A-III-4 Laser

Začneme tým, že spravíme niekoľko pozorovaní, ktoré nám neskôr uľahčia život.
Pozorovanie prvé: Vždy musí existovať optimálne riešenie, v ktorom sa laserový lúč odrazí od každého zrkadla len jedenkrát. Ak by sme totiž mali riešenie, v ktorom sa od toho istého zrkadla odrazí lúč dvakrát, vieme z toho riešenia celý úsek medzi prvým a druhým odrazom vynechať – buď tak, že na dotyčné políčko dáme opačne otočené zrkadlo, alebo tak, že tam to zrkadlo vôbec nedáme. Toto nové riešenie použije nanajvýš toľko zrkadiel ako to pôvodné.
Na zrkadlo sa môžeme dívať ako na možnosť zmeniť na danom políčku smer o 90 stupňov, teda zatočiť doľava alebo doprava. Keďže už vieme, že stačí hľadať riešenie, kde na každom políčku zmeníme smer najviac raz, hľadáme vlastne cestu od lasera k fotoreceptoru, ktorá najmenejkrát zmení smer.

Pozorovanie druhé: Uvažujme trochu inú úlohu: namiesto laserového lúča majme robota, ktorý sa môže zadarmo hýbať dopredu a za jeden peniaz sa otočiť doľava alebo doprava. Potom najlacnejšia cesta robota do cieľa zodpovedá najmenšiemu počtu zrkadiel, ktoré potrebujeme v našej pôvodnej úlohe.
V čom je medzi týmito dvomi úlohami rozdiel? Po našom prvom pozorovaní už ostáva len jeden rozdiel: skutočnosť, že robot sa (za dva peniaze) vie na jednom políčku otočiť aj do protismeru, čo ale v našej úlohe so zrkadlami spraviť nevieme.
Tu si ale stačí uvedomiť, že optimálne riešenie úlohy s robotom otočku o 180 stupňov nikdy nepoužije. Ak by sa robot niekedy otočil o 180 stupňov, znamená to, že sa nejaký čas bude vracať po trase, ktorú už predtým prešiel – až kým nepríde na miesto, kde sa od nej zase odpojí. Potom by ale bolo lacnejšie, keby celú zachádzku vynechal a na danom mieste sa len rovno otočil do smeru, ktorým chce ísť ďalej.

Z vyššie uvedených pozorovaní teda vyplýva, že nám stačí nájsť cestu pre robota, ktorý najmenejkrát zatočí, a potom z nej zostrojiť riešenie našej úlohy tak, že na miesta, kde zatočil, umiestnime správne natočené zrkadlá.

Kubické riešenie

Pre jednoduchosť predpokladajme, že stôl má rozmery $n \times n$. Náš robot sa môže nachádzať v nanajvýš $4n^2$ stavoch: je n^2 možností, kde stojí, a štyri možnosti, ktorým smerom sa pozerá. Na stavový priestor sa môžeme dívať ako na graf. Jednotlivé stavy budú vrcholy a hrany budú zodpovedať úsekom prejdeným medzi jednotlivými zatočeniami. Presnejšie, keď sme v konkrétnom stave, náš ďalší pohyb bude vyzerať tak, že spravíme niekoľko (1 alebo viac) krokov dopredu a potom (na políčku, kde tak smieme spraviť) zatočíme doľava alebo doprava. Každej takejto možnosti bude zodpovedať jedna hrana.
V takto postavenom grafe teda máme $O(n^2)$ vrcholov a z každého vedie $O(n)$ hrán. Každá hrana zodpovedá jednému zatočeniu, na hľadanie najlepšieho spôsobu, ako dosiahnuť jednotlivé stavy, môžeme použiť obyčajné prehľadávanie do šírky (BFS) zo začiatočného stavu.
Implementačný detail: keď skúšame všetky možnosti, kam sa ďalej pohnúť z konkrétneho stavu, a podarí sa nám dosiahnuť cieľ, zjavne sme práve našli najlacnejší spôsob, ako to spraviť. Rovno v tej chvíli teda prehľadávanie prerušíme a spätným prechodom zostrojíme jednu optimálnu cestu.
Vyššie popísané riešenie v najhoršom prípade prezrie celý graf, jeho časová zložitosť je teda $O(n^3)$.

Kvadratické riešenie

Graf na priestore stavov vieme vybudovať aj šikovnejšie. Stavy/vrcholy budeme mať tie isté ako vo vyššie popísanom riešení, hrany však pridáme ináč. Z každého vrcholu povedú nanajvýš tri hrany: hrana s dĺžkou 0 zodpovedajúca jednému kroku dopredu a (ak sa na danom mieste vieme otočiť) hrany s dĺžkou 1 zodpovedajúce otočeniu doľava a doprava. Naďalej zjavne platí, že najkratšia cesta zo štartu do cieľa v tomto grafe zodpovedá najlepšiemu riešeniu našej pôvodnej úlohy.
Keďže tentokrát máme hrany dĺžok 0 a 1, na nájdenie najkratšej cesty nevieme použiť obyčajné prehľadávanie do šírky, vieme ho však vhodne upraviť. V ľubovoľnom grafe, ktorého dĺžky hrán sú 0 a 1, vieme najkratšie cesty z jedného vrcholu do všetkých ostatných vypočítať nasledovným algoritmom:

- Pre každý vrchol si pamätáme najmenšiu vzdialenosť doň, ktorú sme už objavili. Na začiatku máme vzdialenosť 0 pre vrchol, kde začíname, a vzdialenosť ∞ pre všetky ostatné vrcholy.

- Aby sme vedeli na konci aj zostrojiť najkratšiu cestu zo štartu do cieľa, pre každý vrchol si pamätáme, odkiaľ sme doň najkratšou cestou prišli.
- Vrcholy čakajúce na spracovanie nebudeme mať uložené v obyčajnej fronte, ale v tzv. obojstrannej fronte (deque). Tej vieme efektívne pridávať aj odoberať vrcholy na oboch jej koncoch. Na začiatku do fronty zaradíme začiatočný vrchol.
- Rovnako ako v klasickom BFS, kým sa nám fronta nevyprázdni, dokola opakujeme cyklus, v ktorom vyberieme prvý vrchol z fronty a spracujeme ho.
- Spracovanie jedného vrcholu vyzerá nasledovne: Postupne prejdeme všetky hrany, ktoré z neho vedú. Pre každú hranu sa pozrieme, či pomocou nej nevieme zlepšiť vzdialenosť do jej koncového vrchola. Ak áno, zapíšeme si pre ten vrchol jeho novú vzdialenosť a zaradíme ho do fronty na spracovanie.

  Ale pozor, dôležitá zmena: Ak zaraďujeme do fronty vrchol, do ktorého sme prišli hranou dĺžky 1, zaradíme ho klasicky, teda na koniec fronty. Ale ak sme prišli hranou dĺžky 0, zaradíme ho *hneď na začiatok.*

Prečo tento algoritmus funguje a aká je jeho časová zložitosť?
Matematickou indukciou si môžeme dokázať, že vo fronte čakajúcej na spracovanie sú vždy nanajvýš dve skupiny vrcholov: najskôr nejaké vrcholy, do ktorých sa vieme dostať cestou „aktuálnej“ dĺžky x, potom možno nejaké vrcholy, do ktorých sa vieme dostať cestou dĺžky $x + 1$. Tento invariant platí práve vďaka spôsobu, akým nové vrcholy do fronty zaraďujeme.
V každom kroku algoritmu teda spracujeme vrchol, ktorý má spomedzi všetkých nespracovaných najmenšiu doterajšiu vzdialenosť. No a ľahko nahliadneme, že tá už musí byť definitívna – všetky vrcholy s menšou vzdialenosťou sme už totiž spracovali a všetky hrany vedúce z nich ďalej sme už tým pádom prezreli. Z toho teda vyplýva, že keď náš algoritmus prehlási o nejakom vrchole, že je spracovaný, poznáme naozaj správnu dĺžku najkratšej cesty doň.
No a už si len treba uvedomiť, že každý vrchol sa môže vo fronte na spracovanie ocitnúť nanajvýš *dvakrát.* Áno, dvakrát, nie raz. Môže sa napr. stať, že keď spracúvame vrchol v_1, ktorý je vo vzdialenosti 7 od začiatku, objavíme spôsob, ako hranou dĺžky 1 dosiahnuť vrchol w. Vtedy nastavíme vrcholu w vzdialenosť na 8 a zaradíme ho na koniec fronty. Neskôr však môžeme spracovať iný vrchol v_2 a z neho vrchol w dosiahnuť hranou dĺžky 0. Ak sa toto stane, zmenšíme vrcholu w vzdialenosť na 7 a zaradíme ho do fronty na spracovanie druhýkrát – tentokrát však na jej začiatok.
To, že máme vo fronte na spracovanie vrchol w dvakrát, nám nijak neprekáža – pri druhom spracovaní vrcholu w sa už jednoducho nič nestane.
Tento algoritmus teda každý vrchol grafu aj každú hranu grafu nanajvýš dvakrát spracuje, a teda je jeho časová zložitosť lineárna od veľkosti grafu. V našom prípade máme graf s $O(n^2)$ vrcholmi aj hranami, časová zložitosť tohto riešenia je teda tiež $O(n^2)$.
(Na záver dodáme, že vyššie uvedená argumentácia platí pre úplne ľubovoľný graf s hranami dĺžok 0 a 1. Náš graf je navyše špeciálny: pre každý vrchol platí, že všetky cesty doň majú rovnakú paritu dĺžky, lebo tá je jednoznačne určená otočením v ňom. Každý vrchol bude teda do fronty na spracovanie zaradený dokonca len raz.)

Listing programu (C++)

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

const int MAXRS = 2500;
const int DR[] = { 0, 1, 0, -1 };
const int DS[] = { 1, 0, -1, 0 };

struct stav { int r, s, d; };

int vzdialenost[MAXRS][MAXRS][4];
stav odkial[MAXRS][MAXRS][4];
char zrkadlo[MAXRS][MAXRS][4];

#define INDEX(pole,cstav) pole[cstav.r][cstav.s][cstav.d]

int main() {
```

```
    // načítaj mapu
    int R, S;
    cin >> R >> S;
    vector<string> mapa(R);
    for (int r=0; r<R; ++r) cin >> mapa[r];

    // nájdi štartovací stav a vlož ho do fronty
    stav start;
    for (int r=0; r<R; ++r) if (mapa[r][0] == '-') { start.r=r; start.s=0; start.d=0; }
    for (int r=0; r<R; ++r) if (mapa[r][S-1] == '-') { start.r=r; start.s=S-1; start.d=2; }

    for (int r=0; r<R; ++r) for (int s=0; s<S; ++s) for (int d=0; d<4; ++d) vzdialenost[r][s][d] = 987654321;
    INDEX(vzdialenost,start) = 0;
    deque<stav> Q;
    Q.push_back(start);

    // prehľadávaj do šírky
    while (!Q.empty()) {
        stav kde = Q.front();
        Q.pop_front();

        // zisti, či nie sme v cieli, a ak áno, zostroj a vypíš riešenie
        if (mapa[ kde.r ][ kde.s ] == '=') {
            while (kde.r != start.r || kde.s != start.s || kde.d != start.d) {
                if (INDEX(zrkadlo,kde) != '.') mapa[kde.r][kde.s] = INDEX(zrkadlo,kde);
                kde = INDEX(odkial,kde);
            }
            for (string row : mapa) cout << row << "\n";
            return 0;
        }

        // vyskúšaj tri možnosti pohybu: rovno, cez zrkadlo /, cez zrkadlo \.
        for (int pohyb=0; pohyb<3; ++pohyb) {
            stav kam = kde;
            int dlzka = 0;
            if (pohyb == 0) { kam.r += DR[ kde.d ]; kam.s += DS[ kde.d ]; }
            if (pohyb == 1) { kam.d = 3 - kde.d; dlzka = 1; }
            if (pohyb == 2) { kam.d = kde.d ^ 1; dlzka = 1; }

            if (mapa[ kam.r ][ kam.s ] == 'X') continue;
            if (mapa[ kam.r ][ kam.s ] == '-') continue;
            if (pohyb != 0 && mapa[ kam.r ][ kam.s ] == 'O') continue;

            if (INDEX(vzdialenost,kde) + dlzka >= INDEX(vzdialenost,kam)) continue;
            INDEX(vzdialenost,kam) = INDEX(vzdialenost,kde) + dlzka;
            INDEX(odkial,kam) = kde;
            INDEX(zrkadlo,kam) = "./\\"[pohyb];
            if (dlzka == 0) Q.push_front(kam); else Q.push_back(kam);
        }
    }
    cout << "nemozne\n";
}
```

A-III-5 Graffiti

Za úlohu je pomerne ľahké získať nejaké body hrubou silou: jednoducho simulujeme proces popísaný v zadaní, pričom si o každom panele múru pamätáme, či je momentálne pomaľovaný.

Ak umelci maľujú len na jeden panel

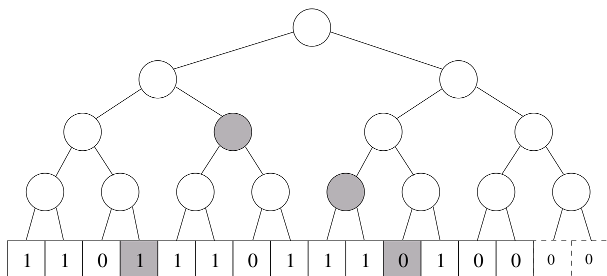
Pre vstupy, v ktorých každý umelec pomaľuje len jeden panel, je ľahké aj efektívne riešenie. Stačí si napríklad v jednej usporiadanej množine pamätať všetky čisté panely a v druhej všetky pomaľované panely. Vždy, keď príde nový umelec, nájdeme v čase $O(\log p)$ prvý čistý panel. A vždy, keď príde poriadková služba, nájdeme v $O(\log p)$ v druhej množine úsek pomaľovaných panelov, ktoré treba vyčistiť, a jeden po druhom ich postupne presunieme medzi čisté.

Dokopy všetkých umelcov spracujeme v čase $O(u \log p)$. No a keďže každú pomaľovanú stenu, ktorú niekedy poriadková služba vyčistí, musel predtým niekto zamaľovať, dokopy za celý beh programu vyčistí služba nanajvýš u stien, a teda aj všetky udalosti týkajúce sa poriadkovej služby vieme spracovať v čase $O(u \log p)$.

(Všimnite si, že je nutné pri spracúvaní poriadkovej služby vedieť, ktoré panely v jej úseku sú pomaľované, a spracovať len tie. Ak by sme sa pozerali na úplne každý panel v čistenom úseku, narástla by časová zložitosť na $O(up)$.)

Všeobecné riešenie

V našom vzorovom riešení použijeme na reprezentáciu múru tzv. intervalový strom: úplný binárny strom, ktorého listy zodpovedajú jednotlivým panelom múru. Vnútorné vrcholy tohto stromu potom zodpovedajú dlhším úsekom múru. Ak listy stromu tvoria úroveň 0, ich rodičia úroveň 1, a tak ďalej, tak každý vrchol na úrovni i predstavuje nejaký konkrétny úsek 2^i po sebe idúcich panelov.
Počet listov tohto stromu je najbližšou mocninou dvoch väčšou alebo rovnou p. (Použijeme len p najľavejších z nich, ostatné sú akoby celý čas pomaľované.) Je zjavné, že listov je menej ako $2p$. Hĺbka intervalového stromu je teda $O(\log p)$.



Na obrázku je príklad intervalového stromu. Pole v spodnom riadku predstavuje listy stromu, teda úroveň 0. Hodnoty 0 a 1 predstavujú pomaľované a prázdne panely.
Hlavné použitie intervalového stromu je nasledovné: ľubovoľný úsek múru vieme rozdeliť na $O(\log p)$ kratších (a navzájom disjunktných) úsekov, z ktorých každý zodpovedá nejakému vrcholu intervalového stromu. Ak teda chceme niečo zistiť o nejakom úseku, prípadne si o ňom zapísať nejakú informáciu, nemusíme v čase $O(p)$ postupne po jednom prejsť všetky jeho políčka, ale stačí v čase $O(\log p)$ nájsť príslušné políčka v strome. Na obrázku sú sivou farbou označené vrcholy stromu, ktoré dokopy reprezentujú úsek začínajúci štvrtým a končiaci jedenástym panelom.

Rozmyslime si, aké operácie potrebujeme vedieť s naším stromom robiť:

- Pre dané d nájsť najľavejší výskyt d voľných panelov vedľa seba.
- Celý daný úsek označiť ako zafarbený, alebo naopak ako čistý.

Aby sme vedeli robiť prvú z nich efektívne, budeme si v každom vrchole stromu pamätať tri hodnoty:

- Aký najdlhší úsek čistých panelov sa nachádza niekde v úseku panelov, ktoré tento vrchol zastupuje?
- Koľkými čistými panelmi jeho úsek panelov začína a koľkými končí?

Tieto informácie si vieme ľahko udržiavať aktuálne. Ak ich poznáme pre oboch synov nejakého vrcholu v, ľahko z nich vypočítame, čo má byť zapísané vo v. (Napr. najdlhší čistý úsek maximom troch možností: najdlhší úsek pod ľavým synom, najdlhší úsek pod pravým synom, alebo úsek, ktorý tvoria čisté panely na konci úseku ľavého syna a na začiatku úseku pravého syna.)
No a pomocou týchto informácií vieme ľahko nájsť najľavejší dostatočne dlhý čistý úsek: stačí začať v koreni stromu a vždy si vybrať najľavejšiu možnosť ktorá ešte vedie k riešeniu.
Aby sme vedeli efektívne značiť úseky ako čisté alebo zafarbené, použijeme techniku nazývanú *lenivé robenie zmien* (lazy updates). Každému vrcholu pridáme ešte jednu premennú, ktorú nazveme stav. Každý vrchol môže byť v jednom z troch stavov: aktuálny, čistý, alebo špinavý. Aktuálny vrchol má v sebe uložené informácie vo vyššie popísanej podobe. Čistý vrchol je vrchol, v ktorom máme poznačené, že všetky vrcholy pod ním sú čisté. No a špinavý vrchol je vrchol, v ktorom máme poznačené, že všetky vrcholy pod ním sú zamaľované.
Keď teda meníme stav panelov (či už umelec niektoré pomaľoval alebo poriadková služba niektoré vyčistila), jediné, čo urobíme, je, že zmeníme stav vrcholov, ktoré tomu úseku zodpovedajú (a prepočítame informácie vo vrcholoch nad nimi).
Takáto zmena môže spôsobiť, že niekde pod čistými/špinavými vrcholmi sú vrcholy, v ktorých sú neaktuálne informácie. Toto je ale práve podstata lenivých zmien – v tejto chvíli si len povieme, že nám tento stav neprekáža. Predstavme si totiž, že ideme od koreňa ľubovoľným smerom dodola po našom intervalovom strome. Skôr, než

prídeme do vrcholu, v ktorom sú neaktuálne informácie, vždy narazíme na príslušný čistý/špinavý vrchol a tam už vieme, na čom sme a nemusíme ísť hlbšie.

Občas sa nám stane, že z nejakého čistého alebo špinavého vrcholu v predsa len potrebujeme ísť ďalej dodola. Napr. máme poznačené, že vrchol predstavujúci nejakých 8 panelov je špinavý, ale teraz prišla nová požiadavka, že treba prvé tri z nich očistiť. Až v tejto situácii prepošleme informáciu ďalej: ak bol v špinavý, tak najskôr oboch synov v zmeníme na špinavých, potom rekurzívne zídeme dole do jedného z nich, urobíme potrebné zmeny, a keď sa z rekurzie vynoríme, tak prepočítame (teraz už aktuálny) vrchol v pomocou informácie v jeho synoch. V našom príklade by teda výsledkom bola situácia, v ktorej by ľavý syn v skončil tiež aktuálny (a pamätali by sme si v ňom, že sú pod ním tri čisté panely), zatiaľ čo pravý syn v by skončil špinavý. Z týchto údajov by sme následne prepočítali v.

Každú operáciu so stromom vieme takto zrealizovať v čase $O(\log p)$, spracovanie u udalostí teda zvládneme s časovou zložitosťou $O(u \log p)$.

Na záver dodávame, že existujú aj iné riešenia s touto časovou zložitosťou. Iné riešenie je napr. založené na tom, že múr si budeme reprezentovať ako množinu súvislých úsekov čistých panelov. Tie si budeme udržiavať vo vyvažovanom binárnom strome usporiadané podľa začiatku. Navyše si v každom vrchole stromu budeme pamätať, aký najdlhší úsek leží v jeho podstrome.

Listing programu (C++)

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

const int CISTY = 0, SPINAVY = 1, AKTUALNY = 2;
const int HLBKA = 19;

struct vrchol {
    int zaciatok, koniec, najviac, stav;
    void zapln(int level, int s) { stav=s; zaciatok = koniec = najviac = (s==CISTY ? (1<<level) : 0); }
};

vector< vector<vrchol> > T;

void prepocitaj_vrchol(int level, int offset) {
    if (level == 0) return;
    vrchol &ja = T[level][offset], lsyn = T[level-1][2*offset], psyn = T[level-1][2*offset+1];
    ja.stav = AKTUALNY;
    ja.zaciatok = lsyn.zaciatok;
    if (ja.zaciatok == 1 << (level-1)) ja.zaciatok += psyn.zaciatok;
    ja.koniec = psyn.koniec;
    if (ja.koniec == 1 << (level-1)) ja.koniec += lsyn.koniec;
    ja.najviac = max( max(lsyn.najviac, psyn.najviac), lsyn.koniec+psyn.zaciatok );
}

void preposli_dole(int level, int offset) {
    if (level == 0) return;
    vrchol &ja = T[level][offset], &lsyn = T[level-1][2*offset], &psyn = T[level-1][2*offset+1];
    if (ja.stav == AKTUALNY) return;
    lsyn.zapln(level-1,ja.stav);
    psyn.zapln(level-1,ja.stav);
    prepocitaj_vrchol(level,offset);
}

int najdi_prvy_usek(int dlzka, int level=HLBKA, int offset=0) {
    preposli_dole(level,offset);
    if (T[level][offset].najviac < dlzka) return -1;
    if (level == 0) return offset;
    int tmp = najdi_prvy_usek(dlzka,level-1,2*offset);
    if (tmp != -1) return tmp;
    vrchol lsyn = T[level-1][2*offset], psyn = T[level-1][2*offset+1];
    if (lsyn.koniec + psyn.zaciatok >= dlzka) return (1<<(level-1))*(2*offset+1) - lsyn.koniec;
    return najdi_prvy_usek(dlzka,level-1,2*offset+1);
}

void zaznac_usek(int lo, int hi, int stav, int level=HLBKA, int offset=0, int vlo=0, int vhi=1<<HLBKA) {
    preposli_dole(level,offset);
    if (hi <= vlo || vhi <= lo) return;
    if (lo <= vlo && vhi <= hi) { T[level][offset].zapln(level,stav); return; }
    zaznac_usek(lo,hi,stav,level-1,2*offset,vlo,(vlo+vhi)/2);
    zaznac_usek(lo,hi,stav,level-1,2*offset+1,(vlo+vhi)/2,vhi);
    prepocitaj_vrchol(level,offset);
}

int main() {
    int P, U;
    cin >> P >> U;
```

```
    T.resize(HLBKA+1);
    T[0].resize( 1<<HLBKA, {0,0,0,AKTUALNY} );
    for (int p=1; p<=P; ++p) T[0][p] = {1,1,1,AKTUALNY};
    for (int h=0; h<=HLBKA; ++h) {
        T[h].resize( 1<<(HLBKA-h) );
        for (int o=0; o<(1<<(HLBKA-h)); ++o) prepocitaj_vrchol(h,o);
    }

    while (U--) {
        string cmd; cin >> cmd;
        if (cmd == "U") {
            int d; cin >> d;
            int ans = najdi_prvy_usek(d);
            cout << ans << "\n";
            if (ans != -1) zaznac_usek(ans,ans+d,SPINAVY);
        } else {
            int x, y; cin >> x >> y;
            zaznac_usek(x,y+1,CISTY);
        }
    }
}
```

A-III-6 Korálky

Našou úlohou je pomôcť Natálke s jej korálkovým problémom. Začnime nie príliš efektívnym algoritmom.

Skúšanie všetkých možností

Nazvime neklesajúcu postupnosť koráliek *náhrdelník*. Pre dostatočne malé n si môžeme vygenerovať všetky validné náhrdelníky a vybrať najdlhší z nich. Označme si postupnosť na vstupe F. Všetky dobré náhrdelníky vieme vygenerovať napríklad pomocou rekurzívnej funkcie $f(ind, lavy, pravy)$, kde ind je index čísla, ktoré ideme spracovať a $lavy$ a $pravy$ sú indexy poslednej korálky už navlečenej na šnúrku zľava a sprava. Táto funkcia postupne vyskúša všetky možnosti, ako náhrdelník dokončiť, a vráti dĺžku najdlhšej z nich. V tele funkcie f stačí vyskúšať tri možnosti a vybrať z nich maximum.

- $f(ind + 1, lavy, pravy)$ (preskočíme číslo na pozícii ind)
- $f(ind + 1, ind, pravy) + 1$, ak $F[ind] < F[lavy]$ (pridáme číslo na pozícii ind na začiatok)
- $f(ind + 1, lavy, ind) + 1$, ak $F[ind] > F[pravy]$ (pridáme číslo na pozícii ind na koniec)

Teraz už stačí zavolať funkciu $f(i + 1, i, i)$, pre každé $i < n$ a vybrať maximum. Časová zložitosť tohto riešenia je $O(3^n)$, pretože máme n pozícii a v každej sa naša rekurzia môže rozdeliť do troch vetiev.
Toto riešenie však vieme vylepšit pridaním memoizácie. Argumenty funkcie f sú ohraničené počtom čísel na vstupe a každý stav vieme vypočítať v konštantnom čase. Ak teda každý stav vypočítame len raz a zapamätáme si jeho riešenie v tabuľke, dostaneme časovú zložitosť priamo úmernú počtu rôznych volaní funkcie f, teda $O(n^3)$.

Vzorové riešenie

Vzorové riešenie je založené na nasledovnej myšlienke: Niektorú korálku sme museli na náhrdelník navliecť ako prvú. Ak by sme vedeli číslo i a teda farbu $F[i]$ tejto prvej korálky, rozdelil by sa nám problém na dve nezávislé časti. O každej z nasledujúcich korálok totiž vieme, že ak je menšia ako $F[i]$, môžeme ju navliecť len zľava, a ak je väčšia ako $F[i]$, tak len sprava.
No a keďže aj zľava aj sprava chceme korálok navliecť čo najviac, potrebujeme medzi „ľavými“ korálkami nájsť čo najdlhšiu klesajúcu a medzi „pravými“ korálkami čo najdlhšiu rastúcu podpostupnosť. Implementovať stačí jeden z týchto dvoch algoritmov. Totiž ak v nejakom poli zmeníme všetkým prvkom znamienko (vynásobíme ich -1), tak sa klesajúce podpostupnosti zmenia na rastúce a naopak.
Takto dostávame riešenie s časovou zložitosťou $O(n^2 \log n)$. Stačí postupne vyskúšať všetky možnosti pre index i prvej navlečenej korálky a zakaždým dvomi volaniami algoritmu z domáceho kola nájsť najdlhšiu klesajúcu a rastúcu podpostupnosť, ktoré k dotyčnej korálke vieme pridať.

Vyššie popísané riešenie však stále robí veľa práce zbytočne veľakrát.

Keď sa lepšie pozrieme na algoritmus z domáceho kola, môžeme si všimnúť, že keď spracujeme konkrétny index, dozvieme sa dĺžku najdlhšej rastúcej podpostupnosti, ktorá na tomto indexe *končí*. (Táto dĺžka zodpovedá indexu políčka, ktoré meníme.)
My teraz tento algoritmus použijeme na zadanú postupnosť F, ale *v opačnom smere, teda od konca*. Tým sa pre každý index dozvieme dĺžku najdlhšej *klesajúcej* podpostupnosti, ktorá na ňom *začína*.
Následne spravíme to isté, ale pre postupnosť F s opačnými znamienkami. Tým sa pre každý index dozvieme aj dĺžku najdlhšej *rastúcej* podpostupnosti, ktorá na ňom začína.
No a keď máme tieto dve informácie, už ľahko úlohu vyriešime v lineárnom čase: pre každý index i vieme v konštantnom čase povedať, koľko najviac korálok vieme použiť, ak začneme korálkou i.
Celková časová zložitosť tohto riešenia je teda len $O(n \log n)$.

Listing programu (C++)

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

vector<int> LIS(const vector<int> &A) {
    // odpoved[i] = dĺžka najdlhšej rastúcej podpostupnosti končiacej na indexe i
    vector<int> odpoved;
    vector<int> konce(1,-(1<<30));
    for (int a : A) {
        auto kam = upper_bound( konce.begin(), konce.end(), a );
        odpoved.push_back( kam - konce.begin() );
        if (kam == konce.end()) konce.push_back(a); else *kam = a;
    }
    return odpoved;
}

int main() {
    int N;
    cin >> N;
    vector<int> F(N);
    for (int n=0; n<N; ++n) cin >> F[n];
    reverse( F.begin(), F.end() );
    vector<int> lds = LIS(F);
    for (int &f : F) f = -f;
    vector<int> lis = LIS(F);

    int odpoved = 0;
    for (int n=0; n<N; ++n) odpoved = max( odpoved, lis[n]+lds[n]-1 );
    cout << odpoved << endl;
}
```

TRIDSIATY TRETÍ ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Eduard Batmendijn, Michal Forišek, Samuel Gurský, Samuel Sládek, Emanuel Tesař
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2018