

Na řešení úloh máte 4,5 hodiny čistého času.

Řešení každého příkladu musí obsahovat:

- **Popis řešení**, to znamená slovní popis použitého algoritmu, argumenty zdůvodňující jeho správnost (případně důkaz správnosti algoritmu), diskusi o efektivitě vašeho řešení (časová a paměťová složitost). Slovní popis řešení musí být jasný a srozumitelný i bez nahlédnutí do samotného zápisu algoritmu (do programu).
- **Program**. V úlohách **P-III-1** a **P-III-2** je třeba uvést dostatečně podrobný zápis algoritmu, nejlépe ve tvaru zdrojového textu nejdůležitějších částí programu v jazyce Pascal nebo C. Ze zápisu můžete vynechat jednoduché operace jako vstupy, výstupy, implementaci jednoduchých matematických vztahů apod. V úloze **P-III-3** zapište navržený algoritmus ve formě programu v jazyce ALIK.

Hodnotí se nejen správnost programu, ale také kvalita popisu řešení a efektivita zvoleného algoritmu.

P-III-1 Náhrdelníky

K. O. Lektor je sběratel náhrdelníků. Náhrdelníky se liší počtem, pořadím a druhy použitých drahokamů. Jeho finanční zdroje nejsou neomezené, proto by se chtěl vyhnout tomu, aby kupoval více kusů stejného typu náhrdelníku. Vymyslel tedy kódování, které funguje takto: každému druhu drahokamů přiřadil písmeno abecedy a tyto kódy zapsal v pořadí, v jakém se nacházejí na náhrdelníku, počínaje libovolným z nich. Dále si pořídil stroj, který pro zadaný kód zjistí, zda již má příslušný náhrdelník ve sbírce.

Obchodníci s náhrdelníky si samozřejmě rychle povšimli slabiny tohoto systému – pokud náhrdelník pootočili, případně obrátili, kód se změnil a K. O. Lektor si tak nakoupil několik duplikátů – například náhrdelník ABCA si koupil i jako AABC a ACBA. Chtěl by tedy svůj stroj vylepšit tak, aby tuto situaci rozeznal. Napište program implementující toto vylepšení.

Program dostane na vstupu několik kódů náhrdelníků x_1, \dots, x_N ($1 \leq N \leq 1\,000\,000$) a měl by pro každý z nich vypsát, zda si ho má K. O. Lektor koupit nebo ne tak, aby získal každý typ náhrdelníků právě jednou. Pokud je náhrdelník x_i různý od všech náhrdelníků x_1, \dots, x_{i-1} (včetně rotací a zrcadlení), program odpoví „Kup to“, v opačném případě odpoví „Podvod“.

Program může využívat starý stroj jako „černou skříňku“, která si pamatuje množinu kódů náhrdelníků (na počátku práce množinu prázdnou) a je schopna do této množiny kódy přidávat (zavoláním procedury `Pridej`) a zjišťovat, jestli stroj již nějaký kód zná (zavoláním funkce `UzMam`, která vrací `true` (1 v C), pokud někdy předtím byla zavolána procedura `Pridej` pro stejný kód):

```
int UzMam (const char *kod);           /* v C */
void Pridej (const char *kod);

function UzMam (var kod:string):boolean; { v Pascalu }
procedure Pridej (var kod:string);
```

K. O. Lektor má na váš program následující požadavky: Paměťová složitost nesmí záviset na počtu testovaných náhrdelníků. Časová složitost by měla být co nejnižší, přičemž volání funkce starého stroje počítáme jako operace běžící v lineárním čase. Ze dvou programů o stejné časové složitosti pak považuje za lepší ten, který funkce starého stroje volá méněkrát.

Příklad: V levém sloupci je vstup, v pravém jediný správný výstup:

ABCA	Kup to
ACBA	Podvod
AABC	Podvod

P-III-2 Mosty

Magické observatoře (MO) nedávno otevřely nový areál v Horních Mokřadech. Podnebí v Horních Mokřadech je bohužel poměrně deštivé, a tak se vedení MO rozhodlo, že propojí všechny budovy v areálu nadzemními krytými mosty. Vedení MO samozřejmě chce, aby si dodatečné stavební úpravy vyžádaly co nejmenší náklady. Proto požaduje, aby celková délka mostů, které propojí budovy v areálu, byla co nejmenší. Navíc, kvůli vlivu energetických zón v okolí Horních Mokřad, musí všechny mosty vést severojižně.

Vášim úkolem je vytvořit algoritmus, který pro zadanou mapu areálu MO navrhne nejlepší možné propojení budov severojižními mosty. Pokud všechny budovy nelze navzájem propojit, pak algoritmus vypíše vhodnou zprávu.

Na vstupu algoritmus obdrží mapu areálu MO jako čtvercovou síť o N řádcích a M sloupcích. Budovy v areálu jsou reprezentovány znaky x , volná prostranství tečkami. Budova je maximální oblast tvořená políčky x , která se mezi sebou dotýkají hranami. Vášim úkolem je navrhnout propojení budov severojižními (na mapě svislými) mosty tak, aby mezi každými dvěma budovami existovala cesta používající pouze políček x a mostů: Mezi dvěma políčky x lze přejít, pokud sousedí hranou, mosty lze používat pouze v severojižním směru, tj. seshora dolů nebo zdola nahoru v jejich směru na mapě.

Při (popisu) řešení této úlohy se soustředte na efektivní nalezení optimálního propojení mezi budovami a zdůvodnění správnosti navrženého algoritmu.

Příklad 1:

Pro $M = 8$ a $N = 5$ uvažme následující mapu areálu:

```

. .xxxxx.
. . . . .x
. . .x. . .x
. x.x. . . .
. xxx. .xx

```

Areál je tvořen čtyřmi budovami (na následujícím obrázku jsou jejich políčka označena čísly 1 až 4; všimněte si, že políčka s čísly 1 a 2 tvoří dvě různé budovy):

```

. .11111.
. . . . .2
. . .3. . .2
. 3.3. . . .
. 333. .44

```

Optimální řešení úlohy pro zadanou mapu je propojit dvojici budov 1 a 3 mostem délky jedna, dvojici budov 2 a 4 rovněž mostem délky jedna a budovy 1 a 4 mostem délky tři:

```

. .11111.
. . . | . . | 2
. . .3. . | 2
. 3.3. . | |
. 333. .44

```

Protože mosty lze používat pouze severojižně, nemůžeme z mostu spojujícího budovy 1 a 4 přejít do budovy 2 a je třeba vybudovat i most mezi budovami 2 a 4.

P-III-3 ALÍK

Sestrojte program pro stroj ALIK, který spočte dvojkový logaritmus zadaného nenulového čísla x , tedy pozici nejlevější jedničky v x . Pozice bitů rostou zprava doleva, nejnižší řád je na pozici 0.

Příklad: Dvojkový logaritmus dvojkového čísla **00110001** je 5, logaritmus z **00000001** je 0.

Studijní text: (stejný jako v krajském kole)

Aritmeticko-logický integerový kalkulátor (zkráceně ALIK) je počítačový stroj pracující s W -bitovými celými čísly v rozsahu 0 až $2^W - 1$ včetně; kdykoliv budeme hovořit o číslech, půjde o tato čísla. Budeme je obvykle zapisovat ve dvojkové soustavě polotučnými číslicemi a vždy si na začátek dvojkového zápisu doplníme příslušný počet nul, aby číslic (bitů) bylo právě W . Většinou také nebudeme rozlišovat mezi číslem a jeho dvojkovým zápisem, takže i -tým bitem čísla budeme rozumět i -tý bit jeho dvojkového zápisu (bity číslujeme zprava doleva od 0 do $W - 1$).

Paměť stroje je tvořena 26 registry pojmenovanými a až z . Každý registr vždy obsahuje jedno číslo.

ALIK se řídí programem, což je posloupnost přiřazovacích příkazů typu $registr := výraz$, přičemž výraz může obsahovat konstanty (čísla zapsaná ve dvojkové soustavě), registry, závorky a následující operátory (řecká písmena značí podvýrazy, v pravém sloupci jsou priority operátorů):

$\alpha + \beta$	sečte čísla α a β . Pokud je výsledek větší než $2^W - 1$, číslice vyšších řádů odřízne. Jinými slovy, počítá součet modulo 2^W .	4
$\alpha - \beta$	odečte od čísla α číslo β . Pokud je $\alpha < \beta$, spočte $2^W + \alpha - \beta$, čili rozdíl modulo 2^W .	4
$\alpha * \beta$	vynásobí dvě čísla, výsledek opět modulo 2^W .	6
α / β	vydělí číslo α číslem β ; dělení nulou dá vždy výsledek 0.	6
$\alpha \% \beta$	vrátí zbytek po dělení čísla α číslem β , čili $\alpha - \beta * (\alpha / \beta)$; pokud je $\beta = 0$, je výsledek roven α .	6
$\neg \alpha$	spočte bitovou negaci čísla α , což je číslo, jehož i -tý bit je 0 právě tehdy, je-li i -tý bit čísla α roven 1 , a naopak.	9
$\alpha \wedge \beta$	bitové operace: <i>and</i> , <i>or</i> a <i>xor</i> . Vyhodnocují se tak, že se i -tý bit výsledku spočte z i -tého bitu	8
$\alpha \vee \beta$	čísla α a i -tého bitu čísla β podle následujících tabulek:	7
$\alpha \oplus \beta$		7

$0 \wedge 0 = 0$	$0 \vee 0 = 0$	$0 \oplus 0 = 0$
$0 \wedge 1 = 0$	$0 \vee 1 = 1$	$0 \oplus 1 = 1$
$1 \wedge 0 = 0$	$1 \vee 0 = 1$	$1 \oplus 0 = 1$
$1 \wedge 1 = 1$	$1 \vee 1 = 1$	$1 \oplus 1 = 0$

Příklad 2:

Pro $M = 6$ a $N = 5$ uvažme následující mapu areálu:

```

xxxxxx
x. . . .x
x.xx.x
x. . . .x
xxxxxx

```

Areál je tvořen dvěma budovami a k jejich propojení stačí vybudovat jeden most délky 1:

```

xxxxxx
x. | . .x
x.xx.x
x. . . .x
xxxxxx

```

Příklad 3:

Pro $M = 5$ a $N = 4$ uvažme následující mapu areálu:

```

xxx. .
. . . . .
xxx. .
. . .xx

```

Areál je tvořen třemi budovami. Protože budovu v pravém dolním rohu nelze spojit s žádnou jinou budovou mostem, všechny budovy nelze vzájemně propojit.

$\alpha \ll \beta$	posune číslo α o β bitů doleva, čili doplní doprava β nul a odřízne prvních β bitů zleva, aby byl výsledek opět W -bitový.	2
$\alpha \gg \beta$	posune číslo α o β bitů doprava, čili doplní doleva β nul a odřízne posledních β bitů vpravo, aby byl výsledek opět W -bitový.	2

Pokud závorky neurčí jinak, vyhodnocují se operátory s vyšší prioritou před operátory s nižší prioritou. V rámci stejné priority se pak vyhodnocuje zleva doprava (s výjimkou operátoru \neg , který je unární, a tudíž se musí vyhodnocovat zprava doleva).

Příklad 0: (jak fungují operátory; zde máme $W = 4$)

$a + b \wedge c + d = (a + (b \wedge c)) + d$	zde zafungují priority operátorů
0101 + 1110 = 0011	nejvyšší bit výsledku 10011 se již ořízнул
0001 - 1111 = 0010	odčítáme modulo 16 = 10000
0101 \wedge 0011 = 0001	takto funguje <i>and</i>
0101 \vee 0011 = 0111	takto <i>or</i>
0101 \oplus 0011 = 0110	a takto <i>xor</i>
(1 \ll 11) - 1 = 1000 - 1 = 0111	jak vyrobit pomocí \ll posloupnost jedniček
$a \vee \neg a = 1111$	jak získat z čehokoliv samé jedničky

Výpočet probíhá takto: Nejprve se do registru x nastaví vstup (to je vždy jedno číslo) a do ostatních registrů nuly. Poté se provedou všechny příkazy v pořadí, v jakém jsou v programu uvedeny, přičemž vždy se nejprve vyhodnotí *výraz* na pravé straně a teprve poté se jeho výsledek uloží do *registru*, takže uvnitř výrazu je ještě možné pracovat s původní hodnotou registru. Po dokončení posledního příkazu se hodnota v registru y interpretuje jako výsledek výpočtu. Hodnoty v ostatních registrech mohou být libovolné.

Často budeme potřebovat, aby program mohl pracovat s většími čísly, než je číslo na vstupu, takže budeme rozlišovat velikost vstupu N (tj. počet bitů potřebných k zapsání vstupní hodnoty) a velikost W registrů a mezivýsledků, kterou si při psaní programu sami určíme. Pokud bychom ovšem povolili exponenciálně velká čísla (tedy $W = 2^N$), mohli bychom cokoliv spočítat v konstantním čase – stačilo by do jedné dlouhatánské konstanty uvedené v programu zakódovat všechny možné výsledky programu pro všechny hodnoty vstupu. Tak dlouhé registry lze však stěží považovat za realistické, proto přijmeme omezení, že W musí být polynomiální ve velikosti vstupu, čili že existuje konstanta k taková, že pro každé N je $W \leq N^k$.

Ne vždy si ovšem vystačíme s jedním programem, který funguje pro všechny velikosti vstupu – mnohdy potřebujeme podle N měnit hodnoty pomocných konstant v programu, někdy také nějakou operaci opakovat vícekrát v závislosti na velikosti vstupu. Povolíme si tedy programy zapisovat obecněji, a to tak, že uvedeme seznam pravidel, jež nám pro každé N vytvoří program, který počítá správně pro všechny vstupy velikosti N . [Formálně bychom tato pravidla mohli zavést třeba jako programy v nějakém klasickém programovacím jazyce. My si ale formalismus odpustíme a budeme je popisovat slovně.]

Při řešení úloh budeme chtít, aby časová složitost vygenerovaných programů, tedy jejich délka v závislosti na N , byla co nejmenší. Mezi stejně rychlými programy je pak lepší ten, který si vystačí s kratšími čísly, čili s menším W (to je analogie prostorové složitosti). Podobně jako u klasických programů ovšem budeme v obou případech přehlížet multiplikační konstanty.

Příklad 1: Sestrojte program pro ALIK, který dostane na vstupu nenulové číslo a vrátí výsledek 1 právě tehdy, je-li toto číslo mocninou dvojky, jinak vrátí nulu.

Řešení: Nejdříve si všimněme, že mocniny dvojky jsou právě čísla, která obsahují právě jeden jedničkový bit. Sledujme chování následujícího jednoduchého programu.

Zmíňme ale ještě konvence, které budeme používat při psaní všech ukázkových programů: V levém sloupci naleznete jednotlivé příkazy, v pravém sloupci obecný tvar spočítané hodnoty pro libovolné N . Pokud se nějaká číslice nebo skupina číslic opakuje vícekrát, značíme opakování exponentem, tedy 0^8 je osm nul, $(01)^3$ je zkratka za **010101**. Řeckými písmeny značíme blíže neurčené skupiny bitů.

$a := x - 1$	$x = \alpha 10^i$
$b := x \wedge a$	$a = \alpha 01^i$
	$b = \alpha 00^i$

Číslo v registru a se od x vždy liší tím, že nejpravější **1** se změní na **0** a všechny **0** vpravo od ní se změní na **1**. Proto $b = x \wedge a$ se musí od x lišit právě přepsáním nejpravější **1** na **0**. (To proto, že bity vlevo od této **1** jsou stále stejné a $\alpha \wedge \alpha = \alpha$, zatímco ve zbytku čísla se vždy *anduje 0 s 1*, což dá nulu.) A jelikož mocniny dvojky jsou právě čísla, v jejichž dvojkovém zápisu je právě jedna **1**, spočte náš program v b nulu právě tehdy, je-li x mocninou dvojky (nebo nulou, což jsme si ale zakázali).

Zbývá tedy vyřešit, jak z nuly udělat požadovanou jedničku a z nenuly nulu. K tomu si zavedeme operaci $r := \text{if}(s, t, u)$, která bude realizovat podmínku: pokud $s \neq 0$, přiřadí $r := t$, jinak $r := u$. Provedeme to jednoduchým trikem: rozšíříme si registry o jeden pomocný bit vlevo, nastavíme v r tento bit na jedničku a sledujeme, zda se zmenšením vzniklého čísla o jedničku tento bit změní na nulu nebo ne:

$v := s \vee 10^N$	$v = 1s$
$v := v - 1$	$v = 1s'$ (je-li $s \neq 0$), jinak 01^N
$v := v \wedge 10^N$	$v = 10^N$ nebo 00^N
$v := v \gg N$	$v = 0^N 1$ nebo $0^N 0$
$v := v - 1$	$v = 0^{N+1}$ nebo 1^{N+1}

$$r := (u \wedge v) \vee (t \wedge \neg v)$$

$$r = t \text{ nebo } u$$

Stačí tedy na konec našeho programu přidat

$$y := \text{if}(b, 0, 1)$$

$$y = \mathbf{0} \text{ nebo } \mathbf{1}$$

a máme program, který rozpoznává mocniny dvojky v konstantním čase a používá k tomu čísla o $N + 1 = O(N)$ bitech.

Ještě si ukažme, jak bude probíhat výpočet pro dva konkrétní 8-bitové vstupy (tehdy je $N = 8$ a $W = 9$):

$a := x - 1$	$x = \mathbf{001011000}$	$x = \mathbf{000100000}$
$b := x \wedge a$	$a = \mathbf{001010111}$	$a = \mathbf{000011111}$
$v := b \vee \mathbf{100000000}$	$b = \mathbf{001010000}$	$b = \mathbf{000000000}$
$v := v - 1$	$v = \mathbf{101010000}$	$v = \mathbf{100000000}$
$v := v \wedge \mathbf{100000000}$	$v = \mathbf{101001111}$	$v = \mathbf{011111111}$
$v := v \gg 8$	$v = \mathbf{100000000}$	$v = \mathbf{000000000}$
$v := v - 1$	$v = \mathbf{000000001}$	$v = \mathbf{000000000}$
$y := (\mathbf{000000001} \wedge v) \vee (\mathbf{000000000} \wedge \neg v)$	$v = \mathbf{000000000}$	$v = \mathbf{111111111}$
	$y = \mathbf{000000000}$	$y = \mathbf{000000001}$

Příklad 2: Sestrojte program pro ALIK, který spočte *binární paritu* vstupního čísla, čili vrátí 0 nebo 1 podle toho, zda je v tomto čísle sudý nebo lichý počet jedničkových bitů.

Řešení: Binární parita $P(x)$ čísla $x = x_{N-1} \dots x_1 x_0$ je podle definice rovna $x_0 \oplus x_1 \oplus \dots \oplus x_{N-1}$. Jelikož operace \oplus je asociativní ($\alpha \oplus (\beta \oplus \gamma) = (\alpha \oplus \beta) \oplus \gamma$) a komutativní ($\alpha \oplus \beta = \beta \oplus \alpha$), můžeme tento vztah pro $N = 2^k$ (to opět můžeme bez újmy na obecnosti předpokládat) přeuspořádat na

$$P(x) = (x_0 \oplus x_{N/2}) \oplus (x_1 \oplus x_{N/2+1}) \oplus \dots \oplus (x_{N/2-1} \oplus x_{N-1}),$$

což je ovšem parita čísla vzniklého v XORování horní a dolní poloviny čísla x . Takže výpočet parity N -bitového čísla můžeme na konstantní počet příkazů převést na výpočet parity $N/2$ -bitového čísla, ten zase na výpočet parity $N/4$ -bitového čísla atd., až po $\log_2 N$ kroků na paritu 1-bitového čísla, která je ovšem rovna číslu samému.

Paritu tedy vypočteme na logaritmický počet příkazů pracujících s N -bitovými čísly takto:

$p := x \gg N/2$	$p =$ horních $N/2$ bitů x
$q := x \wedge \mathbf{1}^{N/2}$	$q =$ dolních $N/2$ bitů x
$x := p \oplus q$	$x = N/2$ -bitové číslo s paritou jako původní x
$x := (x \gg N/4) \oplus (x \wedge \mathbf{1}^{N/4})$	$x = N/4$ -bitové ... (můžeme psát zkráceně)
...	
$x := (x \gg 1) \oplus (x \wedge \mathbf{1})$	$x =$ 1-bitové ...
$y := x$	$y = x$ (už jen zkopírovat výsledek)

Náš programovací jazyk samozřejmě $\mathbf{1}^{N/2}$ a podobné operace nemá, ale to vůbec nevadí, protože je vždy používáme jen na podvýrazy závisící pouze na N , takže je v programu můžeme pro každé N uvést jako konstanty. Například pro $N = 8$ bude výpočet probíhat takto:

$p := x \gg 4$	$x = \mathbf{00110110}$
$q := x \wedge \mathbf{1111}$	$p = \dots \mathbf{0011}$
$x := p \oplus q$	$q = \dots \mathbf{0110}$
$x := (x \gg 2) \oplus (x \wedge \mathbf{11})$	$x = \dots \mathbf{0101}$
$x := (x \gg 1) \oplus (x \wedge \mathbf{1})$	$x = \dots \mathbf{00}$
$y := x$	$x = \dots \mathbf{0}$
	$y = \mathbf{00000000}$

Příklad 3: Ve vzorovém řešení úlohy P-I-4 b) jsme potřebovali přesunout posloupnost jedniček na konec čísla, tedy číslo tvaru $\mathbf{0}^i \mathbf{1}^j \mathbf{0}^k$ převést na $\mathbf{0}^i \mathbf{0}^k \mathbf{1}^j$. To je pomocí dělení možné provést v konstantním čase třeba takto:

$a := x \wedge (x - 1)$	$x = \mathbf{0}^i \mathbf{1}^j \mathbf{0}^k$
$b := x \oplus a$	$a = \mathbf{0}^i \mathbf{1}^{j-1} \mathbf{00}^k$ (viz Příklad 1)
$y := x / b$	$b = \mathbf{0}^i \mathbf{0}^{j-1} \mathbf{10}^k$
	$b = \mathbf{0}^i \mathbf{0}^k \mathbf{1}^j$

Zde jsme využili toho, že dělení mocninou dvojky je možné použít jako bitový posun doprava, ovšem zadaný místo počtu bitů, o které se má posouvat, číslem majícím $\mathbf{1}$ na pozici, která se má po posunu objevit úplně vpravo.

P-III-4 Zaklínadla

Program: `magie.pas` / `magie.c` / `magie.cpp`

Vstup: `magie.in`

Výstup: `magie.out`

Při výzkumu vlivu přesmyček na magická zaklínadla se podařilo dokázat, že výměna takových dvou sousedních písmen v zaklínadle, která se nevyskytují v (anglické) abecedě těsně vedle sebe, nemá vliv na magický účinek zaklínadla. Například místo oblíbeného *abraka* lze stejně dobře použít *arbaka*, ale nikoliv již *baraka*. Samozřejmě lze sousední písmena v zaklínadle prohazovat vícekrát, takže z *abraka* lze postupně odvodit následující (stejně účinná) zaklínadla:

$$abraka \longrightarrow arbaka \longrightarrow rabaka \longrightarrow rabkaa \longrightarrow rakbaa \longrightarrow rkabaa \longrightarrow krabaa.$$

Povšimněte si, že pořadí výměn sousedních písmen v zaklínadle můžeme otočit, a tedy ze zaklínadla *krabaa* lze též získat původní *abraka*.

Dvě zaklínadla nazveme *ekvivalentní*, jestliže je lze mezi sebou převést posloupností výměn dvou sousedních písmen, která se nevyskytují v abecedě těsně vedle sebe. Například zaklínadla *abraka* a *krabaa* jsou ekvivalentní, ale zaklínadla *dabra* a *badar* ne.

Soutěžní úloha: Napište program, který pro zadané dvojice zaklínadel určí, zda jsou či nejsou ekvivalentní.

Vstup: Vstupní soubor `magie.in` obsahuje několik bloků, z nichž každý odpovídá jedné dvojici zaklínadel. Obě zaklínadla v jednom bloku mají vždy stejný počet písmen N , $1 \leq N \leq 1\,000\,000$, který je uveden na prvním řádku každého bloku. Následuje N řádků, z nichž každý obsahuje dvě malá písmena anglické abecedy (tzn. znaky z rozmezí `a..z`), která jsou oddělená jednou mezerou. První znak na i -tém z těchto řádků je i -tý znak prvního zaklínadla, druhý znak je i -tý znak druhého zaklínadla. Vstupní soubor je ukončen řádkem, který obsahuje jediné číslo 0.

Výstup: Výstupní soubor `magie.out` obsahuje pro každý blok vstupního souboru jeden řádek. Tento řádek obsahuje slovo „ekvivalentni“, pokud jsou zadaná zaklínadla ekvivalentní, a slovo „neekvivalentni“, pokud nejsou. Řádky musí být vypsány v pořadí, v jakém se jim odpovídající bloky vyskytují v souboru `magie.in`.

Příklad:

Soubor `magie.in`

```
6
a k
b r
r a
a b
k a
a a
5
d b
a a
b d
r a
a r
0
```

Soubor `magie.out`

```
ekvivalentni
neekvivalentni
```

P-III-5 Asfaltěři

Program: `asfalt.pas / asfalt.c / asfalt.cpp`
Vstup: `asfalt.in`
Výstup: `asfalt.out`

Ve Viácii si občané již dlouho stěžovali na nekvalitní silnice. Když si jednou i vrchní cestář při cestě do práce v kočáře vyrazil zub, rozhodl se k radikální akci. Doslechl se, že v sousední zemi začali na cesty používat novinku zvanou asfalt, a myšlenka na vyasfaltování všech silnic ve Viácii byla na světě. A jak si vrchní cestář usmyslel, tak se i stalo. Při realizaci nápadu se ale nižší cestáři museli potýkat s nemilým problémem: asfalt se do Viacie dovážel v ohromných barelech, ve kterých bylo asfaltu tak akorát na dvě cesty (byly to opravdu ohromné barely). Potíž byla v tom, že jakmile se barel narazil a asfalt z něj začal vytékat, nedal se už proud asfaltu ničím zastavit a bylo tedy nutné vyasfaltovat najednou dvě na sebe navazující cesty. Jenže jak rozvrhnout asfaltování, aby cestáři neskončili ve městě s poloplným barelem a se všemi cestami vedoucími do města už vyasfaltovanými? Důsledky zalití náměstí a několika přilehlých čtvrtí do asfaltu by byly pro cestáře jistě nemilé... Rozhodli se proto přizvat k problému vás, jakožto na asfalt vzaté odborníky.

Soutěžní úloha: Napište program, který pro zadané propojení měst cestami rozhodne, zda je možno cesty podle popsaných pravidel vyasfaltovat, a pokud ano, vypíše jednu z možností, jak rozvrhnout, která dvojice cest bude asfaltována ze kterého barelu.

Vstup: Na prvním řádku vstupního souboru `asfalt.in` se nacházejí dvě celá čísla N a M oddělená mezerou, $1 \leq N \leq 10\,000$, $1 \leq M \leq 40\,000$ – počet měst a počet cest ve Viácii. Dále ve vstupním souboru následuje M řádků popisujících jednotlivé cesty. Každý řádek obsahuje dvě celá čísla A a B oddělená mezerou – čísla měst (města číslujeme od jedné do N), mezi kterými cesta vede. Předpokládejte, že mezi každými dvěma městy se lze po cestách dostat (pokud ne přímo, tak přes jiná města).

Výstup: Výstupní soubor `asfalt.out` bude buď obsahovat jediný řádek s textem „Cesty nelze vyasfaltovat.“, pokud neexistuje způsob, jak vyasfaltovat všechny cesty a neskončit s poloplným barelem v nějakém městě, nebo bude obsahovat $M/2$ řádků s popisem postupu asfaltování. Každý řádek postupu bude popisovat využití jednoho barelu s asfaltem. Bude obsahovat tři celá čísla oddělená mezerou – číslo města, ve kterém má asfaltování začít, číslo města, do kterého se má pokračovat a číslo města, ve kterém má asfaltování skončit. Každá cesta musí být vyasfaltována právě jednou.

Příklad 1:

Soubor `asfalt.in`
3 3
1 2
2 3
3 1

Soubor `asfalt.out`
Cesty nelze vyasfaltovat.

Příklad 2:

Soubor `asfalt.in`
5 8
1 5
1 4
1 3
1 2
3 2
3 4
4 5
4 2

Soubor `asfalt.out`
5 1 4
5 4 3
4 2 3
3 1 2