanajvýš raz ho odtiaľ odoberieme, takže dokopy celé spracovanie zmení množinu kandidátov nanajvýš $2n$-krát.)
Najpomalšou časťou je triedenie. Tu si však stačí uvedomiť, že poradie bodov sa medzi jednotlivými kolami nemení. Môžeme na začiatku raz v zanedbateľnom čase $O(n \log n)$ usporiadať všetkých n bodov. Keď potom spracúvame konkrétny bod A, stačí prechádzať tento usporiadaný zoznam bodov a preskočiť A aj všetky body, čo nie sú od A vľavo hore. Takto spracujeme každý konkrétny bod A v lineárnom čase, a teda dokopy dostaneme časovú zložitosť $O(n^2)$.

Vzorové riešenie

Vo vzorovom riešení využijeme myšlienky kvadratického riešenia a pridáme niečo navyše. Aj v tomto riešení budeme počítať len obdĺžniky, ktoré majú body vľavo hore a vpravo dole.
Prvým vylepšením bude použitie techniky *rozdeľuj a panuj*. Ak máme len jeden bod, riešenie úlohy je triviálne. Ak ich máme viac, rozdelíme ich zvislou čiarou na dve čo najviac rovnaké polovice: ľavú a pravú. Prázdne obdĺžniky sú teraz troch typov: také, čo sú celé v ľavej polovici (ich počet zistíme rekurzívnym volaním pre ľavú polovicu bodov), také, čo sú celé vpravo (druhé rekurzívne volanie) a také, čo majú svoj ľavý horný roh v ľavej a svoj pravý dolný roh v pravej polovici.
Aby sme dostali efektívne riešenie, potrebujeme vedieť efektívne spočítať prázdne obdĺžniky tretieho typu. Na obrázku 3 je príklad toho, ako to bude vyzerať. Pozrite si tento príklad a potom pokračujte v čítaní riešenia.
Pre každý bod A v pravej polovici chceme zistiť, koľko rôznych obdĺžnikov určuje. Prvá vec, ktorú potrebujeme spraviť, je pre každý takýto bod zistiť, kde je „jeho bod B“ – teda najbližší vyšší spomedzi bodov v pravej polovici, ktorý je naľavo od A. Tento bod predstavuje hranicu, po ktorú môže ísť dohora obdĺžnik s rohom v A. Výpočet tejto informácie je v podstate totožný s algoritmom, ktorým sme v predchádzajúcom riešení hľadali množinu kandidátov – len to robíme zdola dohora a vždy, keď pridáme nového kandidáta, tak on je tou hranicou pre tie body, ktoré vtedy spomedzi kandidátov odstránime.
Pre rádovo n bodov toto vieme spraviť v čase $O(n \log n)$ kvôli triedeniu. Keďže aj posledná časť tohto riešenia bude mať rovnakú časovú zložitosť, toto nám bude stačiť.

Hlavná myšlienka zvyšku riešenia teraz bude nasledovná: Úplne všetky body (ľavú aj pravú polovicu dokopy) si usporiadame zhora dole a v tomto poradí ich budeme spracúvať. Na ľavej strane si budeme v zásobníku udržiavať množinu kandidátov. Vždy, keď príde na rad bod z ľavej polovice, túto množinu upravíme (rovnako



Obr. 3: Príklad situácie pri hľadaní obdĺžnikov medzi ľavou a pravou polovicou. Plná zvislá čiara predstavuje hranicu medzi polovicami. Už vieme, že ku bodu A musia ľavé horné rohy byť v aktuálnej množine kandidátov (na čiarkovanej čiare). Teraz si navyše musíme všimnúť, že dohora môže náš obdĺžnik ísť nanajvýš po bod B, a teda len spodní traja kandidáti (8, 7, 6) naozaj určujú prázdne obdĺžniky.

ako v predchádzajúcom riešení). No a vždy, keď príde na rad bod z pravej polovice, pozrieme sa na hornú hranicu pre jeho obdĺžniky a túto súradnicu binárne vyhľadáme v našej množine kandidátov vľavo. Celkový počet prázdnych obdĺžnikov zväčšíme o počet kandidátov, ktorí ležia dostatočne nízko – každý z nich určuje jeden prázdny obdĺžnik.

Akú má tento algoritmus časovú zložitosť? Označíme $T(n)$ celkový čas potrebný na spracovanie n bodov. Toto spracovanie pozostáva z dvoch rekurzívnych volaní na spracovanie $n/2$ bodov, následného predpočítania horných hraníc pre pravú polovicu a potom sčítania obdĺžnikov ktoré zasahujú do oboch polovíc. Každé rekurzívne volanie prebehne v čase $T(n/2)$ a zvyšok výpočtu vieme spraviť v čase $\Theta(n \log n)$. Dostávame teda, že funkcia T spĺňa nasledovný rekurentný vzťah: $T(n) = 2T(n/2) + \Theta(n \log n)$.
Toto je podobná rekurencia ako napr. pri triedení MergeSort, len tam máme jeden logaritmus navyše. Podobnou technikou ako pri analýze MergeSortu sa dá ukázať, že táto funkcia T patrí do triedy $\Theta(n(\log n)^2)$, to je teda celková časová zložitosť nášho riešenia.[1]

Listing programu (C++)

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

struct point { int id, x, y; };

bool compare_X(const point &A, const point &B) { return A.x < B.x; }
bool compare_Y(const point &A, const point &B) { return A.y > B.y; }

ostream& operator<<(ostream &os, const point &P) { return os << "(" << P.x << "," << P.y << ")"; }

long long count(vector<point> body) {
    int N = body.size();
    if (N <= 1) return 0;

    // rozdelime body na lavu a pravu polovicu, spravime rekurzivne volania
    sort( body.begin(), body.end(), compare_X );
    vector<point> lave( body.begin(), body.begin()+(N/2) );
```

[1] Jeden možný argument vyzerá nasledovne: Predstavme si všetky rekurzívne volania nášho algoritmu znázornené ako strom. Celkový čas behu algoritmu vieme zistiť tak, že sčítame počty krokov, ktoré spraví v každom z vrcholov tohto stromu. Toto sčitovanie môžeme robiť po vrstvách: koreň zvlášť, potom dokopy jeho oboch synov, potom dokopy všetky štyri vrcholy kde spracúvame $n/4$ bodov, a tak ďalej. Vrstiev má strom rekurzie zhruba $\log_2 n$ a na každej sa dokopy udeje nanajvýš rádovo $n \log n$ krokov výpočtu.

```
    vector<point> prave( body.begin()+(N/2), body.end() );
    int hranica = prave[0].x;
    long long answer = count(lave) + count(prave);

    // predpocitame pre kazdy bod v pravej casti pokial dohora mozeme
    sort( body.begin(), body.end(), compare_Y );
    vector<int> max_y(N, 1<<30);
    vector<int> visible;
    for (int n=N-1; n>=0; --n) {
        if (body[n].x < hranica) continue;
        while (!visible.empty() && body[n].x < body[visible.back()].x) {
            int b = visible.back(); visible.pop_back();
            max_y[b] = body[n].y;
        }
        visible.push_back(n);
    }

    // pocitame obdlzniky
    visible.clear();
    for (int n=0; n<N; ++n) {
        if (body[n].x < hranica) {
            // bod z lavej polovice
            while (!visible.empty() && body[n].x > body[visible.back()].x) visible.pop_back();
            visible.push_back(n);
        } else {
            // bod z pravej polovice
            int top = max_y[n];
            int lo = -1, hi = visible.size();
            while (hi - lo > 1) {
                int med = (lo + hi) / 2;
                if (body[visible[med]].y < top) hi = med; else lo = med;
            }
            answer += visible.size() - hi;
        }
    }
    return answer;
}

int main() {
    int N; cin >> N;
    vector<point> body(N);
    for (int n=0; n<N; ++n) cin >> body[n].x >> body[n].y;
    long long total = count(body);
    for (int n=0; n<N; ++n) body[n].x *= -1;
    total += count(body);
    cout << total << endl;
}
```

A-III-2 Horiaci strom

Ak je strom malý a konáre krátke, vieme celú úlohu riešiť dvomi vnorenými prehľadávaniami. Okrem zadaných vrcholov stromu si môžeme spraviť vrcholy aj pozdĺž každého konára tak, aby medzi susednými bol vždy jeden centimeter. Takto dostaneme nový strom s jednotkovými hranami. Šírenie ohňa na ňom vieme simulovať prehľadávaním do šírky. No a po každej sekunde simulácie (teda vždy, keď nasledujúci vrchol čakajúci vo fronte na spracovanie, má väčšiu vzdialenosť od začiatku ako naposledy spracovaný vrchol) sa môžeme pozrieť na ešte nehoriace časti stromu a pomocou druhého prehľadávania (tentokrát je jedno, či prehľadávame do šírky alebo do hĺbky) spočítať komponenty súvislosti – teda jednotlivé nehoriace oblasti.

Lepšie riešenie dostaneme tak, že si uvedomíme, že v predchádzajúcom riešení vôbec nie je potrebné simulovať každú sekundu šírenia ohňa. Počas sekúnd, v ktorých sa len oheň spokojne šíri po konároch stromu, sa nijak nemení počet nehoriacich oblastí. To sa môže stať len vtedy, keď oheň dohorí do nejakého z pôvodných vrcholov, alebo keď sa na nejakom konári stretnú dva v protismere horiace ohne. Tieto situácie budeme volať *udalosti*.

Môžeme si všimnúť, že strom s n vrcholmi má $n-1$ hrán, a teda dokopy na ňom môže nastať len $O(n)$ udalostí. Efektívnejšie ako pri prvom riešení by teda bolo simuláciu ohňa robiť tak, že len postupne zisťujeme, kedy a ktoré udalosti vlastne počas požiaru nastanú.

Takúto simuláciu vieme pomerne priamočiaro spraviť v kvadratickom čase, a to tak, že postupne identifikujeme všetky časy, v ktorých nejaká udalosť nastane. Keď máme situáciu v nejakom čase t_i (presnejšie, tesne po ňom), tak len postupne prezrieme všetky horiace hrany a o každej zistíme, kedy na nej nastane nasledujúca udalosť. Minimum z takto zistených časov bude čas t_{i+1} nasledujúcej udalosti.

Tesne po každom z časov t_i (vrátane času $t_0 = 0$, teda hneď po začiatku simulácie) potom spustíme prehľadávanie, aby sme spočítali aktuálny počet nehoriacich oblastí. Jedno takéto prehľadávanie vieme spraviť v lineárnom

čase – stačí si postupne ofarbovať ešte nehoriace vrcholy, pričom nehoriace vrcholy spojené nehoriacou hranou patria do toho istého komponentu. Na záver potom ešte prirátame jednu oblasť za každú hranu, ktorej už horia oba konce ale ešte nezhorela celá.
Toto riešenie strávi $O(n^2)$ času samotnou simuláciou horenia a počas nej $O(n)$-krát spustí prehľadávanie v čase $O(n)$ na spočítanie nehoriacich oblastí. Celková časová zložitosť je teda kvadratická: $O(n^2)$.

Vzorové riešenie 1: simulácia spredu

Na predchádzajúcom riešení zefektívnime dve veci. Aby sme vedeli rýchlejšie spracúvať udalosti, budeme si budúce udalosti, o ktorých už vieme, udržiavať v prioritnej fronte usporiadanej podľa času, kedy nastanú. A aby sme vedeli rýchlejšie povedať počet oblastí, rozmyslíme si, že ho vieme priamo prepočítať vždy, keď spracujeme nejaké udalosti.
Začneme tou druhou zmenou. Keď sa stretnú dva ohne niekde na hrane, prestane existovať oblasť medzi nimi, počet oblastí teda klesne o jednu. Keď dohorí oheň do nejakého dovtedy nehoriaceho vrcholu, vo všeobecnosti počet oblastí stúpne, keďže dovtedy súvislá oblasť sa rozpadne na niekoľko menších. Tu si len treba dať pozor na to, aby sme správne spracovali situácie, kedy viacero udalostí nastane naraz na tom istom mieste. Ak do vrcholu v stupňa d naraz dohorí oheň po k rôznych hranách, rozpadne sa pôvodná jedna oblasť na $d-k$ menších – jedna pre každú hranu, po ktorej sa z v začne oheň šíriť ďalej.
Teraz sa pozrime na efektívnejšie spracúvanie udalostí. Každú udalosť si popíšeme tromi údajmi: čas, kedy nastáva, jej typ (či ide o udalosť na hrane alebo vo vrchole) a identifikátor objektu, kde nastane. Usporadúvať udalosti v prioritnej fronte budeme postupne podľa všetkých parametrov. Potom vieme naraz spracovať všetky udalosti, ktoré sa udejú na tom istom mieste.

Listing programu (C++)

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

const int VRCHOL = 0, HRANA = 1, NEKONECNO = 987654321;

struct udalost { int cas, typ, id; };

bool operator<(const udalost &A, const udalost &B) {
    if (A.cas != B.cas) return A.cas < B.cas;
    if (A.typ != B.typ) return A.typ < B.typ;
    return A.id < B.id;
}
bool operator>(const udalost &A, const udalost &B) { return B<A; }

struct hrana { int k1, k2, d, z1, z2; }; // konce, dlzka, a casy kedy zacali konce horiet

void zapal(hrana &H, int u, int cas) { if (H.k1 == u) H.z1 = cas; else H.z2 = cas; }
bool horia_oba_konce(const hrana &H) { return (H.z1 < NEKONECNO) && (H.z2 < NEKONECNO); }
int kedy_dohori(const hrana &H) {
    int prvy = min(H.z1, H.z2), druhy = max(H.z1, H.z2);
    return prvy + (H.d - (druhy-prvy)) / 2;
}
int opacny_vrchol(const hrana &H, int u) { return H.k1 + H.k2 - u; }

int main() {
    // nacitame vstup
    int N, H; cin >> N >> H;
    vector<int> rodic(N,-1), dlzka(N,-1);
    for (int n=1; n<N; ++n) cin >> rodic[n];
    for (int n=1; n<N; ++n) cin >> dlzka[n];

    // zostrojime si graf
    vector<hrana> hrany(N-1);
    vector< vector<int> > graf(N);
    for (int n=0; n<N-1; ++n) {
        int u = n+1, v = rodic[n+1];
        hrany[n] = { u, v, dlzka[n+1], NEKONECNO, NEKONECNO };
        graf[u].push_back(n);
        graf[v].push_back(n);
    }

    // nachystame si datove struktury pre simulaciu
    int aktualnych_komponentov = 1;
    int najviac_komponentov = 1;
    int posledny_cas = 0;
    priority_queue< udalost, vector<udalost>, greater<udalost> > udalosti;
    vector<int> zacal_horiet(N, NEKONECNO);
```

```
    // zapalime zaciatocne vrcholy
    while (H--) {
        int v; cin >> v;
        zacal_horiet[v] = 0;
        aktualnych_komponentov += int( graf[v].size() ) - 1;
        for (int h : graf[v]) {
            zapal( hrany[h], v, 0 );
            if (horia_oba_konce(hrany[h])) {
                udalosti.push( { kedy_dohori(hrany[h]), HRANA, h } );
            } else {
                udalosti.push( { hrany[h].d, VRCHOL, opacny_vrchol(hrany[h],v) } );
            }
        }
    }
    najviac_komponentov = max( najviac_komponentov, aktualnych_komponentov );

    // spracuvame udalosti
    while (!udalosti.empty()) {
        auto U = udalosti.top(); udalosti.pop();
        if (U.cas > posledny_cas) {
            najviac_komponentov = max( najviac_komponentov, aktualnych_komponentov );
            posledny_cas = U.cas;
        }
        if (U.typ == VRCHOL) {
            // ak ide o udalost vo vrchole, skontrolujeme, ci ozaj nastala alebo zacal horiet skor
            if (U.cas > zacal_horiet[U.id]) continue;
            // ak vrchol zacal horiet teraz z viacerych smerov naraz, ubudaju komponenty
            if (U.cas == zacal_horiet[U.id]) { --aktualnych_komponentov; continue; }
            // ak vrchol este nezacal horiet, zapalime ho
            zacal_horiet[U.id] = U.cas;
            aktualnych_komponentov += int( graf[U.id].size() ) - 2;
            // a zapalime aj vsetky hrany veduce z neho
            for (int h : graf[U.id]) {
                zapal( hrany[h], U.id, U.cas );
                if (horia_oba_konce(hrany[h])) {
                    int t = kedy_dohori(hrany[h]);
                    if (t > U.cas) udalosti.push( { t, HRANA, h } );
                } else {
                    udalosti.push( { U.cas+hrany[h].d, VRCHOL, opacny_vrchol(hrany[h],U.id) } );
                }
            }
        } else {
            --aktualnych_komponentov;
        }
    }
    cout << najviac_komponentov << endl;
}
```

Vzorové riešenie 2: simulácia zozadu

Namiesto toho, aby sme ošetrovali udalosti, ktoré nastanú naraz, môžeme len „naprázdno“ spraviť vyššie popísanú simuláciu. Nebudeme sa počas nej zaujímať o nehoriace oblasti, len vygenerujeme usporiadaný zoznam všetkých udalostí, ktoré počas požiaru nastanú. V tomto zozname stačí mať zjednodušené udalosti „začal horieť tento vrchol“ a „dohorela niekde v strede táto hrana“.

Akonáhle máme tento zoznam udalostí, môžeme si akoby celý proces pustiť odzadu. Keď sa na film s požiarom pozeráme odzadu, vidíme nové dva typy udalostí: „vznikla nová oblasť na hrane“ a „prestal horieť vrchol, čím sa niekoľko oblastí spojilo dokopy“. Takéto typy udalostí vieme efektívne simulovať pomocou algoritmu union-find.[2] No a vždy, keď sa počas tejto simulácie zmení aktuálny čas, tak sa stačí pozrieť na aktuálny počet komponentov súvislosti.

Tento prístup k riešeniu má ešte aj tú výhodu, že v podstate bez zmeny ho vieme zovšeobecniť na ľubovoľné grafy. Tam by prvý typ riešenia už nefungoval, lebo len lokálne nevieme povedať, ako to, že začal horieť nový vrchol, zmenilo počet a tvar nehoriacich oblastí.

Obe verzie vzorového riešenia majú časovú zložitosť $O(n \log n)$. (Pri druhom vzorovom riešení je na to potrebné mať dostatočne efektívnu implementáciu algoritmu union-find.)

A-III-3 Pokazený rover spracúva polia

V podúlohách B a C budeme pre pohodlie predpokladať, že makrá `spoj`, `prvy` a `druhy` máme implementované tak, že výstupná lokalita môže byť totožná s niektorou vstupnou – teda že napríklad keď máme v lokalite `x` kód

[2] Popis tohto algoritmu nájdete napr. v riešení úlohy A-III-5 v 25. ročníku OI.

dvojice (a, b) vieme vykonať inštrukciu `prvy x x` a dostať tak priamo v lokalite `x` hodnotu a.
Makrá, ktoré túto vlastnosť nemajú, vieme do tejto podoby ľahko upraviť: najskôr uložíme výstup do pomocnej lokality a potom, na konci celého makra, prepíšeme príslušnú vstupnú lokalitu správnou výstupnou hodnotou.

Podúloha A: testovanie prvočíselnosti

V tejto podúlohe stačí vhodne použiť cykly a existujúce makrá. Ošetríme $a = 0$ a $a = 1$ ako špeciálne prípady. Pre $a \geq 2$ potrebujeme overiť, že a nie je deliteľné žiadnym z čísel od 2 po $a - 1$. Už máme makro pre delenie so zvyškom, potrebujeme ho len vedieť v cykle veľakrát použiť a zakaždým skontrolovať jeho výstup.

```
MAKRO prvocislo A B
            zapis A [A-kopia]
            vynuluj [D]
            vynuluj [nula]

            rovnake A [D] koniec            # otestujeme ci A=0 alebo A=1
            prenes K J [D] -
            rovnake A [D] koniec
            prenes K J [D] -                # v D mame teraz 2, ideme v cykle skusat ci D deli A

    cyklus: rovnake A [D] nasiel            # ziadne D < A nedelilo A => A je prvocislo
            zapis [A-kopia] A
            vydel A [D] [podiel] [zvysok]
            rovnake [zvysok] [nula] koniec  # nasli sme delitela => A nie je prvocislo
            prenes K J [D] -                # ideme na dalsie D
            skoc cyklus

    nasiel: prenes K J B -
    koniec: zapis [A-kopia] A
END
```

Podúloha B: pridaj na koniec

Pri rozmýšľaní o týchto úlohách pomôže, keď si každú premennú, v ktorej máme uloženú postupnosť čísel, predstavíme ako zásobník. Základné operácie, ktoré vieme ľahko robiť s postupnosťou, sú totiž veľmi podobné práve zásobníkovým operáciám: vieme sa ľahko pozrieť na prvý prvok postupnosti, odobrať ho, alebo naopak na začiatok pridať nový prvok. Na našu úlohu sa potom môžeme dívať ako na úlohu „napíš program, ktorý pridá nové číslo na spodok zásobníka“.
So zásobníkmi sa táto úloha rieši ľahko. Postupnosť prvok po prvku presunieme do druhého zásobníka, čím ju otočíme hore nohami. Jej pôvodný koniec je teraz na začiatku. Tam pridáme nový prvok na vrch zásobníka. No a potom všetky prvky presunieme späť a sme hotoví.
V našej implementácii si najskôr spravíme makrá `push` a `pop`, ktoré budú pracovať s postupnosťou ako so zásobníkom: `push` na začiatok postupnosti pridá novú hodnotu, `pop` odoberie z postupnosti jej prvý prvok do určenej premennej, a ak sa to nepodarí (postupnosť už bola prázdna), skočí na príslušné návestie.
Pomocou týchto makier spravíme samostatné makro `reverz`, ktoré obráti postupnosť, a pomocou neho potom definujeme makro `append`.

```
MAKRO push X prvok
            spoj prvok X X
            prenes K J X -
END

MAKRO pop X prvok nevyslo
            prenes X J K nevyslo    # ak X=0, snazime sa odobrat prvok z prazdnej postupnosti
            prvy X prvok
            druhy X X
END

MAKRO reverz X Y
            vynuluj Y
    cyklus: pop X [prvok] koniec    # odoberieme prvok alebo zistime, ze uz sa minuli
            push Y [prvok]
            skoc cyklus
    koniec: cakaj
END

MAKRO append X prvok
            reverz X [Z]
            push [Z] prvok
            reverz [Z] X
END
```

Podúloha C: triedenie

Jednou možnou cestou k riešeniu tejto úlohy je implementovať si makrá, ktoré nám umožnia pracovať s postupnosťou ako s plnohodnotným poľom – teda makro na čítanie prvku na konkrétnom indexe a makro na zápis na konkrétny index. Ani potom ešte nie je implementácia žiadneho triedenia priamočiara. Odporúčané triedenia tohto typu sú InsertSort alebo MaxSort. Implementovať rekurzívne triedenia ako QuickSort alebo MergeSort je možné ale výrazne bolestivé. V našom vzorovom riešení pôjdeme ešte menej bolestivou cestou: budeme implementovať vhodne upravený BubbleSort.

Pri našej implementácii sa úplne zaobídeme bez funkcií na prácu s poľom. Namiesto toho si všimnime, že výmeny po sebe idúcich prvkov vieme robiť aj počas robenia reverzu. Toto spravíme jednoducho: Namiesto toho, aby sme vždy zobrali prvok z jednej postupnosti a vložili ho do druhej sa vždy pozrieme na prvé dva prvky a do druhej postupnosti vložíme menší z nich.

Jeden takýto prechod postupnosťou presne zodpovedá jednému prechodu poľom počas BubbleSortu. (Teda až na to, že nám celú postupnosť reverzne. To ale ľahko napravíme druhým, tentokrát obyčajným reverzom.)

Ak chceme usporiadať n-prvkovú postupnosť, stačí vyššie popísanú procedúru zopakovať aspoň $(n-1)$-krát. (Na konci postupnosti sa nám postupne zbierajú najväčšie prvky v správnom poradí. Každý prechod postupnosťou ich nechá na mieste a navyše k nim pridá najväčší z prvkov, ktoré ešte neboli na správnom mieste.)

```
MAKRO minimum X Y Z                 # do Z ulozime minimum z hodnot X a Y
            zapis X [Xkopia]
            prenes [Xkopia] Y Z nevyslo
            skoc koniec
   nevyslo: zapis X Z
    koniec: cakaj
END

MAKRO maximum X Y Z                 # do Z ulozime maximum z hodnot X a Y
            vynuluj Z
            prenes K X Z -
            prenes K Y Z -
            minimum X Y [mensi]
            prenes Z [mensi] K -
END

MAKRO reverzsort X Y
            pop X [naboku] koniec   # zoberieme si prvy prvok pola nabok
    cyklus: pop X [prvok] uprac     # odoberieme dalsi prvok, ak uz nie je, skoncime
            minimum [naboku] [prvok] [mensi]
            maximum [naboku] [prvok] [vacsi]
            push Y [mensi]
            zapis [vacsi] [naboku]
            skoc cyklus
     uprac: push Y [naboku]         # vratime naspat do postupnosti prave odlozeny prvok
    koniec: cakaj
END

MAKRO sort X
            dlzka X [D]
    cyklus: prenes [D] J K koniec
            reverzsort X [Y]
            reverz [Y] X
            skoc cyklus
    koniec: cakaj
END
```

TRIDSIATY SIEDMY ROČNÍK OLYMPIÁDY V INFORMATIKE

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