A-I-1 Tancujúci kráľ

Slovom *tanečníci* budeme označovať všetkých $n-1$ ľudí okrem kráľa. Tanečníkov budeme volať *susední*, ak buď stoja priamo vedľa seba, alebo majú medzi sebou len kráľa.
Ak úloha má riešenie, musia v ňom byť všetci tanečníci usporiadaní podľa výšky. Predstavme si teraz, že sme si usporiadali výšky všetkých tanečníkov a medzi nich sme následne vložili na správne miesto kráľa. Ak kráľ pokazil usporiadanie (je prinízky alebo privysoký), vidíme, že naša úloha nemá riešenie.
Všimnime si teraz, že ľubovoľných dvoch susedných tanečníkov vieme vymeniť pomocou tanečnej figúry, ktorá zahŕňa len ich dvoch a prípadne kráľa. Všetkých tanečníkov teda určite vieme z ich úvodného poradia preusporiadať do ľubovoľného iného. A špecificky teda platí, že ich vždy vieme preusporiadať tak, aby všetci tanečníci boli v usporiadanom poradí.
Už teda vieme za úlohu získať štyri body: algoritmom BubbleSort (ktorý si vystačí s výmenami susedných prvkov) môžeme usporiadať všetkých tanečníkov, potom skontrolovať, či kráľ nekazí usporiadané poradie, a podľa toho buď vypísať NIE, alebo postupnosť výmen, ktoré sme spravili.

Riešenie bez kráľa

BubbleSort na usporiadanie x-prvkovej postupnosti potrebuje v najhoršom prípade až $x(x-1)/2$ výmen, a to je omnoho viac, ako si my môžeme dovoliť. Lepšie riešenia úlohy budú musieť vhodne používať aj reverzy dlhších úsekov poľa.
V tejto časti riešenia si ukážeme, že ľubovoľný úsek x tanečníkov, ktorý neobsahuje kráľa, vieme usporiadať pomocou nanajvýš $x-1$ vhodne zvolených reverzov.
Hlavná myšlienka tohto postupu je jednoduchá. Prvým reverzom dostaneme najmenší prvok na začiatok poľa, druhým reverzom druhý najmenší prvok na druhú pozíciu, a tak ďalej. Keď správne umiestnime predposledný prvok, posledný už určite bude na svojom mieste.
Presnejšie bude každá iterácia postupu vyzerať nasledovne: Prejdeme celý nespracovaný zvyšok poľa (teda všetky prvky okrem tých, ktoré sme už skôr umiestnili na správne miesto) a nájdeme v ňom minimum. Následne spravíme reverz úseku od začiatku nespracovanej časti po pozíciu, na ktorej sme aktuálne minimum našli – čím ho presunieme na miesto, kam patrí.

Ľahké riešenie za 8 bodov

Vyššie uvedeným postupom usporiadame zvlášť všetkých tanečníkov naľavo od kráľa a zvlášť všetkých napravo. (Toto vyžaduje dokopy nanajvýš $n-3$ reverzov.)
Tesne naľavo od kráľa je teraz niekoľko tanečníkov, ktorí svojou výškou patria napravo. A tesne napravo od neho sú zase tanečníci, ktorí sú prinízki na to, aby boli v pravej časti. Nájdeme tento úsek a jedným reverzom všetkých tanečníkov presunieme na správnu stranu.
Na záver znova usporiadame zvlášť ľavých a zvlášť pravých tanečníkov. Toto riešenie si teda určite vystačí s nanajvýš $2n-5$ reverzmi.

Riešenie za plný počet bodov

Podobne ako v predchádzajúcom riešení budeme rozlišovať *ľavú časť* (pozície naľavo od kráľa, kde majú skončiť tanečníci od kráľa nižší) a *pravú časť* (kam patria tí vyšší).
Na plný počet bodov potrebujeme riešenie, ktoré nikdy nespraví viac ako $3n/2$ reverzov. Nižšie si ukážeme jedno takéto riešenie. V tomto riešení začneme tým, že sa pozrieme, ktorá časť je menšia. Tú vyriešime ako prvú, pričom vždy použijeme nanajvýš dva reverzy na to, aby sme jedného tanečníka dostali na správne miesto. Dlhšiu časť potom usporiadame už známym postupom, ktorý použije nanajvýš jeden reverz na každého tanečníka.
Pozície $k-i$ a $k+i$ budeme volať *zrkadlové*. Ak spravíme reverz úseku, ktorého konce sú zrkadlové pozície, vymeníme tým ich obsah. (A tiež obsah všetkých dvojíc zrkadlových pozícií medzi nimi, ale to nám bude jedno.)
Bez ujmy na všeobecnosti predpokladajme, že ľavá časť je kratšia. Postupne pre každú pozíciu od 1 po $k-1$ spravíme nasledovné:

- Ak je prvok, ktorý na túto pozíciu patrí, v pravej časti: spravíme v pravej časti reverz, ktorým ho dostaneme na zrkadlovú pozíciu od tej správnej. Následne spravíme reverz úseku od správnej pozície po

jej zrkadlovú, čím dostaneme aktuálny prvok na správne miesto.

- Inak: spravíme v ľavej časti reverz, ktorým aktuálny prvok rovno dostaneme na správne miesto.

Nižšie je znázornený príklad toho, ako dostávame na správne miesto prvky 1, 2 a 3, predstavujúce troch najnižších tanečníkov. Písmeno K predstavuje kráľa, bodky sú ostatní tanečníci, pomlčky ukazujú úsek, ktorý reverzujeme. Všimnite si, že akonáhle niekoho úmyselne umiestnime na správne miesto, už ním zaručene nikdy nebudeme hýbať.

```
..3..2K..........1.
            ------    1 na zrkadlovú pozíciu
..3..2K.....1......
-------------         1 na správne miesto
1.....K2..3........
       -----          2 na zrkadlovú pozíciu
1.....K.3..2.......
 -----------          2 na správne miesto
12..3.K............
  ---                 3 rovno na správne miesto
123...K............
```

Pri implementácii je dôležité si uvedomiť, že maximálne n je malé (nanajvýš 1000). V prvom rade to znamená, že si môžeme naozaj dovoliť spraviť a priamočiaro odsimulovať všetky potrebné reverzy. No a v druhom rade to znamená, že aj operácie, ktoré by išli robiť efektívnejšie, vôbec nemusíme robiť efektívne. Napríklad krok, kedy chceme každému tanečníkovi priradiť jeho výslednú pozíciu, by sme síce mohli implementovať s časovou zložitosťou $O(n \log n)$, omnoho pohodlnejšie je však spraviť to kvadraticky.

Listing programu (C++)

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

int main() {
    // nacitame vstup
    int N, K, L;
    cin >> N >> K >> L;
    --K; // v programe indexujeme od nuly
    vector<int> tanecnici(N);
    for (int &x : tanecnici) cin >> x;

    // overime, ci existuje riesenie
    vector<int> zaver = tanecnici;
    sort( zaver.begin(), zaver.end() );
    if (zaver[K] != tanecnici[K]) { cout << "NIE\n"; return 0; }

    // explicitne priradime kazdemu prvku miesto, kam patri
    vector<int> stav(N, -1);
    stav[K] = K;
    for (int n=0; n<N; ++n) if (n != K) {
        for (int i=0; i<N; ++i) if (tanecnici[i] == zaver[n] && stav[i] == -1) {
            stav[i] = n;
            break;
        }
    }

    // zistime, ktora strana je kratsia, a podla toho si zvolime,
    // od ktoreho konca ideme prvky umiestnovat na spravne miesto
    int start = 0, step = 1, end = N;
    if (K > N-1-K) { start = N-1; step = -1; end = -1; }

    // postupne pre kazde n umiestnime spravny prvok na miesto n
    vector< pair<int,int> > kroky;
    for (int n=start; n!=end; n+=step) {
        if (stav[n] == n) continue; // uz je tam
        int where = 0;
        while (stav[where] != n) ++where;
        if ( (where-K)*(n-K) < 0 ) {
            // prvok mame na nespravnej strane krala
            int mirror = 2*K - n;

            int i = min(mirror,where), j=max(mirror,where)+1;
```

```
            kroky.push_back( {i, j} );
            reverse( stav.begin()+i, stav.begin()+j );

            i = min(mirror,n); j = max(mirror,n)+1;
            kroky.push_back( {i, j} );
            reverse( stav.begin()+i, stav.begin()+j );
        } else {
            // prvok mame na spravnej strane krala
            int i = min(n,where), j = max(n,where)+1;
            kroky.push_back( {i, j} );
            reverse( stav.begin()+i, stav.begin()+j );
        }
    }

    // vypiseme vystup
    cout << "ANO\n";
    cout << kroky.size() << "\n";
    for (const auto &krok : kroky) cout << (krok.first+1) << "␣" << krok.second << "\n";
}
```

A-I-2 Prominencia

Ukážeme si dve možné riešenia: jedno priamočiarejšie, v ktorom vezmeme definíciu zo zadania a ukážeme si, ako ju efektívne implementovať, a jedno, v ktorom si problém mierne preformulujeme.
V oboch riešeniach začneme tým, že už počas načítania vstupu zahodíme po sebe idúce opakujúce sa hodnoty. Takto dostaneme nové pohorie, ktoré má presne rovnaký tvar ako to na vstupe, ale navyše každý vrchol je teraz len bod, nie úsek.

Riešenie podľa definície

Zistiť, ktoré vrcholy sú najvyššie na svete, je ľahké. Aby sme určili prominenciu pre ostatné vrcholy, potrebujeme pre každý z nich vedieť efektívne povedať dve veci: Najskôr potrebujeme vedieť, kde je smerom doľava aj doprava najbližší ostro vyšší bod terénu. A keď to už vieme, tak potrebujeme vedieť, ako najhlbšie medzi nimi klesneme. Inými slovami, na našom poli nadmorských výšok potrebujeme vedieť efektívne robiť operácie „nájdi najbližšiu väčšiu hodnotu v danom smere“ a „nájdi minimum úseku“.
Efektívne nájsť minimum úseku vieme rôznymi spôsobmi, zväčša zahŕňajúcimi nejaké to množstvo predpočítania pomocných informácií. Jedným z najjednoduchších spôsobov je použitie *intervalového stromu*. Viac o intervalových stromoch sa dočítaš napríklad tu: https://www.ksp.sk/kucharka/intervalovy_strom/
Aj hľadanie najbližšej väčšej hodnoty vieme efektívne spraviť pomocou intervalového stromu – len tentokrát takého, kde si v každom vrchole pamätáme maximum úseku, ktorý tento vrchol predstavuje. Existuje však aj ešte efektívnejšie riešenie, ktoré je zároveň implementačne jednoduchšie. Keď chceme o každom políčku poľa vedieť, kde naľavo od neho je najbližšie väčšie, stačí pole raz prejsť zľava doprava a priebežne si vhodné údaje pamätať v zásobníku.
Presnejšie to celé bude fungovať nasledovne: predstavme si rovnako ako v zadaní, že čísla v poli predstavujú výšky po sebe idúcich bodov pohoria. Postupne zľava doprava ich budeme spracúvať. Predstavte si teraz, že sa na už spracovanú časť tohto pohoria pozeráme sprava. Niektoré body vidíme, niektoré sú zakryté neskôr spracovanými. Tie, ktoré vidíme, majú spoločnú vlastnosť: napravo od neho sú zatiaľ len samé hodnoty od neho menšie. Práve tieto body si budeme pamätať v zásobníku. Keď na pravom konci pohoria pribudne ďalší bod, najbližší naľavo, ktorý je od neho väčší, musí nutne byť jeden z tých bodov, ktoré si pamätáme.
Spracovanie každého ďalšieho bodu bude detailnejšie vyzerať nasledovne: Kým máme na vrchu zásobníka bod s nanajvýš rovnakou výškou ako práve spracúvaný, vyhodíme ho. (Tieto body pri pohľade sprava práve prestali byť viditeľné.) Bod, ktorý nám ostal na vrchu zásobníka, je pre práve spracúvaný najbližším väčším vľavo. Na záver už len pridáme práve spracovaný bod do zásobníka ako najnižší spomedzi aktuálne sprava viditeľných.
Spracovanie konkrétneho bodu síce môže trvať dlho (ak toho zo zásobníka postupne potrebujeme veľa vybrať), ľahko však nahliadneme, že dokopy bude celé spracovanie poľa trvať len čas priamo úmerný jeho dĺžke. Totiž každý prvok práve raz do zásobníka vložíme a každý prvok nanajvýš raz niekedy zo zásobníka vyberieme.
Dokopy teda strávime čas $O(n)$ tým, že pre každý prvok poľa nájdeme najbližší väčší vľavo (a potom aj vpravo spustením toho istého algoritmu na obrátené pole), potom čas $O(n)$ tým, že si nad poľom postavíme minimový intervalový strom, no a na záver pre každý vrchol v čase $O(\log n)$ pomocou intervalového stromu zistíme, ako

hlboko treba klesnúť, aby sme sa dostali k nejakému vyššiemu vrcholu. Toto riešenie má teda dokopy časovú zložitosť $O(n \log n)$.

Listing programu (C++)

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

vector<int> get_bigger_left(const vector<int> &H) {
    int N = H.size();
    vector<int> answer(N,-1), idx(1,-1), hei(1,1000000007);
    for (int n=0; n<N; ++n) {
        while (hei.back() <= H[n]) { hei.pop_back(); idx.pop_back(); }
        answer[n] = idx.back();
        hei.push_back(H[n]); idx.push_back(n);
    }
    return answer;
}

vector<int> get_bigger_right(const vector<int> &H) {
    vector<int> Hcopy = H;
    reverse( Hcopy.begin(), Hcopy.end() );
    vector<int> answer = get_bigger_left(Hcopy);
    reverse( answer.begin(), answer.end() );
    for (int &x : answer) x = H.size() - 1 - x;
    return answer;
}

struct intervalac {
    int N, L;
    vector< vector<int> > T;

    int min_query(int lo, int hi, int r=-1, int c=-1, int tlo=-1, int thi=-1) {
        if (r == -1) { r=0; c=0; tlo=0; thi=1<<L; }
        if (thi <= lo || hi <= tlo) return 1000000007;
        if (lo <= tlo && thi <= hi) return T[r][c];
        return min(
            min_query( lo, hi, r+1, 2*c,   tlo, (tlo+thi)/2 ),
            min_query( lo, hi, r+1, 2*c+1, (tlo+thi)/2, thi )
        );
    }

    intervalac(const vector<int> &H) {
        N = H.size();
        L = 0;
        while ((1 << L) < N) ++L;
        T.resize(L+1);
        for (int l=0; l<=L; ++l) T[l].resize(1<<l, 1000000007);
        for (int n=0; n<N; ++n) T[L][n] = H[n];
        for (int l=L-1; l>=0; --l) for (int i=0; i<(1<<l); ++i)
            T[l][i] = min( T[l+1][2*i], T[l+1][2*i+1] );
    }
};

int main() {
    // nacitame vstup a odstranime po sebe iduce duplikaty
    int N;
    cin >> N;
    vector<int> pohorie;
    for (int n=0; n<N; ++n) {
        int vyska;
        cin >> vyska;
        if (n == 0 || vyska != pohorie.back()) pohorie.push_back(vyska);
    }
    N = pohorie.size();
    int everest = *max_element( pohorie.begin(), pohorie.end() );

    // najdeme pre kazdy bod najblizsi vacsi vlavo aj vpravo, predpocitame minima
    vector<int> BL = get_bigger_left(pohorie), BR = get_bigger_right(pohorie);
    intervalac T(pohorie);

    // pre kazdy vrchol najdeme a vypiseme jeho prominenciu
    for (int kto=0; kto<N; ++kto) {
        if (kto-1 >= 0 && pohorie[kto-1] >= pohorie[kto]) continue;
        if (kto+1 <  N && pohorie[kto+1] >= pohorie[kto]) continue;
        if (pohorie[kto] == everest) { cout << everest << "\n"; continue; }
        int left = 1000000007, right = 1000000007;
        if (BL[kto] != -1) left = pohorie[kto] - T.min_query(BL[kto]+1,kto);
        if (BR[kto] != N) right = pohorie[kto] - T.min_query(kto+1,BR[kto]);
        cout << min(left,right) << "\n";
    }
}
```

Riešenie trochu ináč

Predstavme si, že voda v mori stúpla tak, že zaliala celé naše pohorie. Čo sa teraz bude diať, keď bude voda postupne klesať? Každý vrchol sa niekedy vynorí z vody ako samostatný ostrov. No a ako voda ďalej klesá, občas sa nám stane, že sa niektoré ostrovy spoja. Tieto okamihy nás budú zaujímať. Totiž vždy, keď nejaký ostrov pripojíme k inému s vyšším vrcholom, našli sme práve to sedlo, ktoré určuje prominenciu vrcholu na prvom ostrove. (Kým bola voda vyššie, nevedeli sme sa z ostrova dostať. Keď voda klesla až sem, zrazu sa vieme dostať na vyššie položené miesta.)

Vyššie popísaný postup budeme chcieť efektívne simulovať. Samozrejme, prvým krokom ku efektívnej simulácii je uvedomiť si, že klesanie vody nemusíme simulovať ako spojitý proces. Stačí postupne odsimulovať jednotlivé *udalosti* – teda okamihy, kedy sa niečo zaujímavé stane. Inými slovami, stačí, keď zoberieme všetky body zadané na vstupe a usporiadame ich podľa výšky od najvyššieho. Takto dostaneme poradie, v ktorom sa budú vynárať z vody.

Keď sa konkrétny bod vynorí, stane sa jedna z troch možných udalostí: vznikne nový ostrov (ak sa vynoril vrchol), nejaký ostrov sa o políčko zväčší, alebo sa dva ostrovy spoja. Ak sa spoja dva ostrovy s rôznou maximálnou výškou, dozvedeli sme sa práve prominenciu najvyšších bodov na nižšom z nich.

Aby sme vedeli toto celé robiť efektívne, spravíme to nasledovne: Pre každé políčko si budeme pamätať, ktorý ostrov (ak nejaký) ním aktuálne začína a ktorý ním končí. Pre každý ostrov si budeme pamätať, kde má svoj začiatok, koniec a najvyššie body.

Keď sa teraz vynorí nové políčko, stačí sa pozrieť o políčko dopredu a dozadu. Podľa toho, či tam končí, resp. začína nejaký ostrov, zistíme, ktorá situácia nastala, a spracujeme ju.

Toto riešenie má časovú zložitosť $O(n \log n)$ kvôli úvodnému usporiadaniu bodov pohoria podľa výšky. Celú simuláciu vynárania ostrovov vieme potom už spraviť v lineárnom čase.

Listing programu (C++)

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

struct ostrov {
    int zac, kon; // index aktualneho zaciatku a konca
    int vyska; // vyska najvyssieho bodu
    list<int> maxima; // indexy najvyssich bodov
};

int main() {
    // nacitame vstup a odstranime po sebe iduce duplikaty
    int N;
    cin >> N;
    vector<int> pohorie;
    for (int n=0; n<N; ++n) {
        int vyska;
        cin >> vyska;
        if (n == 0 || vyska != pohorie.back()) pohorie.push_back(vyska);
    }
    N = pohorie.size();

    // usporiadame vsetky body vstupu podla vysky od najvyssej
    vector< pair<int,int> > udalosti;
    for (int n=0; n<N; ++n) udalosti.push_back( { pohorie[n], n } );
    sort( udalosti.begin(), udalosti.end() );
    reverse( udalosti.begin(), udalosti.end() );

    // inicializujeme prominenciu pre najvyssie hory sveta
    vector<int> prominencia(N, 0);
    int everest = *max_element( pohorie.begin(), pohorie.end() );
    for (int n=0; n<N; ++n) if (pohorie[n] == everest) prominencia[n] = everest;

    // simulujeme postupne vynaranie ostrovov a pocitame prominenciu ostatnych vrcholov
    vector<ostrov> ostrovy;
    vector<int> tu_zacina(N,-1), tu_konci(N,-1);
    for (const auto &rec : udalosti) {
        int kde = rec.second;
        bool ma_pred = (kde > 0 && tu_konci[kde-1] != -1);
        bool ma_za = (kde < N-1 && tu_zacina[kde+1] != -1);
        if (!ma_pred && !ma_za) {
            // vynoril sa novy ostrov
            ostrovy.push_back( {kde, kde, pohorie[kde], {kde} } );
            tu_zacina[kde] = tu_konci[kde] = ostrovy.size()-1;
            continue;
        }
        if (!ma_za) {
            // predlzujeme predchadzajuci ostrov doprava
            int o = tu_konci[kde-1];
            tu_konci[kde-1] = -1;
```

```
            tu_konci[kde] = o;
            ostrovy[o].kon = kde;
            continue;
        }
        if (!ma_pred) {
            // predlzujeme nasledujuci ostrov dolava
            int o = tu_zacina[kde+1];
            tu_zacina[kde+1] = -1;
            tu_zacina[kde] = o;
            ostrovy[o].zac = kde;
            continue;
        }
        // spajame dva ostrovy dokopy
        int o1 = tu_konci[kde-1], o2 = tu_zacina[kde+1];
        tu_konci[kde-1] = tu_zacina[kde+1] = -1;
        if (ostrovy[o1].vyska == ostrovy[o2].vyska) {
            // len spojime dokopy mnoziny miest kde maju maxima
            ostrovy[o1].maxima.splice( ostrovy[o1].maxima.end(), ostrovy[o2].maxima );
        } else {
            // prezije ten ostrov ktory je vyssi, druheho maximam nastavime prominenciu
            if (ostrovy[o1].vyska < ostrovy[o2].vyska) swap(o1, o2);
            for (int x : ostrovy[o2].maxima) prominencia[x] = pohorie[x] - pohorie[kde];
        }
        ostrovy[o1].zac = min(ostrovy[o1].zac, ostrovy[o2].zac);
        ostrovy[o1].kon = max(ostrovy[o1].kon, ostrovy[o2].kon);
        tu_zacina[ ostrovy[o1].zac ] = tu_konci[ ostrovy[o1].kon ] = o1;
    }

    // vypiseme vystup
    for (int x : prominencia) if (x > 0) cout << x << "\n";
}
```

Riešenie v lineárnom čase

Obe predchádzajúce riešenia sú dostatočne rýchle na to, aby získali plný počet bodov. Ak by nás ale zaujímalo ešte efektívnejšie riešenie našej súťažnej úlohy, za cenu možno trochu komplikovanejšej implementácie vieme túto úlohu riešiť aj v lineárnom čase.
Upravíme naše prvé riešenie tak, aby počas prechodu zľava doprava pomocou zásobníka určilo pre každý bod pohoria nielen najbližší vyšší bod naľavo, ale aj minimum na ceste doň. Toto vieme spraviť nasledovne: spolu s postupnosťou bodov, ktoré máme momentálne v zásobníku, si pre každý úsek medzi nimi budeme pamätať aj minimum na ňom. Keď spracúvame nový bod pohoria a postupne vyhadzujeme niektoré staré body zo zásobníka, stačí potom zobrať minimum z výšky aktuálneho bodu a miním všetkých úsekov, ktoré sme práve odstránili, a dostaneme tak minimum nového úseku – toho, ktorý teraz máme medzi práve spracovaným bodom a tým bodom, ktorý je teraz bezprostredne pod ním v zásobníku.

A-I-3 Strelec

Úlohu budeme riešiť pomocou techniky dynamického programovania. Postupne pre každý počet ťahov i a každé súradnice (j, k) vypočítame hodnotu $P[i, j, k]$: počet spôsobov, ktorými sa vieme dostať na políčko (j, k) s tým, že sme zatiaľ navštívili v správnom poradí prvých i písmen slova w.

Niektoré tieto hodnoty sú zjavné. Označme $S[j, k]$ písmeno na políčku (j, k) na šachovnici. Ak $S[j, k] \neq w[i]$, tak zjavne $P[i, j, k] = 0$. Slovne: po preskákaní prvých i písmen slova w vieme byť na políčku len vtedy, ak obsahuje i-te písmeno slova w.
Tiež je zjavné, že ak $S[j, k] = w[1]$, tak $P[1, j, k] = 1$: jediný spôsob, ako navštíviť prvé písmeno slova w, je začať na ňom.

No a ako vypočítame tie zvyšné hodnoty $P[i, j, k]$? Skončiť po i ťahoch na políčku (j, k) vieme tak, že spravíme prvých $i - 1$ ťahov, tými sa dostaneme na nejaké iné políčko a z toho následne i-tym ťahom prídeme na políčko (j, k). Spôsoby, ktoré nás tam dovedú z rôznych políčok, sú zjavne navzájom rôzne. Celkový počet spôsobov teda dostaneme jednoducho tak, že sčítame počty spôsobov pre všetky možnosti posledného ťahu.
Formálne, $P[i, j, k]$ vieme spočítať ako súčet všetkých $P[i - 1, j', k']$ takých, že strelec vie jedným ťahom prejsť z (j', k') na (j, k).
Riešením našej úlohy je potom súčet všetkých hodnôt $P[n, \star, \star]$, kde n je dĺžka slova w.

Keďže strelec sa pohybuje len šikmo, na šachovnici rozmerov $r \times s$ máme nanajvýš $2\min(r,s) - 2$ možností, odkiaľ prísť na konkrétne políčko. Pri šikovnej implementácii teda každú z hodnôt $P[i,j,k]$ vieme vypočítať v čase $O(\min(r,s))$. Celková časová zložitosť tohto riešenia je teda $O(n \cdot r \cdot s \cdot \min(r,s))$.

Lepšie riešenie

Vyššie popísané zlepšenie vieme ešte zefektívniť, a to tak, že budeme šikovnejšie sčitovať možnosti, ako sa na políčko dostať. Časovú zložitosť tým zlepšíme na $O(nrs)$.

Políčkom so súradnicami (j,k) prechádzajú dve uhlopriečky. Jednu z nich tvoria tie políčka, ktoré majú súčet súradníc $j+k$, druhú tvoria tie, ktoré majú rozdiel súradníc $j-k$.

Označme si $A[i,x]$ celkový počet spôsobov, ako byť po i ťahoch na uhlopriečke so súčtom x, a $B[i,y]$ celkový počet spôsobov, ako byť po i ťahoch na uhlopriečke s rozdielom y.
Keď už poznáme všetky hodnoty $P[i,\star,\star]$, vieme všetky hodnoty $A[i,\star]$ a $B[i,\star]$ spočítať v čase $O(rs)$. Stačí postupne prejsť cez všetky políčka (j,k) a hodnotu $P[i,j,k]$ pripočítať ku $A[i,j+k]$ aj ku $B[i,j-k]$.
(Alebo ekvivalentne: postupne prejsť cez všetky uhlopriečky a pre každú cez všetky jej políčka. Každé políčko spracujeme raz v každom smere. Prvá nami opísaná konštrukcia sa ľahšie implementuje, obe majú ale rovnakú asymptotickú časovú zložitosť.)
No a keď poznáme všetky hodnoty $A[i-1,\star]$ a $B[i-1,\star]$, vieme pomocou nich v čase $O(rs)$ vypočítať všetky hodnoty $P[i,\star,\star]$, teda hodnoty pre jednotlivé políčka po nasledujúcom ťahu. Totiž počet spôsobov, ako sa dostať na konkrétne políčko (j,k) po i ťahoch, vieme teraz vyjadriť ako počet spôsobov, ako sa dostať po $i-1$ ťahoch na jednu z uhlopriečok, ktorá cez neho prechádza. (Od celkového počtu spôsobov, ako sa dostať na konkrétnu uhlopriečku, musíme zakaždým odčítať spôsoby, kedy sme už po $i-1$ ťahoch presne na políčku (j,k), keďže na mieste ostať nesmieme.)

Listing programu (Python)

```
from collections import defaultdict

R, C = [ int(_) for _ in input().split() ]
W = input()
board = [ input() for r in range(R) ]

# zistime pocet sposobov (0 alebo 1) pre prve pismeno
P = [ [ int( board[r][c] == W[0] ) for c in range(C) ] for r in range(R) ]

# postupne pre kazde dalsie pismeno scitame uhlopriecky
# a pomocou nich zistime nove pocty
for n in range(1,len(W)):
    A, B = defaultdict(int), defaultdict(int)
    for r in range(R):
        for c in range(C):
            A[r+c] += P[r][c]
            B[r-c] += P[r][c]

    newP = [ [ 0 for c in range(C) ] for r in range(R) ]
    for r in range(R):
        for c in range(C):
            if W[n] != board[r][c]: continue
            newP[r][c] = A[r+c] + B[r-c] - 2*P[r][c]

    P = newP

print( sum( sum(row) for row in P ) )
```

V implementácii sme pre lepšiu čitateľnosť použili asociatívne polia (`defaultdict`), aby sme nemuseli ošetrovať záporné indexy v poli B. Stačilo by samozrejme namiesto $B[j-k]$ používať $B[(j-k)+(s-1)]$ a mohli by A aj B byť obyčajné polia.
Taktiež sme v implementácii kvôli lepšej čitateľnosti vynechali počítanie modulo 10^9+7.

A-I-4 Pokazený rover

Riešenia podúloh uvádzame v rovnakom poradí ako v zadaní.

Podúloha A: vynásob jamu štyrmi

Počet kamienkov v lokalite `jama` vieme zdvojnásobiť tak, že z kameňolomu do jamy prenesieme toľko kamienkov, koľko je práve v jame. Keď toto dvakrát zopakujeme, máme želaný výsledok. V programe to vyzerá nasledovne:

```
prenes K jama jama -
prenes K jama jama -
```

Podúloha B: porovnaj počty kamienkov

V lokalite A máme a kamienkov, v lokalite B ich je b.
Otestovať, či $a \geq b$, vieme nasledovne: Z lokality A skúsime odniesť do kameňolomu toľko kamienkov, koľko ich je v lokalite B. Ak sa to nepodarí, vieme, že $a < b$. Ak sa to podarí, vieme, že $a \geq b$. V tomto prípade môžeme následne do lokality A pridať späť toľko kamienkov, koľko ich je v lokalite B, aby sme mali zachované pôvodné počty kamienkov.
Test na rovnosť vieme spraviť tak, že najskôr otestujeme, či $a \geq b$, a ak áno, tak otestujeme aj či $b \geq a$.

```
prenes A B K koniec
prenes K B A -
prenes B A K koniec
prenes K A B -
prenes K J C -
```

To isté riešenie vieme zapísať aj stručnejšie – namiesto do kameňolomu môžeme prenášať kamienky naspäť rovno na tú kôpku, z ktorej ich berieme. Tým je zaručené, že počet kamienkov sa nej nezmení. Rover však stále vyhodnotí, či sa operácia podarila, a podľa toho skočí alebo neskočí na príslušné návestie. (Tento trik bol uvedený v Príklade 1 v študijnom texte.)

```
prenes A B A koniec
prenes B A B koniec
prenes K J C -
```

Podúloha C: makro pre delenie so zvyškom

V lokalitách A a B máme vstup: hodnoty a a b, pričom $b > 0$. Do lokality P chceme uložiť celú časť podielu a/b a do lokality Z zvyšok pri tomto delení.
Postup bude jednoduchý. Na začiatku skopírujeme obsah lokality A do lokality Z. Následne budeme v cykle z lokality Z odoberať vždy B kamienkov naraz. Za každé úspešné odobratie pridáme jeden kamienok do lokality P.

Malo by byť zjavné, že tento postup časom skončí (počet kamienkov v lokalite Z sa zmenšuje) a keď sa tak stane, v lokalite P máme počet úspešných odobratí (čiže presne celú časť podielu a/b) a v lokalite Z nám zvýšili presne kamienky predstavujúce zvyšok pri tomto delení.
V nižšie uvedenom programe používame makrá `pridaj`, `skoc` a `cakaj` definované v študijnom texte.

```
MAKRO vydel A B P Z
            pridaj A Z
    cyklus: prenes Z B K koniec
            prenes K J P -
            skoc cyklus
    koniec: cakaj
END
```

Podúloha D: najväčší spoločný deliteľ

V lokalitách A a B máme nejaké neznáme počty kamienkov $a, b > 0$. Chceme vypočítať a do prázdnej lokality C uložiť ich najväčšieho spoločného deliteľa. Použijeme Euklidov algoritmus. Ten je založený na dvoch pozorovaniach. Prvé je zjavné: pre $x > 0$ platí $nsd(x, 0) = x$. Druhé: $nsd(x, y) = nsd(y, x \bmod y)$.

Zdôvodníme si druhé pozorovanie. Ak $x < y$, tak jednoducho tvrdí $nsd(x, y) = nsd(y, x)$, čo zjavne platí. Nech teraz $x \geq y$. Hodnotu x môžeme zapísať v tvare $x = py + z$, kde p je celá časť podielu x/y a z je zvyšok po tomto delení. Potom naše pozorovanie hovorí, že $nsd(x, y) = nsd(y, z)$.
Prečo tento vzťah platí? Každé d, ktoré delí x aj y, musí deliť aj $x - py$, čiže z. No a každé d, ktoré delí y aj z, musí deliť aj $py + z$, čiže x.

Celý Euklidov algoritmus je teraz veľmi jednoduchý: kým sú obe čísla kladné, používame druhé pozorovanie, a keď jedno klesne na nulu, použijeme prvé pozorovanie a sme hotoví.
(Môžeme si všimnúť, že pre $x \geq y$ sa každým použitím nášho druhého pravidla súčet aktuálnych dvoch hodnôt zmenší, takže tento algoritmus určite časom skončí. V tejto súťažnej úlohe nás netrápi jeho presná časová zložitosť.)

```
        # vynuluje X
        MAKRO vynuluj X
            prenes X X K -
        END

        # ak X obsahuje nulu, skoci na navestie N
        MAKRO nulove X N
            prenes X J X N
        END

        # hodnotu X zapise do premennej Y, pricom prepise povodny obsah Y
        MAKRO zapis X Y
            vynuluj Y
            pridaj X Y
        END

        # hlavny program: ak B=0 koncime, inak zmenime (A, B) na (B, A mod B)
cyklus: nulove B koniec
        vynuluj Z
        vydel A B odpad Z
        zapis B A
        zapis Z B
        skoc cyklus
koniec: zapis A C
```

V programe používame makro `vydel` z podúlohy C, pričom sa dopúšťame jednej nepresnosti: premennú `odpad`, kam sa ukladá celá časť podielu, na nič nepotrebujeme, a tak sa ju ani neobťažujeme medzi jednotlivými volaniami `vydel` vyprázdniť.

Podúloha D po druhé

Pripomeňme si na záver ešte raz, že v tejto súťažnej úlohe nás netrápi časová zložitosť programov, len ich korektnosť. Úlohu o najväčšom spoločnom deliteľovi preto môžeme vyriešiť aj nasledovne: Vieme, že $nsd(a, b) \leq b$. Postupne pre každé i od b smerom dole ku 1 otestujeme, či i delí aj a aj b. Prvé takéto i bude zjavne rovné $nsd(a, b)$.

```
        # ak X nie je delitelne Y, skoci na navestie chyba
        MAKRO nedelitelne X Y chyba
                    vydel X Y [podiel] [zvysok]
                    nulove [zvysok] delilo
                    skoc chyba
            delilo: cakaj
        END

        # hlavny program:
        pridaj B I
cyklus: nedelitelne A I zmensi
        nedelitelne B I zmensi
        # ak sme sa dostali sem, I deli A aj B a teda je to nsd(A,B)
        pridaj I C
        skoc koniec
        # ak sme skocili na navestie zmensi, zmensime I o jedna a hladame dalej
zmensi: prenes I J K -
        skoc cyklus
```

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 2021