# 67. ročník Matematické olympiády – 2017/2018

*Řešení úloh ústředního kola kategorie P – 2. soutěžní den*

---

## P-III-4 Laser

Na úvod provedeme několik pozorování, která nám později usnadní práci.

*Pozorování první:* Vždy musí existovat optimální řešení, v němž se laserový paprsek odrazí od každého zrcadla jenom jednou. Kdybychom totiž měli řešení, ve kterém se od nějakého zrcadla odrazí paprsek dvakrát, můžeme z tohoto řešení celý úsek mezi prvním a druhým odrazem vynechat – buď tak, že na dotyčné políčko dáme opačně otočené zrcadlo, nebo tak, že tam to zrcadlo vůbec nedáme. Toto nové řešení použije nejvýše tolik zrcadel, jako řešení původní.

Na zrcadlo se můžeme dívat jako na možnost změnit na daném políčku směr o 90 stupňů, tedy zatočit doleva nebo doprava. Když už víme, že stačí hledat řešení, v němž na každém políčku změníme směr nejvýše jednou, hledáme vlastně cestu od laseru k fotoreceptoru, která nejméně-krát změní směr.

*Pozorování druhé:* Uvažujme trochu jinou úlohu: místo laserového paprsku máme robota, který se zdarma pohybuje vpřed a za jeden peníz se může otočit doleva nebo doprava. Potom nejlevnější cesta robota ze startu do cíle odpovídá nejmenšímu počtu zrcadel, která potřebujeme použít v naší původní úloze.

V čem je rozdíl mezi oběma úlohami? Po našem prvním pozorování zůstává už jen jeden rozdíl – robot se (za dva peníze) dokáže na jednom políčku otočit i do protisměru, což ale v původní úloze se zrcadly není možné.

Stačí si však uvědomit, že optimální řešení úlohy s robotem otočku o 180 stupňů nikdy nepoužije. Kdyby se robot někdy otočil o 180 stupňů, znamenalo by to, že se po nějakou dobu bude vracet po trase, kterou už předtím prošel, dokud nepřijde na místo, kde se z ní zase odpojí. Potom by ale bylo určitě levnější, kdyby celou tuto zacházku vynechal a na daném místě by se rovnou otočil do směru, kterým chce jít dále.

Z uvedených pozorování tedy plyne, že nám stačí nalézt cestu pro robota, který nejméně-krát zatočí. Z ní potom sestrojíme řešení naší úlohy tak, že na místa, kde robot zatočil, umístíme správně natočená zrcadla.

### Kubické řešení

Pro jednoduchost předpokládejme, že stůl má rozměry $n \times n$. Náš robot se může nacházet v nejvýše $4n^2$ stavech: máme $n^2$ možností, kde stojí, a pro každou z nich čtyři možnosti, kterým směrem se dívá. Na stavový prostor se můžeme dívat jako na graf. Vrcholy grafu představují jednotlivé stavy, hrany grafu odpovídají úsekům cesty mezi jednotlivými zatočeními. Přesněji řečeno, když jsme v nějakém stavu, náš další pohyb bude vypadat tak, že provedeme několik (jeden nebo více) kroků dopředu a potom (na políčku, kde smíme) zatočíme doleva nebo doprava. Každé takovéto možnosti bude odpovídat jedna hrana.

V takto postaveném grafu tedy máme $\mathcal{O}(n^2)$ vrcholů a z každého vede $\mathcal{O}(n)$ hran. Každá hrana odpovídá jednomu zatočení. K nalezení nejlepšího způsobu, jak dosáhnout jednotlivé stavy, můžeme použít obyčejné prohledávání do šířky (BFS) z počátečního stavu.

Implementační detail: když zkoušíme všechny možnosti, kam se lze dále pohnout z konkrétního stavu, a podaří se nám při tom dosáhnout cíle, zjevně jsme právě našli nejlevnější způsob, jak to udělat. V takovém okamžiku proto prohledávání přerušíme a zpětným průchodem sestrojíme jednu optimální cestu.

Výše popsané řešení v nejhorším případě projde celý graf, jeho časová složitost je proto $\mathcal{O}(n^3)$.

**Kvadratické řešení**

Graf na prostoru stavů můžeme vybudovat šikovněji. Stavy/vrcholy budeme mít stejné jako v předcházejícím řešení, hrany do grafu ale přidáme jiným způsobem. Z každého vrcholu povedou nejvýše tři hrany: hrana s délkou 0 odpovídající jednomu kroku vpřed a (pokud se na daném místě smíme otočit) hrany s délkou 1 odpovídající otočení doleva a doprava. Nadále zjevně platí, že nejkratší cesta ze startu do cíle v tomto grafu odpovídá nejlepšímu řešení naší původní úlohy.

Protože tentokrát máme hrany délky 0 a 1, k nalezení nejkratší cesty nemůžeme použít obyčejné prohledávání do šířky. Dokážeme ho ale vhodně upravit. V libovolném grafu, jehož délky hran jsou 0 a 1, můžeme nejkratší cesty z jednoho vrcholu do všech ostatních určit následujícím algoritmem:

- Pro každý vrchol si pamatujeme jeho nejmenší vzdálenost od počátečního vrcholu, jakou jsme dosud objevili. Na začátku máme vzdálenost 0 pro počáteční vrchol a vzdálenost $\infty$ pro všechny ostatní vrcholy.
- Abychom dokázali na konci sestrojit nejkratší cestu ze startu do cíle, pro každý vrchol si pamatujeme, odkud jsme do něj nejkratší cestou přišli.
- Vrcholy čekající na zpracování nebudeme mít uložené v obyčejné frontě, ale v tzv. oboustranné frontě (deque). V ní umíme efektivně přidávat i odebírat vrcholy na obou koncích. Na začátku výpočtu do fronty zařadíme počáteční vrchol.
- Stejně jako v klasickém BFS, dokud se fronta nevyprázdní, opakujeme cyklus, v němž vždy vybereme první vrchol z fronty a zpracujeme ho.
- Zpracování jednoho vrcholu vypadá takto: Postupně projdeme všechny hrany, které z něj vedou. Pro každou hranu se podíváme, zda pomocí ní můžeme zlepšit vzdálenost do jejího koncového vrcholu. Pokud ano, zapíšeme si pro tento vrchol jeho novou vzdálenost a zařadíme ho do fronty na zpracování.

  *Ale pozor, důležitá změna:* Jestliže zařazujeme do fronty vrchol, do něhož jsme přišli hranou délky 1, zařadíme ho klasicky, tedy na konec fronty. Jestliže jsme ale přišli hranou délky 0, zařadíme ho *hned na začátek.*

Proč tento algoritmus funguje a jaká je jeho časová složitost?

Matematickou indukcí můžeme dokázat, že ve frontě čekající na zpracování jsou vždy nejvýše dvě skupiny vrcholů: nejprve nějaké vrcholy, do nichž se umíme dostat cestou „aktuální“ délky $x$, potom případně nějaké vrcholy, do nichž se umíme dostat cestou délky $x+1$. Tento invariant platí právě díky způsobu, jakým nové vrcholy do fronty zařazujeme.

V každém kroku algoritmu tedy zpracujeme vrchol, který má mezi všemi nezpracovanými vrcholy nejmenší dosavadní vzdálenost. Snadno nahlédneme, že ta už musí být definitivní – všechny vrcholy s menší vzdáleností jsme totiž zpracovali a všechny hrany vedoucí z nich dále jsme už proto prohlédli. Z toho plyne, že když náš algoritmus označí nějaký vrchol jako zpracovaný, známe skutečně správnou délku nejkratší cesty vedoucí ze startu do něj.

Zbývá uvědomit si, že každý vrchol se může ve frontě na zpracování ocitnout nejvýše *dvakrát.* Může se například stát, že když zpracujeme vrchol $v_1$, který je ve vzdálenosti 7 od počátku, objevíme způsob, jak hranou délky 1 dojít do vrcholu $w$. Nastavíme tedy vrcholu $w$ vzdálenost na 8 a zařadíme ho na konec fronty. Později ale můžeme zpracovat jiný vrchol $v_2$ a z něho vrchol $w$ dosáhnout hranou délky 0. Když se toto stane, zmenšíme vrcholu $w$ vzdálenost na 7 a zařadíme ho do fronty na zpracování podruhé – tentokrát ale na její začátek.

Skutečnost, že máme vrchol $w$ ve frontě na zpracování dvakrát, nám nijak nevadí – při druhém zpracování vrcholu $w$ se už jednoduše nic nestane.

Tento algoritmus tedy zpracuje každý vrchol grafu i každou hranu grafu nejvýše dvakrát, proto je jeho časová složitost lineární vzhledem k velikosti grafu. V našem případě máme graf s $\mathcal{O}(n^2)$ vrcholy a hranami, časová složitost tohoto řešení je tedy také $\mathcal{O}(n^2)$.

(Na závěr dodejme, že výše uvedený postup platí pro úplně libovolný graf s hranami délky 0 a 1. Náš graf je navíc speciální: pro každý vrchol platí, že všechny do něj vedoucí cesty mají stejnou paritu délky. Každý vrchol bude tedy do fronty na zpracování zařazen dokonce jenom jednou.)

```
#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 vzdalenost[MAXRS][MAXRS][4];
stav odkud[MAXRS][MAXRS][4];
char zrcadlo[MAXRS][MAXRS][4];

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

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

```
    // najdi startovací 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)
        vzdalenost[r][s][d] = 987654321;
    INDEX(vzdalenost,start) = 0;
    deque<stav> Q;
    Q.push_back(start);

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

        // zjisti, zda nejsme v cíli; pokud ano, sestroj a vypíš řešení
        if (mapa[ kde.r ][ kde.s ] == '=') {
            while (kde.r != start.r || kde.s != start.s || kde.d != start.d) {
                if (INDEX(zrcadlo,kde) != '.')
                    mapa[kde.r][kde.s] = INDEX(zrcadlo,kde);
                kde = INDEX(odkud,kde);
            }
            for (string row : mapa) cout << row << "\n";
            return 0;
        }

        // vyzkoušej tři možnosti pohybu: rovně, přes zrcadlo /, přes zrcadlo \.
        for (int pohyb=0; pohyb<3; ++pohyb) {
            stav kam = kde;
            int delka = 0;
            if (pohyb == 0) { kam.r += DR[ kde.d ]; kam.s += DS[ kde.d ]; }
            if (pohyb == 1) { kam.d = 3 - kde.d; delka = 1; }
            if (pohyb == 2) { kam.d = kde.d ^ 1; delka = 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(vzdalenost,kde) + delka >= INDEX(vzdalenost,kam)) continue;
            INDEX(vzdalenost,kam) = INDEX(vzdalenost,kde) + delka;
            INDEX(odkud,kam) = kde;
            INDEX(zrcadlo,kam) = "./\\"[pohyb];
            if (delka == 0) Q.push_front(kam); else Q.push_back(kam);
        }
    }
    cout << "nemozne\n";

    return 0;
}
```

## P-III-5 Graffiti

Bylo poměrně snadné získat za úlohu nějaké body hrubou silou: jednoduše simulujeme proces popsaný v zadání, přičemž si o každém panelu zdi pamatujeme, zda je momentálně pomalovaný.

### Jak umělci malují jen na jeden panel

Pro vstupy, v nichž každý umělec pomaluje jenom jeden panel, je snadné napsat i efektivní řešení. Stačí si například v jedné uspořádané množině pamatovat všechny čisté panely a v druhé všechny pomalované panely. Kdykoliv přijde nový umělec, najdeme v čase $\mathcal{O}(\log p)$ první čistý panel. Kdykoliv přijde pořádková služba, najdeme v čase $\mathcal{O}(\log p)$ v druhé množině úsek pomalovaných panelů, které je třeba vyčistit, a jeden po druhém je postupně přesuneme mezi čisté.

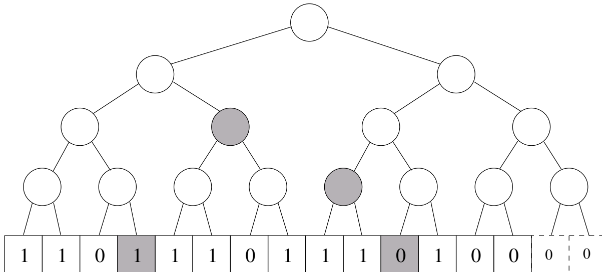
Dohromady všechny umělce zpracujeme v čase $\mathcal{O}(u \log p)$. Protože každý pomalovaný panel, který někdy pořádková služba vyčistí, musel předtím někdo pomalovat, dohromady za celý běh programu vyčistí služba nejvýše $u$ panelů, takže také všechny události týkající se pořádkové služby zpracujeme čase $\mathcal{O}(u \log p)$.

(Všimněte si, že při zpracování pořádkové služby je nutné vědět, které panely v jejím úseku jsou pomalované, a zpracovat jenom ty. Kdybychom se dívali na úplně každý panel v čištěném úseku, vzrostla by časová složitost na $\mathcal{O}(up)$.)

### Obecné řešení

V našem vzorovém řešení použijeme pro reprezentaci zdi tzv. intervalový strom: úplný binární strom, jehož listy odpovídají jednotlivým panelům zdi. Vnitřní vrcholy tohoto stromu potom odpovídají delším úsekům zdi. Jestliže listy stromu tvoří úroveň 0, jejich rodiče úroveň 1, a tak dále, potom každý vrchol na úrovni $i$ představuje nějaký konkrétní úsek $2^i$ po sobě jdoucích panelů.

Počet listů tohoto stromu je nejbližší mocninou dvou větší nebo rovnou $p$. (Použijeme jen $p$ nejlevějších z nich, ostatní jsou jakoby po celou dobu pomalované.) Je zjevné, že listů je méně než $2p$. Hloubka intervalového stromu je proto $\mathcal{O}(\log p)$.




Na obrázku je příklad intervalového stromu. Pole ve spodním řádku představuje listy stromu, tedy úroveň 0. Hodnoty 0 a 1 představují pomalované a čisté panely.

Intervalový strom využijeme následovně: libovolný úsek zdi dokážeme rozdělit na $\mathcal{O}(\log p)$ kratších (a navzájem disjunktních) úseků, z nichž každý odpovídá nějakému vrcholu intervalového stromu. Když tedy chceme něco zjistit o nějakém úseku,

případně si o něm zapsat nějakou informaci, nemusíme v čase $\mathcal{O}(p)$ postupně po jednom projít všechna jeho políčka, ale stačí v čase $\mathcal{O}(\log p)$ nalézt příslušná políčka ve stromu.

Na obrázku jsou šedou barvou označeny vrcholy stromu, které dohromady reprezentují úsek začínající čtvrtým a končící jedenáctým panelem.

Rozmyslíme si, jaké operace potřebujeme s naším stromem provádět:

- Pro dané $d$ nalézt nejlevější výskyt $d$ volných panelů vedle sebe.
- Celý daný úsek označit jako omalovaný, nebo naopak jako čistý.

Abychom mohli provádět efektivně první z nich, budeme si v každém vrcholu stromu pamatovat tyto hodnoty:

- Jaký nejdelší úsek čistých panelů se nachází někde v úseku panelů, které tento vrchol zastupuje?
- Kolika čistými panely jeho úsek panelů začíná a kolika končí?

Tyto informace lze snadno udržovat aktuální. Když je známe pro oba syny nějakého vrcholu $v$, lehce z nich vypočítáme údaje pro vrchol $v$. (Například nejdelší čistý úsek je maximem ze tří možností: nejdelší úsek pod levým synem, nejdelší úsek pod pravým synem, nebo úsek, který tvoří čisté panely na konci úseku levého syna a na začátku úseku pravého syna.)

S pomocí těchto informací snadno určíme nejlevější dostatečně dlouhý čistý úsek: stačí začít v kořenu stromu a vždy si vybrat nejlevější možnost, která ještě vede k řešení.

Abychom dokázali efektivně označovat úseky jako čisté nebo pomalované, použijeme techniku zvanou *líné provádění změn* (lazy updates). Každému vrcholu přidáme ještě jednu proměnnou, kterou nazveme stav. Každý vrchol může být v jednom ze tří stavů: aktuální, čistý, nebo špinavý. Aktuální vrchol má v sobě uloženy informace ve výše popsané podobě. Čistý vrchol je takový vrchol, v němž máme poznačeno, že všechny vrcholy pod ním jsou čisté. Špinavý vrchol je takový vrchol, v němž máme poznačeno, že všechny vrcholy pod ním jsou pomalované.

Když měníme stav panelů (ať už umělec některé pomaloval, nebo pořádková služba některé vyčistila), uděláme jenom to, že změníme stav vrcholů, které tomu úseku odpovídají (a přepočítáme informace ve vrcholech nad nimi).

Taková změna může způsobit, že někde pod čistými/špinavými vrcholy jsou vrcholy, v nichž jsou uloženy neaktuální informace. Toto je ale právě podstata líných změn – v této chvíli si jenom řekneme, že nám tento stav nevadí. Představme si totiž, že jdeme od kořene libovolným směrem dolů po našem intervalovém stromu. Dříve, než přijdeme do vrcholu, v němž jsou neaktuální informace, vždy narazíme na příslušný čistý/špinavý vrchol a tam už víme, na čem jsme, a nemusíme chodit hlouběji.

Občas se nám stane, že z nějakého čistého nebo špinavého vrcholu $v$ přece jen potřebujeme jít dále dolů. Například máme poznačeno, že vrchol představující nějakých 8 panelů je špinavý, ale právě přišel nový požadavek, že je třeba první

tři z nich očistit. Až v této situaci pošleme informaci dále: jestliže byl $v$ špinavý, tak nejprve oba syny vrcholu $v$ změníme na špinavé, potom rekurzívně sestoupíme dolů do jednoho z nich, provedeme potřebné změny, a když se z rekurze vynoříme, přepočítáme (nyní už aktuální) vrchol $v$ pomocí informací v jeho synech. V našem příkladu by tedy výsledkem byla situace, v níž by levý syn $v$ skončil také jako aktuální (a pamatovali bychom si v něm, že jsou pod ním tři čisté panely), zatímco pravý syn $v$ by skončil špinavý. Z těchto údajů bychom následně přepočítali $v$.

Každou operaci se stromem dokážeme takto zrealizovat v čase $\mathcal{O}(\log p)$, zpracování $u$ událostí tedy zvládneme s časovou složitostí $\mathcal{O}(u \log p)$.

Na závěr dodáváme, že existují i jiná řešení se stejnou časovou složitostí. Jedno možné řešení je například založeno na tom, že si budeme zeď reprezentovat jako množinu souvislých úseků čistých panelů. Ty si budeme udržovat ve vyvažovaném binárním stromu uspořádané podle jejich začátku. Navíc si v každém vrcholu stromu budeme pamatovat, jaký nejdelší úsek leží v jeho podstromu.

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

const int CISTY = 0, SPINAVY = 1, AKTUALNI = 2;
const int HLOUBKA = 19;

struct vrchol {
    int zacatek, konec, nejvic, stav;
    void zapln(int level, int s)
        { stav=s; zacatek = konec = nejvic = (s==CISTY ? (1<<level) : 0); }
};

vector< vector<vrchol> > T;

void prepocitej_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 = AKTUALNI;
    ja.zacatek = lsyn.zacatek;
    if (ja.zacatek == 1 << (level-1)) ja.zacatek += psyn.zacatek;
    ja.konec = psyn.konec;
    if (ja.konec == 1 << (level-1)) ja.konec += lsyn.konec;
    ja.nejvic = max( max(lsyn.nejvic, psyn.nejvic), lsyn.konec+psyn.zacatek );
}

void preposli_dolu(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 == AKTUALNI) return;
    lsyn.zapln(level-1,ja.stav);
    psyn.zapln(level-1,ja.stav);
    prepocitej_vrchol(level,offset);
}

int najdi_prvni_usek(int delka, int level=HLOUBKA, int offset=0) {
    preposli_dolu(level,offset);
```

```
    if (T[level][offset].nejvic < delka) return -1;
    if (level == 0) return offset;
    int tmp = najdi_prvni_usek(delka,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.konec + psyn.zacatek >= delka)
        return (1<<(level-1))*(2*offset+1) - lsyn.konec;
    return najdi_prvni_usek(delka,level-1,2*offset+1);
}

void zaznac_usek(int lo, int hi, int stav, int level=HLOUBKA,
                 int offset=0, int vlo=0, int vhi=1<<HLOUBKA) {
    preposli_dolu(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);
    prepocitej_vrchol(level,offset);
}

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

    T.resize(HLOUBKA+1);
    T[0].resize( 1<<HLOUBKA, {0,0,0,AKTUALNI} );
    for (int p=1; p<=P; ++p) T[0][p] = {1,1,1,AKTUALNI};
    for (int h=0; h<=HLOUBKA; ++h) {
        T[h].resize( 1<<(HLOUBKA-h) );
        for (int o=0; o<(1<<(HLOUBKA-h)); ++o) prepocitej_vrchol(h,o);
    }

    while (U--) {
        string cmd; cin >> cmd;
        if (cmd == "U") {
            int d; cin >> d;
            int ans = najdi_prvni_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);
        }
    }

    return 0;
}
```

## P-III-6 Korálky

Naším úkolem je pomoci Natálce s jejím korálkovým problémem. Začneme ne příliš efektivním algoritmem.

### Zkoušení všech možností

Neklesající posloupnost korálků nazveme *náhrdelník.* Pro dostatečně malé $n$ si můžeme vygenerovat všechny náhrdelníky a vybrat nejdelší z nich. Označme si posloupnost čísel na vstupu jako $F$. Všechny dobré náhrdelníky můžeme vygenerovat například pomocí rekurzívní funkce $f(ind, levy, pravy)$, kde $ind$ je index čísla, které chceme zpracovat, a $levy$ a $pravy$ jsou indexy posledního korálku navlečeného na šňůrku zleva a zprava. Tato funkce postupně vyzkouší všechny možnosti, jak náhrdelník dokončit, a vrátí délku nejdelší z nich. V těle funkce $f$ stačí vyzkoušet tři možnosti a vybrat z nich maximum.

- $f(ind + 1, levy, pravy)$ (přeskočíme číslo na pozici $ind$)
- $f(ind + 1, ind, pravy) + 1$, jestliže $F[ind] < F[levy]$ (přidáme číslo na pozici $ind$ na začátek)
- $f(ind + 1, levy, ind) + 1$, jestliže $F[pravy] < F[ind]$ (přidáme číslo na pozici $ind$ na konec)

Nyní už jenom stačí zavolat funkci $f(i + 1, i, i)$, pro každé $i < n$ a vybrat maximum. Časová složitost tohoto řešení je $\mathcal{O}(3^n)$, protože máme $n$ pozic a v každé se naše rekurze může rozdělit do tří větví.

Toto řešení můžeme vylepšit pomocí techniky zvané memoizace (kešování). Povšimneme si, že v průběhu rekurze počítáme hodnotu funkce $f$ pro stejné parametry opakovaně. Při prvním výpočtu hodnoty $f(ind, levy, pravy)$ si vypočtenou hodnotu zapamatujeme a v dalších větvích rekurze místo rekurzivního výpočtu funkce použijeme zapamatovanou hodnotu. Všechny argumenty funkce $f$ jsou mezi 1 a $n$, a tedy počet hodnot, které si musíme zapamatovat, je $\mathcal{O}(n^3)$. Výpočet každé z nich ze tří dalších hodnot lze provést v konstantním čase. Tímto postupem získáme algoritmus s časovou a paměťovou složitostí $\mathcal{O}(n^3)$.

### Vzorové řešení

Vzorové řešení je založeno na následující myšlence: Některý korálek jsme museli na náhrdelník navléknout jako první. Kdybychom znali číslo $i$ a tedy barvu $F[i]$ tohoto prvního korálku, rozdělil by se nám problém na dvě nezávislé části. O každém z následujících korálků totiž víme, že je-li menší než $F[i]$, můžeme ho navléknout jedině zleva, a je-li větší než $F[i]$, tak jedině zprava.

Protože zleva i zprava chceme navléknout co nejvíce korálků, potřebujeme mezi „levými“ korálky nalézt co nejdelší klesající a mezi „pravými“ korálky co nejdelší rostoucí podposloupnost. Stačí implementovat jeden z těchto dvou algoritmů. Když totiž v nějakém poli změníme všem prvkům znaménko (vynásobíme je $-1$), tak se klesající podposloupnosti změní na rostoucí a naopak.

Dostáváme tak řešení s časovou složitostí $\mathcal{O}(n^2 \log n)$. Stačí postupně vyzkoušet všechny možnosti pro index $i$ prvního navlečeného korálku a vždy pomocí dvou volání

algoritmu z domácího kola nalézt nejdelší klesající a rostoucí podposloupnost, které k příslušnému prvnímu korálku můžeme připojit.

Popsané řešení ale stále dělá mnoho práce zbytečně vícekrát.

Když se lépe podíváte na algoritmus z domácího kola, můžete si všimnout, že se při zpracování konkrétního indexu dozvíme délku nejdelší rostoucí podposloupnosti, která na tomto indexu *končí.*

Nyní tento algoritmus použijeme na zadanou posloupnost $F$, ale *v opačném směru, tedy od konce.* Tím se pro každý index dozvíme délku nejdelší *klesající* podposloupnosti, která na něm *začíná.*

Následně uděláme totéž, ale pro posloupnost $F$ s opačnými znaménky. Tím se pro každý index dozvíme také délku nejdelší *rostoucí* podposloupnosti, která na něm začíná.

Když už máme tyto dvě informace, úlohu snadno vyřešíme v lineárním čase: pro každý index $i$ dokážeme v konstantním čase určit, kolik nejvýše korálků můžeme použít, když začneme korálkem $i$.

Celková časová složitost tohoto řešení je tedy pouze $\mathcal{O}(n \log n)$.

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

vector<int> LIS(const vector<int> &A) {
    // odpoved[i] = délka nejdelší rostoucí podposloupnosti končící na indexu 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;

    return 0;
}
```