72. ročník Matematické olympiády – 2022/2023

Řešení úloh domácího kola kategorie P

P-I-1 Tmavá zákoutí

Nejprve si všimněme, že pokud v nějakém sloupci není žádné osvětlené políčko, organizátoři se do takového sloupce, a tedy ani do žádného sloupce napravo od něj, neumí nikdy dostat. V opačném případě je vždy možné dostat se z Matfyzu na kolej například tak, že postupně rozsvítí každý sloupec. Toto pozorování stačí na zisk tří bodů: Nejprve ověříme, zda je v každém sloupci nějaké osvětlené políčko, a pokud ano, tak postupně vyzkoušíme rozsvítit každou podmnožinu sloupců, podíváme se, zda pro něj existuje osvětlená cesta z Matfyzu na kolej, a nakonec vrátíme nejmenší počet rozsvícených sloupců, pro něž taková cesta existovala.

Další důležité pozorování je, že existuje-li cesta z Matfyzu na kolej, na níž organizátoři rozsvítí nejvýše k sloupců, potom lze takovou cestu zvolit tak, aby organizátoři po rozsvícení každého sloupce v tomto sloupci popošli nahoru či dolů na vhodné políčko, z něj přešli doprava a pak se již nikdy do daného sloupce nevrátili. Všimněte si totiž, že pokud organizátoři rozsvítí sloupce $i < j$ a žádný sloupec mezi nimi, pak existuje cesta z nějakého políčka sloupce i do nějakého políčka sloupce j, která ve všech sloupcích $i+1, \ldots, j-1$ využívá od začátku rozsvícená políčka. Potom organizátoři po rozsvícení sloupce i mohou použít tuto cestu, dojít do sloupce j, ten rozsvítit a pak se již nikdy nemusí vracet nalevo od j.

To znamená, že si můžeme představit, že máme dva typy kroků. První typ je krok do sousedního políčka, které bylo od začátku osvětlené. Takový krok nás nestojí žádné rozsvěcování. Druhý typ je krok do libovolného (ne nutně sousedního) políčka ve stejném sloupci, na něj je potřeba rozsvítit jeden sloupec. Potom nás zajímá, kolik nejméně kroků druhého typu potřebujeme, abychom došli z Matfyzu na kolej. To lze spočítat například pomocí Dijsktrova algoritmu, ale stačí i varianta prohledávání do šířky pro hrany délky 0 a 1, kde vrcholy, do nichž přijdeme hranou délky 0, dáváme na začátek fronty, nikoli na konec. (Této variantě se často říká *0-1 BFS*.) Časová složitost takového algoritmu je $\mathcal{O}(n^3)$ (resp. $\mathcal{O}(n^3 \log n)$ při použití Dijkstrova algoritmu), jelikož máme celkem n^2 políček a z každého vede řádově n hran (do všech políček v daném sloupci vedou hrany délky 1, k tomu z něj mohou vycházet až čtyři hrany délky 0 do sousedních osvětlených políček). Takový algoritmus stačil na zisk pěti bodů.

Podívejme se nyní trochu důkladněji na to, co se stane, když se organizátoři rozhodnou rozsvítit nějaký sloupec. V tu chvíli po něm mohou libovolně chodit, ale výše jsme si rozmysleli, že jim stačí dojít na vhodné políčko tohoto sloupce, z něj udělat krok doprava na od začátku rozsvícené políčko v následujícím sloupci a znovu se už do rozsvíceného sloupce nevracet. Proto můžeme místo navštěvování políček, která na začátku nebyla rozsvícená, rovnou zkoušet skákat do rozsvícených políček ve stejném sloupci a ve sloupci o jedna napravo, čímž najednou započítáme rozsvícení sloupce a celou cestu do dalšího rozsvíceného políčka bez toho, abychom

tuto cestu museli explicitně procházet. Takové řešení, implementované pomocí 0-1 BFS, má složitost $\mathcal{O}(n^2 + m^2)$, kde m je počet rozsvícených políček, a stačí proto na zisk sedmi bodů. (Ve složitosti máme m^2 jako horní odhad velikosti grafu na rozsvícených políčkách a n^2 proto, abychom si mohli pomocná data ukládat do polí $n \times n$. Lze se obejít i bez toho například pomocí hešovacích tabulek, ale pro vyřešení úlohy to nebylo nutné.)

Pro zisk plného počtu bodů se potřebujeme zbavit druhé mocniny u m. To nelze jen lepší analýzou popsaného algoritmu, protože kdybychom měli dva sousední sloupce a v každém z nich řekněme $m/4$ rozsvícených políček, tak z každého rozsvíceného políčka vlevo můžeme skočit do každého rozsvíceného políčka vpravo. Toto je samozřejmě neefektivní – pro každé levé políčko totiž znovu zkoušíme rozsvítit celý sloupec, ale přitom jakmile sloupec celý rozsvítíme, nezáleží na tom, ze kterého políčka jsme to udělali. Ve vzorovém řešení tohle uděláme chytřeji:

Pro každý sloupec si přidáme virtuální „políčko“, které bude reprezentovat to, že daný sloupec celý rozsvítíme. Z každého rozsvíceného políčka povede do příslušného virtuálního políčka (orientovaná) hrana délky 1 a z virtuálního políčka reprezentujícího sloupec s povedou (orientované) hrany délky 0 do všech rozsvícených políček ve sloupci $s + 1$, pokud takový existuje. Tím zajistíme, že každé rozsvícené políčko bude incidentní s nejvýše čtyřmi hranami délky 0 do sousedních rozsvícených políček a k tomu s nejvýše dvěma dalšími hranami z/do virtuálních políček, tedy celkem budeme mít jen $\mathcal{O}(m)$ hran a časová složitost řešení bude tudíž $\mathcal{O}(n^2 + m)$, což stačí na plný počet bodů. (Tak, jak řešení popisujeme, tak neumí rozsvítit poslední sloupec a dojít po něm až dolů na kolej. Rozmyslete si ale, že místo toho mohou organizátoři vždy rozsvítit předposlední sloupec, dojít po něm úplně dolů a potom udělat krok doprava na kolej, takže i výše popsaný algoritmus dává správné výsledky.)

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

using namespace std;

typedef long long ll;
typedef pair<int, int> ii;

int n, m;
vector<vector<bool>> lights; // Je políčko rozsvícené?
vector<vector<int>> dist;    // Vzdálenost pro 0-1 BFS
vector<vector<int>> columns; // Seznam rozsvícených políček v každém sloupci
vector<vector<bool>> done;

int main() {
        cin >> n >> m;
        int x, y;

        lights.resize(n, vector<bool>(n, false));
        done.resize(n, vector<bool>(n+1, false));
        dist.resize(n, vector<int>(n+1, -1));
        columns.resize(n);
```

```
for (int i = 0; i < m; ++i) {
        cin >> x >> y;
        --x; --y; // Indexujeme od 0
        lights[x][y] = true;
        columns[x].push_back(y);
}

dist[0][0] = 0;

deque<ii> q;
q.push_back({0,0});

while (!q.empty()) {
        auto cur = q.front();
        q.pop_front();
        x = cur.first;
        y = cur.second;
        if (done[x][y]) continue;
        done[x][y] = true;

        // Políčky (x, n) reprezentujeme pomocný vrchol, že jsme
        // rozsvítili sloupec x-1
        if (y == n) {
                // Projdeme všechna rozsvícená políčka ve sloupci n
                for (auto ny : columns[x]) {
                        // Již jsme tu byli, vzdálenost nelze zmenšit
                        if (dist[x][ny] != -1) continue;

                        dist[x][ny] = dist[x][y];
                        q.push_front({x, ny});
                }
        } else {
                vector<ii> to_visit; // Všechny potenciálně možné kroky
                to_visit.push_back({x-1,y});
                to_visit.push_back({x+1,y});
                to_visit.push_back({x,y-1});
                to_visit.push_back({x,y+1});
                to_visit.push_back({x+1,n});

                for (auto nw : to_visit) {
                        int nx = nw.first,
                                ny = nw.second;
                        // Existuje takové políčko?
                        if (nx < 0 ||
                                ny < 0 ||
                                nx >= n ||
                                ny > n) continue;
                        if (dist[nx][ny] != -1) continue;
                        if (ny != n && !lights[nx][ny]) continue;
                        dist[nx][ny] = dist[x][y];
                        if (ny == n) {
                                ++dist[nx][ny];
                                q.push_back(nw);
                        } else {
                                q.push_front(nw);
                        }
                }
```

```
            }
        }
        cout << dist[n-1][n-1] << endl;
        return 0;
}
```

Alternativní řešení

Předpokládejme, že jsme našli nějaký způsob, jak se organizátoři mohou dostat na kolej. Jejich cestu si můžeme rozdělit na jednotlivé úseky, kdy vždy nějakou dobu chodí po rozsvícených políčcích, potom rozsvítí celý sloupec, někam po něm dojdou, pak se znovu chvíli pohybují po od začátku rozsvícených políčcích atd.

Pro rozsvícené políčko (x, y) označíme jako jeho *komponentu* množinu všech rozsvícených políček, do nichž se z (x, y) umíme dostat bez toho, abychom muselo rozsvěcovat nějaký sloupec. (Tohle přesně odpovídá komponentě souvislosti v grafu složeném z rozsvícených políček.) Organizátoři tedy nějakou dobu chodí po komponentě, pak rozsvítí nějaký sloupec, což jim umožní přejít do jiné komponenty, a pak dále pokračují v rámci ní.

Všimněte si, že pokud si promítneme nějakou komponentu na osu x (tj. podíváme se, ve kterých sloupcích leží nějaké políčko té komponenty), dostaneme vždy interval. Jinak řečeno, pokud má komponenta nějaká políčka ve sloupcích $i < j$, pak má políčko i ve všech sloupcích k takových, že $i \leq k \leq j$. Z toho plyne, že se organizátorům vždy vyplatí jít v rámci konkrétní komponenty co nejvíce doprava: Kdyby se rozhodli rozsvítit sloupec a přejít do jiné komponenty jinde než v nejpravějším políčku aktuální komponenty, mohou místo toho po aktuální komponentě přejít až do nejpravějšího políčka a rozsvítit jeho sloupec (a přejít do nějaké komponenty, v níž dle původního plánu daný sloupec navštívili). Tím se rozhodně nezvýší počet rozsvícených sloupců.

S tímto pozorováním lze vymyslet hladové řešení této úlohy. Nejprve si předpočítáme komponenty a promítneme si je na intervaly na ose x (to lze v čase $\mathcal{O}(m+n^2)$). Komponentu obsahující vrchol (n, n) z technických důvodů protáhneme, aby končila až ve (virtuálním) sloupci $n + 1$. Pro každé x od 1 do n si jako prav(x) označíme nejpravější konec nějakého intervalu, který začíná v x, (nebo -1, pokud v x žádný interval nezačíná) v čase $\mathcal{O}(n + m)$ například tak, že postupně projdeme všechny komponenty a vždy se podíváme, zda ta aktuální nedosahuje více doprava než všechny ostatní, které jsme do té doby viděli a začínají na stejném místě. Potom už spustíme samotné hladové hledání cesty pomocí následujícího algoritmu:

1. *akt* $\leftarrow 1$
   ◁ *Intervaly začínající od akt dále jsme ještě nekontrolovali.*
2. *odpoved* $\leftarrow 0$ ◁ *Celkový počet rozsvícených sloupců*
3. *pravo* $\leftarrow$ pravý konec intervalu komponenty obsahující $(1, 1)$
   ◁ *Sloupec pravo bude vždy ten, který budeme muset rozsvítit.*
4. Dokud *pravo* $\neq n + 1$:

5. *novy* $\leftarrow \max\{\text{prav}(i)$ pro i od *akt* do *pravo* $+ 1\}$
   ◁ *Tím najdeme nejpravější sloupec, kam se umíme dostat tak,*
   ◁ *že rozsvítíme sloupec pravo a přejdeme do jiné komponenty.*
6. Pokud *novy* $\leq$ *pravo*:
7. Skonči, na kolej se nelze dostat.
8. *akt* $\leftarrow$ *pravo* $+ 2$
   ◁ *Zkontrolovali jsme intervaly končící až do pravo* $+ 1$.
9. *pravo* $\leftarrow$ *novy*
10. *odpoved* $\leftarrow$ *odpoved* $+ 1$

Všimněte si, že v každé iteraci cyklu se proměnná *pravo* zvýší alespoň o 1, tedy iterací bude $\mathcal{O}(n)$. Každá iterace nám zabere $\mathcal{O}(1)$ práce plus tolik, kolik zabere řádek, kde nastavujeme proměnnou *novy*. Když se přes celý běh algoritmu podíváme na všechna spuštění řádku nastavujícího proměnnou *novy*, pro každé i od 1 do n se na prav(i) zeptáme nejvýše jednou, celkem tedy uděláme $\mathcal{O}(n)$ práce. Proto toto hladové hledání nejlepší cesty zabere $\mathcal{O}(n)$ času.

P-I-2 Inovativní vizualizace

Budeme používat terminologii teorie grafů. Organizační struktura je zakořeněný strom T, zaměstnanci jsou jeho vrcholy (ředitelka je kořen) a vztah přímé nadřízenosti/podřízenosti odpovídá hraně. Úkolem je strom na vstupu nakreslit tak, aby vrcholy byly ve vrcholech konvexního n-úhelníka, hrany byly vždy úsečky a žádné dvě hrany se nekřížily.

Nejprve si rozmyslíme, jak získat dva body za prostřední podúlohu. Pokud je ředitelka přímou nadřízenou všech ostatních (tj. příslušný graf je *hvězda*), může Michal umístit vrcholy libovolným způsobem, hrany se nikdy křížit nemohou. Pokud bude umisťovat vrcholy postupně, pro první má n možností, pro druhý $n-1$ možností a tak dále, celkem má tedy $n!$ možností.

Ve druhém případě má každý zaměstnanec kromě zaměstnance n právě jednoho přímého podřízeného, tedy graf je *cesta,* a hrany vedou vždy mezi vrcholy i a $i+1$. Předpokládejme, že Michal už všechny vrcholy umístil, a podívejme se na vrcholy 1 a 2. Hrana je spojující dělí n-úhelník na dvě části a musí platit, že všechny zbylé vrcholy budou umístěny v jedné z nich (protože jinak by některá z hran musela překřížit tu spojující vrcholy 1 a 2). To znamená, že druhá část musí být prázdná. Obecně platí, že když se podíváme na hranu spojující vrcholy i a $i+1$, tak ta dělí n-úhelník na dvě části a všechny vrcholy s čísly $i+2, \ldots, n$ musí být umístěné v jedné z nich. Pokud tedy bude Michal umisťovat vrcholy postupně, pro vrchol 1 má n možností, pro vrcholy $2, \ldots, n-1$ má právě dvě možnosti – musí být na jednom ze dvou konců souvislého úseku doposud umístěných vrcholů, a pro vrchol n zbude jen jedna možnost. Celkem je tedy odpověď $n2^{n-2}$.

Řešení obecného případu

Předpokládejme, že už máme nějaké vyhovující umístění vrcholů. Obejděme n-úhelník proti směru hodinových ručiček začínajíce u ředitelky a postupně si za-

pisujme čísla zaměstnanců, na něž jsme narazili, než se znovu vrátíme k ředitelce. Tím dostaneme nějakou posloupnost čísel $1, \ldots, n$ začínající jedničkou, kde se každé vyskytuje právě jednou (tj. *permutaci*). Posloupnosti, které mohou vzniknout takovým způsobem pro nějaké nakreslení, označíme jako *vyhovující*. Řešením úlohy je poté n krát počet vyhovujících posloupností, jelikož n způsoby můžeme vybrat, do kterého vrcholu umístíme ředitelku, a pak budeme postupně umisťovat zaměstnance dle pořadí daného posloupností.

Všimněme si, že posloupnost $(a_1, a_2, \ldots, a_n)$ je vyhovující právě tehdy, když neexistují indexy $i_1 < i_2 < j_1 < j_2$ takové, že $a_{i_1}a_{j_1}$ i $a_{i_2}a_{j_2}$ jsou obě hrany stromu. Jsou-li $a_{i_1}a_{j_1}$ i $a_{i_2}a_{j_2}$ hrany, tak po zanesení do n-úhelníka se budou křížit. Naopak pokud máme nakreslení, v němž se dvě hrany kříží, tak po jeho přepsaní na posloupnost dostaneme takovouto čtveřici. Toto pozorování již stačí pro zisk tří bodů. Postupně projdeme všechny možné permutace, pro každou z nich projdeme všechny dvojice hran stromu a ověříme, v jakém pořadí jsou v permutaci umístěny jejich koncové vrcholy.

Ekvivalentní podmínka pro vyhovující posloupnost

Pro libovolný vrchol v označíme jako *podstrom zakořeněný ve* v podgraf obsahující všechny (ne nutně přímé) potomky v, včetně v. Když řekneme jen *podstrom,* myslíme tím podstrom zakořeněný v nějakém nespecifikovaném vrcholu. Máme-li posloupnost $(a_1, \ldots, a_n)$, jako *interval* označíme libovolnou podposloupnost tvaru $(a_i, a_{i+1}, \ldots, a_j)$. Nyní dokážeme, že posloupnost je vyhovující právě tehdy, když každý podstrom je napsaný v nějakém intervalu.

Pokud posloupnost vyhovující není, výše jsme ukázali, že existují indexy $i_1 < i_2 < j_1 < j_2$ takové, že $a_{i_1}a_{j_1}$ i $a_{i_2}a_{j_2}$ jsou obě hrany stromu. Nechť T_1 je podstrom, který obsahuje a_{i_1} a a_{j_1} a je zakořeněný v jednom z těchto vrcholů a definujme T_2 analogicky pro a_{i_2} a a_{j_2}.

Nejprve dokážeme, že buď $a_{j_1} \notin T_2$, anebo $a_{i_2} \notin T_1$: Předpokládejme, že $a_{j_1} \in T_2$. Jelikož $a_{i_1}a_{j_1}$ je hrana, tak jeden z těchto vrcholů je přímým potomkem druhého. Podívejme se nyní na cestu z tohoto potomka až do kořene celého stromu T ze zadání. První dva vrcholy na této cestě jsou a_{i_1} a a_{j_1} v nějakém pořadí a dále na této cestě musí nutně ležet alespoň jeden z vrcholů a_{i_2}, a_{j_2} jakožto kořen podstromu T_2. Pokud tam leží a_{i_2}, máme hotovo, protože pak jistě $a_{i_2} \notin T_1$. I kdyby však na této cestě ležel jen vrchol a_{j_2}, tak stále vrchol a_{i_2} musí být jeho přímým potomkem, což pořád znamená, že nemůže ležet v T_1. Pokud $a_{i_2} \in T_1$, analogicky dokážeme, že $a_{j_1} \notin T_2$. Tím pádem vidíme, že buď T_1, anebo T_2 není napsaný v nějakém intervalu.

Abychom dokázali ekvivalenci, musíme ještě předpokládat, že existuje podstrom T', který není zapsaný v žádném intervalu, a dokázat, že potom posloupnost není vyhovující. To, že T' není zapsaný v intervalu, znamená, že existují indexy $i < j < k$ takové, že $a_i, a_k \in T'$ a $a_j \notin T'$. Všimněme si, že $i \neq 1$, jelikož $a_1 = 1$ je kořen celého stromu a jediný podstrom obsahující kořen je celý strom T.

Podívejme na cestu z vrcholu a_j do kořene, tedy vrcholu $a_1 = 1$. Žádný vrchol této cesty neleží v podstromě T', jelikož jinak by tam ležel i vrchol a_j. Tato cesta

začíná vrcholem s indexem v intervalu (i, k) a končí vrcholem s indexem mimo tento interval. Nechť $a_\ell a_{\ell'}$ je první hrana této cesty taková, že $i < \ell < k$ a buď $\ell' > k$, nebo $\ell' < i$. Předpokládejme, že $\ell' < i$, druhý případ se dokáže symetricky.

Nyní se podívejme na cestu z a_i do a_k. Tato cesta celá leží v podstromě T' (jelikož oba vrcholy a_i i a_k v něm leží), začíná vrcholem s indexem v intervalu (ℓ, ℓ') a končí vrcholem s indexem mimo tento interval. Označme jako $a_m a_{m'}$ první hranu této cesty takovou, že $\ell < m < \ell'$ a $m' < \ell$ nebo $m' > \ell'$ (rovnost nemůže nastat, jelikož a_ℓ ani $a_{\ell'}$ neleží v T'). Pokud $m' < \ell$, tak máme indexy $m' < \ell < m < \ell'$ takové, že $a_{m'} a_m$ i $a_\ell a_{\ell'}$ jsou hrany, tedy posloupnost není vyhovující. Pokud $m' > \ell'$, tak nevyhovujícnost posloupnosti svědčí indexy $\ell < m < \ell' < m'$.

Pro vyřešení úlohy nám tedy stačí spočítat počet vyhovujících posloupností. Problém si zdánlivě zesložitíme tím, že to budeme chtít spočítat pro každý podstrom. Přesněji řečeno, pro každý vrchol v stromu T spočítáme, kolik existuje vyhovujících posloupností pro podstrom T' zakořeněný ve v takových, že v je na prvním místě. Díky tomu můžeme napsat rekurzivní vztah, který nám daný počet pro v spočítá na základě znalosti příslušných počtů pro všechny potomky v.

Pokud je v list, odpověď je 1. Jinak víme, že každý podstrom T' musí být napsaný v nějakém intervalu. Tedy speciálně pro každého potomka w vrcholu v bude podstrom zakořeněný ve w napsaný v nějakém intervalu. Posloupnost bude mít tedy tvar, kde prvním členem bude v, pak bude interval odpovídající některému z potomků, pak interval odpovídající dalšímu z potomků atd. Řekněme, že v má $\mathrm{pot}(v)$ potomků. Potom je $\mathrm{pot}(v)!$ možností, jak uspořádat intervaly odpovídající jednotlivým potomkům. V intervalu pro potomka w potom téměř bude napsána vyhovující podposloupnost pro podstrom zakořeněný ve w (jejichž počet spočítáme rekurzí) s tou výjimkou, že w nemusí být na prvním místě, nýbrž může být kdekoliv mezi intervaly odpovídajícími podstromům zakořeněných v jeho potomcích, úplně na začátku nebo úplně na konci. To je celkem $\mathrm{pot}(w) + 1$ možností.

Rekurze bude tedy vypadat takto:

Algoritmus POČET(v)

1. Pokud je v list, vrátíme 1.
2. $ret \leftarrow 1$
3. Pro všechny potomky w vrcholu v:
4. $\quad ret \leftarrow ret \cdot$ POČET(w)
5. $\quad ret \leftarrow ret \cdot (\mathrm{pot}(w) + 1)$
6. $ret \leftarrow ret \cdot \mathrm{pot}(v)!$
7. Vrátíme *ret*.

Při implementaci samozřejmě nesmíme zapomenout na to, že výsledek chceme počítat modulo 10^9+7. Abychom dosáhli dobré časové složitosti, nejprve si předpočítáme funkci $\mathrm{pot}(v)$, která počítá počet potomků v, což zřejmě dokážeme v lineárním čase. Také si předpočítáme všechny faktoriály až do n (modulo $10^9 + 7$), což také lze v lineárním čase, pokud si vzpomeneme, že $(n + 1)! = (n + 1) \cdot n!$.

Nakonec nám tedy zbývá samotná funkce Počet(v). Všimněte si, že pro každý vrchol w zavoláme Počet(w) nejvýše jednou, totiž ve chvíli, kdy procházíme potomky jeho rodiče, stejný argument ukáže, že pro každý vrchol w provedeme krok 5 nejvýše jednou. Jelikož ve zbytku funkce Počet(v) děláme jen konstantně mnoho práce, plyne z toho, že celkově náš program na všech voláních Počet(v) dohromady stráví $\mathcal{O}(n)$ času. Tím získáváme program, kterému stačí $\mathcal{O}(n)$ času a $\mathcal{O}(n)$ paměti, a získá tedy plný počet bodů.

Poznámka

Ve skutečnosti lze funkci Počet(v) počítat ještě jednodušeji bez rekurzivního volání. Indukcí „od listů nahoru“ lze dokázat, že pokud označíme jako T' podstrom zakořeněný ve v, pak

$$\text{Počet}(v) = \text{pot}(v)! \prod_{w \in T' \setminus \{v\}} (\text{pot}(w) + 1)!$$

Všimněte si, že označíme-li jako $\deg(v)$ počet sousedů v v T, pak pro každý vrchol v kromě kořene platí $\deg(v) = \text{pot}(v)+1$, a je-li v kořen, pak $\deg(v) = \text{pot}(v)$. Odpověď na úlohu má proto následující jednoduchý vzoreček:

$$n \prod_{v \in T} \deg(v)!$$

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

#define MOD 1000000007

using namespace std;
typedef long long ll;

int n;
vector<vector<int>> e;

// Stupeň vrcholu v celém stromě, tj. pro všechny vrcholy
// kromě kořene to bude počet potomků + 1, pro kořen jen počet potomků.
vector<int> deg;
vector<ll> fac;

ll pocet(int v, int p = -1) {
        ll ret = 1;
        for (auto u : e[v]) if (u != p) {
                ret = ret * pocet(u, v) % MOD;
        }
        return ret * fac[deg[v]] % MOD;
}

int main() {
        cin >> n;
        e.resize(n, vector<int>{});
        deg.resize(n, 0);
        int a;
```

```
        for (int i = 1; i < n; ++i) {
                cin >> a;
                --a;
                e[a].push_back(i);
                e[i].push_back(a);
                ++deg[a];
                ++deg[i];
        }

        // Předpočítáme si faktoriály
        fac.resize(n + 1);
        fac[0] = 1;
        for (int i = 1; i <= n; ++i)
                fac[i] = (fac[i-1] * i) % MOD;

        cout << (n * pocet(0) % MOD) << endl;
        return 0;
}
```

P-I-3 Nevhodný dárek

Předpokládejme, že použitím Pepova algoritmu na posloupnost (x) dostaneme posloupnost $(x_1, \ldots, x_n)$ a že použitím algoritmu na posloupnost (y) dostaneme posloupnost $(y_1, \ldots, y_n)$. Rozmyslete si, že potom použitím algoritmu na posloupnost (x, y) dostaneme posloupnost $(x_1, \ldots, x_n, y_1, \ldots, y_n)$. Z toho plyne obecně, že máme-li posloupnost $(z_1, \ldots, z_k)$ a aplikujeme na ni algoritmus, dostaneme totéž, jako kdybychom aplikovali algoritmus postupně na (z_1), (z_2) až (z_k) a výsledky spojili zpět v jednu posloupnost.

S tímto pozorováním nyní indukcí dokážeme, že začneme-li s posloupností (x) a skončíme s posloupností $(x_1, \ldots, x_n)$, potom v posloupnosti $(x_1, \ldots, x_n)$ bude právě x jedniček. Je-li x rovno nule či jedné, tvrzení zřejmě platí. Předpokládejme nyní, že tvrzení platí pro všechna čísla $0, \ldots, x-1$ a udělejme jeden krok algoritmu na posloupnost (x). Tím dostaneme $(\lfloor \frac{x}{2} \rfloor, x \bmod 2, \lfloor \frac{x}{2} \rfloor)$. Z indukčního předpokladu víme, že po použití algoritmu na $(\lfloor \frac{x}{2} \rfloor)$ dostaneme $\lfloor \frac{x}{2} \rfloor$ jedniček, stejně tak pro $(x \bmod 2)$, což je už samo o sobě nula nebo jedna. To znamená, že po použití algoritmu na (x) dostaneme skutečně $\lfloor \frac{x}{2} \rfloor + (x \bmod 2) + \lfloor \frac{x}{2} \rfloor = x$ jedniček, a získáváme tím dva body.

Předpokládejme znovu, že použitím Pepova algoritmu na posloupnost (x) dostaneme posloupnost $(x_1, \ldots, x_n)$. Jak velké je n v závislosti na x? Označme jako b počet číslic ve dvojkovém zápisu čísla x (čili b je *počet bitů* čísla x). Pro $x > 0$ platí $b = \lceil \log_2(x+1) \rceil$, tedy speciálně $b \in \mathcal{O}(\log(x))$. Indukcí podle b dokážeme, že $n = 2^b - 1$. Pro $b = 1$ to platí, jelikož máme-li $b = 1$, pak $x = 0$ nebo $x = 1$ a v obou případech $n = 1 = 2^1 - 1$. Předpokládejme nyní, že tvrzení platí pro $1, \ldots, b-1$ a mějme nějaké b-bitové číslo x. Použijeme-li jeden krok Pepova algoritmu, dostaneme posloupnost $(\lfloor \frac{x}{2} \rfloor, x \bmod 2, \lfloor \frac{x}{2} \rfloor)$. Počet bitů $\lfloor \frac{x}{2} \rfloor$ je $b-1$ a $x \bmod 2$ má jeden bit, tedy na všechny tři členy můžeme použít indukční předpoklad a dostaneme, že výsledná posloupnost pro (x) bude mít $(2^{b-1}-1) + (2^1-1) + (2^{b-1}-1) = 2 \cdot 2^{b-1} - 1 = 2^b - 1$ členů, což jsme chtěli dokázat.

Čísla $c \leq 10^5$ mají nejvýše 17 bitů, tedy výsledná posloupnost bude mít nejvýše $2^{17} - 1 = 131071$ členů. Tím pádem je možné všechny Pepovy kroky odsimulovat (pokud to člověk implementuje dostatečně efektivně) a získat tak 3 body.

Řekněme, že chceme zjistit člen x_i posloupnosti $(x_1, \ldots, x_n)$, jež vznikne použitím Pepova algoritmu na (x). Použijeme-li na (x) jeden krok Pepova algoritmu, dostaneme $(\lfloor \frac{x}{2} \rfloor, x \bmod 2, \lfloor \frac{x}{2} \rfloor)$. Je-li $i \leq 2^{b-1} - 1$, vznikne x_i použitím Pepova algoritmu na první člen $\lfloor \frac{x}{2} \rfloor$. Je-li $i = 2^{b-1}$, pak x_i je přímo $x \bmod 2$, jinak můžeme x_i nalézt jako $(i - 2^{b-1})$-tý člen posloupnosti, která vznikne použitím Pepova algoritmu na druhý výskyt $\lfloor \frac{x}{2} \rfloor$. To nám dává následující rekurenci pro posloupnost PRVEK(x, i), která vrátí x_i.

Algoritmus PRVEK(x, i)

1. $b \leftarrow$ *počet_bitů*(x)
2. ◁ *Předpokládáme, že i je validní, tj.* $1 \leq i \leq 2^b - 1$.
3. Pokud $i = 2^{b-1}$:
4. Vrátíme $x \bmod 2$.
5. Pokud $i \leq 2^{b-1} - 1$:
6. Vrátíme PRVEK$(\lfloor \frac{x}{2} \rfloor, i)$.
7. Jinak: ◁ *Zde platí* $i > 2^{b-1}$.
8. Vrátíme PRVEK$(\lfloor \frac{x}{2} \rfloor, i - 2^{b-1})$.

Toto stačí na zisk šesti bodů. Všimněte si, že s každým rekurentním voláním funkce PRVEK se sníží počet bitů o 1, proto volání PRVEK(x, i) zabere nejvýše $\mathcal{O}(b) = \mathcal{O}(\log(x))$ rekurentních volání. Proto nalezení počtu jedniček v intervalu $\ell, \ldots, r$ vzniklých z posloupnosti (c) můžeme udělat tak, že postupně zjistíme hodnotu všech prvků daném intervalu, což zabere čas $\mathcal{O}((r-\ell)\log(c))$. (Technicky vzato je ještě potřeba vyřešit, jak efektivně počítat počet bitů x. Nejjednodušší varianta je mírně upravit funkci PRVEK tak, aby jako parametr explicitně přijímala b, $\lfloor \frac{x}{2} \rfloor$ má pak $b - 1$ bitů. Počet bitů c si můžeme na začátku spočítat jednoduše v čase $\mathcal{O}(\log c)$.

Pro zisk plného počtu bodů stačí zkombinovat předchozí algoritmus s pozorováním ze začátku vzorového řešení. Předchozí algoritmus je neefektivní ve chvíli, kdy se ptáme postupně na všechny prvky intervalu, který vznikne z nějakého čísla x, které se v nějaké fázi Pepova algoritmu objeví jako člen aktuální posloupnosti. Tehdy totiž rovnou víme, že v daném intervalu bude x jedniček. Efektivnější funkce bude vypadat následovně:

Algoritmus POČET(x, ℓ, r)

1. $b \leftarrow$ *počet_bitů*(x)
2. Pokud $r < 1$ nebo $\ell > 2^b - 1$:
3. Vrátíme 0.
4. Pokud $\ell \leq 1$ a $r \geq 2^b - 1$:
5. Vrátíme x.

6. Jinak:
7. $\quad posun \leftarrow 2^{b-1} - 1$
8. $\quad$ Vrátíme POČET$(\lfloor \frac{x}{2} \rfloor, \ell, r)$
$\quad\quad$ + POČET$(x \bmod 2, \ell - posun, r - posun)$
$\quad\quad$ + POČET$(\lfloor \frac{x}{2} \rfloor, \ell - posun - 1, r - posun - 1)$.

Rozmyslíme si, že zavoláme-li funkci POČET(c, ℓ, r), tak celkem proběhne nejvýše $\mathcal{O}(\log(c))$ rekurzivních zavolání. To již stačí na plný počet bodů, pokud počet bitů budeme počítat jako v předchozím případě. Pokud znáte důkaz, že zjištění součtu intervalu pomocí intervalového stromu vyžaduje logaritmický počet kroků, tak důkaz našeho tvrzení je velice podobný.

Dokážeme, že pro každý počet bitů b budou nejvýše dvě instance funkce POČET s parametrem s daným počtem bitů, které se budou dál rekurzivně volat. Je-li to pravda, tak z toho již rovnou plyne, že celkem proběhne nejvýše $\mathcal{O}(\log(c))$ rekurzivních zavolání funkce POČET.

Každá možná instance funkce POČET odpovídá nějakému intervalu ve výsledné posloupnosti. Instance, které jsou mimo zadaný úsek, rozhodně nebudeme dále počítat, a pokud je zavoláme, ihned vrátíme nulu. Instance, které jsou celé uvnitř tázaného úseku, také zvládneme vyhodnotit hned, a tedy zbývají pouze instance, které leží na hraně (tedy odpovídají nějakým prvkům uvnitř i mimo zadaný úsek). Ovšem zadaný úsek má jen dvě hranice a přes každou z nich může vést jen jeden úsek odpovídající instanci funkce pro daný počet bitů, tedy rekurzící se instance budou nejvýše dvě.

P-I-4 Hvězdní věštci

Část a)

Obdobně jako v příkladu 3 ze studijního textu nahlédneme, že Hvězdné impérium tvoří kružnici právě tehdy, když každý systém má právě dva sousedy. Nemusíme tedy vůbec používat věštce a dostáváme řešení s K=0: Každý systém odpoví ANO, pokud má právě dva sousedy. Pokud Hvězdné impérium tvoří kružnici, všechny systémy odpoví ANO, jinak alespoň jeden systém odpoví NE. V pseudokódu:

```
if (D == 2)
    return ANO;
else
    return NE;
```

Část b)

S použitím věštce může každý systém v určit jednoho ze svých sousedů s_v, se kterým bude spárovaný, tomuto sousedu poslat *true* a všem ostatním *false*. Pak odpoví ANO, pokud obdrží *true* od souseda s_v a *false* od všech ostatních sousedů; jinak odpoví NE. Na určení s_v pro systém v s D sousedy stačí vyvěštit číslo mezi 0 a $D-1$, tedy $\lceil \log_2 D \rceil$ bitů. Jelikož $D < N$ pro každý systém, dostáváme $K = \lceil \log_2 N \rceil$.

Následuje řešení v pseudokódu.

```
bool send[D], receive[D];
unsigned(ceil(log(N))) R;

for (i = 0, ..., D - 1)
    send[i] = (i == R);

for (i = 0, ..., D - 1)
    if (receive[i] != (i == R))
        return NE;

return ANO;
```

Pokud Hvězdné impérium má nějaké perfektní párování M, věštec v každém systému může uhodnout souseda v M, a výše popsaný algoritmus zajistí, že odpověď bude ANO ve všech systémech. Na druhou stranu, pokud všechny systémy odpoví ANO, můžeme systémy rozdělit do dvojic ve tvaru $\{v, s_v\}$ a Hvězdné impérium tedy má perfektní párování.

Část c)

Tuto úlohu můžeme řešit obdobně jako příklad 3 ze zadání: Každý systém si nechá vyvěštit svoji polohu p na cestě a pošle ji všem sousedům. Odpoví pak ANO, pokud jsou splněny následující dvě podmínky:

- buď p je 0 nebo od alespoň jednoho souseda obdrží $p - 1$, a zároveň
- buď p je $N - 1$ nebo od alespoň jednoho souseda obdrží $p + 1$.

Tento algoritmus zjevně potřebuje vyvěštit $K = \lceil \log_2 N \rceil$ bitů v každém systému. V pseudokódu:

```
typedef unsigned(ceil(log(N))) pozice;

pozice R;
pozice send[D], receive[D];

if (R >= N)
    return NE;

for (i = 0, ..., D - 1)
    send[i] = R;

bool pred_ok = (R == 0);
bool za_ok = (R == N - 1);
for (i = 0, ..., D - 1) {
    if (receive[i] + 1 == R) pred_ok = true;
    if (receive[i] == R + 1) za_ok = true;
}

if (pred_ok && za_ok)
    return ANO;
else
    return NE;
```

Jestliže Hvězdné impérium má hamiltonovskou cestu, věštci mohou přežít tak, že správně vyvěští pozici systémů na cestě, čímž zajistí odpověď ANO ve všech systémech. Naopak, předpokládejme, že všechny systémy odpověděly ANO, a nechť s_0 je jeden ze systémů, kterým jejich věštec vyvěštil nejmenší pozici. Tato pozice musí

být 0, jinak by výše popsaný algoritmus zajistil, že alespoň jednomu sousedovi byla vyvěštěna menší pozice. Algoritmus také zajistí, že nějakému sousedovi s_1 systému s_0 je vyvěštěna pozice 1, sousedovi s_2 systému s_1 je vyvěštěna pozice 2, ..., až dokud takto nedorazíme do systému s_{N-1} s vyvěštěnou pozicí $N-1$. Pak $s_0 s_1 s_2 \dots s_{N-1}$ je hamiltonovská cesta.