54. ročník matematické olympiády na středních školách

Kategorie P

In: Karel Horák (editor); Martin Mareš (editor); Peter Novotný (editor); Jaromír Šimša (editor); Jaroslav Švrček (editor); Pavel Töpfer (editor); Jaroslav Zhouf (editor): 54. ročník matematické olympiády na středních školách. Zpráva o řešení úloh ze soutěže konané ve školním roce 2004/2005. 46. mezinárodní matematická olympiáda. 17. mezinárodní olympiáda v informatice. (Czech). Praha: Jednota českých matematiků a fyziků, 2006. pp. 93–115.

Persistent URL: http://dml.cz/dmlcz/405091

Kategorie P

Texty úloh

P – I – 1

Prádelna

Bořivoj se rozhodl, že začne podnikat — a to ve velkém. Jednou v noci, poté co upadl v koupelně, dostal skvělý nápad: otevře si prádelnu. Každý, kdo přijde, si bude moci za malý obnos půjčit pračku, pokud bude nějaká volná, a vypere si své prádlo. Ihned se tedy zeptal svých přátel, zda by chtěli jeho prádelnu navštěvovat, a zjistil, že zájem je opravdu veliký. Brzy ani nevěděl, kolik praček bude vůbec potřebovat, aby se dostalo na všechny zákazníky. A proto se rozhodl obrátit se na vás s prosbou o pomoc.

Soutěžní úloha. Na vstupu dostanete počet zakázek, které Bořivoj na jeden konkrétní den obdržel. U každé zakázky víte čas příchodu zákazníka a dobu, na jakou si chce pronajmout jednu pračku. Požadavky zákazníků nejsou uvedeny v žádném konkrétním pořadí.

Vaším úkolem je zjistit, kolik nejméně praček bude Bořivoj potřebovat, aby si každý zákazník mohl pronajmout pračku na celou požadovanou dobu od svého příchodu. Kromě minimálního počtu praček musíte pro Bořivoje vytvořit ještě seznam, podle kterého bude posílat zákazníky k volným pračkám.

Formát vstupu: První řádka textového souboru `pradelna.in` obsahuje jediné přirozené číslo $N \leqq 10\,000$ — počet zákazníků. Dalších N řádek obsahuje informace o jednotlivých zákaznících: na i-té z těchto řádek je uveden čas T_i, kdy chce zákazník přijít, a doba T_i', na kterou si chce pronajmout pračku. Můžete předpokládat, že T_i a T_i' jsou celá čísla od 1 do 1 000 000 000.

Formát výstupu: První řádka textového souboru `pradelna.out` obsahuje jediné číslo P — nejmenší možný počet praček, s nimiž může Bořivoj obsloužit všechny zákazníky. Dalších N řádek bude obsahovat N čísel a_1

až a_N, přičemž a_i je číslo pračky, kterou má použít i-tý zákazník. Předpokládejte, že pračky budou očíslovány od 1 do P.

Příklad:

pradelna.in	pradelna.out
4	3
1000 1000	2
1900 900	1
1500 700	3
2000 500	2

P – I – 2

Závody

Letos se po několika letech opět konají slavné závody švábů. Závody probíhají na pečlivě připravené překážkové dráze obsahující takové záludnosti, jako je třeba mistička s cukrem. Švábí závodníci jsou na trať vypouštěni v minutových intervalech a aby je bylo možno v cíli rozeznat, má každý závodník k sobě připevněnu cedulku s minutou startu (první šváb má tedy číslo nula, druhý jedna, atd.). Organizátory závodu by zajímalo, jak moc se švábi během tréninkového běhu promíchali. Pokud by se totiž promíchali hodně, bylo by třeba prodloužit intervaly mezi jednotlivými závodníky, aby se při běhu tolik neovlivňovali. Jako míra promíchanosti závodníků byl stanoven počet dvojic závodníků, kteří doběhli do cíle v opačném pořadí, než v jakém vyběhli na trať. Spočítat míru promíchanosti pro dané pořadí švábů v cíli je již úloha pro vás.

Váš program dostane na vstupu počet švábů N a pořadí, v jakém švábi doběhli do cíle (tedy nějakou permutaci čísel $0, \ldots, N-1$). Na výstup má váš program vypsat míru promíchanosti závodníků.

Formát vstupu: Vstupní textový soubor `zavody.in` obsahuje dva řádky. Na prvním řádku je uvedeno jedno celé číslo N, $1 \leqq N \leqq 30\,000$. Na druhém řádku je N různých celých čísel z intervalu $0, \ldots, N-1$ oddělených jednou mezerou.

Formát výstupu: Výstupní textový soubor `zavody.out` obsahuje jediný řádek s jedním celým číslem — počtem dvojic závodníků, kteří doběhli v opačném pořadí, než v jakém vystartovali.

Příklad:

zavody.in	zavody.out
5	3
1 0 4 2 3	

P – I – 3

Fylogenetika

Fylogenetika je obor biologie zabývající se rozpoznáváním vývojových vztahů mezi organismy. Často používanou metodou je srovnávání genetického kódu. V této úloze se budeme zabývat výrazně zjednodušenou variantou tohoto problému.

Genetický kód budeme mít uložen jako řetězec skládající se z písmen ‚A', ‚C', ‚G' a ‚T'. Budeme předpokládat, že vývoj nového druhu probíhá tak, že se na začátek nebo na konec genetického kódu připojí nové geny — to je samozřejmě pouze idealizace (čti: úplný nesmysl). Dostanete zadán genetický kód několika organismů, vaším úkolem je nalézt mezi nimi všechny dvojice předek — potomek, tj. takové, že genetický kód předka je souvislým podřetězcem genetického kódu potomka.

Formát vstupu: Vstupní textový soubor `fylogen.in` obsahuje několik řetězců složených z písmen ‚A', ‚C', ‚G' a ‚T', reprezentujících genetické kódy jednotlivých organismů. Organismy jsou očíslovány $1, 2, \ldots, n$; na i-tém řádku se nachází kód i-tého organismu. Můžete předpokládat, že řetězců je nejvýše 50, každý z nich má nejvýše 50 znaků a žádné dva řetězce nejsou stejné.

Formát výstupu: Výstupní textový soubor `fylogen.out` tvoří seznam všech dvojic předek — potomek. Každá řádka výstupního souboru popisuje jednu z těchto dvojic a sestává z čísla předka následovaného číslem potomka. Dvojice mohou být uvedeny v libovolném pořadí, nesmějí se však opakovat.

Příklad: Pro vstup

```
ATAT
CATATG
CATATGA
CATATGG
```

je jedním z možných správných řešení výstup

```
1 2
1 3
1 4
2 3
2 4
```

P – I – 4

ALIK

Aritmeticko-logický integerový kalkulátor (zkráceně ALIK) je počítací 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
$\neg\,\alpha$	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: *and*, *or* a *xor*. Vyhodnocují se tak, že	8
$\alpha \vee \beta$	se i-tý bit výsledku spočte z i-tého bitu čísla α a i-tého	7
$\alpha \oplus \beta$	bitu čísla β podle následujících tabulek:	7

$$\begin{array}{lll} \mathbf{0} \wedge \mathbf{0} = \mathbf{0} & \mathbf{0} \vee \mathbf{0} = \mathbf{0} & \mathbf{0} \oplus \mathbf{0} = \mathbf{0} \\ \mathbf{0} \wedge \mathbf{1} = \mathbf{0} & \mathbf{0} \vee \mathbf{1} = \mathbf{1} & \mathbf{0} \oplus \mathbf{1} = \mathbf{1} \\ \mathbf{1} \wedge \mathbf{0} = \mathbf{0} & \mathbf{1} \vee \mathbf{0} = \mathbf{1} & \mathbf{1} \oplus \mathbf{0} = \mathbf{1} \\ \mathbf{1} \wedge \mathbf{1} = \mathbf{1} & \mathbf{1} \vee \mathbf{1} = \mathbf{1} & \mathbf{1} \oplus \mathbf{1} = \mathbf{0} \end{array}$$

$\alpha \ll \beta$	posune číslo α o β bitů doleva, čili doplní na jeho konec β nul a odřízne prvních β bitů, aby byl výsledek opět W-bitový.	2

$\alpha \gg \beta$	posune číslo α o β bitů doprava, čili doplní na jeho začátek β nul a odřízne posledních β bitů, 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
$\mathbf{0101} + \mathbf{1110} = \mathbf{0011}$	nejvyšší bit výsledku **10011** se již oříznul
$\mathbf{0001} - \mathbf{1111} = \mathbf{0010}$	odčítáme modulo 16 = **10000**
$\mathbf{0101} \wedge \mathbf{0011} = \mathbf{0001}$	takto funguje *and*
$\mathbf{0101} \vee \mathbf{0011} = \mathbf{0111}$	takto *or*
$\mathbf{0101} \oplus \mathbf{0011} = \mathbf{0110}$	a takto *xor*
$(\mathbf{1} \ll \mathbf{11}) - \mathbf{1} = \mathbf{1000} - \mathbf{1} = \mathbf{0111}$	jak vyrobit pomocí $\ll$ posloupnost jedniček
$a \vee \neg a = \mathbf{1111}$	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 zápisu 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 spočíst 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 přijměme omezení, že W musí být polynomiální ve velikosti vstupu, čili že existuje konstanta k taková, že pro každé N je $W \leqq N^k$.

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 multiplikativní 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.

Zmiňme 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 $\mathbf{0}^8$ je osm nul, $(\mathbf{01})^3$ je zkratka za **010101**. Řeckými písmeny značíme blíže neurčené skupiny bitů.

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

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 *and*uje **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 $r := if(s,t,u)$, 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:

$v := s \vee \mathbf{10}^N$	$v = \mathbf{1}s$
$v := v - 1$	$v = \mathbf{1}r'$ (je-li $r \neq 0$), jinak $\mathbf{01}^N$
$v := v \wedge \mathbf{10}^N$	$v = \mathbf{10}^N$ nebo $\mathbf{00}^N$
$v := v \gg N$	$v = \mathbf{0}^N\mathbf{1}$ nebo $\mathbf{0}^N\mathbf{0}$
$v := v - 1$	$v = \mathbf{0}^{N+1}$ nebo $\mathbf{1}^{N+1}$
$r := (u \wedge v) \vee (t \wedge \neg v)$	$r = t$ nebo u

Stačí tedy na konec našeho programu přidat

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

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 $W = 9$):

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

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 $x = x_{N-1} \ldots x_1 x_0$ je podle definice rovna $x_0 \oplus x_1 \oplus \ldots \oplus x_{N-1}$. 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

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

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

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

$p := x \gg \frac{1}{2}N$	$p =$ horních $\frac{1}{2}N$ bitů x
$q := x \wedge \mathbf{1}^{N/2}$	$q =$ dolních $\frac{1}{2}N$ bitů x
$x := p \oplus q$	$x = \frac{1}{2}N$-bitové číslo s paritou jako původní x
$x := (x \gg \frac{1}{4}N) \oplus (x \wedge \mathbf{1}^{N/4})$	$x = \frac{1}{4}N$-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 žádné celé části čísel 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:

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

Soutěžní úlohy.

a) Sestrojte program pro Alik, jehož výsledkem bude počet jedničkových bitů ve dvojkovém zápisu čísla na vstupu.

b) Sestrojte program pro ALIK, který k zadanému číslu x spočte nejbližší větší číslo, v jehož dvojkovém zápisu je stejný počet jedniček jako v zápisu x. Pokud takové číslo neexistuje, výsledek může být libovolný.

P – II – 1

Prádelní salón

Z Bořivoje se stal díky vaší pomoci úspěšný podnikatel a jeho klientela zahrnuje i bohatší a malichernější zákazníky. Pokud totiž nějaký zákazník uvidí dva různé zákazníky používat stejnou pračku, nebude už tuto prádelnu dále navštěvovat: „No považte, přece nelze prát prádlo s lidmi, kteří nemají na to, aby si zaplatili pračku sami pro sebe!"

Soutěžní úloha. Na vstupu dostanete $N \leqq 10\,000$ — počet zákazníků, kteří navštíví Bořivojovu prádelnu během jednoho dne. U každého zákazníka je zadán čas jeho příchodu a doba, na jakou si chce pronajmout pračku (obojí jsou celá čísla mezi 1 a 1 000 000 000). Požadavky zákazníků nejsou uvedeny v žádném konkrétním pořadí.

Vaším úkolem je zjistit, kolik nejméně praček Bořivoj potřebuje, aby všichni jeho zákazníci byli zcela spokojeni. Zákazník bude spokojen, pokud si bude moci pronajmout pračku od okamžiku příchodu na dobu, kterou požaduje (je samozřejmé, že jednu pračku nemohou používat dva různí zákazníci současně), a navíc během doby, kdy bude prát, nebude žádnou pračku využívat více zákazníků po sobě.

Kromě určení minimálního počtu praček musíte pro Bořivoje vytvořit ještě seznam, podle kterého bude posílat zákazníky k volným pračkám.

Příklad: Pro 5 zákazníků, jejichž příchody a doby praní jsou

```
1000 1000
3000 2000
4500 500
1500 500
2000 2000
```

jsou potřeba alespoň 3 pračky a přiřazení praček zákazníkům například takové:

```
zákazník 1 bude u pračky 2
zákazník 2 bude u pračky 3
zákazník 3 bude u pračky 1
zákazník 4 bude u pračky 3
zákazník 5 bude u pračky 2
```

Všimněte si, že zákazníci 3 a 5 nemohou dostat stejnou pračku, protože by je viděl zákazník 2 pracovat u stejné pračky.

P – II – 2

Zakázané rozdíly

Mějme dáno celé kladné číslo N, $N \geqq 2$, a soustavu podmínek tvaru $x_i - x_j \neq a_{i,j}$, kde $x_1, \ldots, x_{N+1}$ jsou proměnné, $a_{i,j}$ jsou celá čísla mezi 0 a $N-1$ a pro každou dvojici indexů i a j, $1 \leqq i < j \leqq N+1$ soustava obsahuje právě jednu podmínku.

Soustavu budeme řešit modulo modulo zadané číslo N, tj. všechny aritmetické operace jsou prováděny modulo N. Připomeňme si, že výsledkem aritmetické operace provedené modulo N je zbytek po dělení původního výsledku číslem N, např. $(2+3) \bmod 4 = 1$, $(2-3) \bmod 4 = 3$, $(3 \cdot 2) \bmod 5 = 1$, $(3 \cdot 4) \bmod 6 = 0$, atd. Všimněte si zejména způsobu počítání, pokud je původní výsledek operace záporný.

Nalezněte algoritmus, který pro zadané N a čísla $a_{i,j}$ zjistí, zda zadaná soustava podmínek má řešení, tzn. zda existují čísla $x_1, \ldots, x_{N+1} \in \{0, \ldots, N-1\}$ taková, že rozdíl $x_j - x_i - a_{i,j}$ není dělitelný N pro žádné i a j, $1 \leqq i < j \leqq N+1$. Pokud má soustava řešení, algoritmus musí také libovolné její řešení nalézt a vypsat.

Příklad 1: Pro $N = 3$, máme zadány následující podmínky:

$$\begin{aligned} x_1 - x_2 &\neq 1, \\ x_1 - x_3 &\neq 2, \\ x_1 - x_4 &\neq 2, \\ x_2 - x_3 &\neq 2, \\ x_2 - x_4 &\neq 1, \\ x_3 - x_4 &\neq 0. \end{aligned}$$

Soustava má řešení, např. $x_1 = x_2 = x_4 = 2$ a $x_3 = 1$.

Příklad 2: Pro $N = 2$, máme zadány následující podmínky:

$$\begin{aligned} x_1 - x_2 &\neq 1, \\ x_1 - x_3 &\neq 0, \\ x_2 - x_3 &\neq 1. \end{aligned}$$

Pokud $x_1 = 0$, pak $x_2 = 0$ podle první podmínky a $x_3 = 1$ podle druhé podmínky. Potom ale třetí podmínka není splněna. Podobně pokud

$x_1 = 1$, x_2 musí být rovno 0 a x_3 rovno 1 a třetí podmínka opět není splněna. Zadaná soustava podmínek tedy nemá řešení.

P – II – 3

Redundantní redundance

Úřad pro potírání redundantních repetic (zřízený Komisí pro likvidaci redundantních úřadů) se zabývá odstraňováním zbytečně opakovaných dokumentů v archivech. Procházení archivů je samozřejmě velmi nudná práce, která odvádí úředníky od jiných, mnohem zajímavějších a jistě i prospěšnějších využití jejich pracovního času. Velmi by je proto potěšilo, kdybyste pro ně napsali program řešící následující úlohu:

Je dáno přirozené číslo k a nějaký znakový řetězec T. Určete souvislý podřetězec délky k, který se v T nejvíce opakuje, a také počet jeho výskytů R. Jednotlivé výskyty tohoto řetězce se mohou částečně překrývat. V případě, že existuje více řetězců, které se opakují R-krát, vypište jeden libovolný z nich.

Příklad: Pro vstup `abababa` a $k = 3$ je nejčastějším řetězcem `aba` opakující se 3krát.

P – II – 4

ALÍK

Definici stroje ALIK naleznete ve studijním textu za touto úlohou. Od domácího kola se liší tím, že přibyly operace násobení, dělení a zbytku po dělení a Příklad 3 na tyto operace.

Soutěžní úloha. Sestrojte program pro ALIK, který k zadanému číslu $x = x_{N-1} \ldots x_1 x_0$ nalezne zrcadlové číslo $y = x_0 x_1 \ldots x_{N-1}$, tj. číslo, jehož dvojkový zápis vznikne zapsáním N-bitového dvojkového zápisu čísla x (včetně případných počátečních nul) pozpátku.

Studijní text. Aritmeticko-logický integerový kalkulátor (zkráceně ALIK) je počítací 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
$\alpha \,/\, \beta$	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: *and*, *or* a *xor*. Vyhodnocují se tak, že	8
$\alpha \vee \beta$	se i-tý bit výsledku spočte z i-tého bitu čísla α a i-tého	7
$\alpha \oplus \beta$	bitu čísla β podle následujících tabulek:	7

$\mathbf{0} \wedge \mathbf{0} = \mathbf{0}$	$\mathbf{0} \vee \mathbf{0} = \mathbf{0}$	$\mathbf{0} \oplus \mathbf{0} = \mathbf{0}$
$\mathbf{0} \wedge \mathbf{1} = \mathbf{0}$	$\mathbf{0} \vee \mathbf{1} = \mathbf{1}$	$\mathbf{0} \oplus \mathbf{1} = \mathbf{1}$
$\mathbf{1} \wedge \mathbf{0} = \mathbf{0}$	$\mathbf{1} \vee \mathbf{0} = \mathbf{1}$	$\mathbf{1} \oplus \mathbf{0} = \mathbf{1}$
$\mathbf{1} \wedge \mathbf{1} = \mathbf{1}$	$\mathbf{1} \vee \mathbf{1} = \mathbf{1}$	$\mathbf{1} \oplus \mathbf{1} = \mathbf{0}$

$\alpha \ll \beta$	posune číslo α o β bitů doleva, čili doplní na jeho konec β nul a odřízne prvních β bitů, aby byl výsledek opět W-bitový.	2
$\alpha \gg \beta$	posune číslo α o β bitů doprava, čili doplní na jeho začátek β nul a odřízne posledních β bitů, 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
$\mathbf{0101} + \mathbf{1110} = \mathbf{0011}$	nejvyšší bit výsledku **10011** se již ořízl
$\mathbf{0001} - \mathbf{1111} = \mathbf{0010}$	odčítáme modulo 16 = **10000**
$\mathbf{0101} \wedge \mathbf{0011} = \mathbf{0001}$	takto funguje *and*
$\mathbf{0101} \vee \mathbf{0011} = \mathbf{0111}$	takto *or*
$\mathbf{0101} \oplus \mathbf{0011} = \mathbf{0110}$	a takto *xor*
$(\mathbf{1} \ll \mathbf{11}) - \mathbf{1} = \mathbf{1000} - \mathbf{1} = \mathbf{0111}$	jak vyrobit pomocí $\ll$ posloupnost jedniček
$a \vee \neg\, a = \mathbf{1111}$	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 zápisu 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 spočíst 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 $W \leqq N^k$.

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 ně-

jakém 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 multiplikativní 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.

Zmiňme 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 $\mathbf{0}^8$ je osm nul, $(\mathbf{01})^3$ je zkratka za **010101**. Řeckými písmeny značíme blíže neurčené skupiny bitů.

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

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 *and*uje **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 $r := if(s,t,u)$, 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 vznik-

lého čísla o jedničku tento bit změní na nulu nebo ne:

$v := s \vee \mathbf{10}^N$	$v = \mathbf{1}s$
$v := v - 1$	$v = \mathbf{1}r'$ (je-li $r \neq 0$), jinak $\mathbf{01}^N$
$v := v \wedge \mathbf{10}^N$	$v = \mathbf{10}^N$ nebo $\mathbf{00}^N$
$v := v \gg N$	$v = \mathbf{0}^N\mathbf{1}$ nebo $\mathbf{0}^N\mathbf{0}$
$v := v - 1$	$v = \mathbf{0}^{N+1}$ nebo $\mathbf{1}^{N+1}$
$r := (u \wedge v) \vee (t \wedge \neg v)$	$r = t$ nebo u

Stačí tedy na konec našeho programu přidat

$$y := \mathit{if}(b, 0, 1) \qquad\qquad y = \mathbf{0} \text{ nebo } \mathbf{1}$$

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 $W = 9$):

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

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 $x = x_{N-1} \ldots x_1 x_0$ je podle definice rovna $x_0 \oplus x_1 \oplus \ldots \oplus x_{N-1}$. 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

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

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

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

$p := x \gg \frac{1}{2}N$	$p =$ horních $\frac{1}{2}N$ bitů x
$q := x \wedge \mathbf{1}^{N/2}$	$q =$ dolních $\frac{1}{2}N$ bitů x
$x := p \oplus q$	$x = \frac{1}{2}N$-bitové číslo s paritou jako původní x
$x := (x \gg \frac{1}{4}N) \oplus (x \wedge \mathbf{1}^{N/4})$	$x = \frac{1}{4}N$-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 žádné celé části čísel 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:

	$x =$ **00110110**
$p := x \gg 4$	$p =$ ····**0011**
$q := x \wedge \mathbf{1111}$	$q =$ ····**0110**
$x := p \oplus q$	$x =$ ····**0101**
$x := (x \gg 2) \oplus (x \wedge \mathbf{11})$	$x =$ ····**00**
$x := (x \gg 1) \oplus (x \wedge \mathbf{1})$	$x =$ ····**0**
$y := x$	$y =$ **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 $\mathbf{0}^i\mathbf{0}^j\mathbf{1}^k$. To je pomocí dělení možné provést v konstantním čase třeba takto:

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

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 **1** na pozici, která se má po posunu objevit úplně vpravo.

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 $x_1, \ldots, x_N$ ($1 \leqq N \leqq 1\,000\,000$) a měl by pro každý z nich vypsat, 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 $x_1, \ldots, x_{i-1}$ (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.

Vaším 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. Vaším 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 soustřeďte 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.

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.

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.

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:

$$\begin{aligned} &\textit{abraka} \longrightarrow \textit{arbaka} \longrightarrow \textit{rabaka} \longrightarrow \textit{rabkaa} \longrightarrow \\ &\longrightarrow \textit{rakbaa} \longrightarrow \textit{rkabaa} \longrightarrow \textit{krabaa}. \end{aligned}$$

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, $1 \leqq N \leqq 1\,000\,000$, 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`	Soubor `magie.out`
`6`	`ekvivalentni`
`a k`	`neekvivalentni`
`b r`	
`r a`	
`a b`	
`k a`	
`a a`	
`5`	
`d b`	
`a a`	
`b d`	
`r a`	
`a r`	
`0`	

P – III – 5

Asfaltéři

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 Viácie 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, $1 \leqq N \leqq 10\,000$, $1 \leqq M \leqq$ $\leqq 40\,000$ — 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
```