A-III-1 Veže z kociek

Podúloha A: minimálny počet krokov

Na každú vežu sa môžeme pozrieť nasledovne: začneme od spodnej kocky a ideme dohora, kým má nasledujúca kocka menšie číslo ako tá súčasná. Takto nájdeme najdlhší *sufix* danej veže, ktorý je usporiadaný. Kocky tvoriace usporiadaný sufix ofarbíme na zeleno a všetky kocky nad ním ofarbíme na červeno.
Všimnime si teraz, že pre každú vežu platí, že všetky jej červené kocky musíme niekedy presunúť. Posledná červená kocka má totiž väčšie číslo ako prvá zelená, a ak z tejto veže spravíme menej presunov, obe tieto kocky v nej stále budú a červená stále bude nad zelenou, takže veža nebude usporiadaná. Dostávame teda dolný odhad: ľubovoľné korektné riešenie musí mať aspoň toľko krokov, koľko máme dokopy červených kociek.
No a už si stačí len uvedomiť, že tento dolný odhad je vždy presný – teda že vždy vieme nájsť riešenie, ktoré všetko usporiada tak, že každú červenú kocku presunieme práve raz. Toto vieme spraviť nasledovným algoritmom:
Kým máš vežu, ktorá má na vrchu červenú kocku, tak si vyber jednu takú vežu a zober jej hornú červenú kocku. Vyber si ľubovoľnú inú vežu. V tejto druhej veži sa pozri na jej zelenú usporiadanú časť. Nájdi v nej miesto, kam podľa hodnoty patrí naša červená kocka, vlož ju tam a prefarbi ju na zeleno. Tým sme spravili jeden krok, máme o jednu červenú kocku menej a jeden zelený úsek je o kocku dlhší ako bol pôvodne.

Listing programu (Python)

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

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

Podúloha B: konštrukcia

Aby sme vedeli efektívne počítať, ktorú kocku kam vložiť, budeme vyššie popísaný algoritmus robiť systematickejšie: najskôr vložíme všetky červené kocky z veže 1 do zeleného úseku veže 2, potom červené kocky z veže 2 do zeleného úseku veže 3, a tak ďalej. Každé kolo tohto procesu bude vyzerať rovnako, až na posledné, a aj tam nastane len maličká zmena: keď budeme červené kocky z veže v vkladať do zeleného úseku veže 1, už na vrchu veže 1 nebudú jej pôvodné červené kocky, čo nám trochu zmení čísla pozícií, na ktoré budeme vkladať.
Stačí teda popísať, ako spraviť jedno kolo vyššie popísaného postupu. Aby to bolo efektívne, mal by náš postup mať časovú zložitosť približne lineárnu (napr. s logaritmickým faktorom navyše) od počtu kociek, ktorých sa týka – teda ak máme na začiatku v i-tej veži c_i červených a z_i zelených kociek, tak presun červených kociek z veže 1 do veže 2 by sme mali vedieť spraviť v časovej zložitosti približne lineárnej od $c_1 + z_2$.
Keď dáme do poľa záznamy pre všetky červené kocky z prvej veže a všetky zelené z druhej a toto pole usporiadame, dostaneme poradie, v ktorom musia všetky tieto kocky skončiť. Teraz už len potrebujeme spočítať, ktorú kam vložiť, aby sa tak stalo.
Toto vieme efektívne spraviť napr. tak, že si nad poľom, ktoré sme práve usporiadali, postavíme súčtový intervalový strom v ktorom si v listoch budeme o každej kocke pamätať, či sa už nachádza v druhej veži. Na začiatku sú teda v červených listoch tohto stromu nuly a v zelených jednotky. (A v každom vnútornom vrchole máme súčet, čiže počet kociek, ktoré už existujú v jemu zodpovedajúcom intervale.) No a teraz začneme prechádzať pôvodnú prvú vežu zhora dole. Pre každú červenú kocku sa pozrieme, kde vo výslednom poli má skončiť, no a následne sa intervalového stromu opýtame na to, koľko kociek už existuje naľavo od tejto pozície. Našu práve spracúvanú kocku treba vložiť presne pod všetky tieto kocky. (A nesmieme zabudnúť na c_2 červených kociek, ktoré sú v druhej veži nad nimi.) Následne už len prepočítame intervalový strom po tom, ako si zaznačíme, že aj práve presunutá kocka už existuje v druhej veži.
Toto riešenie má časovú zložitosť $O(x \log x)$, kde $x = c_1 + z_2$ je celkový počet spracúvaných kociek. No a keď takto postupne spracujeme všetky veže, každú kocku započítame v práve jednom prechode. Celkový počet krokov počas celého algoritmu teda môžeme zhora odhadnúť $O(n \log n)$, kde n je celkový počet kociek.
V nižšie uvedenom programe používame namiesto plnohodnotného intervalového stromu jednoduchší Fenwickov strom.

Listing programu (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): # vrati sucet v intervale [0, i)
        res = 0
        while i > 0:
            res += self.tree[i]
            i -= i & (-i)
        return res

def zelene(veza):
    posledne, kde, kolko = max(veza) + 1, len(veza), 0
    while kde > 0 and veza[kde-1] < posledne:
        posledne, kde, kolko = veza[kde-1], kde-1, kolko+1
    return kolko

def presun(veze, odkial, kam, ma_este_cervene):
    pocet_cervenych_kam = len(veze[kam]) - zelene(veze[kam]) if ma_este_cervene else 0

    cervene_odkial = veze[odkial][:-zelene(veze[odkial])]
    zelene_kam = veze[kam][-zelene(veze[kam]):]
    dokopy = cervene_odkial + zelene_kam
    dokopy.sort()
    pozicia = { y:x for x,y in enumerate(dokopy) }

    uz_zije = FenwickTree(len(dokopy))
    for z in zelene_kam: uz_zije.add( pozicia[z] )
    for c in cervene_odkial:
        p = pozicia[c]
        nad = uz_zije.prefix_sum(p) + pocet_cervenych_kam
        print(f'presun {c} z veze {odkial} do veze {kam} pod {nad} kociek')
        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: predpísané výšky

Čo sa zmení, keď dostaneme predpísané výšky p_i?
Prvá zmena je, že ak má nejaká veža dlhší usporiadaný sufix ako je jej hodnota p_i, nemôžeme si tento sufix nechať celý. Zelenú farbu vtedy teda dostane len spodných p_i kociek, všetky ostatné budú červené – musia niekam preč.
Opäť, nech c_i a z_i sú začiatočné počty červených a zelených kociek vo veži i. Pre každú vežu si navyše označme $d_i = p_i - z_i$. Číslo d_i udáva, koľko kociek do zeleného úseku tejto veže musíme pridať, aby mala na konci správnu výšku. Súčet všetkých d_i si označme D. Hodnota D je zjavne tiež rovná celkovému počtu červených kociek, teda súčtu všetkých c_i. V našom riešení teda určite potrebujeme spraviť aspoň D krokov.
Každej červenej kocke by sme chceli priradiť vežu inú ako jej vlastnú, do ktorej ju presunieme, a to tak, aby sme do každej veže i dokopy presunuli práve d_i kociek. Ak sa nám toto podarí spraviť, tak každú červenú kocku presunieme práve raz a tým zjavne dostaneme optimálne riešenie. Ako však ukazuje aj príklad v zadaní úlohy, toto nie vždy pôjde. Problém nastane vtedy, ak máme nejakú vežu, kde je na začiatku veľa červených kociek (tie musíme všetky dať preč) ale zároveň má na konci byť vysoká (takže musíme veľa kociek presunúť aj z iných veží sem).

Problémové veže: Vežu i nazveme *problémová*, ak platí $c_i + d_i > D$. Takáto veža nám sama osebe spôsobí, že naše riešenie bude musieť mať aspoň $c_i + d_i$ krokov, lebo aj niekedy musíme dať c_i červených kociek odtiaľto preč, aj niekedy musíme presunúť d_i kociek z iných veží sem.

Tvrdenie 1: Problémová veža je vždy nanajvýš jedna.

Dôkaz: Je zjavné, že nemôžu byť dve takéto veže i a j, lebo potom by už $(c_i + d_i) + (c_j + d_j)$ dokopy bolo viac ako $2D$, čo je spor s tým, že súčet všetkých c_i je D aj súčet všetkých d_i je D.

Tvrdenie 2: Ak je niektorá veža i problémová, vieme celú úlohu vyriešiť na presne $c_i + d_i$ krokov.

Dôkaz: Keďže $c_i + d_i > D$ a zároveň $d_i +$ (súčet všetkých ostatných d_j) $= D$, je c_i väčšie ako súčet všetkých ostatných d_j. Vieme teda začať tým, že v prvých c_i krokoch riešenia rozmiestnime všetky červené kocky z veže i do ostatných veží tak, aby sme každú vložili do niektorého zeleného úseku tam, kam patrí, a aby sme na konci v každej inej veži j mali zelený úsek dĺžky (aspoň) p_j.
Takto sme dosiahli stav, v ktorom veži i chýba d_i kociek do správnej výšky a ostatné veže už majú každá na spodku p_j usporiadaných kociek a nad tým majú dokopy d_i kociek navyše. Z každej privysokej veže teraz teda stačí presunúť jej prebytočné kocky do veže i a sme hotoví.

Tvrdenie 3: Ak nemáme žiadnu problémovú vežu, tak existuje riešenie na D krokov.

Dôkaz: Ukážeme, že vždy vieme červené kocky postupne presúvať tak, aby nám po žiadnom kroku nevznikla problémová veža.
Všimnime si, že ak má nejaká veža $c_i > 0$, tak z $c_i + d_i \leq D$ vyplýva, že nutne $d_i < D$, takže nutne existuje iná veža s $d_j > 0$. Preto vždy vieme spraviť ťah, v ktorom nejakú červenú kocku presunieme z veže, kde sú (veža i) do veže, kde ich treba (veža j).
Každý presun zmenší D o jedna. Vežu i nazveme *kritická*, ak platí $c_i + d_i = D$. Ak máme kritickú vežu, musíme v každom kroku riešenia buď presunúť jednu červenú kocku preč z nej (zmenšiť c_i) alebo presunúť jednu červenú kocku z inej veže sem (zmenšiť d_i), aby sa z nej nestala problémová veža.
Rovnako, ako sme vyššie zdôvodnili, že problémová veža je najviac jedna, vieme nahliadnuť, že kritické veže sú (pre $D > 0$) vždy najviac dve. Ak sú dve, vždy vieme presunúť červenú kocku z jednej do druhej. Ak je jedna, presunieme červenú kocku z nej, ak nejaké ešte má, resp. do nej, ak už nie. A ak nie sú kritické veže žiadne, môžeme spraviť ľubovoľný platný krok.
Náznak iného dôkazu: Tvrdenie 3 priamočiaro vyplýva z Hallovej vety o existencii perfektného párovania v bipartitnom grafe – Hall's Marriage Theorem. Vrcholy predstavujúce jednu vežu majú dosť susedov v opačnej partícii, a vrcholy predstavujúce viac ako jednu vežu majú ako susedov úplne celú opačnú partíciu.

Listing programu (Python)

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

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) ) ) )
```

A-III-2 Hnusný plot

Kubické riešenie

Označme $P[x][y][s]$ počet hnusných plotov, ktoré sú tvorené latami 1 až x, pričom prvá lata je y a potom plot pokračuje smerom s (teda dohora, na vyššiu hodnotu, ak $s = \uparrow$, resp. dodola ak $s = \downarrow$).
Tieto hodnoty vieme všetky spočítať dynamickým programovaním.

Pre $x = 0$ máme jeden prázdny plot a pre $x = 1$ jeden plot, ktorý nepokračuje dohora ani dodola.
Pre $x = 2$ máme $P[2][1][\uparrow] = P[2][2][\downarrow] = 1$.

Pre väčšie x platí nasledovný rekurentný vzťah: Pozrime sa na konkrétnu hodnotu $P[x][y][\uparrow]$. Druhá lata v tomto plote má dĺžku $z > y$ a následne plot pokračuje dodola. Celkový počet týchto plotov teda vieme zistiť tak, že zistíme počet pre každé konkrétne z a tieto počty sčítame.

A ako zistíme tento počet pre konkrétne z?
Všimnime si, že hnusných plotov s latami dĺžok 1 až x, ktoré začínajú latou y, je presne rovnako veľa, ako hnusných plotov, ktoré sú tvorené inými latami x rôznych dĺžok a začínajú y-tou najmenšou spomedzi nich. Inými slovami, zjavne nezáleží na konkrétnych dĺžkach lát, len na ich relatívnom poradí.
Ak zoberieme ľubovoľný hnusný plot dĺžky x začínajúci $[y, z, \dots]$ pre $z > y$ a zmažeme z neho úvodné y, dostaneme hnusný plot tvorený $x-1$ latami. Tieto laty majú všetky dĺžky od 1 po x okrem y. Náš hnusný plot teda začína $(z-1)$-tou najmenšou spomedzi použitých lát a potom pokračuje dodola. Platí preto, že takýchto hnusných plotov je presne $P[x-1][z-1][\downarrow]$.
Hľadaná hodnota $P[x][y][\uparrow]$ je teda rovná súčtu hodnôt $P[x-1][z-1][\downarrow]$ pre všetky z od $y+1$ po x.
Hodnoty pre hnusné ploty začínajúce dodola spočítame analogicky. Pozor len na rozdiel v ± 1: keď ideme z y do z dodola, tak po odstránení y je nový začiatok z-tou najmenšou spomedzi ostávajúcich lát, keďže tentokrát bolo y väčšie ako z.
Toto riešenie má časovú zložitosť $O(n^3)$: počítame rádovo n^2 rôznych hodnôt a každú z nich zistíme prejdením a sčítaním rádovo n možností.

Výpočet odpovede pre daný prefix plota

Ak máme daný prefix dĺžky 0, sčítame počty hnusných plotov pre všetky možné začiatky a smery pokračovania.
Ak máme daný prefix dĺžky 1, čiže je daný začiatok ale nie smer, sčítame počty hnusných plotov pre tento začiatok a oba možné smery pokračovania.
Ak je daný prefix dlhší, začneme tým, že skontrolujeme, či je samotný daný prefix hnusný – musí ísť striedavo hore a dole. Ak to nerobí, je odpoveď 0. Nech teraz M je množina obsahujúca poslednú latu daného prefixu (tú označme p) a všetky laty, ktoré ešte musíme umiestniť. Hľadanou odpoveďou je potom hodnota $P[x][y][s]$, kde $x = |M|$, y je poradové číslo laty p v množine M a smer s je opačný od posledného smeru v danom prefixe.

Kvadratické riešenie

Na to, aby sme časovú zložitosť zlepšili z kubickej na kvadratickú, nám stačí použiť vzorce, ktoré sme si už odvodili vyššie, len ich šikovnejšie vyhodnocovať. Všimnime si napríklad nasledovné vzťahy:

$$\begin{aligned} P[10][7][\uparrow] &= \quad\quad + P[9][7][\downarrow] + P[9][8][\downarrow] + P[9][9][\downarrow] \\ P[10][6][\uparrow] &= P[9][6][\downarrow] + P[9][7][\downarrow] + P[9][8][\downarrow] + P[9][9][\downarrow] \end{aligned}$$

Vidíme, že druhý súčet sa od prvého líši len v jednom člene navyše. A rovnaké vzťahy platia aj vo všeobecnosti: keď počítame hnusné ploty z x lát začínajúce latou $y-1$ a pokračujúce dohora, môžeme všetky možnosti rozdeliť do dvoch skupín:

- Ak je ďalšia lata dĺžky $z \geq y+1$, dostávame ploty v podstate identické s tými, ktoré začínajú latou y. (Bijekcia: Keď zoberieme ľubovoľný plot začínajúci $[y, z, \dots]$ a vymeníme v ňom laty y a $y-1$, dostaneme takýto plot.)
- Navyše nám pribudli nové možnosti jediného typu: po začiatočnej late $y-1$ pokračovať latou y. Len tieto teda musíme samostatne spočítať.

Toto nové pozorovanie nám umožní každú z hodnôt P spočítať v konštantnom čase, a teda celkovo dostaneme kvadratickú časovú zložitosť.

Listing programu (C++)

```
#include <iostream>
#include <vector>
using namespace std;
const long long MOD = 1000000007; // výsledok počítame modulo toto, aby nám nepretiekli premenná
const int MAXN = 5007;
long long dohora[MAXN][MAXN], dodola[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;
```

```
    dohora[2][0] = dodola[2][1] = 1;
    spolu[2] = 2;

    for (int n=3; n<=N; ++n) {
        // ploty z n lát, začínajú p, idú dohora
        for (int p=n-2; p>=0; --p) dohora[n][p] = (dohora[n][p+1] + dodola[n-1][p]) % MOD;
        // ploty z n lát, začínajú p, idú dodola
        for (int p=1; p<=n-1; ++p) dodola[n][p] = (dodola[n][p-1] + dohora[n-1][p-1]) % MOD;

        for (int p=0; p<n; ++p) spolu[n] = (spolu[n] + dohora[n][p] + dodola[n][p]) % MOD;
    }

    if (K == 0) {
        cout << spolu[N] << endl;
    } else if (K == 1) {
        int z = prefix[0];
        cout << (dohora[N][z] + dodola[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], mensie = 0;
            for (int k=0; k<K-1; ++k) if (prefix[k] < z) ++mensie;
            bool klesala = (prefix[K-2] > prefix[K-1]);
            cout << (klesala ? dohora[N-K+1][z-mensie] : dodola[N-K+1][z-mensie]) << endl;
        } else {
            cout << 0 << endl;
        }
    }
}
```

A-III-3 Zhasni

Potrebujeme nájsť takú okružnú cestu, pri ktorej do každej miestnosti vstúpime nepárne veľa ráz.
Pri riešení budeme používať terminológiu z teórie grafov. Miestnosti v dome sú vrcholy grafu, hrany zodpovedajú dverám medzi nimi.

Na mriežke: kedy neexistuje riešenie

Keď si mriežku ofarbíme ako šachovnicu, vidíme, že každé dvere vedú medzi bielou a čiernou miestnosťou. Graf popisujúci takýto dom je teda nutne *bipartitný*.
V bipartitnom grafe má nutne každá okružná cesta párny počet krokov, lebo musíme skončiť v tej istej partícii, v ktorej sme začínali, a pri každom kroku zmeníme farbu miestnosti.
Na druhej strane, dĺžku konkrétnej okružnej cesty vieme spočítať tak, že sa pre každý vrchol pozrieme, koľkokrát sme doň vošli, a tieto čísla sčítame. Preto ak je celkový počet vrcholov n nepárny a do každého z nich máme vojsť nepárny počet ráz, musí byť celková dĺžka takejto cesty nepárna.
Z toho vyplýva, že pre nepárne n v bipartitnom grafe zadaná úloha nemá riešenie.

Na mriežke: ako vyriešiť zvyšok

Keďže náš graf je súvislý, má nejakú kostru – množinu $n-1$ hrán, ktoré stačia na to, aby všetko bolo naďalej prepojené.
Zvoľme si ľubovoľnú konkrétnu kostru. Napr. tak, že z vrcholu 0 spustíme prehľadávanie do hĺbky (DFS) a pre každý iný vrchol zoberieme do kostry tú hranu, ktorou tento vrchol objavíme.
Našu okružnú cestu vieme teraz zostrojiť tak, že sa len po hranách tejto kostry prejdeme pomocou druhého, vhodne upraveného DFS.
Úprava bude vyzerať nasledovne: Vždy, keď naposledy opúšťame nejakú miestnosť x a vraciame sa z nej do miestnosti y, pozrieme sa, či v x zostalo zhasnuté svetlo. Ak nie, tak sa z y ešte raz vrátime do x, aby sme tam zhasli, a následne sa z x druhýkrát vrátime do y (a odvtedy už do x nepôjdeme).
Je zjavné, že takto zabezpečíme, že v každej miestnosti *inej ako miestnosť 0, kde prehľadávanie začíname aj končíme* takto dosiahneme, že tam na konci bude zhasnuté. Čo vieme povedať o miestnosti 0? V každej z $n-1$ zvyšných miestností sme zhasli, čiže každú z nich sme navštívili nepárne veľakrát. Keďže n je párne, znamená to, že $n-1$ je nepárne, a teda do ostatných miestností sme dokopy vstúpili nepárny počet krát. Tiež vieme, že dokopy má naša okružná cesta párnu dĺžku. Preto sme museli aj do miestnosti 0 vstúpiť nepárny počet krát, a teda je na konci zhasnuté aj tam.

Na všeobecnom grafe

Aby sme predchádzajúce riešenie zovšeobecnili na domy s ľubovoľnou topológiou, stačí nám vymyslieť, ako vyriešiť súvislé grafy, ktoré nie sú bipartitné. Ukáže sa, že v každom takomto grafe budeme vedieť všetko zhasnúť, a teda bipartitné grafy s nepárnym n ostanú jediným prípadom, kedy riešenie neexistuje.
Vyššie popísané riešenie (vyberieme si kostru a prejdeme sa po nej) vždy vyrobí okružnú cestu párnej dĺžky. Čo sa stane, ak ho použijeme na grafe, ktorý nie je bipartitný? Pre párne n tam toto riešenie bude naďalej fungovať, keďže bipartitnosť grafu pri dôkaze správnosti nijak nevyužívame. Pre nepárne n dostaneme takmer funkčné riešenie s jediným problémom: jedine v spálni, teda vrchole 0, ostane na konci rozsvietené. Toto potrebujeme nejako napraviť.

V každom grafe, ktorý nie je bipartitný, niekde existuje aspoň jeden jednoduchý cyklus nepárnej dĺžky. Nižšie si toto dokážeme a zároveň popíšeme algoritmus, ako jeden takýto cyklus nájsť.
Upravíme DFS, ktorým zostrojujeme kostru nášho grafu, nasledovne: Vrcholy si budeme explicitne farbiť dvoma farbami: štartový vrchol bude biely; všetci susedia bieleho vrcholu dostanú čiernu farbu a naopak. Ak sa nám podarí prejsť celý graf a nikde nestretnúť konflikt, práve sme overili, že graf je bipartitný (a ofarbenie, ktoré sme zostrojili, sú jeho partície). Vo všetkých ostatných prípadoch sa nám niekedy počas behu algoritmu stane, že spracúvame nejaký vrchol u a počas toho narazíme na hranu vedúcu do skôr spracovaného vrcholu v, ktorý dostal priradenú rovnakú farbu ako v.
Akonáhle nastane táto situácia, vieme, že náš graf nemôže byť bipartitný. A tiež vieme nájsť sľubovaný jednoduchý cyklus: stačí zobrať hranu uv a k nej jedinú cestu, ktorá vedie z u do v po našej DFS kostre. Táto cesta má párnu dĺžku (striedajú sa na nej biele a čierne vrcholy a má oba konce rovnakej farby), takže dokopy s hranou uv dostávame sľubovaný cyklus.

Kompletný algoritmus pre nepárne n teraz bude vyzerať nasledovne:

1. Spustíme DFS aby sme zostrojili jednu kostru a overili, či je náš graf bipartitný.
2. Ak sme zistili, že máme bipartitný graf, podáme správu, že riešenie neexistuje.
3. Ak sme našli spor, vyššie popísaným spôsobom zostrojíme cyklus nepárnej dĺžky a zapamätáme si, ktoré vrcholy ho tvoria.
4. Následne dokončíme naše prvé DFS a zostrojíme si jednu konkrétnu kostru nášho grafu.
5. Spustíme druhú fázu algoritmu: postupne sa prechádzame po zvolenej kostre a zabezpečujeme, že každý vrchol zhasneme, keď ho naposledy opúšťame.
6. Počas tejto fázy algoritmu spravíme jednu vec navyše: keď naše prehľadávanie prvýkrát vstúpi do niektorého vrcholu nami zapamätaného nepárneho cyklu, predtým, než budeme pokračovať ďalej v DFS po kostre, celý tento cyklus raz obehneme.
7. Keď naše DFS dobehne, vieme, že všetky vrcholy okrem vrcholu 0 sú zhasnuté. No tentokrát vďaka obehnutiu nepárneho cyklu vieme, že celková dĺžka našej okružnej cesty je nepárna. A ak sme počas nej každý z $n-1$ zvyšných vrcholov (čo je párny počet) navštívili nepárne veľakrát, znamená to, že aj do vrcholu 0 sme museli vstúpiť nepárne veľakrát, a teda máme zhasnuté aj tam.

Zjednodušenie implementácie

Namiesto zostrojenia jednoduchého cyklu môžeme zobrať celú okružnú cestu nepárnej dĺžky, ktorú tvorí cesta 0 do u po DFS kostre, hrana uv a cesta z v do 0 po DFS kostre.
Keďže táto okružná cesta zaručene obsahuje vrchol 0, môžeme sa po nej prebehnúť úplne na začiatku prepínania svetiel, a až následne spustiť DFS, ktoré v každom vrchole zhasne, keď ho opúšťa.
Celková časová zložitosť nášho riešenia (či už pôvodného alebo práve popísanej jednoduchšej verzie) je zjavne lineárna od veľkosti grafu, čiže $O(n+d)$, kde n je počet miestností a d je počet dvier v našom dome.
Ešte podotkneme, že nižšie uvedený program používa vstupný formát z podúlohy B, pričom predpokladáme, že graf na vstupe je jednoduchý – teda medzi každými dvoma miestnosťami vedú najviac jedny dvere.

Listing programu (C++)

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

int N, D;
vector< vector<int> > G;
vector<int> farba, rodic;
int u, v; // hrana na neparnom cykle

void dfs1(int kde, int odkial) {
    // zakorenime si strom, overime bipartitnost, ak nie je bipartitny, najdeme u+v
    for (int kam : G[kde]) if (kam != odkial) {
        if (farba[kam] == -1) {
            farba[kam] = !farba[kde];
            rodic[kam] = kde;
            dfs1(kam, kde);
        } else {
            if (farba[kam] == farba[kde] && u == -1) { u=kde; v=kam; }
        }
    }
}

vector<int> do_korena(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> &svieti) {
    // rekurzivne vyriesime cely podstrom s korenom kde
    for (int kam : G[kde]) if (rodic[kam] == kde) {
        cout << kam << "␣";
        svieti[kam] = !svieti[kam];
        dfs2(kam, svieti);
        cout << kde << "␣";
        svieti[kde] = !svieti[kde];
    }
    if (svieti[kde]) {
        cout << rodic[kde] << "␣" << kde << "␣";
        svieti[kde] = !svieti[kde];
        svieti[rodic[kde]] = !svieti[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); }

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

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

    vector<bool> svieti(N, true);
    if (N%2 == 1) {
        // obehneme raz neparny cyklus
        auto cesta_u = do_korena(u), cesta_v = do_korena(v);
        cesta_u.pop_back();
        reverse(cesta_u.begin(), cesta_u.end());
        for (int x : cesta_u) { cout << x << "␣"; svieti[x] = !svieti[x]; }
        for (int x : cesta_v) { cout << x << "␣"; svieti[x] = !svieti[x]; }
    }
    dfs2(0, svieti);
}
```

ŠTYRIDSIATY PRVÝ ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Forišek, Sebastian Hajdu, Paulína Smolárová
Recenzia: Michal Anderle, Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: NIVAM – Národný inštitút vzdelávania a mládeže, Bratislava 2026