50. ročník matematické olympiády na středních školách

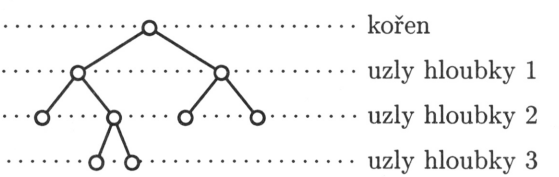
Kategorie P

In: Leo Boček (editor); Karel Horák (editor); Tomáš Pitner (editor); Jaromír Šimša (editor); Jaroslav Švrček (editor); Pavel Töpfer (editor): 50. ročník matematické olympiády na středních školách. Zpráva o řešení úloh ze soutěže konané ve školním roce 2000/2001. 42. mezinárodní matematická olympiáda. 13. mezinárodní olympiáda v informatice. (Czech). Praha: Jednota českých matematiků a fyziků, 2001. pp. 83–136.

Persistent URL: http://dml.cz/dmlcz/405031

P – I – 1

Binární strom je struktura tvořená jednotlivými uzly. Jeden z uzlů je význačný — říkáme mu kořen. Každý z uzlů buď nemá žádného následníka (pak se nazývá list), nebo má právě dva následníky (další uzly stromu). Hloubkou uzlu rozumíme jeho vzdálenost od kořene stromu. Uvědomte si, že kořen může být i listem — pak je binární strom tvořen jediným vrcholem hloubky 0. Příklad binárního stromu si můžete prohlédnout na následujícím obrázku:



Abychom mohli binární stromy jednoduše popisovat, zavedeme si následující kódování: k-tý řádek kódování (pro $k = 0, 1, 2 \ldots$) popisuje právě uzly hloubky k v pořadí zleva doprava. Uzel binárního stromu, který není listem, budeme v našem kódování zobrazovat znakem `U`, listy budeme označovat znakem `L`. Binární strom z předchozího obrázku tedy bude zakódován jako:

```
U
UU
LULL
LL
```

Soutěžní úloha. Je dán počet listů N ($N \leqq 10\,000$) a jejich hloubky v binárním stromě (nějakých N přirozených čísel). Napište program, který sestaví strom se zadanými hloubkami listů a ten vypíše v našem kódování. Jestliže vstupním datům vyhovuje více různých binárních stromů,

program vypíše libovolný jeden z nich. Pokud vyhovující strom neexistuje, program o tom vypíše zprávu.

Formát vstupu: První řádek vstupního souboru `stromy.in` obsahuje jediné číslo N (počet listů). Druhý řádek obsahuje N čísel — hloubky listů hledaného stromu.

Formát výstupu: Výstupní soubor `stromy.out` bude obsahovat kódování nalezeného stromu ve výše uvedeném formátu, případně zprávu ‚`Odpovidajici strom neexistuje.`'.

Příklad 1:

`stromy.in`
```
4
2 3 1 3
```

`stromy.out`
```
U
UL
LU
LL
```

Příklad 2:

`stromy.in`
```
3
1 1 2
```

`stromy.out`
```
Odpovidajici strom neexistuje.
```

P – I – 2

Na království krále Mírumila III. zaútočila nepřátelská vojska a podařilo se jim obsadit několik měst. Král nyní potřebuje dát svému generálovi příkaz k protiútoku (bez příkazu přeci generál nemůže bojovat). Generál však momentálně provádí inspekci vojsk v jiném městě. Je proto třeba vyslat posla, který příkaz co nejrychleji doručí. Příkaz ovšem v žádném případě nesmí padnout do rukou nepřítele! Proto se posel musí neustále držet co nejdále od nepřítelem obsazených měst. Vaším úkolem je navrhnout pro posla co nejlepší trasu.

Soutěžní úloha. Program dostane na vstupu zadaný počet měst N ($1 \leqq N \leqq 100$). Jednotlivá města budeme označovat čísly $1 \ldots N$. Dále je na vstupu uveden počet cest M ($1 \leqq M \leqq 10\,000$) a seznam těchto cest vedoucích mezi městy. Každá cesta je určena dvojicí čísel měst, která spojuje. Cesty se kříží pouze ve městech a je možno se po nich dostat z libovolného města do libovolného (případně přes města jiná). Další údaj K zadaný na vstupu určuje počet měst obsazených nepřítelem, následuje seznam obsazených měst. Nakonec program dostane číslo města, odkud vyráží posel, a číslo města, kde se zdržuje generál. Váš program má nalézt trasu, jejíž vzdálenost od měst obsazených nepřítelem je maximální. Pokud existuje takových tras více, program určí libovolnou nejkratší z nich. Vzdálenost měst $\mathcal{A}$ a $\mathcal{B}$ počítáme jako minimální

počet cest, po kterých musíme projít, abychom se dostali z města $\mathcal{A}$ do města $\mathcal{B}$. Vzdálenost trasy od města $\mathcal{A}$ je pak nejmenší ze vzdáleností města $\mathcal{A}$ od jednotlivých měst ležících na uvažované trase. Vzdáleností trasy od obsazených měst rozumíme nejmenší ze vzdáleností mezi trasou a některým z obsazených měst nebo nulu, pokud některé město na trase samé je obsazeno.

Formát vstupu: První řádek vstupního souboru `posel.in` obsahuje čísla N (počet měst) a M (počet cest). Po něm následuje M řádků, z nichž každý obsahuje popis jedné cesty. Cesta je popsána dvojicí čísel koncových měst. Následuje řádek s číslem K (počet obsazených měst) a za ním K řádků s čísly obsazených měst. Poslední řádek vstupního souboru obsahuje číslo města, odkud vyjíždí posel, a číslo města, kde dlí generál.

Formát výstupu: Výstupem programu v souboru `posel.out` jsou čísla měst na nejlepší nalezené trase uvedená v pořadí, v jakém jimi má posel projíždět. Všechna čísla měst jsou zapsána na jediném řádku výstupního souboru a jsou oddělena mezerami.

Příklad: `posel.in`

```
10 12
1 2
2 3
3 4
4 5
2 5
1 6
6 7
7 8
8 5
1 9
9 10
10 5
1
3
1 5
```

`posel.out`

```
1 9 10 5
```

P – I – 3

Skupina přátel se rozhodla, že v létě podniknou společný výlet na kolech. Většina zvolené trasy však vede přírodní rezervací, a proto mohou

nocovat pouze v kempech. Kempy, ve kterých na svém výletě přespí, ještě nevybrali.

Celková délka naplánované trasy je L ($1 \leqq L \leqq 1\,000\,000\,000$). Maximální vzdálenost, kterou naši přátelé mohou urazit za jeden den, je K, tj. ve dvou po sobě následujících dnech musí přespat v kempech vzdálených nejvýše o K. Na naplánované trase se nachází celkem N kempů ($0 \leqq N \leqq 10\,000$); i-tý kemp je ve vzdálenosti l_i od začátku jejich výletu a cena za přespání v něm je c_i ($1 \leqq c_i \leqq 20\,000$). Čísla L, K, l_i a c_i jsou celá kladná; všechna l_i jsou navzájem různá a platí $0 < l_1 < l_2 < \ldots < l_N < L$.

Vaším úkolem je rozhodnout, zda skupina může naplánovanou trasu projet. Pokud lze trasu takto projet, pak určete, ve kterých kempech mají přespat tak, aby:

a) jejich výlet trval co nejmenší počet dní.
b) celková cena za přespání v kempech byla co nejmenší.

Úlohy a) a b) řešte zvlášť; v případě, že jednu z těchto úloh neumíte vyřešit, řešte pouze druhou z nich.

Formát vstupu: Vstupní soubor se jmenuje `vylet.in`. Na prvním řádku jsou čísla L, K a N oddělená mezerou. Na dalších N řádcích následují dvojice čísel l_i a c_i oddělených mezerou, postupně pro $i = 1$ až $i = N$.

Formát výstupu: Výstupní soubor se jmenuje `vylet-a.out` pro úlohu a) a `vylet-b.out` pro úlohu b). Na prvním řádku je věta ‚`Trasu nelze projet.`', pokud výlet nelze uskutečnit tak, aby naši přátelé nikdy neujeli za den vzdálenost větší než K. V opačném případě první řádek obsahuje dvě čísla — M a C. První z nich, M ($0 \leqq M$), je počet kempů, ve kterých skupina přespí, druhé z nich, C, je cena, kterou za přespání v těchto kempech zaplatí. Druhý řádek souboru obsahuje M mezerou oddělených čísel kempů, v nichž naši přátelé budou nocovat. Kempy jsou číslovány od jedné. Pokud je $M = 0$, nemusí být druhý řádek vůbec uveden. V případě, že existuje více řešení splňujících podmínku a) nebo b), program může vypsat libovolné jedno z nich.

Příklad 1: `vylet.in`

```
15 10 2
2 11
4 12
```

`vylet-a.out, vylet-b.out`

```
Trasu nelze projet.
```

Příklad 2:

vylet.in
```
8 10 1
7 11
```

vylet-a.out, vylet-b.out
```
0 0
```

Příklad 3:

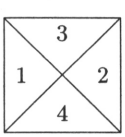
vylet.in
```
25 5 9
4 2
5 8
8 2
10 8
12 2
15 8
16 2
20 8
24 2
```

vylet-a.out
```
4 32
2 4 6 8
```

vylet-b.out
```
5 16
1 3 5 7 8
```

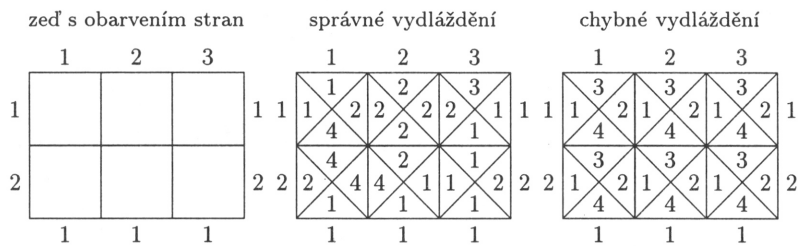
P – I – 4

Nejprve několik definic: *Dlaždice* jsou stejně velké čtverce s obarvenými hranami. Konkrétnímu přiřazení barev hranám dlaždice budeme říkat *typ* dlaždice a budeme jej zapisovat jako uspořádanou čtveřici (l, p, h, d) udávající barvu v pořadí levé, pravé, horní a dolní hrany. Abychom si usnadnili práci, budeme barvy označovat různými symboly — písmeny, čísly apod. Dlaždice typu $(1, 2, 3, 4)$ bude tedy vypadat následovně:



Prostor, který budeme dláždit (budeme mu říkat *zeď*), má tvar obdélníku o velikosti $m \times n$ (m i n jsou přirozená čísla; jednotkou délky budiž délka hrany dlaždice). Strany obdélníku jsou rozděleny na úseky jednotkové délky a každému úseku je opět přiřazena barva. Naším cílem je pokrýt zeď dlaždicemi tak, aby v každém z $m \cdot n$ jednotkových čtverců zdi byla umístěna právě jedna dlaždice, sousední dlaždice se dotýkaly vždy hranami téže barvy a rovněž krajní dlaždice přiléhaly k okraji zdi vždy hranou takové barvy, jakou má i příslušný úsek okraje zdi. Dlaždice není povoleno otáčet.

Příklad:



Pomocí dláždění můžeme snadno řešit úlohy, jejichž výsledkem je buďto odpověď ANO, nebo NE: sestavíme vhodnou množinu typů dlaždic (ta je pro daný problém pevná — nezávisí na vstupu), vezmeme vhodně velkou zeď, její horní okraj obarvíme podle vstupu našeho problému, ostatní okraje ponecháme jednobarevné a budeme se ptát, zda je tuto zeď možno vydláždit či nikoliv. Přitom chceme, aby tento výsledek byl shodný s řešením naší úlohy.

Abychom se nemuseli zabývat tím, jak přesně velkou zeď máme zvolit pro ten či onen vstup úlohy, budeme šířku zdi volit vždy stejnou, jako je délka vstupu (horní okraj tedy bude celý zaplněn vstupem), zatímco výšku zdi použijeme nejmenší, pro níž existuje vydláždění s použitím naší sady dlaždic.

Když tento způsob počítání srovnáme s klasickým programováním, zjistíme, že zvolená sada dlaždic tvoří v našem modelu něco podobného programu a potřebná výška zdi vzdáleně odpovídá době běhu výpočtu — budeme se proto snažit, aby u našich řešení byla co nejmenší.

Formálně řečeno, *dlaždicový program* je uspořádaná čtveřice

$$D = (T, l_0, p_0, d_0),$$

kde T je konečná množina typů dlaždic $\{(l_1, p_1, h_1, d_1), \ldots, (l_k, p_k, h_k, d_k)\}$ a l_0, p_0 a d_0 jsou okrajové barvy. *Rozhodovací úlohou* $P(x)$ rozumíme úlohu zjistit, zda vstup x (konečná posloupnost symbolů, resp. barev z předem určené konečné množiny) má požadovanou vlastnost P. Říkáme, že dlaždicový program řeší rozhodovací úlohu $P(x)$, jestliže platí, že $P(x) =$ ANO právě tehdy, když existuje $v > 0$ takové, že je možno vydláždit dlaždicemi typů obsažených v množině T zeď velikosti $|x| \times v$ s horní hranou obarvenou vstupem x a levou, pravou a dolní hranou obarvenou po řadě barvami l_0, p_0 a d_0. Od každého typu je možno použít libovolně mnoho dlaždic. *Složitostí* programu D pro daný vstup

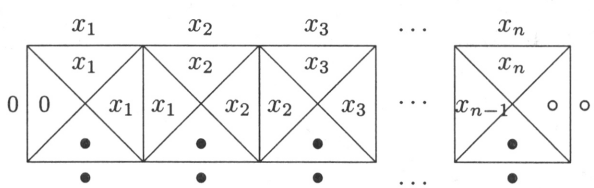
x nazveme nejmenší v, pro něž to je možné; pokud takové neexistuje, a tedy $P(x) = \text{NE}$, definujeme složitost jako nulovou. Složitost programu je funkce délky vstupu n, jejíž hodnota udává maximum ze složitostí programu pro jednotlivé vstupy této délky.

Příklad: Zkusme nyní zkonstruovat dlaždicový program, který bude ověřovat, zda je daná posloupnost tvořená přirozenými čísly $x_1, \ldots, x_n$ $(0 \leqq x_i \leqq 9)$ neklesající. Použijeme dlaždice následujících typů:

$$T = \left\{ \begin{array}{|c|c|}\hline & x \\ \hline i & x \\ \hline & \bullet \\ \hline \end{array}, \begin{array}{|c|c|}\hline & x \\ \hline i & \circ \\ \hline & \bullet \\ \hline \end{array} ; 0 \leqq i \leqq x \leqq 9 \right\},$$

levý okraj obarvíme barvou 0, pravý $\circ$, dolní barvou $\bullet$ a tvrdíme, že tento program řeší danou úlohu se složitostí $O(1)$. To je ovšem třeba dokázat.

Především si ověříme, že každá zeď, kterou je možno vydláždit dlaždicemi typů z množiny T, má jednotkovou výšku. To jasně plyne z toho, že spodní hrana každé dlaždice má barvu $\bullet$, která se nevyskytuje na žádné horní hraně. Z téhož důvodu se dlaždice mající na své pravé hraně barvu $\circ$ musí vyskytovat těsně u pravého okraje zdi a nikde jinde. Každé korektní dláždění proto musí vypadat takto:



což je ovšem možné právě tehdy, když $0 \leqq x_1 \leqq x_2 \leqq \ldots \leqq x_{n-1} \leqq x_n$, tedy když posloupnost na vstupu je neklesající.

Soutěžní úlohy.

a) Sestrojte dlaždicový program, který o dané posloupnosti nul a jedniček zjistí, zda je dvojkovým zápisem nějakého přirozeného čísla dělitelného pěti.

b) Sestrojte dlaždicový program, který o dané posloupnosti přirozených čísel $x_1, \ldots, x_n$ $(0 \leqq x_i \leqq 9)$ rozhodne, je-li nekonstantní (to jest vydláždění existuje právě tehdy, když existují indexy i, j takové, že $x_i \neq x_j$).

P – II – 1

Pepík našel na půdě u babičky krabici s dřevěnými tyčkami. Začal si s nimi hrát a sestavovat z nich trojúhelníky různých tvarů. Uviděl ho jeho tatínek a začalo ho zajímat, kolik různých trojúhelníků lze z těchto tyček sestavit. Vaším úkolem je pomoci mu s nalezením odpovědi na tuto otázku.

Váš program na vstupu obdrží celé číslo N (počet tyček) a dále N navzájem různých kladných čísel d_1 až d_N (délky tyček). Úkolem vašeho programu je určit počet trojic $i < j < k$ takových, že čísla d_i, d_j a d_k splňují trojúhelníkovou nerovnost, tj. $d_i < d_j + d_k$, $d_j < d_i + d_k$ a $d_k < d_i + d_j$.

Příklad. Pro $N = 5$ a $d_1 = 5.5$, $d_2 = 1.5$, $d_3 = 2.0$, $d_4 = 2.5$, $d_5 = 7.5$ váš program odpoví číslem 2, neboť trojúhelník lze sestavit pouze z trojice tyček o délkách d_1, d_4 a d_5 a dále z trojice o délkách d_2, d_3 a d_4. Všimněte si, že trojice tyček o délkách d_1, d_3 a d_5 netvoří trojúhelník.

P – II – 2

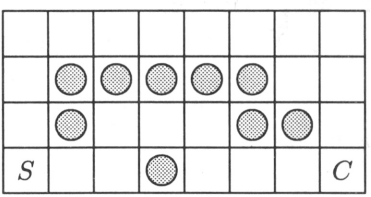
Společnost pro Rovnoprávnost robotů a lidí se snaží vyvinout robota, který by se mohl sám pohybovat v místnosti s překážkami. Bohužel této společnosti chybí softwarový expert a proto se rozhodla, že vás požádá o pomoc.

Robot se má pohybovat v obdélníkové místnosti, na jejíž podlaze je nakreslena čtvercová síť. Na některých políčkách v místnosti jsou postaveny překážky — na tato políčka nesmí robot během svého pohybu vstoupit. Robot se bude po místnosti pohybovat pouze rovnoběžně s některou ze stěn. Společnost však trpí i nedostatkem schopných techniků, a tak jsou možnosti pohybu robota po místnosti značně limitovány. Robot rozpoznává celkem tři příkazy: `Krok`, `Doleva` a `Doprava`. Při obdržení příkazu `Krok` se robot přesune na sousední políčko v tom směru, ve kterém je právě natočen. Při obdržení příkazu `Doleva` se otočí o 90 stupňů doleva a při obdržení `Doprava` se otočí o 90 stupňů doprava.

Vaším úkolem je vytvořit program, který jako vstup dostane rozměry čtvercové sítě na podlaze místnosti (M a N), souřadnice políčka, na kterém se robot právě nachází, a souřadnice políčka, na které se má robot přesunout. Souřadnice políček číslujeme od 1, první souřadnice udává řádek, druhá sloupec. Levý horní roh místnosti má souřadnice $[1, 1]$ (viz příklad níže). Každý z následujících M řádků obsahuje N čísel 0 nebo

1. Je-li j-té číslo na i-tém řádku 1, pak na políčku se souřadnicemi $[i, j]$ je překážka, pokud je toto číslo 0, pak se na políčku se souřadnicemi $[i, j]$ překážka nenachází. Úkolem je nalézt a vypsat posloupnost příkazů, podle nichž robot dojde z počátečního políčka na cílové. Věc má však ještě jeden háček: Provedení příkazů `Doleva` a `Doprava` je časově velmi náročné a vámi vytvořená posloupnost instrukcí pro pohyb robota v místnosti by měla obsahovat co nejmenší počet těchto dvou příkazů. Počet příkazů `Krok` může být libovolný. Počáteční natočení robota si můžete zvolit. V případě, že robot nemůže přejít z počátečního na cílové políčko, vypište vhodnou zprávu.

Příklad. Představme si místnost se čtvercovou sítí 4×8 z následujícího obrázku:



Úkolem je přesunout robota z políčka označeného S (o souřadnicích $[4, 1]$) na políčko označené C (o souřadnicích $[4, 8]$). Vstup vašeho programu by tedy vypadal následovně:

```
4 8
4 1
4 8
0 0 0 0 0 0 0 0
0 1 1 1 1 1 0 0
0 1 0 0 0 1 1 0
0 0 0 1 0 0 0 0
```

Optimální program pro přesun robota je následující (počáteční natočení robota je nahoru):

```
Krok Krok Krok Doprava Krok Krok Krok Krok Krok
Krok Krok Doprava Krok Krok Krok
```

Při této cestě robot udělá 13 kroků a dvakrát se otočí; všimněte si též existence cesty s 9 kroky a 4 otočeními — tato cesta je sice kratší, ale

podle zadání úlohy není optimální, neboť se během ní robot musí otočit vícekrát.

P – II – 3

Malému Pepíčkovi se jednoho dne dostaly do ruky nůžky. A protože byl Pepíček tvořivý po tatínkovi, jenž byl moderním malířem, rozhodl se vylepšit jeden jeho obraz ve tvaru konvexního n-úhelníku. Vybral si dva vrcholy tohoto n-úhelníku a obraz přestřihl po spojnici těchto dvou vrcholů. Pak si na jedné ze vzniklých částí opět vybral dva vrcholy a část opět přestřihl. Když si takto Pepíček chvilku hrál, přistihl ho tatínek, nůžek ho nekompromisně zbavil a začal zachraňovat, co se dá. Po chvilce zjistil, že obraz dohromady už nesloží. Rozhodl se tedy, že alespoň nalezne zbylou část s největším počtem vrcholů a tu vystaví na své nadcházející výstavě jako miniaturu. A právě s hledáním mu máte pomoci vy.

Navrhněte co nejefektivnější algoritmus, který dostane na vstupu počet vrcholů původního obrazu n, počet Pepíčkových střihů k a popis jednotlivých střihů a na základě těchto údajů určí počet vrcholů té zbylé části, která jich má nejvíce. Každý střih je popsán dvojicí čísel (a_i, b_i), což jsou čísla vrcholů v původním n-úhelníku, mezi kterými Pepíček střih vedl. Vrcholy n-úhelníku jsou číslovány po obvodu po řadě čísly od 1 do n. Snažte se, aby časová ani paměťová složitost vašeho řešení nezávisela na počtu vrcholů obrazu.

Příklad. Pro $n = 10$, $k = 3$ a střihy $(1, 8)$, $(7, 5)$ a $(4, 2)$ má největší část 6 vrcholů.

P – II – 4

(Definice dlaždicových programů je stejná jako v úloze P–I–4, pouze příklad jejich použití je složitější a ukazuje, jak je možno využívat více řádků dlaždic.)

Příklad. Zkonstruujme dlaždicový program, který bude ověřovat, zda je daná posloupnost tvořená přirozenými čísly $x_1, \ldots, x_n$ $(0 \leqq x_i \leqq 9)$ *vyvážená*, tzn. zda obsahuje stejný počet sudých a lichých čísel.

Myšlenka našeho řešení je velice jednoduchá: sestrojíme sadu dlaždic, která bude umožňovat právě taková vydláždění, v nichž v každém řádku přepíšeme právě jedno sudé a jedno liché číslo na •. Dolní okraj zdi obarvíme též barvou •. Jelikož obarvení spodního okraje je vyvážené a vydláždění každého řádku vyváženost zachovává, pak jakákoliv

vstupní posloupnost, pro kterou vydláždění existuje, je opravdu vyvážená. A naopak: pokud máme vyváženou posloupnost, pak snadno ověříme, že vydláždění existuje: vybereme si libovolné sudé a libovolné liché číslo (z vyváženosti víme, že v posloupnosti taková dvojice je), ta jedním řádkem dlaždic přepíšeme na • a toto opakujeme tak dlouho ($n/2$-krát), dokud nebudou všechna čísla přepsána. Pokud se nám tedy podaří takový dlaždicový program sestrojit, bude zadanou úlohu řešit se složitostí $O(n)$.

Hledaný program může vypadat například následovně:

$$T = \left\{ \begin{array}{|c|}\hline x \\ \hline A \quad A \\ \hline x \\ \hline \end{array}, \begin{array}{|c|}\hline x \\ \hline S \quad S \\ \hline x \\ \hline \end{array}, \begin{array}{|c|}\hline x \\ \hline L \quad L \\ \hline x \\ \hline \end{array}, \begin{array}{|c|}\hline x \\ \hline B \quad B \\ \hline x \\ \hline \end{array}, \begin{array}{|c|}\hline s \\ \hline A \quad S \\ \hline \bullet \\ \hline \end{array}, \begin{array}{|c|}\hline l \\ \hline A \quad L \\ \hline \bullet \\ \hline \end{array}, \begin{array}{|c|}\hline s \\ \hline L \quad B \\ \hline \bullet \\ \hline \end{array}, \begin{array}{|c|}\hline l \\ \hline S \quad B \\ \hline \bullet \\ \hline \end{array}; \right.$$
$$\left. x \in \{0, \ldots, 9, \bullet\},\ s \in \{0, 2, 4, 6, 8\},\ l \in \{1, 3, 5, 7, 9\} \right\},$$

levý okraj obarvíme barvou A, pravý barvou B a dolní barvou •. Z těchto dlaždic je možno konstruovat výhradně řádky typu

$$A \begin{array}{|c|}\hline x_1 \\ \hline A \quad A \\ \hline x_1 \\ \hline \end{array} \cdots \begin{array}{|c|c|c|}\hline x_{i-1} & x_i & x_{i+1} \\ \hline A \quad A & A \quad S & S \quad S \\ \hline x_{i-1} & \bullet & x_{i+1} \\ \hline \end{array} \cdots$$

$$\cdots \begin{array}{|c|c|c|}\hline x_{k-1} & x_k & x_{k+1} \\ \hline S \quad S & S \quad B & B \quad B \\ \hline x_{k-1} & \bullet & x_{k+1} \\ \hline \end{array} \cdots \begin{array}{|c|}\hline x_n \\ \hline B \quad B \\ \hline x_n \\ \hline \end{array} B$$

kde x_i je sudé a x_k liché, případně

$$A \begin{array}{|c|}\hline x_1 \\ \hline A \quad A \\ \hline x_1 \\ \hline \end{array} \cdots \begin{array}{|c|c|c|}\hline x_{i-1} & x_i & x_{i+1} \\ \hline A \quad A & A \quad L & L \quad L \\ \hline x_{i-1} & \bullet & x_{i+1} \\ \hline \end{array} \cdots$$

$$\cdots \begin{array}{|c|c|c|}\hline x_{k-1} & x_k & x_{k+1} \\ \hline L \quad L & L \quad B & B \quad B \\ \hline x_{k-1} & \bullet & x_{k+1} \\ \hline \end{array} \cdots \begin{array}{|c|}\hline x_n \\ \hline B \quad B \\ \hline x_n \\ \hline \end{array} B$$

pro x_i liché a x_k sudé. To jsou přesně řádky, jaké jsme potřebovali.

Soutěžní úloha. Sestrojte dlaždicový program, který o dané posloupnosti přirozených čísel $x_1, x_2, \ldots, x_n$ ($0 \leqq x_i \leqq 9$) rozhodne, zda je symetrická, tj. zda $x_1 = x_n$, $x_2 = x_{n-1}$, $\ldots$, $x_i = x_{n-i+1}$, $\ldots$, $x_n = x_1$.

P – III – 1

Pan Kašparov byl náruživý hráč šachů. Protože ale často postrádal vhodného protihráče a hrát sám proti sobě ho už nebavilo (vždy si odhalil všechny léčky), vymyslel si následující hru: Na šachovnici o rozměrech $N \times N$ je třeba rozmístit N věží tak, aby se vzájemně neohrožovaly. Aby hra nebyla příliš jednoduchá, pro každou věž je určen obdélník, do kterého se věž musí umístit.

Vaším úkolem je navrhnout algoritmus, který bude tuto hru hrát. Na vstupu dostane rozměr šachovnice a počet věží N a dále popis N obdélníků. Jeden obdélník je popsán čtveřicí čísel A_x, A_y, B_x, B_y, $1 \leqq \leqq A_x \leqq B_x \leqq N$, $1 \leqq A_y \leqq B_y \leqq N$, kde A_x, A_y jsou souřadnice levého horního rohu obdélníku a B_x, B_y jsou souřadnice pravého dolního rohu obdélníku. Řádky číslujeme od 1 do N shora dolů a sloupce od 1 do N zleva doprava. Na výstup pak algoritmus vypíše souřadnice jednotlivých rozmístěných věží, nebo zprávu, že rozmístění dle pravidel hry neexistuje. Pokud existuje více různých řešení, stačí najít jedno libovolné z nich.

Příklad 1:
```
N = 4
1 1 1 1
4 4 4 4
1 1 3 3
3 2 4 4
```
Rozmístění věží může být:
$(1,1)$, $(4,4)$, $(2,2)$, $(3,3)$.

Příklad 2:
```
N = 3
1 1 3 1
2 1 3 1
2 2 3 3
```
Rozmístění věží neexistuje.

P – III – 2

Napište program, který na vstupu obdrží přirozené číslo N a nalezne nejmenší takové přirozené číslo x, že x je dělitelné číslem N a zároveň dekadický zápis čísla x je tvořen pouze ciframi nula a jedna. V případě, že takové číslo x neexistuje, vypíše váš program vhodnou zprávu. Například pro $N = 6$ je hledaným číslem x číslo 1110.

(Definice dlaždicových programů je stejná jako v úloze P–I–4, pouze navíc vysvětluje, jak lze dlaždicové programy používat k výpočtu funkcí a uvádí příklad tohoto použití.)

Dlaždicové programy je možno používat nejen k řešení rozhodovacích problémů, ale také k výpočtu hodnot funkcí. Výpočet funkce $f(x)$ totiž můžeme snadno převést na rozhodovací problém $P(x,y)$ = = „je $y = f(x)$?“, o kterém budeme vědět, že pro každé x bude $P(x,y)$ splněno pro právě jednu hodnotu y. Navíc můžeme dlaždicovému programu x i y zadat jako jeden vstup tak, že barvy dlaždic nebudou odpovídat hodnotám, nýbrž jejich uspořádaným dvojicím.

Příklad. Zkonstruujme dlaždicový program, který pro každé číslo zapsané ve dvojkové soustavě spočte dvojkový zápis tohoto čísla vyděleného třemi (předpokládejme, že je dělitelné beze zbytku). Jinými slovy máme o dané posloupnosti dvojic $(x_1,y_1),\ldots,(x_n,y_n)$ zjistit, zda $\langle y_1,\ldots,y_n\rangle = \frac{1}{3}\langle x_1,\ldots,x_n\rangle$.

Řešení založíme na tradičním algoritmu na písemné dělení (ten je na použité číselné soustavě nezávislý): zvolíme $z_0 = 0$ a spočteme postupně pro všechna k hodnoty $z_k = (2z_{k-1} + x_k) \bmod 3$ a $y_k = \lfloor \frac{1}{3}(2z_{k-1} + x_k) \rfloor$. Nyní dokážeme indukcí, že pro každé k je

$$\langle x_1,\ldots,x_k\rangle = 3\cdot\langle y_1,\ldots,y_k\rangle + z_k.$$

Pro $k = 0$ rovnost platí. Platí-li pro $k-1$, pak pro k dostaneme:

$$\begin{aligned}\langle x_1,\ldots,x_k\rangle &= 2\cdot\langle x_1,\ldots,x_{k-1}\rangle + x_k = \\ &= 2\cdot(3\cdot\langle y_1,\ldots,y_{k-1}\rangle + z_{k-1}) + x_k = \\ &= 3\cdot 2\cdot\langle y_1,\ldots,y_{k-1}\rangle + 2z_{k-1} + x_k = \\ &= 3\cdot 2\cdot\langle y_1,\ldots,y_{k-1}\rangle + 3y_k + z_k = \\ &= 3\cdot\langle y_1,\ldots,y_k\rangle + z_k.\end{aligned}$$

Nyní si stačí zvolit následující sadu dlaždic:

$$T = \left\{ \boxed{\begin{array}{c} xy \\ a \quad b \\ \bullet \end{array}} ;\ x,y\in\{0,1\}, a\in\{0,1,2\}, b = (2a+x) \bmod 3,\ y = \left\lfloor \frac{1}{3}(2a+x)\right\rfloor \right\},$$

levý a pravý okraj budou mít barvu 0, spodní barvu •.

Z těchto dlaždic je možno konstruovat výhradně jednořádková vydláždění (• není barvou horního okraje žádné dlaždice), v nichž má k-tá dlaždice na svém okraji z_k a pro (x_k, y_k) na jejím horním okraji platí $y_k = \lfloor \frac{1}{3}(2z_{k-1} + x_k) \rfloor$. Jinými slovy tato vydláždění odpovídají přesně hodnotám spočteným naším algoritmem, tedy i požadovanému výsledku. Tím je problém vyřešen.

Soutěžní úloha. Sestrojte dlaždicový program, který bude uspořádávat posloupnosti nul a jedniček vzestupně, to znamená, že na posloupnost dvojic nul a jedniček $(x_1, y_1), \ldots, (x_n, y_n)$ odpoví ANO právě tehdy, pokud $y_1, \ldots, y_n$ je posloupnost vzniklá vzestupným uspořádáním posloupnosti $x_1, \ldots, x_n$, tzn. $y_1 \leqq \ldots \leqq y_n$ a posloupnosti x a y obsahují tytéž prvky, nanejvýš v jiném pořadí.

Příklad. Na posloupnost $(1,0),(0,0),(0,0),(1,1),(0,1),(1,1)$ program odpoví ANO, na $(1,1),(0,0)$ NE, na $(1,0),(1,1)$ taktéž NE.

P – III – 4

Program: FORMAT.PAS / FORMAT.C / FORMAT.CPP
Vstup: FORMAT.IN
Výstup: FORMAT.OUT

Pro textový editor potřebujeme napsat program sloužící k formátování textu. Editor pracuje ve znakovém režimu s neproporcionálním písmem. Všechny znaky tedy mají stejnou šířku a také mezera má pevnou šířku stejnou jako každý jiný znak. Rovněž pomocné symboly obsažené v textu (jako jsou interpunkční znaménka, závorky či uvozovky) se zpracovávají stejně jako jakékoliv jiné znaky. Editor je poměrně jednoduchý, takže dělení slov nepřipouští. Program budeme používat vždy k formátování jednoho odstavce textu.

Pro potřeby formátování rozumíme slovem každou souvislou posloupnost nemezerových znaků, která je na obou koncích ukončena mezerou nebo začátkem či koncem řádku. Pomocné symboly obsažené v textu jsou tedy součástí těch slov, od nichž nejsou odděleny mezerou.

Cílem formátování textu je vhodné rozložení slov na jednotlivé řádky tak, aby byl celý text zarovnán „do bloku“ (tzn. k levému i pravému okraji) při zadané šířce řádku. Přitom mezery mezi slovy musí být co možná nejmenší a v textu co nejrovnoměrněji rozloženy. Tyto obecné požadavky si nyní upřesníme: Velikosti mezer mezi slovy na témže řádku (kromě posledního řádku odstavce) se mohou lišit maximálně o 1, před

prvním a za posledním slovem na řádku nesmí být mezera. Pokud je na řádku pouze jedno slovo, je rozložení mezer libovolné. Na posledním řádku odstavce musí být slova oddělena právě jednou mezerou a před prvním slovem nesmí být mezera.

Splňuje-li text tyto závazné požadavky, potom kvalitu zformátování odstavce hodnotíme trestnými body. Ohodnocení odstavce je součtem ohodnocení jednotlivých řádků. Ohodnocení jednoho řádku je dáno výsledkem funkce $F(\langle Width\rangle, \langle Chars\rangle, \langle Words\rangle, \langle Last\rangle)$, kde $\langle Width\rangle$ je šířka stránky (tj. počet znaků na řádce zformátovaného textu včetně všech mezer), $\langle Chars\rangle$ je počet nemezerových znaků na řádce, $\langle Words\rangle$ je počet slov na řádce a $\langle Last\rangle$ značí, zda se ohodnocuje poslední řádek odstavce či nikoliv.

Ohodnocovací funkce F zapsaná v programovací jazyce C vypadá následovně:

```
int F(int Width, int Chars, int Words, int Last)
{
 int Spaces = Width - Chars - Words + 1; /* Počet zbytečných mezer */
 int BasePen = LINEPENALTY;      /* Základní trestné body za řádek */

 if (Spaces < 0)                 /* Nevejde se text na řádek? */
   return INFTYPEN;
 if (Last)                       /* Poslední řádek? */
 {
   if (4*(Chars + Words -1) <= Width) /* Je poslední řádek moc krátký? */
     BasePen += SMALLLINEPEN;
   return BasePen;
 }
 if (Words == 1)                 /* Pouze jedno slovo na řádku? */
   BasePen += SINGLEWORDPEN;
 return Spaces * Spaces + BasePen;   /* Ohodnocení celého řádku */
}
```

V Pascalu je zápis funkce F obdobný:

```
function F(Width, Chars, Words: Integer; Last: Boolean): Integer;
var
  Spaces : Integer;          { Počet zbytečných mezer na řádku }
  BasePen : Integer;         { Základní trestné body za řádek }
begin
  BasePen := LINEPENALTY;
  Spaces := Width - Chars - Words + 1;
  if Spaces < 0 then         { Nevejdou se slova na řádek? }
    F := INFTYPEN
  else if Last then begin    { Ohodnocujeme poslední řádek? }
    if 4*(Chars + Words - 1) <= Width then  { Je řádek moc krátký? }
      Inc(BasePen, SMALLLINEPEN);
    F := BasePen;
```

```
  end
  else begin
    if Words = 1 then        { Je jen jedno slovo na řádku? }
      Inc(BasePen, SINGLEWORDPEN);
    F := Spaces * Spaces + BasePen; { Spočteme výsledné ohodnocení }
  end;
end;
```

Hodnoty konstant jsou:

`LINEPENALTY` = 10

`SMALLLINEPEN` = 5

`SINGLEWORDPEN` = 20

`INFTYPEN` = 30 000

Vaším úkolem je zformátovat odstavec textu při zadané šířce řádku co nejkvalitněji, tzn. splnit všechny závazné požadavky kladené na formátování a přitom dosáhnout co nejnižšího ohodnocení odstavce trestnými body podle funkce F.

Vstup: Ve vstupním souboru `FORMAT.IN` je na prvním řádku zadána požadovaná šířka stránky po zformátování. Na dalších řádcích se nachází text odstavce určený ke zformátování. Můžete předpokládat, že žádný z těchto řádků není delší než 100 znaků, na začátku ani na konci žádného řádku není mezera a mezi jednotlivými slovy na řádku je vždy právě jedna mezera. Vstupní soubor nebude delší než 10 000 znaků (počítáno včetně mezer mezi slovy).

Výstup: Do výstupního souboru `FORMAT.OUT` zapište zadaný text odstavce zformátovaný co nejkvalitněji podle výše uvedených zásad (tj. s nejnižší možnou hodnotou ohodnocovací funkce).

Příklad.

`FORMAT.IN`

```
40
Each section in this document will have the string "<section>" at the
right-hand side of the section title. Each subsection will have
"<subsection>" at the right-hand side. These strings are meant to make
it easier to search through the document.
```

`FORMAT.OUT` (jedno z možných řešení; symbol '␣' označuje mezeru)

```
Each␣␣section␣␣in␣␣this␣␣document␣␣␣will
have␣␣the␣␣string␣␣"<section>"␣␣at␣␣␣the
right-hand␣side␣of␣␣the␣␣section␣␣title.
Each␣subsection␣will␣have␣"<subsection>"
at␣the␣right-hand␣␣side.␣␣These␣␣strings
are␣meant␣to␣make␣it␣␣easier␣␣to␣␣search
through␣the␣document.
```

P – III – 5

Program: OKRUZNI.PAS / OKRUZNI.C / OKRUZNI.CPP
Vstup: OKRUZNI.IN
Výstup: OKRUZNI.OUT

Ve městě Turiststadt začal vzkvétat turistický ruch. Aby podpořili jeho další rozkvět, rozhodli se moudří radní založit společnost City-tour. Posláním této společnosti je provozovat ve městě několik vyhlídkových okružních autobusových linek. Vaším úkolem je vytvořit program, který navrhne trasy vyhlídkových autobusových linek městem podle požadavků radních, popřípadě zjistí, že autobusové linky nelze dle jejich požadavků vytvořit.

Město je tvořeno křižovatkami, které jsou navzájem spojeny ulicemi. Každá ulice spojuje právě dvě křižovatky. Dvě stejné křižovatky mohou být spojeny více různými ulicemi. Křižovatkou rozumíme i místo, do kterého vede jen jedna nebo dvě ulice. Radní kladou na plánované trasy autobusových linek následující požadavky: Aby si turisté mohli pohodlně prohlédnout každou ulici ve městě a přitom se zbytečně neplýtvalo náklady na provoz autobusových linek, musí každou ulicí projíždět právě jedna autobusová linka. Žádná z linek nesmí projet některou z křižovatek více než jednou. Trasy linek musí být navrženy tak, aby první a poslední křižovatka na trase byla stejná — jinak by se linky provozované touto společností daly jen stěží nazývat okružní.

Vstup: První řádek vstupního souboru OKRUZNI.IN obsahuje dvě čísla oddělená mezerou — počet křižovatek (N, $1 \leqq N \leqq 120$) a počet ulic (M). Křižovatky jsou očíslovány čísly od 1 do N. Následujících M řádků vstupního souboru obsahuje popis jednotlivých ulic ve městě: Každý z těchto řádků obsahuje dvě čísla představující čísla křižovatek, které příslušná ulice spojuje. První číslo na každém z těchto řádků je menší než druhé z nich. Tyto řádky jsou v souboru setříděny podle prvního čísla; v případě, že se shoduje více ulic v prvním čísle, jsou setříděny podle druhého čísla. Můžete předpokládat, že počet různých ulic spojujících dvě stejné křižovatky je nejvýše 200.

Výstup: Výstupní soubor OKRUZNI.OUT obsahuje tolik řádků, kolik má společnost provozovat autobusových linek. Každý řádek obsahuje popis právě jedné autobusové linky. Trasa autobusové linky je popsána jako posloupnost čísel křižovatek, kterými linka projíždí. První a poslední číslo uvedené na řádku je tedy stejné (linka začíná a končí na stejné křižovatce). Jednotlivá čísla jsou na každém řádku oddělena právě jednou

mezerou. Pokud nelze trasy linek navrhnout tak, aby vyhovovaly podmínkám ze zadání úlohy, potom výstupní soubor obsahuje jediný řádek se slovem „Nelze“.

Příklad 1.

Vstupní soubor `OKRUZNI.IN`:

```
5 12
1 2
1 2
1 2
1 2
1 3
1 3
2 3
2 5
3 4
3 5
3 5
4 5
```

Výstupní soubor `OKRUZNI.OUT`:

```
1 2 3 1
2 1 2
3 4 5 3
2 5 3 1 2
```

Příklad 2.

Vstupní soubor `OKRUZNI.IN`:

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

Výstupní soubor `OKRUZNI.OUT`:

```
Nelze
```

P – I – 1

Je jasné, že za každé dva uzly hloubky K, $K > 0$ musí existovat jeden uzel hloubky $K - 1$, který není listem — každé dva uzly musíme pod nějaký uzel „zavěsit". Protože navíc pod každý uzel můžeme zavěsit pouze žádný nebo dva uzly, musí být počet uzlů hloubky K sudý. Výše uvedená pozorování nám již dávají návod na sestrojení algoritmu. Pro každou hloubku si budeme pamatovat počet listů a počet ostatních uzlů. Budeme postupovat od uzlů s největší hloubkou. Pro každou hloubku K, $K > 0$ zkontrolujeme, zda je počet uzlů v ní sudý. Pokud není, strom neexistuje. Pokud je počet uzlů sudý, za každé dva uzly umístěné ve stromu na dané hloubce přidáme do předchozí hloubky jeden uzel. Když se takto dostaneme až k uzlům hloubky 0, stačí ověřit, jestli v této hloubce leží právě jeden uzel, jak je vyžadováno v definici binárního stromu. Pokud není, strom neexistuje. To plyne z toho, že pokud neexistuje uzel hloubky 0, nebyl dán žádný list, a tedy strom nemůže mít žádné uzly. To je ale ve sporu s požadavkem, že každý strom musí mít alespoň kořen. Pokud je uzlů hloubky 0 naopak více, strom neexistuje, protože všechny uzly, které jsme přidávali, byly vynucené a každý strom s danými počty listů tedy musí mít na jednotlivých hloubkách alespoň tolik uzlů jako náš strom. Pokud existuje právě jeden uzel hloubky 0, snadno již ze spočítaných počtů listů a ostatních uzlů v jednotlivých hloubkách vytvoříme požadovaný zápis stromu. Jednoduše vypíšeme tolik `L`, kolik je počet listů dané hloubky, a tolik `U`, kolik je počet ostatních uzlů dané hloubky. Algoritmus má časovou i paměťovou složitost $O(N)$. Program je přímou implementací algoritmu.

P – I – 2

Algoritmus řešící tuto úlohu se dá rozdělit do tří fází. V první fázi se pro každé město spočítá, jaká je jeho vzdálenost od nepřítelem obsazených měst (ve smyslu definice uvedené v zadání). Ve druhé fázi se zjistí, jakou maximální vzdálenost od nepřátelských měst dokážeme udržet při cestě z počátečního do cílového města. Ve třetí fázi pak nalezneme nejkratší z tras vedoucích z počátečního do cílového města, které udržují spočtenou vzdálenost.

První fáze: Vzdálenost od obsazených měst budeme hledat pomocí prohledávání do šířky. U každého města si budeme udržovat informaci, zda jsme v něm již byli (na počátku bude nastaveno právě u všech obsazených měst) a jeho vzdálenost od nepřítele. Pro města obsazená nepřítelem bude tato vzdálenost rovna 0. Dále si budeme udržovat frontu měst ke zpracování, do které na začátku uložíme všechna nepřátelská města. V každém kroku výpočtu vždy vezmeme jedno město z fronty a u všech jeho sousedů, ve kterých jsme dosud nebyli, nastavíme vzdálenost o jedna větší, než je vzdálenost vybraného města. U všech těchto sousedů také označíme, že jsme v nich už byli, a přidáme je na konec fronty. První fáze výpočtu končí, když se vyprázdní fronta. Tehdy jsme prošli všechna města a určili jsme vzdálenost každého z nich od nepřítele.

Druhá fáze: V této fázi si budeme udržovat front hned několik, pro každou vzdálenost od nepřátelských měst jednu. Dále si pro každé město budeme zaznamenávat, zda jsme v něm už byli. Také si budeme pamatovat dosud největší nalezenou vzdálenost, kterou dokážeme udržet od nepřítele. Na začátku nastavíme udržitelnou vzdálenost od nepřítele na hodnotu vzdálenosti královského města od nepřítele a toto město vložíme do fronty pro příslušnou vzdálenost. U tohoto města také nastavíme, že jsme v něm už byli. Výpočet probíhá tak, že postupně vyzvedáváme města z fronty pro aktuální udržitelnou vzdálenost, dokud se tato fronta nevyprázdní. Když se fronta vyprázdní, snížíme udržitelnou vzdálenost o jedna a opět začneme vybírat města z příslušné fronty. Vždy, když vezmeme nějaké město z fronty, projdeme všechny jeho sousedy, u dosud nenavštívených z nich nastavíme příznak, že už jsme je nenavštívili, a přidáme je do fronty — jestliže je vzdálenost takového města od nepřátelských měst větší, než je aktuální udržitelná vzdálenost, přidáme vrchol do fronty odpovídající aktuální udržitelné vzdálenosti, jinak město přidáme do fronty odpovídající jeho vzdálenosti od nepřátelských měst. Druhá fáze končí, jakmile vybereme z fronty cílové město. Aktuální udržitelná vzdálenost je pak výslednou udržitelnou vzdáleností.

Třetí fáze: Tato fáze představuje opět prosté prohledávání do šířky. Pro každé město si pamatujeme, zda jsme v něm již byli, a pokud ano, zaznamenáme si také město, ze kterého jsme do něj přišli. Opět používáme frontu na dosud nezpracovaná města. Na začátku vložíme do fronty cílové město. U něj nastavíme, že jsme v něm již byli, a jako jeho předchůdce nastavíme je samé. V každém kroku výpočtu pak vezmeme jedno město z fronty a projdeme všechny jeho sousedy. Každého souseda, kterého jsme dosud nenavštívili a jehož vzdálenost od nepřátelských

měst je větší nebo rovna výsledné udržitelné vzdálenosti, označíme jako navštíveného a přidáme ho na konec fronty. Také u něj jako město, ze kterého jsme přišli, nastavíme právě vybrané město. Prohledávání končí ve chvíli, když je z fronty vyzvednuto počáteční (královské) město. Poté už jenom projdeme cestu z počátečního do cílového města (to je velmi snadné díky odkazům na města, odkud jsme do nich při prohledávání přišli) a cestu vypíšeme.

Algoritmus má časovou složitost $O(M + N)$, kde M je počet cest a N je počet měst.

Správnost algoritmu budeme ukazovat opět po fázích. To, že algoritmus spočte správně vzdálenosti od nepřátelských měst v první fázi, plyne z následujícího: Na počátku mají všechny vrcholy se vzdáleností nula tuto vzdálenost přiřazenu. V okamžiku, kdy jsou zpracovány všechny vrcholy vzdálenosti nula, prošli jsme všechny jejich sousedy, přiřadili jsme jim vzdálenost jedna a zařadili je do fronty. Protože jiné vrcholy vzdálenost jedna mít nemohou, je vzdálenost jedna přiřazena právě všem správným vrcholům. Tuto úvahu lze snadno zobecnit pro libovolnou vzdálenost D. Prohledávání tedy skutečně určí vzdálenosti od nepřátelských měst správně.

Ve druhé fázi se správně spočítá maximální udržitelná vzdálenost od obsazených měst. Sledujeme v ní totiž souběžně všechny možné trasy vedoucí z počátečního města tak dlouho, dokud dokážeme udržet vzdálenost počátečního města (výsledná vzdálenost od nepřítele zřejmě nemůže být větší než vzdálenost počátečního města). Když už neexistuje město s dostatečně velkou vzdáleností, do kterého bychom mohli jít, snížíme udržitelnou vzdálenost o jedna. Všechny vrcholy se vzdáleností o jedna nižší, do kterých se dokážeme dostat přes vrcholy s dosavadní udržitelnou vzdáleností, máme již připraveny v příslušné frontě a začneme tedy prohledávat z nich. Protože udržitelnou vzdálenost snižujeme až když jsme se již dostali všude, kam to bylo možné, její výsledná hodnota bude zřejmě nejvyšší možná.

To, že ve třetí fázi nalezneme nejkratší trasu s danou vzdáleností, je zřejmé. Provádíme totiž jednoduché prohledávání do šířky s tím, že ignorujeme města s příliš malou vzdáleností od nepřítele. Nalezneme tedy určitě trasu s dostatečnou vzdáleností od nepřítele. Skutečnost, že to bude trasa nejkratší možná, plyne z vlastností prohledávání do šířky uvedených v první části důkazu. Program je přímou implementací uvedeného algoritmu.

Řešení úlohy rozdělíme na několik částí — nejprve zformulujeme nutné a postačující podmínky pro to, aby skupina mohla projet trasu dle podmínek v zadání úlohy, poté vyřešíme úlohu a) a nakonec nalezneme řešení úlohy b).

Plánovanou trasu lze projet právě tehdy, když tato trasa není delší než vzdálenost, kterou skupina urazí za den, tj. $L \leqq K$, anebo když jsou splněny zároveň všechny čtyři následující podmínky:

1. Na trase je alespoň jeden kemp.
2. Vzdálenost prvního kempu od začátku trasy je nejvýše K, tj. $l_1 \leqq K$.
3. Vzdálenost libovolných dvou po sobě následujících kempů není větší než K, tj. $l_{i+1} - l_i \leqq K$ pro $1 \leqq i \leqq N - 1$.
4. Vzdálenost posledního kempu od konce trasy je nejvýše K, tj. $L - l_N \leqq K$.

Nutnost všech uvedených podmínek je zřejmá; pokud jsou tyto podmínky splněny, pak plán cesty, ve kterém skupina bude cestovat $N+1$ dní a i-tý den přespí v i-tém kempu, splňuje podmínky ze zadání úlohy.

Nyní vyřešíme úlohu a). Na chvíli si představme, že jsme každému kempu přiřadili číslo d_i, které udává, kolikátý den nejdříve můžeme do tohoto kempu dorazit. Číslo d_i přiřazené kempu i zřejmě splňuje jednu z následujících dvou podmínek:

1. Je $d_i = 1$ a $l_i \leqq K$ — do kempu lze dorazit hned první den.
2. Existuje $j < i$ takové, že $l_i - l_j \leqq K$ a $d_j = d_i - 1$ (do i-tého kempu dorazíme tak, že den před tím přespíme v j-tém kempu), ale neexistuje $j < i$ takové, že $l_i - l_j \leqq K$ a $d_j < d_i - 1$ (jinak by bylo možné do j-tého kempu dorazit již dříve).

Podle těchto podmínek by bylo možné spočítat všechna d_i v čase $O(N)$, náš program však d_i počítat nebude. Plán cesty splňující podmínku a) by mohl vypadat například tak, že h-tý den skupina přespí v i-tém kempu, pokud $d_i = h$ a $d_{i+1} = h+1$; skupina navíc přespí v N-tém kempu, pokud plán cesty bez tohoto kempu nesplňuje podmínku omezující maximální vzdálenost, kterou lze urazit za jeden den. Budeme přímo vytvářet takovýto plán cesty — pokud dosud vytvořený plán cesty končí kempem ve vzdálenosti l od začátku výletu a $L - l > K$ (nelze bez přespání dorazit na konec trasy), pak další den skupina přespí v i-tém kempu, pokud $l_i - l \leqq K$ a i je maximální s touto vlastností. Vytvořit program pracující podle právě popsaného postupu je triviální; časová složitost algoritmu je $O(N)$.

Dále vyřešíme úlohu b). Každému kempu přiřadíme číslo e_i, které udává, kolik by cyklisté museli zaplatit za přespání na cestě z i-tého kempu na konec výletu (počítáno včetně poplatku za přespání v i-tém kempu). Čísla e_i náš algoritmus spočítá postupně pro $i = N$ až $i = 1$; kromě těchto čísel, si pro každý z kempů uložíme informaci, do kterého následujícího kempu se z něj máme vydat, abychom za noclehy zaplatili optimální cenu e_i. Čísla e_i lze spočítat podle následujícího předpisu:

1. Pokud $L - l_i \leqq K$, potom $e_i = c_i$; z i-tého kempu lze dorazit na konec trasy během jednoho dne.
2. Pokud $L - l_i > K$, potom $e_i = min_j(c_i + e_j) = c_i + min_j e_j$, kde se minimum počítá přes všechna $j > i$ taková, že $l_j - l_i \leqq K$; to j, pro které se nabývá minima, určuje pořadí kempu, ve kterém přespíme ten den, kdy vyjedeme z i-tého kempu.

První den, pokud $L > K$, přespíme v kempu s číslem i s minimálním e_i, pro který platí $l_i \leqq K$. Jak bude algoritmus pracovat je nyní již jasné. Zbývá ještě určit, jak rychle lze najít j, které minimalizuje vztah v druhém bodě.

K rychlému nalezení indexu j, použijeme datovou strukturu, která se nazývá halda. Halda je datová struktura, která umožňuje v konstantním čase určit nejmenší z prvků v haldě; prvek do haldy přidat nebo vyjmout nejmenší prvek umí v čase logaritmickém v počtu prvků obsažených v haldě. V haldě si budeme udržovat čísla kempů setříděná podle hodnot e_i; na začátku budeme mít v haldě navíc konec trasy, jehož hodnotu budeme považovat za rovnu nule (bude nejmenším prvkem haldy). Nalezení vhodného j bude probíhat následovně: Zjistíme, zda nejmenší prvek haldy je od i-tého kempu vzdálený nejvýše o K — pokud ano, nalezli jsme příslušné j, v opačném případě z haldy odstraníme tento prvek a celý postup zopakujeme. Poté do haldy přiřadíme i-tý kemp. Za předpokladu logaritmického času přidání prvku do haldy a vyjmutí nejmenšího prvku z haldy je celková doba běhu algoritmu $O(N \log N)$.

Haldu budeme reprezentovat v poli. Bude-li v haldě n prvků, pak její prvky budou uloženy v poli na pozicích s čísly 0 až $n - 1$. Pro prvek x s indexem k budeme prvky na pozicích $2k + 1$ a $2k + 2$ nazývat syny prvku x a prvek x budeme nazývat otcem těchto prvků. Všimněte si, že každý prvek, mimo prvku na pozici 0, má právě jednoho otce. Při práci s haldou budeme dodržovat následující invariant: Každý prvek je větší než jeho otec. Nejmenším prvkem v haldě je proto prvek na nulté pozici; zjištění nejmenšího prvku haldy lze tedy provést v konstantním čase. Přidání prvku do haldy bude probíhat následovně: Je-li v haldě n

prvků, pak nový prvek umístíme na pozici n; pokud je jeho otec větší, vyměníme přidávaný prvek s jeho otcem a celý postup opakujeme tak dlouho, dokud nový prvek není nultým prvkem nebo jeho otec není menší než on. V každém kroku se index nového prvku v poli zmenší alespoň na polovinu a tedy se po logaritmicky mnoha krocích zastavíme. Odebrání prvku z haldy bude probíhat podobně: Je-li v haldě n prvků, pak nejmenší prvek haldy nahradíme prvkem z $(n-1)$-té pozice; tento prvek porovnáme s oběma jeho syny a popřípadě ho zaměníme s menším z obou jeho synů. Skončíme, pokud je tento prvek menší než oba jeho synové. Protože se v každém kroku posuneme na prvek s alespoň dvojnásobným indexem, odebrání nejmenšího prvku z haldy bude trvat čas logaritmický v počtu prvků v haldě.

P – I – 4

a) Mějme zadáno nějaké dvojkové číslo $\langle x_1, \ldots, x_n \rangle$, o němž máme rozhodnout, zda je dělitelné pěti. Sestrojíme sadu dlaždic, jíž bude možno vydláždit pouze zeď o jednom řádku, a to tak, aby barva pravé hrany i-té dlaždice (tu budeme značit p_i) odpovídala zbytku po dělení dvojkového čísla $\langle x_1, \ldots, x_i \rangle$ pěti. Když navíc zvolíme barvu pravého okraje zdi tak, aby odpovídala zbytku 0, půjde zeď vydláždit právě tehdy, je-li zadané číslo dělitelné pěti, a to je přesně to, co potřebujeme.

Použijeme dlaždice následujících typů:

$$T = \left\{ \begin{array}{|c|} \hline y \\ \hline x \quad z \\ \hline \bullet \\ \hline \end{array} \; ; \; 0 \leqq x \leqq 4, 0 \leqq y \leqq 1, z = (2x + y) \bmod 5 \right\},$$

levý i pravý okraj budou mít barvu 0 a dolní okraj barvu •. Jelikož barva • se nevyskytuje na horní hraně žádné dlaždice, musí být každé korektní vydláždění tvořeno jediným řádkem. Zbývá dokázat, že barvy pravých hran dlaždic odpovídají zbytkům, což učiníme indukcí:

- $p_1 = \langle x_1 \rangle = \langle x_1 \rangle \bmod 5$ (existuje právě jedna dlaždice, která může být na prvním políčku — ta, která má na levém okraji nulu a na horním okraji x_1).
- Je-li $p_i = \langle x_1, \ldots, x_i \rangle \bmod 5$, může být na $i+1$-ním políčku pouze jediná dlaždice (mající na levém okraji p_i a na horním okraji x_{i+1}) a její pravý okraj má barvu $p_{i+1} = (2p_i + x_{i+1}) \bmod 5 = (2(\langle x_1, \ldots, x_i \rangle \bmod 5) + x_{i+1}) \bmod 5 = (2\langle x_1, \ldots, x_i \rangle + x_{i+1}) \bmod 5 = \langle x_1, \ldots, x_{i+1} \rangle \bmod 5$.

Z toho plyne, že popsaný dlaždicový program řeší zadanou úlohu se složitostí $O(1)$.

b) Použijeme následující typy dlaždic:

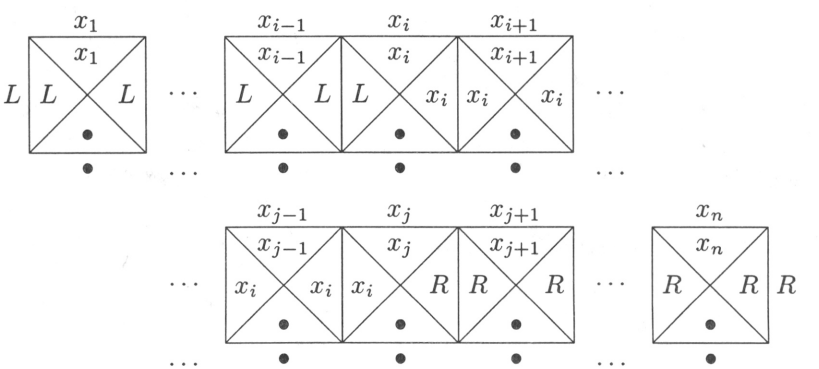
▷ „levé“ dlaždice $\left\{ \begin{array}{|c|}\hline x \\ L \times x \\ \bullet \\ \hline \end{array} ; 0 \leqq x \leqq 9 \right\}$,

▷ „pravé“ dlaždice $\left\{ \begin{array}{|c|}\hline y \\ x \times R \\ \bullet \\ \hline \end{array} ; 0 \leqq x, y \leqq 9, x \neq y \right\}$,

▷ „opakovací“ dlaždice $\left\{ \begin{array}{|c|}\hline x \\ L \times L \\ \bullet \\ \hline \end{array}, \begin{array}{|c|}\hline x \\ R \times R \\ \bullet \\ \hline \end{array}, \begin{array}{|c|}\hline x \\ y \times y \\ \bullet \\ \hline \end{array} ; 0 \leqq x, y \leqq 9 \right\}$.

Levý okraj zdi obarvíme barvou L, pravý barvou R a spodní barvou $\bullet$. Jelikož barva $\bullet$ se nevyskytuje na horní hraně žádné dlaždice, musí být každé korektní vydláždění tvořeno jediným řádkem.

Z toho, jak jsme si typy dlaždic nadefinovali, ihned plyne, že každé vydláždění zdi musí vypadat takto:



(zleva doprava: nejprve [možno i prázdná] posloupnost dlaždic opakujících L, pak jedna levá dlaždice, následuje opět několik opakovacích dlaždic, jedna pravá dlaždice a případně opakování R). Takové vydláždění je ovšem korektní právě tehdy, bylo-li možné nalézt indexy i a j takové, že $x_i \neq x_j$ (díky definici barev hran pravé dlaždice), takže náš dlaždicový program řeší zadanou úlohu se složitostí $O(1)$.

Poznámka. Naše řešení využívá toho, že dlaždicové programy jsou nedeterministické (to znamená, že nemají pevně definovaný průběh výpočtu a místo toho připouštějí více různých výpočtů s tím, že odpovědí

programu je ANO, pokud existuje alespoň jeden korektní výpočet) a že nám nedeterminismus „uhodne“ polohu nějakých dvou různých prvků vstupní posloupnosti.

P – II – 1

Předpokládejme, že máme délky dřevěných tyček setříděny podle velikosti, tj. $d_1 < d_2 < \ldots < d_N$. Potom trojice indexů $i < j < k$ určuje tyčky tvořící trojúhelník právě tehdy, pokud $d_k < d_i + d_j$. Zbývající dvě trojúhelníkové nerovnosti jsou totiž splněny triviálně, neboť $d_i, d_j < d_k$. Pro libovolnou dvojici indexů $i < j$ označme symbolem $l(i,j)$ největší číslo k takové, že $d_k < d_i + d_j$; speciálně tedy platí $l(i,j) \geqq j$. Zvolenou dvojici indexů $i < j$ lze doplnit na trojici $i < j < k$ určující trojúhelník právě těmi k, pro která platí $j < k \leqq l(i,j)$. Pro pevnou dvojici indexů i a j tedy existuje právě $l(i,j) - j$ takových k, že tyčky s indexy $i < j < k$ tvoří trojúhelník. K určení počtu trojúhelníků proto stačí spočítat hodnoty $l(i,j)$ pro všechna $i < j$ a sečíst výrazy $l(i,j) - j$.

Z definice $l(i,j)$ plyne, že $N = l(i,N) \geqq l(i,N-1) \geqq l(i,N-2) \geqq \geqq \ldots \geqq l(i,i+1)$. Náš program bude pracovat následovně: Pro každé i spočteme hodnoty $l(i,N-1), l(i,N-2), \ldots, l(i,i+1)$ a současně budeme počítat součet $(l(i,N-1)-(N-1))+\ldots+(l(i,i+1)-(i+1))$, který představuje počet trojúhelníků, jejichž nejkratší strana má délku d_i. Hodnotu $l(i,j)$ spočítáme tak, že hodnotu $l(i,j+1)$ budeme zmenšovat o jedničku tak dlouho, dokud $d_{l(i,j)} \geqq d_i + d_j$. Protože celkový počet zmenšení o jedničku během výpočtu hodnot $l(i,N-1), l(i,N-2), \ldots, l(i,i+1)$ je nejvýše $N-2$, je doba výpočtu pro jedno pevné i lineární v N. Celková doba výpočtu pro všech $N-2$ možných hodnot i je tedy $O(N^2)$. V tomtéž čase můžeme provést i úvodní setřídění zadaných délek tyček, a tedy časová složitost našeho algoritmu je $O(N^2)$ a paměťová pak $O(N)$.

```
program trojuhelniky;              { P-II-1 }
const MAXN=1000;
var N:word; { počet tyček }
    T:longint; { počet trojúhelníků }
    i,j,k:word;
    d:array[1..MAXN] of real; { délky tyček }
    e:real;
begin
 read(N);
 for i:=1 to N do read(d[i]);
 for i:=1 to N-1 do { setřídíme délky tyček }
  for j:=i+1 to N do
   if d[i]>d[j] then
```

```
    begin
     e:=d[i]; d[i]:=d[j]; d[j]:=e
    end;
 T:=0;
 for i:=1 to N-2 do { i - nejkratší tyčka z trojice }
  begin
   j:=N-1; k:=N; { j - druhá nejkratší tyčka z trojice }
   repeat
    while (j<k) and (d[k]>=d[i]+d[j]) do { podmínku (j<k) lze vypustit }
     dec(k); { hledáme nejdelší tyčku do trojice }
    T:=T+k-j;
    dec(j);
   until j=i;
  end;
 writeln(T);
end.
```

P – II – 2

Nejprve si rozmysleme, jak bychom zadanou úlohu řešili, kdybychom chtěli najít nejkratší cestu robota mezi dvěma zadanými políčky. Algoritmus pro řešení této úlohy je znám pod názvem prohledávaní do šířky nebo též algoritmus vlny. Každému políčku v průběhu výpočtu přiřadíme číslo, jež udává minimální počet kroků, které robot potřebuje k přemístění z počátečního políčka na uvažované políčko. Algoritmus pracuje ve fázích. Nejprve počátečnímu políčku přiřadí nulu. V i-té fázi přiřadí číslo i všem políčkům, která sousedí s nějakým políčkem, kterému bylo přiřazeno číslo $i-1$ a kterým jsme dosud žádné číslo nepřiřadili. Je zřejmé, že takto přiřazená čísla určují minimální počet kroků potřebný k přemístění robota z počátečního políčka.

Nyní si rozmyslíme, jak lze tento algoritmus modifikovat tak, aby řešil úlohu ze zadání. V i-té fázi nebudeme číslovat políčka ve vzdálenosti i kroků, ale políčka, na které se lze přesunout cestou s i změnami směru. Přesněji v i-té fázi budeme číslovat ta políčka, která leží ve stejném řádku nebo sloupci jako některé políčko s číslem $i-1$ a nejsou od něj v tomto řádku či sloupci oddělena překážkou. Správnost tohoto algoritmu je zřejmá. Zbývá domyslet detaily jeho implementace. Políčka si budeme uchovávat v poli tak, že políčka se stejným číslem budou tvořit souvislé úseky a políčka s nižším číslem budou předcházet políčkům s vyšším číslem. Pokud přijdeme v průběhu fáze na dosud neočíslované políčko, zařadíme ho na konec pole. V průběhu algoritmu vyjmeme vždy první nezpracované políčko z pole a prohledáme jeho řádek a sloupec. Pole, se kterým se pracuje právě popsaným způsobem, se obvykle nazývá

fronta — políčka se stavějí na konec fronty a čekají, až na ně přijde řada (budou zpracována). Nechť M a N jsou rozměry čtvercové sítě, potom zpracování každého z $M \times N$ políček vyžaduje čas $O(M + N)$. Celková časová složitost našeho algoritmu by tedy byla $O(M^2N + MN^2)$.

Čas potřebný k výpočtu však lze ještě zlepšit. Jeden a tentýž souvislý úsek v řádku bez překážek je totiž prohledáván několikrát — pro každé políčko z tohoto souvislého úseku jednou. Přitom by ale stačilo prohledat ho jenom z toho políčka, na které přijdeme nejdříve. Proto si budeme u každého políčka pamatovat, zda jsme prohledali souvislý úsek řádku (sloupce) bez překážek, ve kterém se toto políčko nachází. Před prohledáváním řádku (sloupce) otestujeme tento příznak a pokud jsme již příslušný souvislý úsek prohledali, tak vyjmeme ke zpracování další políčko z fronty. Všimněte si, že pro každé políčko je třeba udržovat dva příznaky — jeden pro souvislý úsek bez překážek v řádku a jeden pro úsek ve sloupci. Celkový čas strávený načítáním vstupních dat, prací s frontou a výpisem řešení je zřejmě $O(MN)$. Zbývá stanovit čas potřebný k prohledávání řádků a sloupců. Každý souvislý úsek bez překážek je prohledán právě jednou a součet délek všech souvislých úseků bez překážek je určitě nejvýše MN (tolik je totiž políček ve čtvercové síti). Prohledání jednoho souvislého úseku lze snadno provést v čase lineárním v jeho délce a tedy i celková časová složitost našeho algoritmu je $O(MN)$; paměťová složitost je též $O(MN)$.

```
program robot;          { P-II-2 }
const MAX=20;         { maximální rozměry čtvercové sítě v místnosti }
type policko = record
                x,y : word;
               end;
var sirka, vyska: word; { rozměry místnosti }
    prekazky: array[1..MAX,1..MAX] of boolean;
     { rozložení překážek v místnosti }
    navstiveno, svisle, vodorovne: array[1..MAX,1..MAX] of boolean;
     { indikátory navštívení políček v místnosti }
    predchozi: array[1..MAX,1..MAX] of policko;
     { předchozí políčko na optimální cestě }
    start, cil: policko;
     { počáteční a cílové políčko }
    fronta: array[1..MAX*MAX] of policko;
    zpracovano, vefronte: word;
     { fronta prohledávání do šířky }
    x, y: word; { pomocné proměnné }
    i: integer;
procedure vypis( x, y: word);
 var x0, y0: word;
     k: integer;
```

```
 begin
  x0:=predchozi[x,y].x; { předchozí políčko na optimální cestě }
  y0:=predchozi[x,y].y;
  if ( x0 = predchozi[x0,y0].x ) and ( y0 = predchozi[x0,y0].y ) then
    begin
      { Jsme na počátečním políčku ... }
      if x0 < x then
                write('Program pro robota (počáteční natočení DOLŮ):');
      if x0 > x then
                write('Program pro robota (počáteční natočení NAHORU):');
      if y0 < y then
                write('Program pro robota (počáteční natočení DOPRAVA):');
      if y0 > y then
                write('Program pro robota (počáteční natočení DOLEVA):');
    end
  else
    begin
      { Nejprve vypíšeme předchozí políčko a potom směr našeho otočení }
      vypis(x0,y0);
      k:=(x-x0)*(y0-predchozi[x0,y0].y)-(x0-predchozi[x0,y0].x)*(y-y0);
      if k > 0 then write(' <DOPRAVA>');
      if k < 0 then write(' <DOLEVA>');
    end;
  k:=x+y-x0-y0; { Spočítáme počet kroků, které je třeba udělat }
  if k < 0 then k:=-k;
  while k > 0 do
    begin
      write(' <KROK>');
      dec(k)
    end
 end;
begin
  { Načteme rozměry sítě, překážky, počáteční a cílové políčko }
  readln(vyska,sirka);
  readln(start.x,start.y);
  readln(cil.x,cil.y);
  for x:= 1 to vyska do
    for y:= 1 to sirka do
      begin
        read(i);
prekazky[x,y]:=(i=1);
        navstiveno[x,y]:=false;
svisle[x,y]:=false;
vodorovne[x,y]:=false;
      end;
  { Test na shodu počátečního a cílového políčka }
  if ( start.x = cil.x ) and ( start.y = cil.y ) then
    begin
      writeln('Počáteční a cílové políčko jsou stejné.');
      halt;
    end;
  { Inicializace prohledávání do šířky }
  zpracovano:=0;
  vefronte:=1;
```

```
  fronta[1]:=start;
  navstiveno[start.x,start.y]:=true;
  predchozi[start.x,start.y]:=start;
  { Cyklus prohledávání do šířky }
  while ( zpracovano < vefronte ) do
    begin
      inc(zpracovano);
      x:=fronta[zpracovano].x;
      y:=fronta[zpracovano].y;
      if not vodorovne[x,y] then
        begin
  { Projedeme síť vodorovně ( řádek ) }
          vodorovne[x,y]:=true;
          i:=1;
  while ( y+i <= sirka ) and not ( prekazky[x,y+i] ) do
    begin
              vodorovne[x,y+i]:=true;
              if not navstiveno[x,y+i] then
                begin
                  navstiveno[x,y+i]:=true;
                  predchozi[x,y+i]:=fronta[zpracovano];
                  inc(vefronte);
                  fronta[vefronte].x:=x;
                  fronta[vefronte].y:=y+i;
                end;
      inc(i);
    end;
          i:=-1;
  while ( y+i >= 1 ) and not ( prekazky[x,y+i] ) do
    begin
              vodorovne[x,y+i]:=true;
              if not navstiveno[x,y+i] then
                begin
                  navstiveno[x,y+i]:=true;
                  predchozi[x,y+i]:=fronta[zpracovano];
                  inc(vefronte);
                  fronta[vefronte].x:=x;
                  fronta[vefronte].y:=y+i;
                end;
      dec(i);
    end;
end;
      if not svisle[x,y] then
        begin
  { Projedeme síť svisle ( sloupeček ) }
          svisle[x,y]:=true;
          i:=1;
  while ( x+i <= vyska ) and not ( prekazky[x+i,y] ) do
    begin
              svisle[x+i,y]:=true;
              if not navstiveno[x+i,y] then
                begin
                  navstiveno[x+i,y]:=true;
                  predchozi[x+i,y]:=fronta[zpracovano];
```

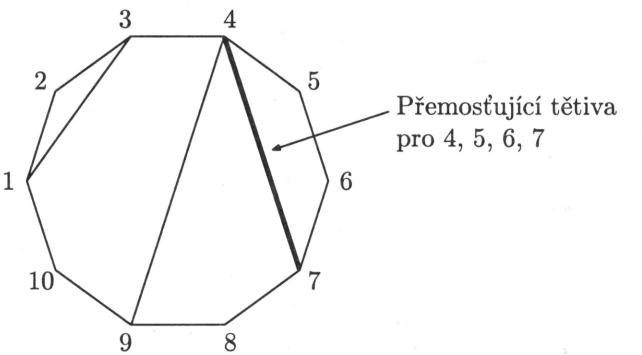
```
                inc(vefronte);
                fronta[vefronte].x:=x+i;
                fronta[vefronte].y:=y;
              end;
      inc(i);
    end;
        i:=-1;
  while ( x+i >= 1 ) and not ( prekazky[x+i,y] ) do
    begin
          svisle[x+i,y]:=true;
          if not navstiveno[x+i,y] then
            begin
              navstiveno[x+i,y]:=true;
              predchozi[x+i,y]:=fronta[zpracovano];
              inc(vefronte);
              fronta[vefronte].x:=x+i;
              fronta[vefronte].y:=y;
            end;
      dec(i);
    end;
end;
    end;
  if not navstiveno[cil.x,cil.y] then
    begin
      { Na cílové políčko se nelze dostat ... }
      writeln('Cesta z počátečního na cílové políčko neexistuje.');
      halt
    end;
  { A vypíšeme nalezenou cestu ... }
  vypis(cil.x,cil.y);
  writeln;
end.
```

P – II – 3

Úlohu si nejdříve lehce přeformulujeme. Obraz je vlastně nějaký konvexní n-úhelník a střihy jsou neprotínající se tětivy tohoto mnohoúhelníku. Úkolem je nalézt mnohoúhelník s největším počtem vrcholů, ve kterém neleží žádná tětiva (nadále budeme tento mnohoúhelník označovat jako „největší mnohoúhelník“).

Algoritmus nejdříve upraví popis každé tětivy tak, aby počáteční vrchol tětivy měl menší číslo než vrchol koncový. Pak všechny tětivy setřídí. Při třídění se porovnává nejdříve číslo počátečního vrcholu. Pokud je u více tětiv stejné, bere se obrácené pořadí čísel jejich koncových vrcholů (tedy $(1,2) < (2,3)$, ale $(1,2) > (1,3)$). Po setřídění začne algoritmus hledat největší mnohoúhelník. Algoritmus při hledání využívá toho, že v každém mnohoúhelníku (až na jeden) existuje právě jedna tětiva „přemosťující“ stranu $(1,n)$. Problematickým mnohoúhelníkem je ten, který

stranu $(1, n)$ obsahuje. Problémy s tímto mnohoúhelníkem řešíme tak, že uvažujeme pomocnou tětivu $(1, n)$. „Přemosťující" tětiva bude mít ze všech tětiv a stran mnohoúhelníku nejmenší počáteční číslo vrcholu a největší koncové číslo vrcholu. (Kdyby měla nějaká tětiva větší koncové i počáteční číslo vrcholu, musela by „přemosťující" tětivu někde protnout. Stejně tak pokud by byla obě čísla vrcholů menší. Pokud by bylo počáteční číslo menší a koncové větší, nebyla by zase naše tětiva „přemosťující".) Také platí, že každá tětiva je „přemosťující" pro právě jeden mnohoúhelník (mnohoúhelník ohraničený tětivou (i, j) bude obsahovat vrcholy i, j a ještě nějaké vrcholy z $i \ldots j$, obr. 27).



Obr. 27

A nyní již k hledání největšího mnohoúhelníku. To má následující ideu: Algoritmus postupně prochází vrcholy n-úhelníku od 1 do n. Udržuje si přitom zásobník, v němž jsou uloženy tětivy, u kterých prošel jejich počátečním vrcholem, ale dosud ne jejich koncovým vrcholem. Jsou to tedy „přemosťující" tětivy pro dosud neuzavřené mnohoúhelníky. U každé tětivy na zásobníku si také udržuje dosud napočítaný počet vrcholů pro mnohoúhelník omezený danou tětivou. Protože aktuální vrchol vždy patří k mnohoúhelníku omezenému nejpozději začínající „přemosťující" tětivou, stačí vždy upravovat jen počet vrcholů u tětivy na vrcholu zásobníku.

Konkrétní implementace hledání: Vždy když se v algoritmu posuneme do dalšího vrcholu, odebereme ze zásobníku tětivy, které v tomto vrcholu končí. Prošli jsme totiž všechny vrcholy, které mohly ležet v mnohoúhelnících ohraničených těmito tětivami. K těmto tětivám už tedy byly spočítány počty vrcholů v jimi ohraničených mnohoúhelnících, a tak stačí podle těchto hodnot upravit dosud nalezené maximum. Při každém odebrání tětivy ze zásobníku algoritmus přičte jedna k počtu vrcholů u tětivy

na vrcholu zásobníku, protože do příslušného mnohoúhelníku se dostal i aktuální vrchol. Pak přidá všechny tětivy začínající v daném vrcholu do zásobníku. Počty vrcholů u nových tětiv nastavuje na jedna, protože se musí započítat aktuální vrchol. Díky setřídění tětiv stačí pouze tětivy po řadě odebírat z pole, dokud je jejich počáteční vrchol shodný s aktuálním. Setřídění také zajistí, že z tětiv začínajících v aktuálním vrcholu budou ty, které skončí později, vloženy do zásobníku dříve. Když jsou přidány všechny tětivy začínající v aktuálním vrcholu, přičte se k tětivě na vrcholu zásobníku jedna za vrchol, do kterého se algoritmus přesouvá.

Právě uvedený algoritmus můžeme ještě zrychlit. Stačí si uvědomit, že je zbytečné posouvat se po obvodu pouze po jednom vrcholu. Stačí nám vlastně jen obejít vrcholy, ve kterých nějaká tětiva začíná nebo končí. To, o kolik se máme posunout, snadno zjistíme jako minimum z konce tětivy na vrcholu zásobníku a počátku první dosud nezařazené tětivy. Získáme tak algoritmus s časovou složitostí $O(k \log k)$. Čas $O(k \log k)$ totiž strávíme tříděním. Samotný průchod n-úhelníkem nám zabere pouze $O(k)$, protože každou tětivu pouze jednou přidáme na zásobník a jednou ji z něj odebereme. Počty vrcholů upravujeme dohromady pouze $2k$-krát. Správnost algoritmu byla ukázána v popisu.

```
program Pepik;  { P-II-3 }
const
  MAXK = 100;
type
  Cut = record
          a, b : Integer;
        end;
  CutA = Array[1..MAXK] of cut;
var
  n, k : Integer;        {Počet vrcholů; Počet střihů}
  c : CutA;              {Popis jednotlivých střihů}

{Načte vstup}
procedure ReadInp;
var
  i, tmp : Integer;
begin
  Write('Zadejte pocet vrcholu a pocet strihu: ');
  Read(n, k);
  Write('Zadejte jednotlive strihy: ');
  {Načte popis střihu}
  for i := 1 to k do begin
    Read(c[i].a, c[i].b);
    {První číslo bude menší}
    if c[i].a > c[i].b then begin
      tmp := c[i].a;
      c[i].a := c[i].b;
```

```
      c[i].b := tmp;
    end;
  end;
  {Přidáme pomocnou tětivu}
  Inc(k);
  c[k].a := 1;
  c[k].b := n;
end;

{Porovná dva střihy}
function CmpCut(a, b : cut) : ShortInt;
begin
  if (a.a < b.a) or ((a.a = b.a) and (a.b > b.b)) then
    CmpCut := -1
  else if (a.a = b.a) and (a.b = b.b) then
    CmpCut := 0
  else
    CmpCut := 1;
end;

{Setřídí pole se střihy QuickSortem}
procedure SortCut(d, u : Integer);
var
  m, tmp : cut;          {Pivot}
  i, j : Integer;
begin
  m := c[(d+u) div 2];  {Vybereme pivota}
  i := d; j := u;
  while i <= j do begin
    {Nalezneme prvky ve špatných částech}
    while CmpCut(c[i], m) = -1 do
      Inc(i);
    while CmpCut(c[j], m) = 1 do
      Dec(j);
    if i <= j then begin
      {Zaměníme prvky ve špatných částech}
      tmp := c[i];
      c[i] := c[j];
      c[j] := tmp;
      Inc(i);
      Dec(j);
    end;
  end;
  if i < u then  {Je co třídit v pravé části?}
    SortCut(i, u);
  if d < j then  {Je co třídit v levé části?}
    SortCut(d, j);
end;

{Nalezne největší mnohoúhelník}
function FindMax(n, d, u : Integer) : Integer;
var
  StackC : Array[1..MAXK] of Integer;
  StackN : Array[1..MAXK] of Integer;
```

```
  AV, SP, AChord, Max : Integer;
begin
  SP := 0;
  AV := c[1].a;
  AChord := 1;
  Max := 0;
  while True do begin
    {Končí tu nějaký mnohoúhelník?}
    while (SP > 0) and (c[StackC[SP]].b = AV) do begin
      if StackN[SP] > Max then {Je největší?}
        Max := StackN[SP];
      Dec(SP);
      if SP > 0 then
        Inc(StackN[SP]);  {Přidáme vrchol za právě ukončenou tětivu}
    end;
    if AV = n then       {Už jsme prošli celý mnohoúhelník?}
      break;
    {Začíná zde nějaký mnohoúhelník?}
    while (AChord <= k) and (AV = c[AChord].a) do begin
      Inc(SP);
      StackC[SP] := AChord;
      StackN[SP] := 1;
      Inc(AChord);
    end;
    {Začíná dříve nějaká tětiva, než končí jiná?}
    if (AChord <= k) and (c[AChord].a < c[StackC[SP]].b) then begin
      Inc(StackN[SP], c[AChord].a - AV);
      AV := c[AChord].a;
    end
    else begin  {Nějaká tětiva nejdříve končí}
      Inc(StackN[SP], c[StackC[SP]].b - AV);
      AV := c[StackC[SP]].b;
    end;
  end;
  FindMax := Max;
end;

begin
  ReadInp;        {Načte vstup}
  SortCut(1, k); {Setřídíme pole se střihy}
  {Nalezne největší část a vypíše ji}
  WriteLn('Nejvetsi cast ma ', FindMax(n, 1, k), ' vrcholu.');
end.
```

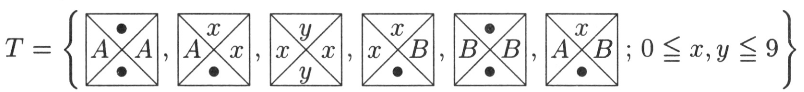
P – II – 4

Využijeme jednoduchého pozorování:

- Posloupnost $x_1, \ldots, x_n$ je symetrická právě tehdy, když $x_1 = x_n$ a posloupnost $x_2, \ldots, x_{n-1}$ je symetrická.
- Jednoprvková i prázdná posloupnost jsou vždy symetrické.

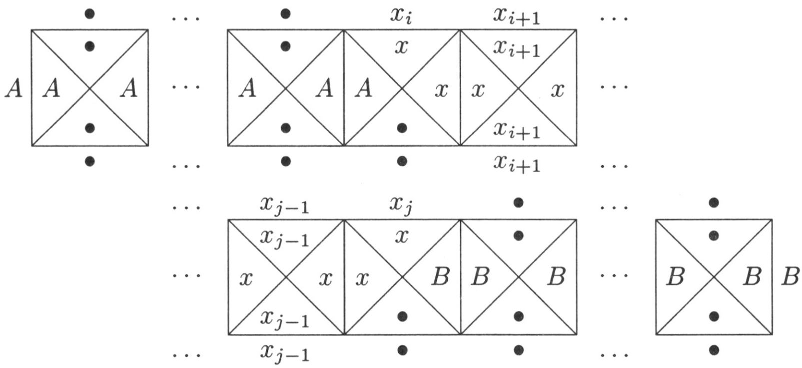
Sestrojíme dlaždicový program, který bude připouštět právě taková vydláždění, v nichž každý řádek ověří rovnost krajních prvků posloupnosti a odstraní je (přepíše na barvu •), přičemž poslední řádek akceptuje buďto prázdnou, nebo jednoprvkovou posloupnost. Takový program odpovídá ANO právě na symetrické vstupní posloupnosti: pokud je posloupnost $x_1, \ldots, x_n$ symetrická, první řádek z ní odstraní x_1 a x_n, druhý x_2 a x_{n-1} atd. až poslední řádek akceptuje buďto prostřední prvek původní posloupnosti (měla-li lichou délku), nebo prázdnou posloupnost (byla-li její délka sudá). A opačně: Pokud program posloupnost akceptuje, pak podle prvního řádku je $x_1 = x_n$, podle druhého $x_2 = x_{n-1}$ atd., tudíž je zadaná posloupnost symetrická. Proto náš program řeší zadanou úlohu se složitostí $O(n)$.

Abychom dosáhli tohoto cíle, použijeme následující sadu dlaždic:

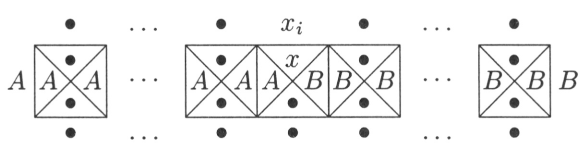


levý okraj zdi barvy A, pravý barvy B a dolní barvy •.

Každý korektně vydlážděný řádek musí vypadat buďto takto:



(to odpovídá kontrole a odstranění krajních prvků posloupnosti), anebo takto:



(takový řádek akceptuje libovolnou jednoprvkovou posloupnost; posloupnost prázdná — taková, jejíž všechny prvky již byly přepsány na • — je akceptována přímo dolním okrajem zdi).

Poznámka. Časové složitosti lepší než lineární již není možno dosáhnout. Idea důkazu: Z definice vydláždění víme, že zeď sudé šířky je možno vydláždit právě tehdy, je-li možno vydláždit její levou polovinu i její pravou polovinu tak, aby v každém řádku měly dlaždice, jimiž se tyto poloviny dotýkají, stejnou barvu společné hrany (těmto hranám budeme říkat prostřední sloupec). Pro každý začátek posloupnosti $x_1, \ldots, x_{n/2}$ ovšem existuje právě jedno doplnění prvky $x_{n/2+1}, \ldots, x_n$ takové, že $x_1, \ldots, x_n$ je symetrická posloupnost. Kdyby nějakým dvěma různým začátkům $x_1, \ldots, x_{n/2}$ a $y_1, \ldots, y_{n/2}$ odpovídalo stejné obarvení středního sloupce, pak by po doplnění prvky $x_{n/2}, \ldots, x_1$ vydláždění pravé poloviny existovalo buď pro oba začátky, nebo pro žádný (závisí totiž pouze na pravé polovině vstupu a středním sloupci), přestože pro první začátek má existovat a pro druhý nikoliv. Proto různých obarvení středního sloupce (těch je $\leqq b^h$, kde b je počet barev vyskytujících se v programu a h maximalní výška zdi, tedy složitost programu) musí být minimálně tolik, kolik je možných začátků posloupnosti ($10^{n/2}$), a tak musí být

$$h \geqq \log_b 10^{n/2} = n \cdot \frac{\log_b 10}{2}.$$

To znamená, že složitost libovolného dlaždicového programu řešícího naši úlohu musí být alespoň lineární.

P – III – 1

Nejdříve je třeba si uvědomit, že jednotlivé souřadnice pozic věží je možno volit nezávisle na sobě. To platí proto, že volba sloupce nijak neomezí rozsah řádek, ve kterém může být věž umístěna. Můžeme tedy nejdříve pro každou věž zvolit sloupec, ve kterém se bude nacházet, a pak pro každou věž zvolit řádek. Převedli jsme tak původní úlohu do jednoho rozměru. Jednotlivé obdélníky se promítnou na intervaly a v každém intervalu chceme nalézt číslo i (umístění věže) takové, aby se čísla žádných dvou intervalů neshodovala.

Čísla budeme hledat následujícím způsobem: Budeme postupně procházet čísla od 1 do N a budeme si udržovat informaci, které intervaly obsahují dané číslo a ještě nemají žádné číslo přiděleno. Z těchto intervalů vybereme ten, který končí nejdříve, a přidělíme mu aktuální číslo. Pokud vybraný interval již neobsahuje přidělované číslo (skončil dříve), úloha nemá řešení. Přímá implementace této myšlenky je samozřejmě možná, ale pomalá ($O(N^2)$). My implementaci zrychlíme následujícím způsobem:

Abychom mohli rychle upravovat informace o tom, které intervaly obsahují dané číslo, setřídíme si je nejdříve vzestupně podle jejich počátku. Abychom byli dále schopni rychle nalézt interval, který končí nejdříve, budeme si intervaly obsahující dané číslo uchovávat v haldě srovnané podle konců intervalů. Přidělování čísel tedy ve vylepšené implementaci probíhá následovně: Procházíme všechna čísla od 1 do N. Pro každé číslo přidáme do haldy všechny intervaly začínající na daném čísle. Pak z haldy (pokud je neprázdná) odebereme minimum (tj. nejdříve končící interval) a přidělíme mu aktuální číslo. Pokud interval již neobsahuje aktuální číslo, řekneme, že požadované rozestavení věží neexistuje. S těmito vylepšeními má algoritmus časovou složitost $O(N \cdot \log N)$ a paměťovou složitost $O(N)$.

Nechť máme nějaké korektní rozestavení věží R. Indukcí dokážeme, že toto rozestavení lze upravit na takové, jaké navrhne náš algoritmus, a tím tedy ukážeme, že náš algoritmus nalezne korektní rozestavení věží.

- ▷ Počátek indukce: Nechť C je nejmenší číslo, které náš algoritmus chce přidělit nějakému (vlastně prvnímu) intervalu I. Z chování našeho algoritmu je zřejmé, že v žádném rozestavení se nižší číslo vyskytovat nemůže. Pokud se v R nevyskytuje ani číslo C, můžeme R upravit tak, že I dáme číslo C. Tím jistě získáme korektní rozestavení, které navíc má pro první interval I přiděleno stejné číslo, jaké by navrhl náš algoritmus. Pokud je v R číslo C již přiděleno nějakému intervalu J, můžeme snadno intervalu J přidělit číslo přidělené intervalu I a intervalu I přiřadit C. Protože interval I končil dříve než interval J, jistě máme opět korektní rozestavení.
- ▷ Indukční krok: Z indukčního předpokladu víme, že existuje rozestavení, které se shoduje s rozestavením navrhovaným naším algoritmem v prvních k číslech. Chceme ukázat, že existuje rozestavení, které se shoduje v prvních $k+1$ číslech. Protože myšlenka důkazu je naprosto stejná jako v počátku indukce, necháváme tuto část důkazu na čtenáři.

Program je přímou implementací algoritmu. Halda je implementována v poli, kde prvek na pozici i má syny na pozicích $2i$ a $2i+1$.

```
program Veze;                    {P-III-1}
const
  MAXN = 100;
type
  {Popis jednoho intervalu}
  Int = record
          s, e : Integer;        {Počátek a konec intervalu}
```

```
          n : Integer;          {Číslo věže, které interval patří}
        end;
  {Popis jednoho bodu}
  Point = array [1..2] of Integer;
  {Popis jednoho obdélníku}
  Rectangle = record
                a, b : Point;   {Levý horní a pravý dolní roh}
              end;
  {Intervaly v jedné souřadnici}
  IntList = array[1..MAXN] of Int;
  CmpProc = function(A, B : Int) : ShortInt;
var
  N : Integer;  {Pocet vezi}
  Rec : Array[1..MAXN] of Rectangle;     {Obdélníky}
  T : Array[1..MAXN] of Point;           {Umístění věží}

{Načte vstup}
procedure ReadInp;
var
  i : Integer;
begin
  Write('Pocet vezi: ');
  Read(N);
  WriteLn('Souradnice obdelniku:');
  for i := 1 to N do
    Read(Rec[i].a[1], Rec[i].a[2], Rec[i].b[1], Rec[i].b[2]);
end;

{Přidá interval do haldy}
procedure AddHeap(var N: Integer; var H: IntList; I: Int; Cmp: CmpProc);
var
  A : Integer;
  Tmp : Int;
begin
  Inc(N);
  H[N] := I;
  A := N;
  while A <> 1 do begin {Nejsme na vrcholu}
    if Cmp(H[A], H[A div 2]) <> -1 then    {Je splněna podmínka haldy?}
      break;
    {Zaměníme prvky, aby byla podmínka splněna}
    Tmp := H[A];
    H[A] := H[A div 2];
    H[A div 2] := Tmp;
    A := A div 2; {Posun o úroveň výše}
  end;
end;

{Odebere z haldy minimum}
procedure
  GetHeapMin(var N: Integer; var H: IntList; Cmp: CmpProc; var Res: Int);
var
  A, M : Integer;
  Tmp : Int;
```

```
begin
  Res := H[1];
  H[1] := H[N];
  Dec(N);
  A := 1;
  while A * 2 < N do begin      {Nejsme na dně?}
    {Nalezneme menšího ze synů}
    if Cmp(H[A*2], H[A*2+1]) = -1 then
      M := A*2
    else
      M := A*2+1;
    if Cmp(H[M], H[A]) <> -1 then  {Podmínka haldy splněna?}
      break;
    {Zaměníme prvky, aby byla podmínka splněna}
    Tmp := H[M];
    H[M] := H[A];
    H[A] := Tmp;
    A := M; {Posun o úroveň níže}
  end;
end;

{Porovná dva intervaly podle počátku}
function CmpInt(a, b : Int) : ShortInt; far;
begin
  if (a.s < b.s) or ((a.s = b.s) and (a.e < b.e)) then
    CmpInt := -1
  else if (a.s = b.s) and (a.e = b.e) then
    CmpInt := 0
  else
    CmpInt := 1;
end;

{Setřídí pole intervalů}
procedure SortInts(var A : IntList);
var
  C, i : Integer;
  H : IntList;
begin
  C := 0;
  for i := 1 to N do
    AddHeap(C, H, A[i], CmpInt);
  for i := 1 to N do
    GetHeapMin(C, H, CmpInt, A[i]);
end;

{Srovná intervaly podle konce}
function CmpBack(a, b : Int) : ShortInt; far;
begin
  if a.e < b.e then
    CmpBack := -1
  else if a.e = b.e then
    CmpBack := 0
  else
    CmpBack := 1;
```

```
end;

{Spočte jednu souřadnici pro každou věž}
function CountCoord(c : Integer) : Boolean;
var
  Ints : IntList;  {Intervaly}
  i, a : Integer;
  ActInts : IntList;    {Halda intervalů k uplatnění}
  ActIntsN : Integer;   {Počet intervalů v haldě}
  Tmp : Int;
begin
  {Vytvoří pole intervalů z obdélníku}
  for i := 1 to N do begin
    Ints[i].s := Rec[i].a[c];
    Ints[i].e := Rec[i].b[c];
    Ints[i].n := i;
  end;
  SortInts(Ints); {Setřídí intervaly}
  {Určí souřadnici věží}
  ActIntsN := 0;
  a := 1;
  for i := 1 to N do begin
    {Přidá do haldy všechny intervaly začínající na aktuální pozici}
    while (a <= N) and (Ints[a].s = i) do begin
      AddHeap(ActIntsN, ActInts, Ints[a], CmpBack);
      Inc(a);
    end;
    if ActIntsN > 0 then begin
      {Vybereme nejdříve končící interval}
      GetHeapMin(ActIntsN, ActInts, CmpBack, Tmp);
      if Tmp.e < i then begin {Už skončil?}
        CountCoord := False;
        Exit;
      end;
      T[Tmp.n][c] := i;
    end;
  end;
  CountCoord := True;
end;

{Vytiskne souřadnice věží}
procedure Print;
var
  i : Integer;
begin
  WriteLn('Souradnice vezi jsou:');
  for i := 1 to N do
    WriteLn(T[i][1], ' ', T[i][2]);
end;

begin
  ReadInp;              {Načte vstup}
  if CountCoord(1) and CountCoord(2) then
                          {Určí souřadnice věží - vešly se?}
```

```
    Print
  else
    WriteLn('Rozmisteni vezi neexistuje.');
end.
```

P – III – 2

Nejprve ukážeme, že pro každé přirozené číslo N existuje přirozené číslo x, jehož ciframi jsou pouze číslice 0 a 1 a které je dělitelné číslem N. Označme $x_1 = 1$, $x_2 = 11$, $x_3 = 111$ atd. Dále označme jako m_i číslo $x_i \bmod N$. Čísla m_i mohou nabývat pouze hodnot od 0 do $N-1$ a proto alespoň dvě z čísel m_1 až m_{N+1} musí být stejná — nechť např. $m_i = m_j$ $(i < j)$. Potom ale číslo $x_j - x_i$ je dělitelné číslem N, neboť $(x_j - x_i) \bmod N = m_j - m_i = 0$. Ciframi čísla $x_j - x_i$ jsou zřejmě pouze číslice 0 a 1 a tedy číslo $x_j - x_i$ splňuje podmínky ze zadání úlohy.

Nadále budeme uvažovat pouze ta čísla, jejichž dekadický zápis je tvořen pouze číslicemi 0 a 1. Předchůdcem čísla x nazveme číslo x div 10, tedy číslo x bez své poslední cifry; naopak následníky čísla x nazveme čísla $10x$ a $10x+1$, tedy ta čísla, která lze vytvořit přidáním jedné cifry na konec čísla x. Číslo x budeme nazývat minimálním číslem pro zbytek z, jestliže $x \bmod N = z$, v dekadickém zápisu čísla x se vyskytují pouze číslice 0 a 1 a x je nejmenší číslo s těmito vlastnostmi. Nyní si dokážeme jednoduché lemma:

Lemma. *Nechť x je minimální číslo pro zbytek z, nechť x' je předchůdcem x; označme $z' = x' \bmod N$. Potom x' je minimální číslo pro zbytek z'.*

Důkaz. Postupujme sporem, tedy předpokládejme, že x' není minimální číslo pro zbytek z', tj. že existuje číslo $x'' < x'$ takové, že $x'' \bmod N = x' \bmod N$. Nechť c je poslední cifra čísla x. Protože $(10x' + c) \bmod N = z$ a $x' \bmod N = x'' \bmod N$, musí nutně platit i $(10x'' + c) \bmod N = z$. Ale potom by číslo $x = 10x' + c$ nemohlo být minimální číslo pro zbytek z, neboť číslo $10x'' + c$ má požadované vlastnosti a je menší — tedy nemůže existovat x'' a tedy x' je minimální číslo pro zbytek z'.

Toto lemma nám dává návod, jak počítat minimální čísla pro různé zbytky. Pro daný zbytek z lze spočítat minimální číslo x tak, že budeme postupně testovat v rostoucím pořadí následníky minimálních čísel pro ostatní zbytky a první následník y nějakého minimálního čísla, pro kterého platí $y \bmod N = z$ je zřejmě hledané minimální číslo pro zbytek z.

Tímto postupem lze vygenerovat minimální čísla pro všechny zbytky, pro které existují: Položme $l_1 = 1$ a pokud už známe $l_1, \ldots, l_k$, položme

l_{k+1} rovno nejmenšímu následníkovi některého z čísel $l_1, \ldots, l_k$ takovému, že $l_{k+1} \bmod N$ je různý od všech $l_i \bmod N$ pro $1 \leqq i \leqq k$. Z předchozích úvah ale vyplývá, že jednotlivá čísla l_i jsou minimální čísla pro zbytky $z_i = l_i \bmod N$. Navíc podle prvního odstavce je jedno z čísel z_i rovno nule.

Náš algoritmus bude pracovat přesně podle výše uvedeného popisu. V poli `fronta` si budeme pamatovat hodnoty z_i a na m-té pozici v poli `predchozi` si budeme pamatovat zbytek předchůdce od minimálního čísla pro zbytek m — z hodnot v tomto poli jsme schopni jednoduše zkonstruovat minimální číslo pro zadaný zbytek. Postupně budeme brát hodnoty z_i z pole `fronta`, spočítáme $(10 * z_i) \bmod N$ a $(10 * z_i + 1) \bmod N$ a pokud jsme dosud nenašli číslo, jehož zbytek by byla jedna z těchto hodnot, tak tento zbytek přidáme na konec pole `fronta` a příslušně zmodifikujeme pole `predchozi`. Časová a paměťová složitost algoritmu je zřejmě $O(N)$.

```
program nulajed; { P-III-2 }
const MAXN=1000;
var N:word; { zadané číslo N }
    predchozi:array[0..MAXN-1] of integer;
     { zbytek předchůdce minimálního čísla s daným zbytkem
  Hodnoty se zvláštním významem:
-2 ... dosud nenalezeno minimální číslo pro tento zbytek
-1 ... nemá předchůdce (číslo 1)
}
    fronta:array[0..MAXN-1] of word;
{ fronta použitá pro generování minimálních čísel }
    ukazatel:word;
{ právě zpracovávaný prvek v poli fronta }
    vefronte:word;
{ počet prvků v poli fronta }
procedure pridej(puvodni,novy:word);
begin { přidá minimální prvek pro zbytek novy do fronty;
  jeho předchůdce je minimální pro zbytek původní }
 if predchozi[novy]<>-2 then exit;
 fronta[vefronte]:=novy;
 inc(vefronte);
 predchozi[novy]:=puvodni;
end;
procedure vypis(p:word);
begin { vypíše číslo se zadaným zbytkem }
 if predchozi[p]>=0 then
  begin
   vypis(predchozi[p]);
   if (10*predchozi[p]) mod N=p then write(0) else write(1)
  end
 else
  write(1)
end;
begin
```

```
 readln(N); { načteme číslo N a inicializujeme pole predchozi }
 for ukazatel:=0 to N-1 do predchozi[ukazatel]:=-2;
 fronta[0]:=1 mod N;
 predchozi[fronta[0]]:=-1;
 ukazatel:=0; vefronte:=1;
 while (predchozi[0]=-2) do
  begin { generujeme čísla s různými zbytky... }
   pridej(fronta[ukazatel],(10*fronta[ukazatel]) mod N);
   pridej(fronta[ukazatel],(10*fronta[ukazatel]+1) mod N);
   inc(ukazatel);
  end;
 vypis(0);  { vypíšeme výsledek a odřádkujeme }
 writeln
end.
```

P – III – 3

Abychom nemuseli vše popisovat zbytečně složitě, zaveďme si jednoduché značení: *Posloupnosti* budou vždy složeny jen z nul a jedniček a všechny budou mít n prvků. Budeme je značit tučnými písmeny a jejich prvky písmeny s indexy, tedy například $\mathbf{x}$ je posloupnost obsahující prvky $x_1, \ldots, x_n$. Počtu jedniček v posloupnosti $\mathbf{x}$ budeme říkat její *váha* a značit $\sharp\mathbf{x}$.

Posloupnost $\mathbf{y}$ je setříděním posloupnosti $\mathbf{x}$ právě tehdy, když platí současně následující dvě podmínky:

1. $y_1 \leqq \ldots \leqq y_n$ (je to neklesající posloupnost)
2. $\sharp\mathbf{y} = \sharp\mathbf{x}$ ($\mathbf{x}$ a $\mathbf{y}$ obsahují stejný počet jedniček a jelikož mají stejnou délku, tak i stejný počet nul, čili se navzájem liší pouze pořadím prvků)

Zabývejme se nejprve tím, jak ověřit druhou podmínku. Mohli bychom postupovat podobně, jako u příkladu v zadání oblastního kola: v každém řádku vydláždění odstranit po jedné jedničce z obou posloupností, a to opakovat tak dlouho, až obě posloupnosti budou obsahovat pouze nuly, což snadno ověříme barvou spodního okraje zdi. Tím úlohu vyřešíme s lineární složitostí, což ovšem, jak si ukážeme, není optimální. U myšlenky postupného „zjednodušování" testovaných posloupností však zůstaneme:

Tvrzení. $\sharp\mathbf{x} = \sharp\mathbf{y} \iff \sharp\mathbf{x} \bmod 2 = \sharp\mathbf{y} \bmod 2 \wedge \sharp\hat{\mathbf{x}} = \sharp\hat{\mathbf{y}}$, *kde* $\hat{\mathbf{x}}$ *je posloupnost, která vznikne z posloupnosti* $\mathbf{x}$ *přepsáním každé druhé jedničky na nulu (to jest přepsáním první jedničky, ponecháním druhé, přepsáním třetí, ponecháním čtvrté atd.).*

Důkaz. Pokud $\sharp\mathbf{x} = \sharp\mathbf{y}$, pak jistě $\sharp\mathbf{x} \bmod 2 = \sharp\mathbf{y} \bmod 2$, ale jelikož $\sharp\hat{\mathbf{x}} = \lfloor \sharp\mathbf{x}/2 \rfloor$ (v $\hat{\mathbf{x}}$ zbyla právě polovina jedniček z $\mathbf{x}$, případná lichá jed-

nička na konci byla odstraněna), musí platit i $\sharp\hat{\mathbf{x}} = \sharp\hat{\mathbf{y}}$. A naopak: pokud $\sharp\hat{\mathbf{x}} = \sharp\hat{\mathbf{y}}$, mohou se posloupnosti $\mathbf{x}$ a $\mathbf{y}$ lišit pouze případnou lichou poslední jedničkou, což ovšem není možné, protože váhy $\sharp\mathbf{x}$ a $\sharp\mathbf{y}$ jsou buďto obě liché, nebo obě sudé.

Vytvořme nejprve dlaždicový podprogram, který pro danou posloupnost $\mathbf{x}$ zadanou barvami horního okraje spočte $\hat{\mathbf{x}}$ na okraji dolním a $\sharp\mathbf{x}$ mod 2 na okraji pravém (v tom smyslu, že vydláždění bude existovat právě tehdy, když barvy těchto okrajů splňují dané podmínky, a toto vydláždění bude mít jediný řádek), předpokládaje, že levý okraj má barvu 0. K tomu nám poslouží následující množina typů dlaždic:

$$T_0 = \left\{ \begin{array}{|c|}\hline 0 \\ \hline 0 \quad 0 \\ \hline 0 \\ \hline \end{array}, \begin{array}{|c|}\hline 1 \\ \hline 0 \quad 1 \\ \hline 0 \\ \hline \end{array}, \begin{array}{|c|}\hline 0 \\ \hline 1 \quad 1 \\ \hline 0 \\ \hline \end{array}, \begin{array}{|c|}\hline 1 \\ \hline 1 \quad 0 \\ \hline 1 \\ \hline \end{array} \right\}$$

Označíme-li si z_i barvu pravé hrany a y_i barvu dolní hrany i-té dlaždice vydlážděného řádku (z_0 budiž barva levé hrany první dlaždice, tedy barva levého okraje zdi, což je 0), snadno ukážeme, že:

▷ $z_i = \left(\sum_{k=1}^{i} x_k\right) \bmod 2$ ... Indukcí: pro $i = 0$ platí, pro $i > 0$ je

$$z_i = (z_{i-1} + x_k) \bmod 2 = \\ = \left(\left(\sum_{k=1}^{i-1} x_k\right) \bmod 2 + x_i\right) \bmod 2 = \left(\sum_{k=1}^{i} x_k\right) \bmod 2.$$

▷ $y_i = 1 \iff x_i = 1 \wedge z_{i-1} = 1$, jinými slovy právě tehdy, je-li x_i jednička, před níž byl lichý počet jedniček, čili jednička sudá — právě ta, která se má objevit v $\hat{\mathbf{x}}$.

Zkombinujme nyní dva takové programy tak, aby pracovaly současně: jeden s posloupností $\mathbf{x}$ a druhý s $\mathbf{y}$ (posloupnosti jsou kódovány dvojicemi (x_i, y_i)), počítaly stejně kódované posloupnosti $\hat{\mathbf{x}}$ a $\hat{\mathbf{y}}$ a rovněž zbytky $\sharp\mathbf{x}$ mod 2 a $\sharp\mathbf{y}$ mod 2. K tomu nám stačí sestrojit množinu T_1 obsahující „součiny“ dvojic dlaždic z množiny T_0, to znamená pro každé dvě dlaždice

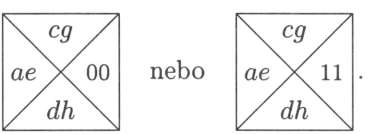
$$A = \begin{array}{|c|}\hline c \\ \hline a \quad b \\ \hline d \\ \hline \end{array} \quad \text{a} \quad B = \begin{array}{|c|}\hline g \\ \hline e \quad f \\ \hline h \\ \hline \end{array}$$

z T_0 do T_1 přidat dlaždici

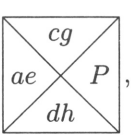
$$\begin{array}{|c|}\hline cg \\ \hline ae \quad bf \\ \hline dh \\ \hline \end{array},$$

kde ae, bf, cg a dh jsou uspořádané dvojice barev. Existuje-li vydláždění řádku, který má na horní hraně dvojice $(x_1, y_1), \ldots, (x_n, y_n)$, na dolní hraně $(\hat{x}_1, \hat{y}_1), \ldots, (\hat{x}_n, \hat{y}_n)$, na levé $(0, 0)$ a na pravé (z_x, z_y), dlaždicemi typů z množiny T_1, musí díky tomu, jak jsme si tyto typy nadefinovali, existovat i vydláždění řádku s horní hranou $x_1, \ldots, x_n$, dolní $\hat{x}_1, \ldots, \hat{x}_n$, levou 0 a pravou z_x, jakož i řádku s horní hranou $y_1, \ldots, y_n$, dolní $\hat{y}_1, \ldots \hat{y}_n$, levou 0 a pravou z_y dlaždicemi typů T_0. A opačně — existují-li tato dvě vydláždění, existuje i jejich složení sestavené z typů z množiny T_1. Proto $z_x = \sharp\mathbf{x} \bmod 2$, $z_y = \sharp\mathbf{y} \bmod 2$ a spodní hrana opravdu obsahuje dvojice prvků posloupností $\hat{\mathbf{x}}$ a $\hat{\mathbf{y}}$.

My ovšem potřebujeme akceptovat právě ta vydláždění, u nichž $z_x = z_y$, tedy s pravým okrajem jak $(0, 0)$, tak $(1, 1)$, ale máme k dispozici pouze jedinou barvu pravého okraje zdi. Proto rozšíříme množinu T_1 na T_2 tak, že ke každému typu dlaždice z T_1 tvaru



přidáme do T_2 typ



kde P je barva pravého okraje, která se neshoduje s žádnou jinou barvou. Jelikož se tato dlaždice může vyskytnout výhradně těsně u pravého okraje (žádná dlaždice totiž nemá levou hranu barvy P, na kterou bychom mohli navázat), odpovídají korektní vydláždění pomocí této rozšířené sady dlaždic právě těm vydlážděním pomocí původní sady, u nichž je $z_x = z_y$.

Zbývá nám ještě přidat test vzestupnosti posloupnosti $\mathbf{y}$, což vyřešíme přidáním dlaždic typů

$$T_z = \left\{ \begin{array}{c} \boxed{\begin{array}{cc} \mathbf{x0} & \\ 0 & 0 \\ & x0 \end{array}}, \boxed{\begin{array}{cc} \mathbf{x1} & \\ 0 & 1 \\ & x1 \end{array}}, \boxed{\begin{array}{cc} \mathbf{x1} & \\ 1 & 1 \\ & x1 \end{array}}, \boxed{\begin{array}{cc} \mathbf{x0} & \\ 0 & P \\ & x0 \end{array}}, \boxed{\begin{array}{cc} \mathbf{x1} & \\ 0 & P \\ & x1 \end{array}}, \boxed{\begin{array}{cc} \mathbf{x1} & \\ 1 & P \\ & x1 \end{array}}; \\ x \in \{0,1\}, \mathbf{x} = \text{„tučná verze“ } x \end{array} \right\},$$

kde **00**, **01**, **10** a **11** jsou barvy dvojic kódujících zadané posloupnosti $\mathbf{x}$ a $\mathbf{y}$ a 00, 01, 10 a 11 analogické kódy používané všemi ostatními dosud

nadefinovanými dlaždicemi. Tím z T_2 vznikne množina typů T, o které tvrdíme, že použita s barvou levého okraje 0, barvou pravého P a barvou dolního okraje 00, řeší zadanou úlohu se složitostí $O(\log n)$, což také ihned dokážeme:

- První řádek vydláždění musí obsahovat výhradně dlaždice typů T_z, protože žádné jiné neobsahují na svých horních hranách barvy, jimiž je kódován vstup. Jak jsme již ukázali v příkladu v zadání domácího kola, typy T_z zajišťují neklesání posloupnosti $\mathbf{y}$. Navíc spodní okraj tohoto řádku předává níže obě vstupní posloupnosti, pouze jinak kódované.
- Všechny ostatní řádky obsahují pouze dlaždice typů T_2, přičemž každý řádek přepíše posloupnosti $\mathbf{x}_i$ a $\mathbf{y}_i$ zadané na svém horním okraji na $\mathbf{x}_{i+1} = \hat{\mathbf{x}}_i$ a $\mathbf{y}_{i+1} = \hat{\mathbf{y}}_i$ a ověří, zda $\sharp\mathbf{x}_i \bmod 2 = \sharp\mathbf{y}_i \bmod 2$. Pokud $\sharp\mathbf{x} = \sharp\mathbf{y}$, pak po maximálně $\lceil \log_2 n \rceil$ takových řádcích budou obě posloupnosti zredukovány na samé nuly, což odpovídá barvě dolního řádku, čili vydláždění celé zdi existuje a má hloubku $\leqq 1 + \lceil \log_2 n \rceil$; pokud $\sharp\mathbf{x} \neq \sharp\mathbf{y}$, vydláždění nemůže existovat, protože alespoň v jednom kroku by $\sharp\mathbf{x}_i \bmod 2$ nebylo rovno $\sharp\mathbf{y}_i \bmod 2$ (viz Tvrzení výše).

Poznámka. Lepší hloubky než $\Omega(\log n)$ již není možno dosáhnout. To můžeme zdůvodnit podobně, jako jsme v předchozím kole dokazovali, že pro ověření symetrie posloupnosti potřebujeme hloubku minimálně lineární. Opět budeme počítat možná obarvení středního sloupce — tentokráte si uvědomíme, že tato obarvení musí být rozdílná pro každé dva různé počty jedniček v $x_1, \ldots, x_{n/2}$ — pokud jsou jedničky pouze mezi těmito prvky a $x_{n/2+1}, \ldots, x_n$ jsou všechny nulové, musí setříděná posloupnost $\mathbf{y}$ obsahovat naopak ve své levé polovině samé nuly a v polovině pravé stejný počet jedniček, jako má $\mathbf{x}$ v polovině levé. Možných počtů jedniček mezi $x_1, \ldots, x_{n/2}$ je $n/2+1$, možných obarvení středního sloupce $\leqq b^h$, kde b je počet barev použitých v našem dlaždicovém programu (to je nějaká konstanta) a h je jeho složitost = výška středního sloupce. Z toho dostaneme:

$$b^h \geqq n/2 + 1 \implies b^h > n/2 \implies h > \log_b n - \log_b 2 \implies h = \Omega(n).$$

P – III – 4

Řešení této úlohy je založeno na dynamickém programování. Pro každé slovo si budeme pamatovat, kde nejlépe zalomit předchozí řádek, když

toto slovo bude na konci řádku. Také si budeme pamatovat celkový počet trestných bodů textu při tomto zalomení. Když spočítáme nejlepší zalomení pro poslední slovo, můžeme ze zapamatovaných informací snadno zrekonstruovat, jak celý text zalámat. Z informací u posledního slova zjistíme, za kterým slovem měl být zalomen předposlední řádek. Z informací u posledního slova na předposledním řádku zjistíme, kde měl být zalomen předpředposlední řádek atd. Když víme, kde měly být jednotlivé řádky zalámány, stačí již jen správně doplnit mezi vypisovaná slova mezery. To uděláme tak, že pokud máme umístit M mezer mezi $S+1$ slov, tak vytvoříme $M \bmod S$ oddělení slov s $(M \operatorname{div} S)+1$ mezerami a $S-(M \bmod S)$ oddělení s $M \operatorname{div} S$ mezerami. Tím se zjevně žádná dvě oddělení slov neliší o více jak jednu mezeru a počet mezer na řádce je také správný. Formátování posledního řádku a řádku s jedním slovem jsou triviální.

A nyní již k zajímavé části algoritmu. Jak rychle spočítat, kde nejlépe zalomit předchozí řádek, když bude aktuální slovo na konci řádku? Tuto informaci budeme postupně počítat od prvního slova. U prvního slova nastavíme počet trestných bodů i místo zalomení na nula. Když máme spočteny hodnoty pro prvních N slov, začneme je počítat pro slovo $N+1$-ní. Jdeme od N-tého slova směrem k začátku textu. Pro každé slovo si spočteme počet trestných bodů za řádek, který začíná za ním a končí $N+1$-vým slovem (pokud je $N+1$-ní slovo posledním v textu, bude jím ukončovaná řádka řádkou poslední, a tak hodnotící funkci informujeme, že hodnotí poslední řádku). K tomuto počtu trestných bodů ještě přičteme trestné body za předchozí text (ty máme již spočteny a uloženy u slova, za kterým jsme se rozhodli řádek zlomit) a zjistíme, zda jsme získali pro $N+1$-ní slovo text s menším počtem trestných bodů. Pokud ano, zapamatujeme si počet bodů pro tento text a místo zalomení. Když už je poslední řádek moc dlouhý (hodnotící funkce nám vrátila ∞), tak víme, že už lepší zalomení nenajdeme. Prošli jsme už totiž všechna možná zalomení předposledního řádku a vybrali z nich to nejlepší. Můžeme tedy počítat hodnoty pro další slovo. Algoritmus má časovou složitost $O(T+W \cdot L)$, kde T je délka textu, W je počet slov a L je délka řádku. Paměťová složitost algoritmu je $O(T)$. Správnost algoritmu plyne z popisu.

Program je přímou implementací algoritmu:

```
#include <stdio.h>
#include <stdlib.h>

#define INPUT "format.in"
```

```
#define OUTPUT "format.out"

#define TEXTLEN 10000 /* Maximalni delka textu */
#define MAXWORDS 5000 /* Maximalni pocet slov v textu */
#define MAXLINELEN 100 /* Maximalni delka vstupni radky */
#define MAXLINES 5000 /* Maximalni pocet radek ve vystupu */

#define LINEPENALTY 10 /* Cena radky */
#define LASTLINESMALLPEN 3 /* Cena za maly posledni radek */
#define SINGLEWORDPEN 20 /* Penalta za jedno slovo na radku */
#define INFTYPEN 30000 /* Nekonecna penalta */

typedef unsigned long price_t;

int Width; /* Sirka radku */
char Text[TEXTLEN]; /* Text nacteny ze souboru */
int Words[MAXWORDS]; /* Text rozsekany na slova */
int WCnt; /* Pocet slov */
price_t WrapPrice[MAXWORDS];
                /* Ohodnoceni textu, pokud zalomime za timto slovem */
int WrapPos[MAXWORDS];
        /* Pozice, kde jsme zalomili, kdyz jsme pridavali toto slovo */

/* Ohodnoti radek */
int LinePrice(int WordLen, int Words, int Last)
{
  int Pts = Width - WordLen - Words + 1;
  int Base = LINEPENALTY;

  if (Pts < 0)
    return INFTYPEN;
  if (Last)
    return LINEPENALTY + (((WordLen+Words-1)*4 < Width) ?
           LASTLINESMALLPEN : 0);
  if (Words == 1)
    Base += SINGLEWORDPEN;
  return Pts * Pts + Base;
}

/* Nacte vstup a rozdeli ho na slova */
void ProcessInput(void)
{
  FILE *In;
  char Buf[MAXLINELEN]; /* Buffer na radku */
  int i, TPos = 0, WPos = 0;    /* ; Pozice v bufferu na text ;
                                   Cislo aktualniho slova */
  int SWPos = 0; /* Pozice pocatku slova */

  if (!(In = fopen(INPUT, "r")))
  {
    puts("Can't open input file.");
    exit(1);
  }
  /* Nacteme delku radku */
```

```
  fgets(Buf, MAXLINELEN, In);
  sscanf(Buf, "%d", &Width);

  while (fgets(Buf, MAXLINELEN, In))
  {
    for (i = 0; Buf[i] != '\0'; i++, TPos++)
    {
      if (Buf[i] == '\n')
        Buf[i] = ' ';
      Text[TPos] = Buf[i];
      if (Buf[i] == ' ')
      {
        Words[WPos++] = TPos - SWPos;
        SWPos = TPos + 1;
      }
    }
  }
  fclose(In);
  WCnt = WPos;
}

void FindBestSep(void)
{
  int i, j;
  price_t Min, Price;
       /* Minimalni dosazene ohodnoceni; Cena aktualniho zlomu */
  int WordsLen, MinPos;
       /* Celkova delka slov na radce; Pozice minimalniho zalomeni  */

  WrapPrice[0] = 0;
  WrapPos[0] = 0;

  for (i = 0; i < WCnt; i++) /* Postupne pridavame jednotliva slova */
  {
    Min = INFTYPEN;
    MinPos = 0;
    WordsLen = 0;
    for (j = i; j >= 0; j--) /* Vyzkousime vsechna mozna zalomeni */
    {
      WordsLen += Words[j];
      Price = LinePrice(WordsLen, i - j + 1, i == WCnt - 1);
      if (Price == INFTYPEN) /* Uz jsem prekrocili velikost radku? */
        break;
      Price += WrapPrice[j];
      if (Price < Min)
      {
        Min = Price;
        MinPos = j;
      }
    }
    WrapPrice[i+1] = Min;
    WrapPos[i+1] = MinPos;
  }
}
```

```
/* Vytiskne dalsi slovo */
void PrintWord(FILE *Out)
{
  static int WPos = 0; /* Pozice slova k vypsani */
  int WStart;

  for (WStart = WPos; Text[WPos] != ' '; WPos++);
  fwrite(Text + WStart, 1, WPos - WStart, Out);
  WPos++;
}

/* Vytiskne radku */
void PrintLine(FILE *Out, int First, int Cnt)
{
  int i, j;
  int Len = 0, TotSpc;    /* Pocet znaku na radce; Celkovy pocet mezer */
  int Spc; /* Pocet mezer v aktualni mezere */

  /* Spocteme pocet znaku ve slovech */
  for (i = 0; i < Cnt; i++)
    Len += Words[First+i];
  TotSpc = Width - Len;
  if (Cnt == 1) /* Jedno slovo? */
  {
    for (j = 0; j < TotSpc; j++)
      fputc(' ', Out);
  }
  else
    for (i = 0; i < Cnt-1; i++)
    {
      PrintWord(Out); /* Vytiskne dalsi slovo */
      /* Spocteme a vytiskneme potrebny pocet mezer */
      Spc = TotSpc / (Cnt - 1) + (i < TotSpc % (Cnt-1));
      for (j = 0; j < Spc; j++)
        fputc(' ', Out);
    }
  PrintWord(Out); /* Jeste vytiskneme posledni slovo */
  fputc('\n', Out);
}

/* Vypiseme nejlepsi vysledek */
void PrintBestText(void)
{
  FILE *Out;
  int Lines = 0, ActWord;
                /* Pocet radek vysledneho textu; Aktualni slovo */
  int LB[MAXLINES], i; /* Pozice jednotlivych zalomeni */

  /* Zjistime pozice jednotlivych zalomeni */
  for (ActWord = WCnt; ActWord > 0; Lines++, ActWord = WrapPos[ActWord])
    LB[Lines] = ActWord;
  LB[Lines] = 0;
  if (!(Out = fopen(OUTPUT, "w")))
```

```
  {
    puts("Can't open output file.");
    exit(1);
  }
  /* Nechame vytisknout radku */
  for (i = Lines - 1; i > 0; i--)
    PrintLine(Out, LB[i+1], LB[i] - LB[i+1]);

  /* Ted jeste vytiskneme posledni radku */
  for (i = LB[1]; i < LB[0] - 1; i++)
  {
    PrintWord(Out);
    fputc(' ', Out);
  }
  PrintWord(Out);
  fputc('\n', Out);

  fclose(Out);
}

int main(void)
{
  ProcessInput(); /* Nacte vstup a rozdeli ho na slova */
  FindBestSep(); /* Nalezneme nejlepsi rozdeleni na radky */
  PrintBestText(); /* Vypise text podle spoctenych zalomeni */
  return 0;
}
```

P – III – 5

Nejprve si rozmysleme, že platí následující tvrzení: Trasy linek městem lze navrhnout právě tehdy, pokud ze všech křižovatek vychází sudý počet ulic. Kdybychom si trasy jednotlivých linek vyznačili různými barvami v plánu města, potom by každá z nich tvořila cyklus. Každá ulice by měla jednoznačně určenu svou barvu. Pokud si trasy linek zakreslíme do mapy, bude z každé křižovatky vycházet buď žádná nebo právě dvě ulice od jedné určité barvy — žádná, pokud trasa příslušné barvy křižovatkou neprochází, a dvě, pokud ano; více ulic stejné barvy nemůže z křižovatky vycházet, neboť každá linka projíždí křižovatkou nejvýše jedenkrát. Obarvení ulic v mapě tedy „páruje“ ulice vycházející z křižovatky a tudíž počet ulic vycházejících z jedné křižovatky musí být sudý.

Nyní si naopak rozmysleme, že pokud z každé křižovatky vychází sudý počet ulic, potom lze navrhnout trasy linek tak, aby splňovaly požadavky zadání úlohy. Postupně obarvujme ulice ve městě tak, aby ulice stejné barvy tvořily cyklus (tj. odpovídaly nějaké autobusové lince). Uli-

ce, které jsme již obarvili, nebudeme nadále považovat za součást města; tím zmenšíme počet ulic vycházejících z jedné křižovatky o sudé číslo (o nulu nebo o dvě), takže počet ulic vycházejících z každé křižovatky bude stále sudý. Vyberme si nějakou křižovatku ve městě a označme si ji na mapě (položme do ní kamínek); vydejme se z této křižovatky po libovolné (dosud neobarvené) cestě a položme do křižovatky, do které jsme dorazili, kamínek. Z každé křižovatky lze vždy pokračovat alespoň jednou ulicí — do křižovatky jsme po jedné ulici přišli, a protože počet neobarvených ulic, které z ní vedou, je sudý, musí z ní vést tedy alespoň dvě neobarvené ulice — tou druhou můžeme pokračovat. Skončíme, pokud bychom na nějakou křižovatku měli položit druhý kamínek — tehdy jsme našli cyklus z ulic a tento cyklus obarvíme nějakou dosud nepoužitou barvou (a prohlásíme ho za novou autobusovou linku). Kamínky odstraníme z mapy a celý proces opakujeme tak dlouho, dokud mapa města obsahuje nějaké neobarvené ulice.

Předchozí důkaz nám dává návod k vytvoření algoritmu, který řeší zadanou úlohu. Nejprve ověříme, zda z některé křižovatky vychází lichý počet ulic; je-li tomu tak, potom rovnou vypíšeme „Nelze“. V opačném případě začneme aplikovat postup z minulého odstavce; kamínky samozřejmě nahradíme nastavováním vhodného příznaku v programu. Pokud nalezneme cyklus, příslušné ulice z mapy rovnou vymažeme a cyklus vypíšeme na výstup. Abychom ušetřili čas, ponecháme v mapě „kamínky“ na křižovatkách, které jsou mezi výchozí křižovatkou a křižovatkou, kam jsme měli položit dva kamínky; na tyto křižovatky bychom mohli položit kamínky i ve městě, ve kterém by byl vynechán právě nalezený cyklus. Pokládání kamínků budeme realizovat jednoduchou rekurzivní funkcí. Zbývá vyřešit, jak právě obarvené ulice rychle odstraňovat z mapy města uložené v paměti počítače. Ulice spojující dvě stejné křižovatky si budeme pamatovat jako jednu ulici s uvedením počtu ulic, které vedou paralelně s touto ulicí (tj. spojují dvě stejné křižovatky). Ulice si uložíme do dvojrozměrného pole; jeden jeho index bude představovat číslo křižovatky a jeho druhý index bude představovat pořadové číslo ulice vycházející z dané křižovatky. Rozměry tohoto pole tedy budou počet křižovatek × maximální počet ulic vycházejících z jedné křižovatky. U každé křižovatky jsou uvedeny všechny ulice, které z ní vycházejí. U každé ulice si pamatujeme, kam vede, kolik ulic s ní vede paralelně a index do pole určující, kde jsou informace o této ulici rovněž uloženy u křižovatky na jejím opačném konci. Příslušnou ulici vymažeme jednoduše tak, že upravíme informace o ní u obou křižovatek, které spojuje, což

lze provést v konstantním čase, neboť si pamatujeme její index u druhé křižovatky.

Paměťové i časové nároky našeho algoritmu jsou $O(N+M)$, kde N je počet křižovatek ve městě a M je počet ulic ve městě. Paměťové nároky algoritmu lze reprezentací pouze jedné z paralelních hran (viz minulý odstavec) snížit na $O(N+H)$, kde H je maximální počet ulic, z nichž žádné dvě nejsou paralelní.