P-III-1 Věže z kostek

Podúloha A: minimální počet kroků

Na každou věž se můžeme podívat následovně: začneme od spodní kostky a postupujeme nahoru, dokud má následující kostka menší číslo než ta aktuální. Takto najdeme nejdelší *sufix* dané věže, který je uspořádaný. Kostky tvořící uspořádaný sufix obarvíme na zeleno a všechny kostky nad ním obarvíme na červeno.

Všimněme si nyní, že pro každou věž platí, že všechny její červené kostky musíme někdy přesunout. Poslední červená kostka má totiž větší číslo než první zelená, a pokud z této věže provedeme méně přesunů, obě tyto kostky v ní stále zůstanou a ta červená bude stále nad tou zelenou, takže věž nebude uspořádaná. Dostáváme tedy dolní odhad: libovolné správné řešení musí mít alespoň tolik kroků, kolik máme celkem červených kostek.

A teď už stačí si jen uvědomit, že tento dolní odhad je vždy přesný, tedy že vždy lze věže uspořádat postupem, který každou červenou kostku přesune přesně jednou: Dokud existuje věž, která má nahoře červenou kostku, vybereme si libovolnou takovou věž a odebereme z ní horní červenou kostku. Poté si vybereme libovolnou jinou věž a podíváme se na její zelenou uspořádanou část. V této části najdeme místo, kam podle hodnoty patří naše červená kostka, vložíme ji tam a přebarvíme ji na zelenou. Tím jsme udělali jeden krok, máme o jednu červenou kostku méně a jeden zelený úsek je o kostku delší, než byl původně.

Program (Python):

```
def zelene(vez):
    posledni, kde, kolik = max(vez) + 1, len(vez), 0
    while kde > 0 and vez[kde-1] < posledni:
        posledni, kde, kolik = vez[kde-1], kde-1, kolik+1
    return kolko

v = int( input() )
veze = [ [ int(_) for _ in input().split() ] for __ in range(v) ]
print( sum(len(vez)-zelene(vez) for vez in veze) )
```

Podúloha B: pokyny

Nyní si ukažme, jak výše popsaný algoritmus na nalezení optimálního postupu implementovat efektivně. Budeme postupovat systematicky: nejprve vložíme všechny červené kostky z věže 1 do zelené části věže 2, poté červené kostky z věže 2 do zelené části věže 3 a tak dále. Každé kolo tohoto procesu bude vypadat stejně, až na poslední, a i tam dojde jen k nepatrné změně: když budeme vkládat červené kostky z věže v do zelené části věže 1, na vrcholu věže 1 již nebudou její původní červené kostky, což nám trochu změní čísla pozic, na které budeme vkládat.

Stačí tedy popsat, jak provést jedno kolo výše popsaného postupu. Aby výsledný algoritmus byl efektivní, měla by časová složitost každého kola být přibližně lineární (např. s logaritmickým faktorem navíc) vzhledem k počtu kostek, kterých se týká – tedy pokud máme na začátku v i-té věži c_i červených a z_i zelených kostek, tak přesun červených kostek z věže 1 do věže 2 bychom měli dokázat provést v časové složitosti přibližně lineární k $c_1 + z_2$.

Pokud do jednoho pole vložíme záznamy pro všechny červené kostky z první věže a všechny zelené z druhé a toto pole seřadíme, získáme pořadí, v němž musí všechny tyto kostky skončit. Nyní už jen potřebujeme spočítat, na které pozice je potřeba vložit jednotlivé červené kostky, abychom tohoto pořadí na konci dosáhli.

To můžeme efektivně provést např. tak, že nad polem, které jsme právě uspořádali, postavíme součtový intervalový strom, ve kterém si v listech budeme u každé kostky pamatovat, zda se již nachází ve druhé věži. Na začátku jsou tedy v červených listech tohoto stromu nuly a v zelených jedničky. V každém vnitřním vrcholu máme součet hodnot v listech pod tímto vrcholem, tedy počet kostek, které již existují v jemu odpovídajícím intervalu. Nyní začneme procházet první věž shora dolů. U každé červené kostky se podíváme, kde má skončit ve výsledném poli, a následně se intervalového stromu zeptáme, kolik kostek již existuje nalevo od této pozice. Naši právě zpracovávanou kostku je třeba vložit přesně pod všechny tyto kostky (nesmíme také zapomenout na c_2 červených kostek, které jsou ve druhé věži nad nimi). Následně už jen přepočítáme intervalový strom poté, co si zaznamenáme, že i právě přesunutá kostka se již nachází ve druhé věži.

Tento postup má časovou složitost $\mathcal{O}(x \log x)$, kde $x = c_1 + z_2$ je celkový počet zpracovávaných kostek. Když takto postupně zpracujeme všechny věže, každou kostku započítáme právě v jednom kole. Celkový počet kroků během celého algoritmu tedy můžeme odhadnout shora na $\mathcal{O}(n \log n)$, kde n je celkový počet kostek.

V níže uvedeném programu používáme místo plnohodnotného intervalového stromu jednodušší Fenwickův strom.

Program (Python):

```
class FenwickTree:
    def __init__(self, size):
        self.tree = [0] * (size + 1)

    def add(self, i):
        i += 1
        while i < len(self.tree):
            self.tree[i] += 1
            i += i & (-i)

    def prefix_sum(self, i): # vrátí součet v intervalu [0, i)
        res = 0
        while i > 0:
            res += self.tree[i]
            i -= i & (-i)
        return res

def zelene(vez):
    posledni, kde, kolik = max(vez) + 1, len(vez), 0
```

```
    while kde > 0 and vez[kde-1] < posledni:
        posledni, kde, kolik = vez[kde-1], kde-1, kolik+1
    return kolik

def presun(veze, odkud, kam, ma_jeste_cervene):
    pocet_cervenych_kam = \
        len(veze[kam]) - zelene(veze[kam]) if ma_jeste_cervene else 0

    cervene_odkud = veze[odkud][:-zelene(veze[odkud])]
    zelene_kam = veze[kam][-zelene(veze[kam]):]
    dohromady = cervene_odkud + zelene_kam
    dohromady.sort()
    pozice = { y:x for x,y in enumerate(dohromady) }

    uz_zije = FenwickTree(len(dohromady))
    for z in zelene_kam: uz_zije.add pozice[z])
    for c in cervene_odkud:
        p = pozice[c]
        nad = uz_zije.prefix_sum(p) + pocet_cervenych_kam
        print(f'presun {c} z veze {odkud} do veze {kam} pod {nad} kostek')
        uz_zije.add(p)

v = int(input())
veze = [ [ int(_) for _ in input().split() ] for __ in range(v) ]
for i in range(v-1): presun(veze, i, i+1, True)
presun(veze, v-1, 0, False)
```

Podúloha C: předepsané výšky

Co se změní, když dostaneme předepsané výšky p_i? První změnou je, že pokud má nějaká věž delší uspořádaný suffix než je její hodnota p_i, nemůžeme si tento suffix nechat celý. Zelenou barvu tedy dostane pouze spodních p_i kostek, všechny ostatní budou červené – musí někam pryč.

Opět si jako c_i a z_i označíme počáteční počty červených a zelených kostek ve věži i. Pro každou věž navíc označme $d_i = p_i - z_i$. Číslo d_i udává, kolik kostek do zelené části této věže musíme přidat, aby měla na konci správnou výšku. Součet všech d_i označíme jako D. Hodnota D je zjevně také rovna celkovému počtu červených kostek, tedy součtu všech c_i. V našem řešení tedy určitě potřebujeme provést alespoň D kroků.

Každé červené kostce bychom chtěli přiřadit jinou věž, než je ta její, do které ji přesuneme, a to tak, aby jsme do každé věže i přesunuli celkem přesně d_i kostek. Pokud se nám to podaří, přesuneme každou červenou kostku přesně jednou a tím zjevně získáme optimální řešení. Jak však ukazuje i příklad v zadání úlohy, to nebude vždy možné. Problém nastane tehdy, máme-li nějakou věž, kde je na začátku mnoho červených kostek (ty musíme všechny odstranit), ale zároveň má být na konci vysoká (takže musíme sem přesunout i mnoho kostek z jiných věží).

Věž i proto nazveme *problémovou*, pokud platí $c_i + d_i > D$. Taková věž nám sama o sobě způsobí, že naše řešení bude muset mít alespoň $c_i + d_i$ kroků, protože z ní musíme odstranit c_i červených kostek a naopak na ni přesunout d_i kostek z jiných věží.

Povšimněme si nejdříve, že nejvýše jedna věž je problémová: Je zřejmé, že

nemohou existovat dvě takové věže i a j, protože pak by součet $(c_i + d_i) + (c_j + d_j)$ byl větší než $2D$, což je v rozporu s tím, že součet všech c_i je D a součet všech d_i je také D.

Pokud takovou problémovou věž i skutečně máme, tvrdíme, že úlohu lze vyřešit na přesně $c_i + d_i$ kroků: Jelikož $c_i + d_i > D$ a zároveň d_i + (součet všech ostatních d_j) $= D$, je c_i větší než součet všech ostatních d_j. Můžeme tedy začít tím, že v prvních c_i krocích řešení rozmístíme všechny červené kostky z věže i do ostatních věží tak, že každou vložíme do některého zeleného úseku tam, kam patří, a že na konci v každé jiné věži j máme zelený úsek délky alespoň p_j.

Tímto způsobem jsme dosáhli stavu, ve kterém jsou všechny věže až do své požadované výšky uspořádané, pouze věži i chybí d_i kostek do správné výšky a ostatní věže mají dohromady o d_i kostek navíc. Z každé příliš vysoké věže tedy stačí přesunout její přebytečné kostky do věže i a máme hotovo.

Naopak, pokud žádná problémová věž není, pak existuje řešení na D kroků: Stačí červené kostky přesouvat tak, aby po žádném tahu nevznikla problematická věž. Proč to vždy lze?

Nejprve si všimněme, že pokud má nějaká věž $c_i > 0$, pak z $c_i + d_i \leq D$ vyplývá, že nutně platí $d_i < D$, takže nutně existuje jiná věž s $d_j > 0$. Proto vždy můžeme provést tah, při kterém přesuneme nějakou červenou kostku z věže, kde jsou (věž i), do věže, kde jsou potřeba (věž j).

Každý přesun zmenší D o jedna. Věž i nazveme *kritickou*, pokud platí $c_i + d_i = D$. Máme-li kritickou věž, musíme v každém kroku řešení buď přesunout jednu červenou kostku pryč z ní (snížit c_i), nebo přesunout jednu červenou kostku z jiné věže sem (snížit d_i), aby se z ní nestala problémová věž.

Stejně jako jsme výše zdůvodnili, že problémová věž je nanejvýš jedna, můžeme vidět, že kritické věže jsou (pro $D > 0$) vždy nanejvýš dvě. Jsou-li dvě, můžeme vždy přesunout červenou kostku z jedné do druhé. Je-li jen jedna, přesuneme červenou kostku z ní, pokud v ní ještě nějakou je, resp. do ní, pokud již ne. A pokud nejsou žádné kritické věže, můžeme provést libovolný platný tah.

Program (Python):

```
def zelene(vez):
    posledni, kde, kolik = max(vez) + 1, len(vez), 0
    while kde > 0 and vez[kde-1] < posledni:
        posledni, kde, kolik = vez[kde-1], kde-1, kolik+1
    return kolik

v = int( input() )
veze = [ [ int(_) for _ in input().split() ] for __ in range(v) ]
P = [ int(_) for _ in input().split() ]

Z = [ zelene(veze[i]) for i in range(v) ]
C = [ len(veze[i]) - Z[i] for i in range(v) ]
D = [ P[i] - Z[i] for i in range(v) ]

print( max( sum(C), max( c+d for c,d in zip(C,D) ) ) )
```

P-III-2 Hnusný plot

Nejprve popišme trochu pomalé řešení dynamickým programováním s kubickou časovou složitostí (tedy za 7 bodů). Označme $P[x][y][s]$ počet hnusných plotů, které jsou tvořeny laťkami s výškami 1 až x, přičemž první laťka plotu má výšku y a poté plot pokračuje směrem s (tedy nahoru, k vyšší hodnotě, pokud $s = \uparrow$, resp. dolů, pokud $s = \downarrow$).

Pro $x = 1$ máme jeden plot, který dále nepokračuje (tedy ho můžeme použít jako plot směřující nahoru i plot směřující dolů). Proto $P[1][1][\uparrow] = P[1][1][\downarrow] = 1$.

Jak můžeme určit hodnotu $P[x][y][\uparrow]$ pro větší x? Jelikož plot musí nejprve růst, jeho druhá laťka má nějakou výšku $z > y$ a od druhé laťky pak plot pokračuje dolů. Celkový počet plotů určující $P[x][y][\uparrow]$ tedy můžeme zjistit tak, že zjistíme počet pro každou volbu výšky z druhé laťky a tyto počty sečteme.

Jak zjistit počet pro zafixované z? Všimněme si, že hnusných plotů s laťkami o výškách 1 až x', které začínají laťkou y', je přesně stejně mnoho jako hnusných plotů, které jsou tvořeny jinými laťkami x' různých výšek a začínají y'-tou nejmenší z nich. Jinými slovy, nezáleží na konkrétních výškách latěk, ale pouze na jejich relativním pořadí.

Nyní si uvědomme, že vezmeme-li libovolný hnusný plot z latěk $1, \ldots, x$ začínající laťkami výšek y a z (kde $z > y$) a odstraníme z něj úvodní laťku výšky y, dostaneme hnusný plot tvořený $x - 1$ laťkami výšek od 1 do x kromě y a začínající laťkou výšky z. Tento hnusný plot tedy začíná $(z-1)$-tou nejmenší z použitých latěk a poté pokračuje dolů. Platí tedy, že takových hnusných plotů je přesně $P[x-1][z-1][\downarrow]$.

Hledaná hodnota $P[x][y][\uparrow]$ se tedy rovná součtu hodnot $P[x-1][z-1][\downarrow]$ pro všechna z od $y+1$ do x.

Hodnoty pro ošklivé ploty začínající dolů spočítáme obdobně, pozor jen na rozdíl v ± 1: Když jdeme z y do z dolů, tak po odstranění y je nový začátek z-tým nejmenším ze zbývajících úseků, protože tentokrát bylo y větší než z. Proto je $P[x][y][\downarrow]$ rovno součtu hodnot $P[x-1][z][\uparrow]$ pro všechna z od 1 do $y-1$.

Výsledný výpočet všech hodnot v poli P má časovou složitost $\mathcal{O}(n^3)$: počítáme řádově n^2 různých hodnot a každou z nich zjistíme procházením a sčítáním řádově n možností.

Nyní se zbývá vypořádat se zadaným začátkem plotu délky k.

- Jestliže $k = 0$, stačí posčítat počty pro všechny možné začátky a směry, tedy hodnoty $P[n][y][\star]$ pro všechna y.
- Máme-li $k = 1$, známe začátek y, ale neznáme směr, výsledek je tedy $P[n][y][\uparrow] + P[n][y][\downarrow]$.
- Jestliže $k \geq 2$, nejprve ověříme, zda je začátek plotu hnusný, a pokud ne, vypíšeme 0. Jinak označme jako y výšku k-té laťky z plotu, s jako směr opačný tomu mezi $(k-1)$-ní a k-tou laťkou, a M jako množinu všech délek latěk kromě prvních $k-1$ v plotu. Nechť laťka y je y'-tá nejmenší v množině M. Výsledkem pak je právě hodnota $P[|M|][y'][s]$.

Rychlejší řešení

Nyní zbývá toto řešení zrychlit. K tomu použijeme stejné vzorce, které jsme již odvodili výše, pouze je budeme chytřeji vyhodnocovat. Všimněme si například následujících vztahů:

$$P[10][7][\uparrow] = \quad P[9][7][\downarrow] + P[9][8][\downarrow] + P[9][9][\downarrow]$$
$$P[10][6][\uparrow] = P[9][6][\downarrow] + P[9][7][\downarrow] + P[9][8][\downarrow] + P[9][9][\downarrow]$$

Vidíme, že druhý součet se od prvního liší pouze jedním členem navíc. A stejné vztahy platí i obecně: když počítáme hnusné ploty z x latěk začínající laťkou y a pokračující nahoru, můžeme všechny možnosti rozdělit do dvou skupin:

- Má-li druhá laťka výšku alespoň $y+2$, dostáváme ploty v podstatě identické s těmi, které začínají laťkou $y+1$, jen v nich prohodíme laťky y a $y+1$; počet takových plotů je tedy $P[x][y+1][\uparrow]$.
- Oproti tomu počet takových plotů, v nichž druhá laťka má výšku $y+1$, je roven $P[x-1][y][\downarrow]$, jak jsme si rozmysleli výše.

Platí tedy $P[x][y][\uparrow] = P[x][y+1][\uparrow] + P[x-1][y][\downarrow]$. Jestliže pole P vyplňujeme v rostoucím pořadí x a klesajícím pořadí y, tyto hodnoty již máme k dispozici a hodnotu $P[x][y][\uparrow]$ dopočítáme v konstantním čase. Celé pole P tedy vyplníme v čase $\mathcal{O}(n^2)$.

Zbytek řešení má lineární časovou složitost, dostáváme tedy řešení s časovou složitostí $\mathcal{O}(n^2)$. Paměťová složitost je také $\mathcal{O}(n^2)$, ale dá se vylepšit na $\mathcal{O}(n)$, když si všimneme, že z pole P si v průběhu výpočtu stačí udržovat poslední dva sloupce.

Program (C++):

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

using namespace std;
const long long MOD = 1000000007;
    // výsledek počítáme modulo toto, aby nám nepřetekly proměnné
const int MAXN = 5007;
long long nahoru[MAXN][MAXN], dolu[MAXN][MAXN], spolu[MAXN];

int main() {
    int N, K;
    cin >> N >> K;
    vector<int> prefix(K);
    for (int k=0; k<K; ++k) { cin >> prefix[k]; --prefix[k]; }

    spolu[0] = spolu[1] = 1;
    nahoru[2][0] = dolu[2][1] = 1;
    spolu[2] = 2;

    for (int n=3; n<=N; ++n) {
        // ploty z n laťek, začínají p a směrem nahoru
        for (int p=n-2; p>=0; --p)
            nahoru[n][p] = (nahoru[n][p+1] + dolu[n-1][p]) % MOD;

        // ploty z n latěk, začínají p a směrem dolů
        for (int p=1; p<=n-1; ++p)
            dolu[n][p] = (dolu[n][p-1] + nahoru[n-1][p-1]) % MOD;
```

```
        for (int p=0; p<n; ++p)
            spolu[n] = (spolu[n] + nahoru[n][p] + dolu[n][p]) % MOD;
    }

    if (K == 0) {
        cout << spolu[N] << endl;
    } else if (K == 1) {
        int z = prefix[0];
        cout << (nahoru[N][z] + dolu[N][z]) % MOD << endl;
    } else {
        bool ok = true;
        for (int i=0; i+2<K; ++i)
            if ((prefix[i]<prefix[i+1]) == (prefix[i+1]<prefix[i+2]))
                ok = false;
        if (ok) {
            int z = prefix[K-1], mensi = 0;
            for (int k=0; k<K-1; ++k)
                if (prefix[k] < z) ++mensi;
            bool klesala = (prefix[K-2] > prefix[K-1]);
            cout << (klesala ? nahoru[N-K+1][z-mensi] : dolu[N-K+1][z-mensi])
                 << endl;
        } else {
            cout << 0 << endl;
        }
    }
}
```

P-III-3 Zhasínání

Musíme najít takovou okružní trasu, při které do každé místnosti vstoupíme lišekrát. Při řešení budeme používat terminologii z teorie grafů. Místnosti v domě jsou vrcholy grafu, hrany odpovídají dveřím mezi nimi.

Na mřížce: kdy neexistuje řešení

Když mřížku vybarvíme jako šachovnici, vidíme, že každé dveře vedou mezi bílou a černou místností. Graf popisující takový dům je tedy *bipartitní*. V bipartitním grafu má nutně každá okružní trasa sudý počet kroků, protože při každém kroku změníme barvu místnosti.

Na druhou stranu, délku konkrétní okružní trasy můžeme spočítat takto: U každého vrcholu se podíváme, kolikrát jsme do něj vstoupili, a tato čísla sečteme. Proto pokud je celkový počet vrcholů n lichý a do každého z nich máme vstoupit lišekrát, musí být celková délka takové cesty lichá.

Z toho vyplývá, že pro liché n v bipartitním grafu nemá zadaná úloha řešení.

Na mřížce: jak najít trasu

Předpokládejme nyní, že n je sudé. Jelikož je náš graf souvislý, má nějakou kostru, tedy množinu $n-1$ hran, které stačí k tomu, aby vše zůstalo propojené. Můžeme ji najít například tak, že z vrcholu 0 spustíme prohledávání do hloubky (DFS) a pro každý další vrchol vezmeme do kostry tu hranu, přes kterou tento vrchol poprvé navštívíme.

Naši okružní trasu nyní můžeme sestavit tak, že se budeme pohybovat pouze po hranách této kostry pomocí druhého, vhodně upraveného DFS. Úprava bude vypadat následovně: Vždy, když naposledy opouštíme nějakou místnost x a vracíme se z ní do místnosti y, podíváme se, zda v x zůstalo zhasnuté světlo. Pokud ne, vrátíme se z y ještě jednou do x, abychom tam světlo zhasli, a následně se z x podruhé vrátíme do y (a od té chvíle už do x nepůjdeme).

Je zřejmé, že tímto způsobem zajistíme, že zhasneme ve všech místnostech, možná s výjimkou ložnice (místnosti číslo 0, kde jsme začali). Co dokážeme říct o stavu světla v této místnosti? V každé z $n-1$ zbývajících místností jsme zhasli, tedy každou z nich jsme navštívili lišekrát. Jelikož n je sudé, znamená to, že $n-1$ je liché, a tedy do ostatních místností jsme celkem vstoupili lišekrát. Také víme, že celkově má naše okružní trasa sudou délku, jelikož graf je bipartitní. Proto jsme museli i do místnosti 0 vstoupit lišekrát, a tedy je i tam zhasnuto.

V obecném grafu

Výše popsaný postup funguje obecně v bipartitních grafech, nejen v těch získaných z mřížky. Zbývá nám tedy vyřešit případ nebipartních souvislých grafů. Ukážeme, že v tomto případě jsme vždy schopni všude zhasnout, a tedy bipartitní grafy s lichým n zůstanou jediným případem, kdy řešení neexistuje.

Výše popsané řešení (vybereme si kostru a projdeme ji) vždy vytvoří okružní trasu sudé délky. Co se stane, pokud ho použijeme na grafu, který není bipartitní? Pro sudé n bude toto řešení i nadále fungovat, protože z celého grafu se díváme pouze

na jeho kostru, což je bipartitní podgraf. Pro lichá n dostaneme téměř funkční řešení s jediným problémem: pouze v ložnici, tedy vrcholu 0, zůstane na konci rozsvíceno. To musíme nějak napravit.

Obecně známé tvrzení říká, že v každém nebipartitním grafu existuje někde alespoň jeden jednoduchý cyklus liché délky. Toto tvrzení mimo jiné plyne z následujícího algoritmu, který takový lichý cyklus najde: Upravíme DFS, pomocí kterého sestavujeme kostru našeho grafu. Vrcholy při něm budeme barvit dvěma barvami tak, že počáteční vrchol bude bílý, všichni sousedé bílých vrcholů dostanou černou barvu a naopak. Pokud se nám podaří projít celý graf a nikde nenarazíme na konflikt, právě jsme ověřili, že graf je bipartitní (jeho vrcholy rozdělit do dvou částí podle barev tak, aby hrany vedly pouze mezi těmito částmi).

Ve všech ostatních případech se nám někdy během běhu algoritmu stane, že zpracováváme nějaký vrchol u a přitom narazíme na hranu vedoucí do dříve zpracovaného vrcholu v, kterému byla přiřazena stejná barva jako v. Jakmile nastane tato situace, víme, že náš graf nemůže být bipartitní, a také umíme najít slíbený lichý cyklus: Stačí vzít hranu uv a k ní jedinou cestu, která vede z u do v po naší DFS kostře. Tato cesta má sudou délku (střídají se na ní bílé a černé vrcholy a má oba konce stejné barvy), takže dohromady s hranou uv dostáváme lichý cyklus.

Celý algoritmus pro liché n bude nyní vypadat následovně:

- Spustíme DFS, abychom sestavili jednu kostru a ověřili, zda je náš graf bipartitní.
- Pokud zjistíme, že máme bipartitní graf, oznámíme, že řešení neexistuje.
- Jinak najdeme liché délky a zapamatujeme si, které vrcholy jej tvoří.
- Následně dokončíme naše první DFS a vytvoříme tak kostru našeho grafu.
- Spustíme druhou fázi algoritmu: postupně procházíme zvolenou kostrou a zajišťujeme, že každý vrchol označíme jako navštívený, když jej opouštíme naposledy.
- Během této fáze algoritmu uděláme ještě jednu věc navíc: když naše prohledávání poprvé vstoupí do některého vrcholu námi zapamatovaného lichého cyklu, než budeme pokračovat dále v DFS po kostře, celý tento cyklus jednou obejdeme.
- Když naše DFS skončí, víme, že všechny vrcholy kromě vrcholu 0 jsou zhasnuté. Díky obcházení lichého cyklu navíc víme, že celková délka naší okružní trasy je lichá. A pokud jsme během ní každý z $n-1$ zbývajících vrcholů (což je sudý počet) navštívili lišekrát, znamená to, že i do vrcholu 0 jsme museli vstoupit lišekrát, a tedy máme zhasnuto i tam.

Algoritmus můžeme ještě trochu zjednodušit: Místo toho, abychom si lichý cyklus pamatovali a později ho obíhali, můžeme na začátku obejít okružní trasu liché délky tvořenou cestou z 0 do u v kostře, hranou uv a cestou z v do 0 v kostře, a až poté spustit DFS, který v každém vrcholu zhasne, když jej opouští.

Celková časová složitost našeho řešení (ať už původního, nebo právě popsané jednodušší verze) je zjevně lineární vzhledem k velikosti grafu, tedy $\mathcal{O}(n+d)$, kde n

je počet místností a d je počet dveří v našem domě. Stejná je i paměťová složitost.

Ještě podotkneme, že níže uvedený program používá vstupní formát z podúlohy B, přičemž předpokládáme, že graf na vstupu je jednoduchý – tedy mezi každými dvěma místnostmi vedou nanejvýš jedny dveře.

Program (C++):

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int N, D;
vector< vector<int> > G;
vector<int> barva, rodic;
int u, v; // hrana na lichém cyklu

void dfs1(int kde, int odkud) {
    // zakořeníme si strom, ověříme bipartitnost,
    // a případně najdeme hranu uv na lichém cyklu
    for (int kam : G[kde]) if (kam != odkud) {
        if (barva[kam] == -1) {
            barva[kam] = !barva[kde];
            rodic[kam] = kde;
            dfs1(kam, kde);
        } else {
            if (barva[kam] == barva[kde] && u == -1) { u=kde; v=kam; }
        }
    }
}

vector<int> do_korene(int start) {
    vector<int> odpoved(1, start);
    while (start != 0) { start = rodic[start]; odpoved.push_back(start); }
    return odpoved;
}

void dfs2(int kde, vector<bool> &sviti) {
    // rekurzivně vyřešíme celý podstrom s kořenem kde
    for (int kam : G[kde]) if (rodic[kam] == kde) {
        cout << kam << " ";
        sviti[kam] = !sviti[kam];
        dfs2(kam, sviti);
        cout << kde << " ";
        sviti[kde] = !sviti[kde];
    }
    if (sviti[kde]) {
        cout << rodic[kde] << " " << kde << " ";
        sviti[kde] = !sviti[kde];
        sviti[rodic[kde]] = !sviti[rodic[kde]];
    }
}

int main() {
    cin >> N >> D;
    G.resize(N);
    for (int d=0; d<D; ++d) {
```

```
        int x, y;
        cin >> x >> y;
        G[x].push_back(y);
        G[y].push_back(x);
    }

    barva.resize(N, -1);
    barva[0] = 0;
    rodic.resize(N, -1);
    u = -1;
    v = -1;
    dfs1(0, -1);

    if (N%2 == 1 && u == -1) { cout << "NELZE\n"; return 0; }

    vector<bool> sviti(N, true);
    if (N%2 == 1) {
        // oběhneme lichou trasu
        auto cesta_u = do_korene(u), cesta_v = do_korene(v);
        cesta_u.pop_back();
        reverse(cesta_u.begin(), cesta_u.end());
        for (int x : cesta_u)
            { cout << x << " "; sviti[x] = !sviti[x]; }
        for (int x : cesta_v)
            { cout << x << " "; sviti[x] = !sviti[x]; }
    }

    dfs2(0, sviti);
}
```