51. 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); Jaroslav Zhouf (editor): 51. ročník matematické olympiády na středních školách. Zpráva o řešení úloh ze soutěže konané ve školním roce 2001/2002. 43. mezinárodní matematická olympiáda. 14. mezinárodní olympiáda v informatice. (Czech). Praha: Jednota českých matematiků a fyziků, 2003. pp. 96–137.

Persistent URL: http://dml.cz/dmlcz/405046

P – I – 1

Matice

Je dána čtvercová matice A o rozměrech $n \times n$, jejíž prvky jsou nuly a jedničky. Pojmem *řádková* resp. *sloupcová výměna* budeme označovat vzájemnou záměnu libovolných dvou řádků, resp. libovolných dvou sloupců matice. Bude nás zajímat, zda je možné posloupností řádkových a sloupcových výměn převést danou matici A na takovou matici, která má na hlavní diagonále pouze jedničky (hlavní diagonálu tvoří prvky $A[1,1], A[2,2], \ldots, A[n,n]$).

V mnoha aplikacích (např. při řešení soustav lineárních rovnic) je výhodné danou matici transformovat na ekvivalentní matici tak, aby hlavní diagonála obsahovala „velké" prvky. V této úloze vlastně nuly reprezentují „malé" a jedničky „velké" prvky matice.

Soutěžní úloha. Navrhněte co nejefektivnější algoritmus, který převede zadanou čtvercovou matici pomocí řádkových a sloupcových výměn na matici, která má na hlavní diagonále samé jedničky, příp. zjistí, že to není možné.

Příklad 1: Matice A:

```
0 1 0 1
1 1 0 1
1 1 0 0
0 0 1 0
```

Výsledek:

```
1 1 0 0
1 1 0 1
0 0 1 0
0 1 0 1
```

Matici je možné transformovat — stačí vyměnit třetí a čtvrtý řádek a potom první a čtvrtý sloupec.

Příklad 2: Matice A:

```
0 1 0
1 1 0
1 1 0
```

Výsledek:

Matici není možné transformovat do požadovaného tvaru.

P – I – 2

Robot

Ve výzkumném ústavu potrubí a rour vyvinuli nový druh robota určeného na čištění teplovodních potrubí. Robot se soustavou potrubí pohybuje podle předem stanoveného plánu. Technologie čištění vyžaduje, aby robot prošel každou trubkou v soustavě dvakrát (nezáleží na tom, jakým směrem) — při prvním průchodu provádí chemické čištění a při druhém mechanické dočišťování. Robot může vlézt do trubky z kteréhokoliv konce, ale jakmile do ní vstoupí, musí ji už celou projít.

Soustava potrubí se skládá z n uzlů očíslovaných od 1 do n, mezi nimiž vede m teplovodních trubek očíslovaných od 1 do m. Každá trubka vede mezi dvojicí uzlů a nemá žádné odbočky ani větvení. Mezi každou dvojicí uzlů vede nejvýše jedna trubka. Soustava potrubí je souvislá, tj. robot se trubkami může dostat na libovolné místo soustavy. Robot začíná svou práci v uzlu číslo 1 a po skončení čištění se musí opět do tohoto uzlu vrátit.

Soutěžní úloha. Napište program, který umožní plánovat trasu čisticího robota. Program přečte ze vstupního souboru popis soustavy potrubí a zjistí, zda existuje trasa robota, která začíná i končí v uzlu číslo 1 a prochází každou trubkou právě dvakrát. Pokud taková trasa existuje, program ji vypíše.

Formát vstupu: První řádek vstupního souboru `robot.in` obsahuje dvě kladná celá čísla n a m ($n \leqq 100$), oddělená jednou mezerou. Každý z následujících m řádků popisuje jednu trubku. Obsahuje vždy dvě kladná celá čísla oddělená mezerou — čísla koncových uzlů této trubky.

Příklad:

`robot.in`

```
5 6
1 3
1 4
1 5
2 4
3 5
4 5
```

`robot.out`

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

Formát výstupu: Výstupní soubor `robot.out` je tvořen jediným řádkem, který obsahuje $2m+1$ čísel uzlů oddělených mezerami. Jsou to čísla uzlů v pořadí, v jakém je má robot navštívit během svého čištění. První a poslední číslo na řádku musí být 1. V případě, že neexistuje trasa, která prochází každou trubkou právě dvakrát, výstupní soubor bude obsaho-

vat jediný řádek se slovem NE. Pokud existuje více vhodných tras, vypište jednu libovolnou z nich.

P – I – 3

Přímka

V rovině je dáno $2n$ bodů, z toho n je bílých a n černých. *Spravedlivá přímka* je taková přímka, která

- ▷ prochází bodem $[0,0]$,
- ▷ neprochází žádným černým ani bílým bodem,
- ▷ rozděluje rovinu na dvě poloroviny, přičemž v jedné z těchto polorovin je stejný počet bílých bodů jako ve druhé polorovině černých, a naopak.

Soutěžní úloha. Napište program, který ze vstupního souboru `primka.in` přečte souřadnice bílých a černých bodů a do souboru `primka.out` vypíše jednu spravedlivou přímku.

Můžete předpokládat, že žádné tři zadané body neleží na jedné přímce a že bod $[0,0]$ neleží na žádné přímce určené dvěma zadanými body. Všechny zadané body mají celočíselné souřadnice.

Formát vstupu: První řádek vstupního souboru `primka.in` obsahuje jediné číslo n. Každý z následujících $2n$ řádků obsahuje souřadnice jednoho zadaného bodu oddělené mezerou. Prvních n bodů je bílých, dalších n bodů je černých.

Formát výstupu: Výstupní soubor `primka.out` je tvořen jediným řádkem, který obsahuje souřadnice jednoho libovolného bodu různého od $[0,0]$, jímž nalezená spravedlivá přímka prochází. Souřadnice jsou od sebe odděleny jednou mezerou. Pokud spravedlivá přímka pro zadanou množinu bodů neexistuje, výstupní soubor bude obsahovat jediný řádek se slovem NE.

Příklad: `primka.in`

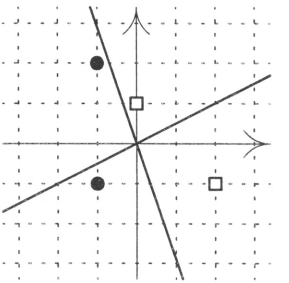
```
2
0 1
2 -1
-1 -1
-1 2
```

`primka.out`

```
2 1
```

Jiné správné řešení:

```
-1.0 2.9
```



Návod. Nechť $A_1 = [x_1, y_1]$, $A_2 = [x_2, y_2]$ a $A_3 = [x_3, y_3]$ jsou body v rovině. Jestliže je hodnota výrazu $(x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1)$ kladná, bod A_3 leží nalevo od polopřímky A_1A_2. Jestliže je tato hodnota záporná, potom leží napravo, a pokud je tato hodnota nulová, bod A_3 leží na přímce A_1A_2.

P – I – 4

Komparátorové sítě

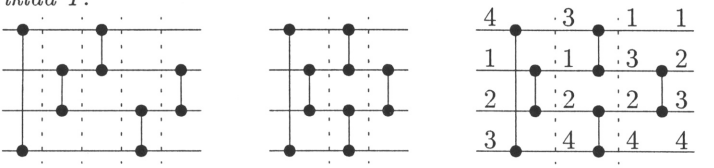
Komparátorové sítě se využívají při návrhu paralelních algoritmů. Mohou se také snadno realizovat pomocí elektronických obvodů. *Komparátor* je jednoduché zařízení, které obdrží na vstupu dvě čísla, porovná je, na vrchním ze svých výstupů vrátí menší z těchto dvou čísel a na spodním větší z nich. Z komparátorů lze sestavovat složitější obvody, kterým budeme říkat komparátorové sítě.

Komparátorová síť je tvořena n vodorovně uspořádanými *vodiči*, které jsou na několika místech propojeny pomocí komparátorů. Komparátory jsou uspořádány do *vrstev*, které odpovídají jednotlivým krokům výpočtu. Na začátku výpočtu (v kroku 0) síť dostane na vstupu n čísel. Po skončení kroku $k - 1$ se výstupy z kroku $k - 1$ přenesou vodiči na komparátory ve vrstvě k. Komparátor ve vrstvě k spojuje vždy dva vodiče (nemusí to ovšem být sousední vodiče). Jestliže je na spodním z nich menší hodnota než na vrchním, vymění tyto hodnoty, v opačném případě je nechá beze změny. V jedné vrstvě může být umístěno i více komparátorů (jejich výpočet probíhá najednou, paralelně), ale v jedné vrstvě může jeden vodič vstupovat nejvýše do jednoho komparátoru. Po skončení celého výpočtu jsou na výstupech sítě tatáž čísla jako na jejích vstupech, pouze může být zaměněno jejich pořadí.

Graficky se vodiče zobrazují jako vodorovné čáry, komparátory jako svislé spojnice svých vstupních vodičů. Komparátory umístěné v jedné vrstvě se kreslí svisle nad sebe, případně do několika sousedních sloupců. Jednotlivé vrstvy sítě oddělujeme svislou čárkovanou čarou. Výpočet komparátorové sítě probíhá zleva doprava.

Při návrhu sítí se snažíme sestrojit je tak, aby doba výpočtu byla co nejmenší, tj. aby síť měla co nejméně vrstev. Druhým kritériem hodnocení kvality sítě je počet použitých komparátorů (na tomto počtu může záviset například výrobní cena sítě).

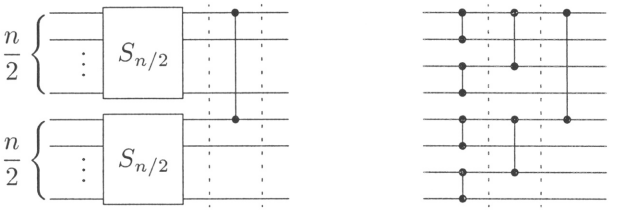
Příklad 1:



Uvažujme nejlevější síť na obrázku. Tato síť dostane čtyři vstupy a vrátí je uspořádané od nejmenšího k největšímu. Po prvních dvou krocích výpočtu bude nejmenší vstup buď na prvním, nebo na druhém vodiči a největší vstup na třetím nebo na čtvrtém. Další dva kroky umístí nejmenší a největší hodnotu na své místo a v posledním kroku se správně uspořádají hodnoty na druhém a třetím vodiči. Všimněte si, že první a druhý komparátor (stejně jako třetí a čtvrtý) je možné sloučit do jedné vrstvy, aniž by se tím změnil výsledek výpočtu. Výsledná rychlejší síť je na prostředním obrázku. Pravý obrázek ukazuje průběh výpočtu pro vstup 4, 1, 2, 3.

Příklad 2: Sestrojte komparátorovou síť, která na vstupu obdrží n čísel a na výstupu umístí nejmenší z těchto čísel na první vodič. Na pořadí ostatních čísel nezáleží. Můžete předpokládat, že n je mocnina dvou.

Řešení. Síť sestrojíme rekurzívně. Označme S_n síť, která úlohu řeší pro n vstupů. Jestliže $n = 1$, S_n nebude obsahovat žádný komparátor, neboť máme jen jediný vstup. Předpokládejme tedy, že $n > 1$. Vstupy rozdělíme na dvě poloviny — horní a dolní. Na každou polovinu vstupů aplikujeme síť $S_{n/2}$. Tyto dvě sítě poloviční velikosti mohou pracovat paralelně. Po skončení jejich výpočtu budeme mít na vodiči číslo 1 nejmenší hodnotu z horní poloviny vstupů a na vodiči číslo $\frac{1}{2}n+1$ nejmenší hodnotu z dolní poloviny. Nyní proto stačí přidat jeden komparátor mezi vodiče 1 a $\frac{1}{2}n+1$ a dostaneme celkové minimum na prvním vodiči. Síť S_n se tedy skládá ze dvou sítí $S_{n/2}$ a z jednoho dalšího komparátoru. Konstrukce sítě S_n je znázorněna na následujícím obrázku vlevo, vpravo je příklad výsledné sítě pro $n = 8$.



Všimněte si, že hloubka rekurze je $\log_2 n$, neboť velikost vstupu se v každém rekurzívním kroku sníží na polovinu. Každá úroveň rekurze přidá do výsledné sítě jednu vrstvu, takže síť S_n má $\log_2 n$ vrstev. Počet použitých komparátorů je v poslední vrstvě 1, v každé další vrstvě odzadu se vždy zdvojnásobí. Nechť počet vstupů je $n = 2^k$. Potom počet použitých komparátorů je $1 + 2 + 4 + \ldots + 2^{k-1} = 2^k - 1 = n - 1$ (součet geometrické posloupnosti). Získali jsme tedy síť, která má $O(\log n)$ vrstev a používá $O(n)$ komparátorů.

Soutěžní úlohy. a) Napište program, který bude simulovat výpočet komparátorové sítě. Program dostane ve vstupním souboru popis zkoumané komparátorové sítě a hodnoty jejích vstupů. Přesný tvar popisu komparátorové sítě si sami navrhněte a popište ho. Program by měl spočítat výsledky výpočtu komparátorové sítě a také by měl umožňovat graficky nebo semigraficky zobrazovat průběh výpočtu sítě. V řešení uveďte také příklady vstupu a výstupu svého programu.

b) Na vstupu je dáno $n-1$ čísel seřazených od nejmenšího po největší (prvních $n-1$ vstupů). Poslední vstup obsahuje libovolné číslo. Navrhněte komparátorovou síť, která zatřídí poslední číslo do této posloupnosti na správné místo podle velikosti (tj. na výstupu bude všech n čísel uspořádáno od nejmenšího k největšímu). Můžete předpokládat, že n je mocnina dvou. Pokuste se navrhnout co nejrychlejší síť.

P – II – 1

Matice

Uvažujme matici A o rozměrech $m \times n$, která obsahuje pouze nuly a jedničky. *Orámovaným obdélníkem* budeme nazývat takovou podmatici, která má aspoň dva řádky, aspoň dva sloupce a jejíž první a poslední řádek, stejně jako první a poslední sloupec, obsahují samé jedničky. Uvnitř obdélníka mohou být libovolné prvky.

Soutěžní úloha. Navrhněte algoritmus, který v dané matici A najde největší orámovaný obdélník. Velikost orámovaného obdélníka s i řádky a j sloupci je rovna $i \cdot j$ (tj. hledáme orámovaný obdélník, pro který bude součin $i \cdot j$ největší možný). Pokud existuje více takových obdélníků, stačí nalézt jeden libovolný z nich. Můžete předpokládat, že aspoň jeden orámovaný obdélník se v matici A nachází.

Příklad: Vstup:

$m = 11$, $n = 24$

```
000000000000000000000000
001111100000000001100000
001100111111111111111110
001011111111111111111110
001010100000100011100000
001111100100100011100000
000010001100100111111000
000010000000100100101000
010011111111100100001000
001000000000000111111000
000000000000000000000000
```

Výstup:

Největší orámovaný obdélník má rozměry 6×9 a jeho levý horní roh je ve 4. řádku a 5. sloupci.

P – II – 2

Robot

Výzkumný ústav potrubí a rour má nový problém, tentokrát se soustavou orientovaných trubek. Pro vyčištění této soustavy je zapotřebí, aby čisticí robot prošel každou trubkou právě jednou. Soustava je orientovaná, což znamená, že pro každou trubku je předepsán směr, kterým musí robot trubkou projít. S vynaložením velkého úsilí se programátorům výzkumného ústavu podařilo napsat program, který pro danou soustavu trubek najde jednu možnou trasu čisticího robota, nebo zjistí, že taková trasa neexistuje. Někdy je ovšem užitečné vědět, zda existuje více různých tras čištění (střídáním čisticích tras je totiž možné snížit opotřebení robota v zatáčkách).

Soustava potrubí je složena z n uzlů očíslovaných $1, \ldots, n$, mezi nimiž vede m jednosměrných trubek očíslovaných $1, \ldots, m$. Každá trubka vede mezi dvojicí navzájem různých uzlů a nemá žádné odbočky nebo větvení. Z každého uzlu vede do každého jiného uzlu nejvýše jedna trubka. Pro tuto soustavu je zaručeno, že existuje trasa robota, která začíná i končí v uzlu 1 a projde každou trubkou právě jednou (v souladu se stanovenou orientací trubky). Pracovníci výzkumného ústavu navíc pomocí svého programu jednu takovouto trasu našli a tato trasa je vám k dispozici. Vaším úkolem je zjistit, zda existuje ještě jiná trasa začínající i končící ve vrcholu 1, která projde každou trubkou právě jednou. Tuto trasu nemusíte vypisovat, stačí, když váš program odpoví ano/ne.

Příklad. Vstup:
$n = 5, m = 7$
trubky:
1 2
1 5
2 3
3 1
3 4
4 1
5 3
trasa: 1 2 3 4 1 5 3 1
Výstup:
Existuje jiná trasa.
Poznámka. 1 2 3 1 5 3 4 1 je příklad jiné trasy.

Vstup:
$n = 5, m = 6$
trubky:
1 2
2 3
3 1
3 4
4 5
5 3
trasa: 1 2 3 4 5 3 1
Výstup:
Neexistuje jiná trasa.

P – II – 3

Učitel

Ve třídě sedí učitel a dává pozor na n žáků, kteří píšou písemku. Učitel se po většinu času dívá jedním směrem. Tento směr budeme nazývat *základní směr*. Jakmile však učitel zaslechne odněkud podezřelé zvuky, rychle se otočí, aby viděl, co se děje. Úkolem je zvolit základní směr tak, aby úhel, o který se musí otočit, byl co nejmenší. Jelikož k různým žákům je třeba natočení o různý úhel, chceme minimalizovat průměrný úhel.

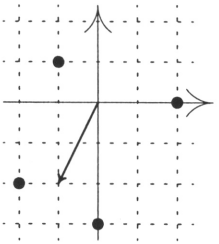
Soutěžní úloha. Na vstupu je dáno číslo n a souřadnice n bodů v rovině $[x_1, y_1], [x_2, y_2], \ldots, [x_n, y_n]$. Každý bod určuje polohu jednoho žáka ve třídě. Učitel sedí v bodě $[0, 0]$. Můžete předpokládat, že bod $[0, 0]$ neleží na žádné přímce určené dvěma zadanými body.

V základním směru je učitel otočen směrem k nějakému bodu $[x, y]$. Když zaslechne vyrušovat žáka i, otočí se směrem k bodu $[x_i, y_i]$. Otáčí se buď po směru hodinových ručiček, nebo proti směru hodinových ručiček, podle toho, kterým směrem je úhel otočení menší (tj. pro každého žáka je úhel otočení nejvýše 180°).

Průměrný úhel otočení je aritmetický průměr úhlů otočení pro všech n bodů. Úkolem je zvolit takový bod $[x, y]$ určující základní směr, aby průměrný úhel otočení byl nejmenší možný.

Pomůcka: Můžete předpokládat, že máte k dispozici funkci $uhel(x, y)$, která vrátí úhel otočení učitele mezi bodem $[1, 0]$ a bodem $[x, y]$ měřeno proti směru pohybu hodinových ručiček (tj. úhel mezi 0° a 360°).

Příklad:
Vstup:
$n = 4$, body: $[-1, 1]$, $[0, -3]$, $[-2, -2]$, $[2, 0]$
Výstup:
Základní směr je směrem k bodu $[-1, -2]$.
Poznámka. Průměrný úhel otočení je 67,5°.
Správných řešení existuje více, například také bod $[-2, -2]$ (žák 3).

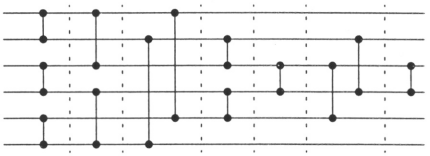


P – II – 4

Komparátorové sítě

(Definice komparátorové sítě je obsažena v textu úlohy P–I–4.)

Soutěžní úlohy. a) Na následujícím obrázku je nakreslena komparátorová síť se šesti vstupy. Nalezněte vstupní data (tj. šestici čísel), která tato síť nesetřídí (neuspořádá je podle velikosti). Zobrazte také průběh výpočtu sítě pro tato vstupní data.



b) V komparátorové síti uvedené v části a) se nachází komparátor, po jehož odstranění bude síť správně třídit libovolnou vstupní posloupnost šesti čísel. Určete tento komparátor a vysvětlete, proč po jeho odstranění síť správně třídí.

c) Na vstupu komparátorové sítě je $2n$ navzájem různých čísel. Čísla na prvních n vstupech jsou seřazena od nejmenšího po největší a čísla na druhých n vstupech jsou také seřazena od nejmenšího po největší. Navrhněte komparátorovou síť, která na prvních n výstupech vrátí n nejmenších čísel (v libovolném pořadí) a na druhých n výstupech vrátí n největších čísel (v libovolném pořadí). Snažte se, aby vaše síť byla co nejrychlejší.

Příklad. $n = 3$, vstupy 1, 4, 5, 2, 3, 6. Na výstupu první tři vodiče obsahují čísla 1, 2, 3 v libovolném pořadí a druhé tři vodiče obsahují čísla 4, 5, 6 v libovolném pořadí. Například 2, 3, 1, 6, 5, 4 je správný výstup.

P – III – 1

Je dána matice A s m řádky a n sloupci. Každý její prvek $a_{i,j}$ (první souřadnice indexu je číslo řádku a druhá je číslo sloupce) je buď celé kladné číslo, nebo takzvaný *žolík*. Žolík budeme označovat znakem $*$. Chceme přerovnat prvky matice tak, aby každý řádek matice tvořil neklesající posloupnost, tj. aby pro každé i, $1 \leqq i \leqq m$, platilo:

$$a_{i,1} \leqq a_{i,2} \leqq \ldots \leqq a_{i,n-1} \leqq a_{i,n}$$

Za žolíky lze do matice dosadit libovolná celá čísla. Tedy například řádek $(1, *, 4, *)$ je neklesající, neboť za žolíky je možné dosadit třeba čísla 2 a 5. Naopak řádek $(5, *, *, 4)$ není neklesající, protože neexistují žádná celá čísla x a y, která by splňovala podmínky $5 \leqq x \leqq y \leqq 4$. Při vyměňování prvků matice jsme omezeni tím, že můžeme mezi sebou zaměnit vždy pouze takové prvky, které jsou umístěny ve stejném sloupci.

Soutěžní úloha. Navrhněte program, který pro zadanou matici A rozhodne, zda je možné přerovnat její prvky v rámci každého sloupce tak, aby všechny řádky výsledné matice byly neklesající. V kladném případě program jednu z možných výsledných matic vypíše.

Příklad:

Vstup:
$m = 4$, $n = 4$
```
 1 10  8 12
 *  9 10  *
 8  * 11 10
12  *  * 10
```

Výstup:
Ano. Jedno možné přeuspořádání prvků je:
```
 1  *  8 10
12  *  *  *
 * 10 11 12
 8  9 10 10
```

Vstup:
$m = 3$, $n = 4$
```
* * * *
* * * *
4 3 2 1
```

Výstup:
Prvky nelze přeuspořádat.

P – III – 2

V jednom lyžařském středisku pořádají závody ve sjezdu. Organizátoři závodů se snaží, aby trasa při každém závodu byla trochu jiná. Na svahu je trvale vyznačeno N *orientačních bodů*, jejichž polohu není možné měnit. Trasa pak prochází některými z uvedených orientačních bodů a splňuje následující podmínky:

- Trasa začíná v nejvýše položeném orientačním bodě a končí v nejníže položeném orientačním bodě.
- Trasa mezi každými dvěma po sobě následujícími orientačními body vede po přímce.
- Nadmořská výška orientačních bodů trasy na svahu ostře klesá.
- V žádném orientačním bodě nemění trasa svůj směr o více než o 45°.

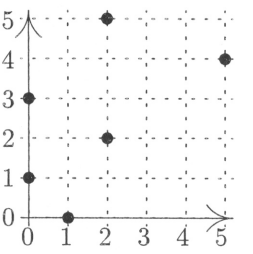
Vaším úkolem je určit počet vyhovujících tras pro závody.

Soutěžní úloha. Na vstupu je dán počet orientačních bodů N a dvojice čísel $(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)$, které popisují polohu jednotlivých orientačních bodů na svahu. Svah si představujeme jako obdélník, který se svažuje shora dolů. Číslo x_i je vzdálenost i-tého orientačního bodu od levého okraje svahu a y_i je jeho vzdálenost od dolního okraje, tj. čím větší je y-ová souřadnice bodu, tím větší má tento bod nadmořskou výšku.

Navrhněte program, který zjistí počet vyhovujících tras na zadaném lyžařském svahu. Můžete předpokládat, že žádné dva orientační body nemají stejnou y-ovou souřadnici a že žádné tři orientační body neleží na jedné přímce.

Pomůcka: Ve svém programu můžete používat funkci $uhel(x, y)$, která vrátí velikost úhlu mezi bodem (x, y) a x-ovou osou, tj. určí úhel, o který je třeba otočit kolem bodu $(0, 0)$ proti směru hodinových ručiček polopřímku vycházející z bodu $(0, 0)$ a procházející bodem $(1, 0)$ tak, aby tato polopřímka po otočení procházela bodem (x, y). Funkce vrací výsledný úhel ve stupních, tj. jako číslo z intervalu $\langle 0, 360)$.

Příklad. Vstup: $N = 6$, orientační body:
$(2, 5), (2, 2), (5, 4), (0, 3), (0, 1), (1, 0)$
Výstup: Existují 3 vyhovující trasy.
Poznámka. Vyhovující trasy jsou:
$(2, 5), (0, 3), (0, 1), (1, 0)$
$(2, 5), (2, 2), (1, 0)$
$(2, 5), (1, 0)$



P – III – 3

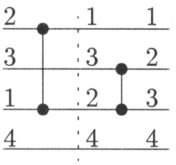
(Definice komparátorové sítě je obsažena v textu úlohy P–I–4.)

Soutěžní úloha. Je dána jedna konkrétní permutace čísel $1, 2, \ldots, n$, tj. taková posloupnost čísel $a_1, a_2, \ldots, a_n$, ve které se každé z čísel $1, 2, \ldots, n$ vyskytuje právě jednou. Máme zaručeno, že naše komparátorová síť bude mít n vodičů a na vstupu bude i-tý vodič obsahovat číslo a_i. Úlohou je navrhnout takovou komparátorovou síť, která tento konkrétní vstup utřídí (navržená síť tedy nemusí třídit žádný jiný vstup).

Protože navržená komparátorová síť bude obecně různá pro různé permutace, je vaším úkolem vytvořit algoritmus, který navrhne konkrétní komparátorovou síť pro zadané číslo n a zadanou posloupnost čísel $a_1, a_2, \ldots, a_n$. Navrženou síť vypište jako posloupnost komparátorů podle jejich výskytu v jednotlivých vrstvách zleva doprava; jednotlivé komparátory udávejte jako dvojici vodičů, které do nich vstupují.

Vámi vytvořený algoritmus musí pracovat v čase, který je polynomiální v n. Snažte se, aby síť, kterou váš algoritmus navrhne, měla co nejméně vrstev (a pokud možno i malý počet komparátorů). Existuje efektivní algoritmus, který pro zadanou permutaci nalezne síť s $O(n)$ komparátory a s $O(\log n)$ vrstvami.

Příklad:
Vstup:
$n = 4$, $a_1 = 2, a_2 = 3, a_3 = 1, a_4 = 4$.
Výstup:
Vstup je možné utřídit například pomocí sítě:
První vrstva: komparátor $(1, 3)$
Druhá vrstva: komparátor $(2, 3)$



P – III – 4

Program: `roury.pas / roury.c / roury.cpp`
Vstupní soubor: `roury.in`
Výstupní soubor: `roury.out`

Ve výzkumném ústavu potrubí a rour mají opět nový problém. Dostali zakázku od Vodáren a kanalizací města Blatislavy, kde je potřeba pročistit městskou kanalizaci od nánosů bláta. Postup při čištění trubek od bláta je jednoduchý: Čistící robot musí projít každou trubkou právě jednou; kdyby robot prošel již vyčištěnou trubkou, hrozilo by nebezpečí, že jeho silné čistící prostředky tuto trubku poškodí. Robot může do

trubky vstoupit z jejího libovolného konce; pokud však již do trubky vstoupí, musí ji celou projít.

Pracovníci výzkumného ústavu si rychle uvědomili, že za těchto podmínek se může stát, že jeden robot na vyčištění celé městské kanalizace nebude stačit. Doporučili tedy do kanalizace poslat více robotů najednou a naprogramovat je tak, aby dohromady pročistili celou kanalizaci.

Nový typ čistícího robota ale není zrovna levný, a proto je potřeba navrhnout takový postup, aby celá kanalizace byla vyčištěna pomocí co nejmenšího počtu robotů. A to je už úkol pro vás.

Kanalizační síť je tvořena n uzly očíslovanými od 1 do n, mezi kterými vede m trubek. Každá trubka spojuje dvojici uzlů a nemá žádné odbočky, ani se nikde nevětví. Mezi každou dvojicí uzlů vede nejvýše jedna trubka.

Soutěžní úloha. Vytvořte program, který pomůže naplánovat trasy pro čistící roboty. Program načte popis kanalizační sítě, určí minimální počet robotů postačující pro vyčištění celé sítě a navrhne jejich trasy tak, aby každou trubkou prošel právě jeden z nich právě jednou.

Formát vstupu: První řádek vstupního souboru `roury.in` obsahuje dvě kladná celá čísla: počet uzlů n ($1 \leqq n \leqq 500$) a počet trubek m, oddělená mezerou. Každý z následujících m řádků popisuje jednu trubku; obsahuje vždy dvě čísla oddělená mezerou, což jsou čísla koncových uzlů trubky.

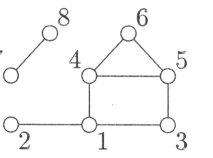
Formát výstupu: Na prvním řádku výstupního souboru `roury.out` bude zapsáno jediné celé číslo R — nejmenší počet robotů, který postačuje k vyčištění kanalizační sítě. Následuje R řádků, z nichž i-tý popisuje trasu i-tého robota. Pokud i-tý robot začíná čistící proces v uzlu a_1, odtud pokračuje trubkou do uzlu a_2, pak do a_3, atd. až skončí v uzlu a_k, potom příslušný řádek výstupního souboru obsahuje k čísel $a_1, a_2, \ldots, a_k$ (v tomto pořadí) oddělená od sebe vždy jednou mezerou.

Příklad. Soubor `roury.in`:

```
8 8
1 2
1 3
1 4
3 5
4 5
4 6
5 6
7 8
```

Soubor `roury.out`:

```
3
2 1 3 5 4 1
4 6 5
8 7
```



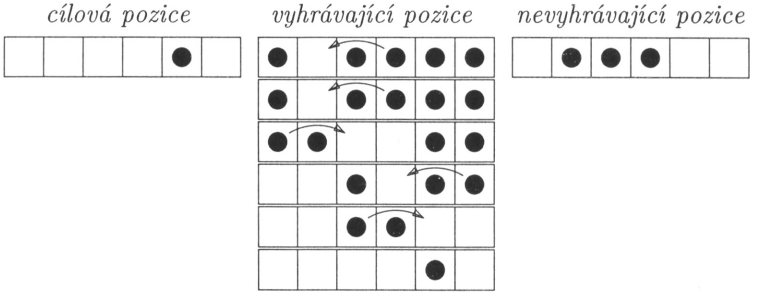
Jestliže existuje více optimálních řešení, váš program má za úkol nalézt a vypsat právě jedno z nich.

P – III – 5

Program: `kameny.pas / kameny.c / kameny.cpp`
Vstupní soubor: `kameny.in`
Výstupní soubor: `kameny.out`

Uvažujme následující hru pro jednoho hráče. Hrací plán je tvořen K políčky uspořádanými do jedné řady. Na začátku hry je na tomto plánu umístěno několik hracích kamenů (na každém políčku leží nejvýše jeden kámen). Je zadána tzv. *cílová pozice*, tj. rozestavení kamenů, kterého je třeba dosáhnout. V jednom tahu můžeme táhnout kamenem z políčka p tak, že jím přeskočíme kámen ležící na sousedním políčku. Přesněji: Pokud je na sousedním políčku napravo (nalevo) od políčka p kámen a následující políčko tímto směrem je volné, lze kámen z políčka p přesunout na volné políčko a přeskočený kámen odstranit z hracího plánu. Jak již bylo řečeno, úkolem je dosáhnout pomocí těchto tahů cílové pozice. Pokud ze zadané pozice lze cílové pozice dosáhnout, říkáme, že zadaná pozice je *vyhrávající*.

Například, pokud pozice vlevo na následujícím obrázku je cílová, pak v prostředním sloupci je jedna z vyhrávajících pozic vzhledem k této cílové pozici (s uvedením příslušné posloupnosti tahů, kterými lze cílové pozice dosáhnout). Naopak pozice v pravé části obrázku není vyhrávající, neboť v ní lze na začátku táhnout jen prostředním ze tří kamenů a tím se dostaneme do pozice se dvěma kameny oddělenými dvěma prázdnými políčky, ze které již nelze dál pokračovat ve hře.



Soutěžní úloha. Vytvořte program, který přečte dvě celá čísla K a N a cílovou pozici a vypíše počet různých vyhrávajících pozic tvořených nejvýše N kameny.

Formát vstupu: Vstupní soubor `kameny.in` bude obsahovat na prvním řádku dvě celá čísla K a N oddělená jednou mezerou. Na druhém řádku souboru bude zadána cílová pozice jako posloupnost K nul a jedniček oddělených mezerami, kde jednička představuje políčko obsazené kamenem a nula prázdné políčko. Můžete předpokládat, že platí $1 \leqq N \leqq K \leqq 100$.

Formát výstupu: Výstupní soubor `kameny.out` bude obsahovat jedno celé číslo, které udává počet vyhrávajících pozic tvořených nejvýše N kameny. Můžete předpokládat, že toto číslo nepřesáhne 10 000.

Příklad. Soubor `kameny.in`:

```
6 3
0 0 1 1 0 0
```

Soubor `kameny.out`:

```
3
```

Soubor `kameny.in`:

```
8 5
1 0 0 0 0 0 0 1
```

Soubor `kameny.out`:

```
10
```

P – I – 1

Úlohu nejprve převedeme do terminologie teorie grafů. *Bipartitní graf* je takový graf, ve kterém můžeme vrcholy rozdělit do dvou disjunktních množin R a S tak, aby každá hrana grafu spojovala některý vrchol z množiny R s některým vrcholem z množiny S. Čtvercovou matici A nul a jedniček rozměrů $n \times n$ můžeme chápat jako reprezentaci bipartitního grafu G s $2n$ vrcholy, kde vrcholy $r_1, r_2, \ldots r_n$ odpovídají řádkům a vrcholy $s_1, s_2, \ldots, s_n$ sloupcům matice. Vrcholy r_i a s_j jsou spojeny hranou právě tehdy, když prvek $A[i, j] = 1$. Hranu mezi r_i a s_j budeme značit (r_i, s_j).

Řekneme, že bipartitní graf má *perfektní párování*, jestliže se jeho vrcholy dají uspořádat do dvojic tak, že v každé dvojici je jeden vrchol z množiny R a jeden vrchol z množiny S a tyto dva vrcholy jsou spojeny hranou. Každý vrchol grafu se přitom musí nacházet právě v jedné takovéto dvojici.

Ukážeme, že jestliže bipartitní graf G má perfektní párování, potom v naší matici A lze přerovnat řádky a sloupce takovým způsobem, aby na hlavní diagonále byly samé jedničky. Pokud párování obsahuje dvojice $(r_{i_1}, s_{j_1}), (r_{i_2}, s_{j_2}), \ldots, (r_{i_n}, s_{j_n})$, pak stačí řádky uspořádat v pořadí $i_1, i_2, \ldots, i_n$ a sloupce v pořadí $j_1, j_2, \ldots, j_n$. Označme takto přerovnanou matici A'. Platí $A'[k, k] = A[i_k, j_k]$. Jelikož mezi vrcholy r_{i_k} a s_{j_k} v grafu G vede hrana, platí $A[i_k, j_k] = 1$, a proto matice A' má na hlavní diagonále samé jedničky.

Naopak, pokud lze matici A přerovnat tak, aby měla na hlavní diagonále samé jedničky, pak v grafu G existuje perfektní párování. Nechť v přerovnané matici A' jsou řádky v pořadí $i_1, i_2, \ldots, i_n$ a sloupce v pořadí $j_1, j_2, \ldots, j_n$. Víme, že pro všechna k platí $A'[k, k] = 1$, a proto také $A[i_k, j_k] = 1$. Proto v grafu G dvojice $(r_{i_1}, s_{j_1}), (r_{i_2}, s_{j_2}), \ldots, (r_{i_n}, s_{j_n})$ tvoří perfektní párování. Dokázali jsme tedy následující tvrzení:

Matici A je možné transformovat na tvar se samými jedničkami na hlavní diagonále právě tehdy, když existuje perfektní párování v grafu G.

Jestliže tedy chceme zjistit, jak zadanou matici A přetransformovat na tvar se samými jedničkami na hlavní diagonále, nalezneme nejprve perfektní párování v bipartitním grafu G. Pokud perfektní párování existuje, přerovnáme řádky a sloupce matice podle postupu uvedeného výše. Pokud takové párování neexistuje, matici nelze transformovat do poža-

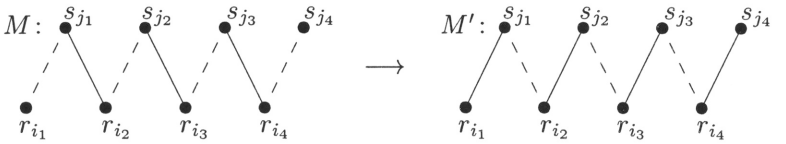
dovaného tvaru. Jediným problémem v tuto chvíli zůstává, jak nalézt perfektní párování bipartitního grafu.

Připomeňme si, že v perfektním párování jsou v každé dvojici vrcholy spojeny hranou. Proto můžeme perfektní párování chápat také jako množinu hran M takovou, že každý vrchol grafu je koncovým vrcholem právě jedné hrany z M. Podobně můžeme definovat *párování* jako množinu hran M takovou, že každý vrchol grafu je koncovým vrcholem nejvýše jedné hrany z M. To znamená, že v párování, které není perfektní, nemusí být každý vrchol zařazen do některé dvojice. Ukážeme si nyní algoritmus, jak nalézt v bipartitním grafu *maximální párování*, tj. párování s nejvyšším možným počtem hran.

Maximální párování budeme hledat postupně. Začneme s prázdným párováním $\mathsf{M} = \emptyset$ a v každém kroku zvýšíme počet párovacích hran o jedna. Když se nám v některém kroku nepodaří zvýšit počet hran v párování, prohlásíme aktuální nalezené párování za maximální a výpočet ukončíme.

Počet hran v párování budeme zvyšovat pomocí takzvaných zlepšujících cest. Uvažujme libovolné pevně zvolené párování M. *Alternující cesta* pro párování M je posloupnost vrcholů $r_{i_1}, s_{j_1}, r_{i_2}, s_{j_2}, \ldots, r_{i_k}, s_{j_k}$, která začíná řádkovým vrcholem, končí sloupcovým vrcholem, každá dvojice po sobě jdoucích vrcholů je spojena hranou a střídají se párovací a nepárovací hrany, přičemž první hrana (r_{i_1}, s_{j_1}) je nepárovací. *Zlepšující cesta* je alternující cesta, která začíná i končí nespárovaným vrcholem.

Všimněte si, že zlepšující cesta $P = r_{i_1}, s_{j_1}, r_{i_2}, s_{j_2}, \ldots, r_{i_k}, s_{j_k}$ se skládá z $k-1$ párovacích a k nepárovacích hran. Mějme tedy párování M a zlepšující cestu P. Jestliže všechny párovací hrany v P změníme na nepárovací a naopak, dostaneme nové párování M′, které má o jednu hranu více (uvědomte si, že M′ skutečně splňuje podmínky párování).



(plné čáry jsou párovací, čárkované čáry jsou nepárovací)

Dokážeme nyní, že naším postupem vždy dostaneme maximální párování, tzn. že vždy existuje zlepšující cesta pro párování, které není maximální.

Tvrzení. *Jestliže párování* M *není maximální, existuje pro něj zlepšující cesta v* G.

Důkaz tvrzení. Nechť M je párování, které není maximální. Jelikož M není maximální, musí existovat nějaké párování M′, které obsahuje více hran než M. Hrany patřící do M, ale nepatřící do M′, nazveme *modré*, zatímco hrany patřící do M′ a ne do M nazveme *červené*. Protože |M′| > > |M|, červených hran je více než modrých.

Uvažujme graf G' tvořený všemi modrými a červenými hranami. Každý vrchol v G' sousedí s nejvýše jednou modrou a jednou červenou hranou. Každá komponenta souvislosti grafu G' proto musí být buď kružnice, nebo cesta, přičemž na každé kružnici i cestě se střídají červené a modré hrany. To znamená, že každá kružnice obsahuje stejný počet modrých a červených hran, počty červených a modrých hran na každé cestě se liší nejvýše o 1. Protože červených hran je více než modrých, musí existovat komponenta P, která obsahuje více červených hran než modrých. Tato komponenta musí být cestou, která začíná i končí červenou hranou. Cesta P tedy obsahuje lichý počet hran. Z toho vyplývá, že jeden z jejích konců musí být řádkovým vrcholem (nazveme ho r_i) a druhý sloupcovým vrcholem (nazveme ho s_j). Jelikož r_i resp. s_j je nespárovaný řádkový resp. sloupcový vrchol a každá druhá hrana patří do párování M, musí být P zlepšující cesta pro M.

Implementace algoritmu. Graf G reprezentujeme samotnou maticí, tj. dvojrozměrným polem A. Pro každý vrchol r_i (s_j) si budeme v poli `par_r` (`par_s`) pamatovat, zda je spárovaný s nějakým vrcholem a pokud ano, číslo tohoto vrcholu. Začneme se všemi vrcholy nespárovanými. Funkce `Zlepsi` hledá zlepšující cestu a když ji najde, použije ji na zlepšení stávajícího párování. Hledání zlepšující cesty budeme realizovat prohledáváním do šířky ze všech vrcholů r_i, které ještě nejsou spárované. Jakmile najdeme nějakou alternující cestu do nespárovaného vrcholu s_j, prohledávání ukončíme. V tomto okamžiku je třeba projít po nalezené cestě z s_j zpět do r_i a změnit párovací hrany na nepárovací a naopak, což znamená aktualizování záznamů v polích `par_r` (`par_s`) pro všechny vrcholy na nalezené cestě. Funkce `Zlepsi` bude volána opakovaně v cyklu tak dlouho, dokud bude možné zvyšovat počet hran v párování. Podaří-li se vylepšit párování n krát, máme nalezeno perfektní párování, které použijeme na přerovnání matice do požadovaného tvaru. V opačném případě program podá zprávu o tom, že se zadaná matice uspořádat nedá.

Efektivita. Každé volání funkce `Zlepsi` potřebuje čas $O(n^2)$, celkově se provede nejvýše n volání této funkce, což dává celkovou časovou složitost $O(n^3)$. Na uložení matice A potřebujeme paměť $O(n^2)$.

Úlohu nejprve převedeme do terminologie teorie grafů. Uzly nacházející se v naší soustavě potrubí budeme nazývat *vrcholy*, trubky nazveme *hranami* a celou soustavu potrubí nazveme *grafem*. Posloupnost vrcholů a hran grafu $v_0, e_1, v_1, e_2, \ldots, v_{k-1}, e_k, v_k$ označíme jako *sled*, jestliže každá hrana e_i spojuje vrcholy v_{i-1} a v_i. Pokud navíc $v_0 = v_k$, tento sled nazveme *uzavřeným*. V řeči teorie grafů je naším úkolem v daném souvislém grafu G nalézt uzavřený sled, který projde každou hranou grafu právě dvakrát.

Ukážeme si algoritmus, který v libovolném souvislém grafu najde sled požadovaných vlastností. Tento algoritmus tedy bude zároveň důkazem, že trasa pro robota vždy existuje. Řešení je založeno na prohledávání grafu do hloubky. O každém vrcholu si budeme pamatovat, zda jsme ho již během prohledávání někdy navštívili. Pro každý navštívený vrchol v nechť `rodic`$[v]$ označuje vrchol, z něhož robot do v poprvé přišel. Vrchol `rodic`$[v]$ je *rodičem* vrcholu v, naopak vrchol v nazveme *potomkem* vrcholu `rodic`$[v]$. Pro vrchol 1 není `rodic`[1] definován.

Prohledávání do hloubky začíná ve vrcholu 1 a probíhá takto:

1. Označ aktuální vrchol v jako navštívený.
2. Postupně kontroluj všechny hrany vedoucí z v a vždy, když najdeš hranu, která vede do dosud nenavštíveného vrcholu u, projdi robotem do vrcholu u a pokračuj rekurzívně v prohledávání z u (v se stane rodičem vrcholu u).
3. Když jsou všechny hrany vedoucí z v zkontrolovány a $v \neq 1$, vrať se do vrcholu `rodic`$[v]$ (a pokračuj v kontrolování jeho hran). Jestliže $v = 1$, skonči.

Uvažujme množinu hran, po nichž robot poprvé přišel do nějakého vrcholu. Jsou to právě hrany (`rodic`$[v], v$) pro ty vrcholy v, pro které je `rodic`$[v]$ definováno. Tyto hrany nazveme *stromové* (nazývají se tak proto, že tvoří souvislý acyklický graf, tzv. strom). Každou stromovou hranou (u, v), kde u je rodičem v, projde robot při prohledávání právě dvakrát — nejprve z vrcholu u poprvé navštíví vrchol v a podruhé při návratu z vrcholu v do vrcholu u. Při prohledávání robot nikdy neprojde po žádné nestromové hraně. Jelikož náš graf G je souvislý, robot při prohledávání navštíví všechny vrcholy. Kdyby totiž existoval nenavštívený vrchol, musel by existovat takový navštívený vrchol u a nenavštívený vrchol v, že u a v jsou spojeny hranou. To však není možné, neboť al-

goritmus prohledávání se z vrcholu u do `rodic`$[u]$ nevrátí, dokud nejsou všechny sousední vrcholy vrcholu u navštíveny.

Podle dosud popsaného prohledávání do hloubky tedy robot navštíví každý vrchol a projde každou stromovou hranou právě dvakrát. Zbývá rozhodnout, kdy a jak bude robot procházet nestromové hrany. Během prohledávání lze snadno detekovat všechny nestromové hrany. Jestliže se robot nachází ve vrcholu u a kontroluje hranu (u, v), tato hrana je nestromová tehdy, pokud vrchol v již byl navštíven a u není rodičem v ani v není rodičem u. Když robot nacházející se ve vrcholu u narazí na nestromovou hranu (u, v), může příslušnou trubku vyčistit tak, že jí projde z vrcholu u do v a zpět.

Implementace algoritmu. Prohledávání do hloubky můžeme snadno implementovat rekurzívní procedurou. Jelikož rekurze si pro aktuální vrchol automaticky pamatuje svého rodiče, není třeba explicitně udržovat záznamy `rodic`$[v]$. Graf lze v programu reprezentovat maticí velikosti $n \times n$, z čehož vyplývá časová i paměťová složitost programu $O(n^2)$. Při použití efektivnější reprezentace hran spojovým seznamem je možné dosáhnout časové i paměťové složitosti $O(m+n)$, kde m je počet hran grafu, ovšem za cenu o něco komplikovanějšího programu.

P – I – 3

Dokážeme nejprve, že spravedlivá přímka vždy existuje. Uvažujme libovolnou orientovanou přímku vedoucí bodem $[0, 0]$. Nechť x je rozdíl počtu bílých bodů nalevo a černých bodů napravo od ní. Jestliže je x rovno nule, přímka je spravedlivá. Pokud ne, budeme přímkou otáčet kolem bodu $[0, 0]$. Vždy, když přímka projde přes bílý nebo černý bod, hodnota x se sníží nebo zvýší o jedna. Když přímku otočíme přesně o 180°, naše sledovaná hodnota se změnila z počátečního x na $-x$ (neboť všechny body, které byly původně nalevo, jsou nyní napravo od přímky a naopak). To znamená, že hodnota musela být aspoň při jedné poloze přímky nulová. V této poloze byla přímka spravedlivá.

Řekneme, že bod B leží nalevo od bodu A, jestliže je nalevo od orientované přímky vedoucí z bodu $[0, 0]$ do A. Základem algoritmu je následující pozorování. Uvažujme nějakou spravedlivou přímku p. Nechť A je první bod, na který narazíme, pokud budeme přímkou p otáčet ve směru hodinových ručiček kolem bodu $[0, 0]$. Bod A a všechny body, které leží od něj napravo, se nacházejí v jedné polorovině určené přímkou p. Body, které jsou od A nalevo, leží ve druhé polorovině. Vidíme, že nezáleží na

konkrétní poloze přímky p, rozdělení bodů je určeno polohou „hraničního“ bodu A.

Stačí tedy uvažovat každý ze zadaných bodů jako kandidáta na bod A, spočítat počet bílých a počet černých bodů nalevo a napravo od něj a zkontrolovat, zda tyto počty splňují podmínky spravedlivé přímky. Když najdeme úspěšného kandidáta, spravedlivou přímku sestrojíme tak, že přímku vedoucí z $[0,0]$ tímto bodem trochu pootočíme kolem bodu $[0,0]$ proti směru hodinových ručiček tak, abychom s ní při tomto otáčení nepřešli přes žádný jiný bod. To lze provést například tak, že určíme první bod X, přes který bychom při takovémto otáčení přímkou přešli, a výslednou přímku vedeme středem mezi kandidátem A a bodem X. Popsaný algoritmus má časovou složitost $O(n^2)$, neboť pro každého kandidáta zjišťujeme polohu každého bodu vzhledem k tomuto kandidátovi.

Algoritmus lze ještě zefektivnit tím, že si všechny bílé a černé body nejprve setřídíme podle jejich pořadí ve směru hodinových ručiček kolem bodu $[0,0]$. Kandidáty na hraniční bod A pak budeme zkoušet v tomto utříděném pořadí. Kdykoliv budeme přecházet od kandidáta $i-1$ ke kandidátovi i, nemusíme již pro každý bod znovu zjišťovat, zda leží od bodu i nalevo nebo napravo, neboť mnoho bodů zůstane v původní polorovině. Budeme si proto v každém kroku udržovat informaci o počtu bílých a černých bodů ležících v pravé polorovině a index j, který ukazuje na poslední bod v pravé polorovině ve směru hodinových ručiček. Když se přesuneme na kandidáta i, stačí bod $i-1$ přemístit do levé poloroviny (tj. odebrat ho z počtu bodů příslušné barvy, které jsou v pravé polorovině) a potom posouvat index j ve směru hodinových ručiček, dokud nenajdeme první bod, který leží nalevo od bodu i. Všechny body, přes které jsme s indexem j přešli, se přesunou z levé poloroviny do pravé a je proto třeba připočítat je k zaznamenanému počtu bodů příslušné barvy. Pokud při posouvání bodu j dojdeme na konec pole, budeme pokračovat cyklicky opět od začátku. Nesmíme zapomenout ošetřit okrajové případy, když například všechny body leží od bodu j nalevo nebo napravo.

Utřídit body ve směru hodinových ručiček můžeme v čase $O(n \log n)$ některým ze standardních třídicích algoritmů, pouze namísto běžného porovnávání hodnot si musíme napsat funkci, jež pro dva body určí, který z nich má větší úhel. Pro první bod musíme projít všechny ostatní body a zjistit, které leží vlevo a které vpravo (v čase $O(n)$). Pro každý další bod už jen budeme posouvat indexy i a j a sledovat pouze body, přes které

přecházíme. Přes každý bod přejdeme každým indexem nejvýše jednou, takže celkový čas posouvání pro všechny kandidátské body dohromady bude $O(n)$. Výsledná časová složitost algoritmu bude proto $O(n \log n)$ a paměťová složitost $O(n)$.

P – I – 4

Část a. Vstupní soubor může mít například následující formát (váš program samozřejmě může používat jiný formát vstupních dat). Na prvním řádku souboru je zadán počet vodičů a počet komparátorů v síti, na druhém řádku jsou uvedeny jednotlivé vstupy sítě v pořadí od vrchního vodiče po spodní a zbývající řádky obsahují popis sítě. Každý komparátor je určen dvěma čísly zapsanými na jednom řádku. Tato čísla udávají vrchní a spodní vodič, které jsou spojené příslušným komparátorem. Vodiče v síti jsou očíslovány shora dolů počínaje jedničkou. Komparátory jsou uvedeny v pořadí zleva doprava, jednotlivé vrstvy sítě se oddělují řádkem obsahujícím dvě nuly.

Program může pro jednoduchost zobrazovat výpočet sítě semigraficky. Nejprve přečte vstupní údaje, přičemž informace o všech komparátorech a oddělovačích vrstev uloží do jednoho pole. Potom bude vykreslovat síť postupně zleva doprava. Pokaždé, když program nakreslí komparátor, odsimuluje jeho činnost na aktuálním stavu vodičů. To znamená, že porovná příslušné dvě hodnoty v poli a pokud jsou v opačném pořadí, vymění je. Po ukončení každé vrstvy (tj. vždy, když v seznamu komparátorů narazíme na oddělovač vrstev) program vypíše aktuální stav hodnot na všech vodičích.

Jediný problém, který je třeba vyřešit, spočívá v umístění jednotlivých komparátorů téže vrstvy do sloupců pod sebe tak, aby se žádné dva komparátory v jednom sloupci nepřekrývaly. Program si proto bude pamatovat, které z vodičů jsou v aktuálním sloupci už obsazeny komparátorem. Jestliže se další komparátor překrývá s některým komparátorem zakresleným v aktuálním sloupci, založíme nový sloupec a komparátor umístíme do nového sloupce.

Celková časová složitost takto navrženého programu je úměrná velikosti nakresleného obrázku, tj. $O(nk)$, kde n je počet vodičů a k je počet sloupců, v nichž jsou komparátory zobrazeny.

Na závěr uvedeme pro ilustraci příklad vstupu a výstupu programu napsaného podle výše uvedeného popisu (jedná se o komparátorovou síť ze studijního textu):

Vstup:

```
4 5
4 1 2 3
1 4
2 3
0 0
```

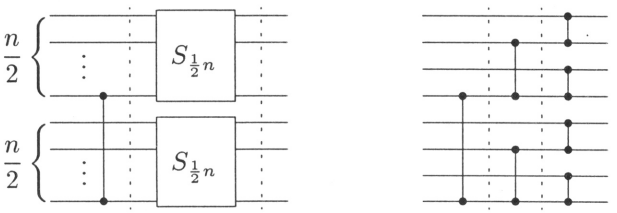
```
1 2
3 4
0 0
2 3
0 0
```

Výstup:

```
4--*-----3--*--1-----1
   |        |
1--|--*--1--*--3--*--2
   |  |           |
2--|--*--2--*--2--*--3
   |        |
3--*-----4--*--4-----4
```

Část b. Komparátorovou síť sestrojíme rekurzívně. Označme S_n síť, která úlohu řeší pro n vstupů. Síť S_1 neobsahuje žádný komparátor, neboť máme jen jeden vstup a ten je jistě setříděn. Předpokládejme, že $n > 1$. První vrstva sítě obsahuje pouze jediný komparátor umístěný mezi vodiči n a $\frac{1}{2}n$. Potom rozdělíme n vodičů na dvě poloviny — horní a dolní — a na každou polovinu vodičů použijeme síť $S_{\frac{1}{2}n}$. Tyto dvě podsítě budou pracovat paralelně.

Konstrukce sítě S_n je zobrazena na následujícím obrázku vlevo, vpravo je příklad výsledné sítě pro $n = 8$.



Ukážeme si, proč takto sestavená síť řeší zadanou úlohu. Postupujeme indukcí podle počtu vodičů. Označme vstupní hodnoty sítě $x_1, x_2, \ldots, x_n$ (umístěny na vodičích v pořadí shora dolů). Mohou nastat dva případy podle toho, zda je x_n menší nebo větší než $x_{\frac{1}{2}n}$. Předpokládejme nejprve, že $x_n \geqq x_{\frac{1}{2}n}$. V takovém případě komparátor v první vrstvě nevymění vstupní hodnoty. Protože $x_n \geqq x_{\frac{1}{2}n}$, prvek x_n patří v uspořádaném pořadí na některý z dolní poloviny vodičů. („Dolní" polovinou vodičů budeme rozumět vodiče se vstupními hodnotami $x_{\frac{1}{2}n+1}$ až x_n, tj. vodiče, které na schématu komparátorové sítě kreslíme v dolní polovině obrázku.)

Po prvním kroku výpočtu tedy platí, že horní polovina je celá utříděna a obsahuje $\frac{1}{2}n$ nejmenších prvků ze vstupu. Dolní polovina je utříděna s případnou výjimkou posledního prvku a obsahuje $\frac{1}{2}n$ největších prvků. Vstupy obou podsítí $S_{\frac{1}{2}n}$ tudíž splňují podmínku ze zadání, že všechny hodnoty kromě poslední mají být na vstupu utříděny (to nevylučuje případ, že také poslední hodnota bude utříděna). Na výstupu proto budou

podle indukčního předpokladu obě poloviny v setříděném pořadí. Jelikož už po prvním kroku každá polovina obsahovala správné prvky, bude správně uspořádána i celá posloupnost.

Druhý případ nastane, jestliže $x_n < x_{\frac{1}{2}n}$. V této situaci komparátor v první vrstvě vymění hodnoty obou těchto vodičů. Víme, že x_n patří někam do horní poloviny prvků a dostane se na nejspodnější vodič horní poloviny. Naopak, poslední prvek z horní poloviny, tj. $x_{\frac{1}{2}n}$, patří ve skutečnosti do dolní poloviny (při třídění je „vytlačen" prvkem x_n). Prvek $x_{\frac{1}{2}n}$ se však prvním komparátorem dostane na nejspodnější vodič, tj. do dolní poloviny. Podobně jako v předchozím případě i nyní tedy máme po prvním kroku v každé polovině ty prvky, které tam v setříděném pořadí patří. Obě poloviny jsou navíc setříděny s případnou výjimkou svého nejspodnějšího prvku. Po aplikování podsítí $S_{\frac{1}{2}n}$ proto opět podle indukčního předpokladu dostaneme dvě utříděné poloviny, které společně vytvoří celou správně uspořádanou posloupnost.

Hloubka rekurze je v naší konstrukci $\log_2 n$, neboť každá úroveň rekurze sníží počet vodičů na polovinu. V každé úrovni rekurze přidáme do sítě jednu vrstvu, celkový počet vrstev je tedy také roven $\log_2 n$.

První vrstva obsahuje jeden komparátor, v každé další vrstvě se počet komparátorů zdvojnásobí. Nechť počet vstupů je $n = 2^k$. Potom počet použitých komparátorů je roven $1 + 2 + 4 + \ldots + 2^{k-1} = 2^k - 1 = n - 1$ (součet geometrické řady). Naše síť má tedy $O(\log n)$ vrstev a používá $O(n)$ komparátorů.

P – II – 1

Každého asi napadne triviální řešení — vyzkoušet všechny možné obdélníky. Obdélník je určen dvojicí protilehlých vrcholů, polohu každého vrcholu můžeme zvolit mn způsoby. V každém zvoleném obdélníku projdeme po jeho obvodě a ověříme, zda všechny jeho obvodové prvky jsou jedničky. Tento jednoduchý algoritmus pracuje v čase $O((mn)^2(m+n))$.

Popsaný algoritmus můžeme vylepšit tím, že zrychlíme ověřování, zda je zvolený obdélník orámovaný. Nechť $h[i,j]$ označuje délku maximálního souvislého úseku jedniček ve sloupci *nad* prvkem (i,j), počítáno včetně tohoto prvku. Přesněji řečeno, $h[i,j] = d$ je takové číslo, že platí $A[i,j] = A[i-1,j] = \ldots = A[i-d+1,j] = 1$ a zároveň buď $d = i$, nebo $A[i-d,j] = 0$. Všimněte si, že když $A[i,j] = 0$, podle této definice je také $h[i,j] = 0$. Podobně nechť $l[i,j]$ označuje délku maximálního souvislého úseku jedniček v řádku *nalevo* od prvku (i,j), včetně prvku

(i,j). Hodnoty $h[i,j]$ a $l[i,j]$ si můžeme vypočítat napřed pro všechny prvky matice. Pro každý prvek zvlášť dokážeme zjistit délku úseku jedniček nalevo a nahoru od něho v čase $O(m+n)$. Celkem máme mn prvků, takže tento předvýpočet polí h a l snadno implementujeme v čase $O(mn(m+n))$. Můžeme ho však ještě urychlit. Jestliže totiž $A[i,j]=0$, víme, že také $h[i,j]=0$. Jestliže $A[i,j]=1$, počet jedniček v souvislém úseku nad prvkem (i,j) je o 1 větší než počet jedniček v úseku nad prvkem $(i-1,j)$, tj. $h[i,j]=h[i-1,j]+1$. Stačí nám tedy počítat hodnoty h po sloupcích shora dolů a při jednom průchodu maticí A v čase $O(mn)$ získáme všech mn hodnot $h[i,j]$.

Předpokládejme nyní, že chceme zjistit, zda je obdélník s levým horním rohem (i_1,j_1) a pravým dolním rohem (i_2,j_2) orámovaný. Obdélník má dolní hranu ze samých jedniček právě tehdy, když úsek jedniček nalevo od prvku (i_2,j_2) má délku aspoň j_2-j_1+1. Podobně ověříme, zda úsek nalevo od prvku (i_1,j_2) (horní hrana) má délku aspoň j_2-j_1+1 a zda úseky nahoru od prvků i_2,j_1 a i_2,j_2 (levá a pravá hrana) mají délku aspoň i_2-i_1+1. Pomocí polí l a h tedy dokážeme v konstantním čase zjistit, zda je daný obdélník orámovaný. Opět vyzkoušíme všechny možnosti umístění levého horního a pravého dolního rohu, pro každý obdélník zjistíme, jestli je orámovaný, a z orámovaných vybereme ten s největší plochou. Tento algoritmus potřebuje $O(mn)$ času na přípravu pomocných polí a $O((mn)^2)$ času na průchod všemi obdélníky. Celkový čas výpočtu je tedy $O((mn)^2)$.

Existuje ale ještě rychlejší řešení úlohy. Pro každou dvojici řádků $i_1<$ $<i_2$ určíme největší orámovaný obdélník, jehož horní hrana leží v řádku i_1 a dolní hrana v řádku i_2. Uvažujme dvojici pevně zvolených řádků i_1 a i_2. Sloupec j nazveme *okrajovým*, jestliže v úseku mezi řádky i_1 a i_2 (včetně) obsahuje samé jedničky. Všimněte si, že sloupec j je okrajový právě tehdy, když $h[i_2,j]\geqq i_2-i_1+1$. Sloupec j nazveme *jedničkovým*, pokud v řádcích i_1 a i_2 obsahuje jedničku. Sloupec, který obsahuje nulu alespoň v jednom z těchto dvou řádků, nazveme *nulovým*.

Každý orámovaný obdélník s vodorovnými hranami ležícími v řádcích i_1 a i_2 odpovídá úseku od sloupce j_1 do sloupce j_2 pro nějaká $j_1<j_2$ taková, že j_1 a j_2 jsou okrajové sloupce a sloupce $j_1+1,\ldots,j_2-1$ jsou jedničkové. Uvažujme maximální úsek po sobě jdoucích jedničkových sloupců (pojmem maximální rozumíme, že se nedá rozšířit, tj. na obou koncích sousedí buď s nulovým sloupcem, nebo s okrajem matice). Jestliže tento úsek neobsahuje aspoň dva okrajové sloupce, zjevně nemůže obsahovat ani žádný orámovaný obdélník. Jestliže obsahuje aspoň

dva okrajové sloupce, potom největší orámovaný obdélník v daném úseku je určen nejlevějším a nejpravějším okrajovým sloupcem daného úseku. K nalezení největšího orámovaného obdélníka v pásu mezi řádky i_1 a i_2 nám tedy stačí v každém maximálním jedničkovém úseku určit nejlevější a nejpravější okrajový sloupec. Toho lze snadno dosáhnout v čase $O(n)$. Musíme vyzkoušet všechny dvojice řádků, celková časová složitost je proto $O(m^2 n)$.

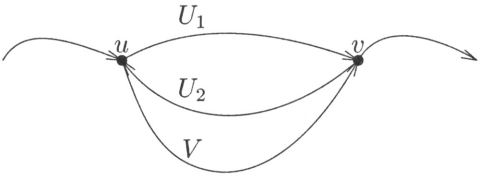
P – II – 2

Úlohu převedeme do řeči teorie grafů. Soustava potrubí tvoří jednoduchý orientovaný graf bez násobných hran a smyček. Uzly představují vrcholy tohoto grafu a trubky jsou jeho orientované hrany. Počet hran vstupujících do vrcholu grafu nazýváme vstupním stupněm tohoto vrcholu a počet vycházejících hran jeho výstupním stupněm. V našem grafu se vstupní a výstupní stupeň každého vrcholu sobě rovnají. Číslo, jemuž se rovnají, budeme také označovat jako počet průchodů vrcholem.

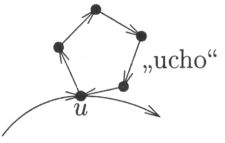
Uvažujme nejprve, co by se stalo, kdyby některý vrchol v měl počet průchodů větší než 2. Trasa robota zadaná na vstupu by takovým vrcholem v procházela alespoň třikrát. Nechť X je úsek trasy mezi prvním a druhým příchodem do v a Y úsek mezi druhým a třetím příchodem. Snadno nyní sestrojíme novou trasu robota. Až do prvního příchodu do v se jde podle původní trasy zadané na vstupu. Po příchodu do v půjde robot nejprve úsek Y (čímž se vrátí do v), potom úsek X (čímž se opět vrátí do v) a dále dokončí svoji cestu podle původní zadané trasy. Jinými slovy, oproti trase zadané na vstupu jsme jen vyměnili pořadí úseků X a Y. Tím jsme ale sestrojili odlišnou trasu. Nemá-li tedy žádná jiná trasa existovat, musí mít nutně každý vrchol počet průchodů 1 nebo 2.

Mějme nyní takový graf a v něm vyznačenou trasu, která začíná i končí ve vrcholu 1 a prochází každou hranou právě jednou. Pokud tento graf neobsahuje žádné hrany, trasa je zjevně jediná (prázdná trasa). Jestliže náš graf nějaké hrany má, půjdeme po vyznačené trase, dokud se na ní poprvé nějaký vrchol u nezopakuje. To určitě dříve či později nastane. Všimněte si úseku cesty mezi prvním a druhým příchodem do u — označme tento úsek U. Má-li některý vrchol úseku U (jiný než u) počet průchodů 2, znamená to, že se tento vrchol ještě někdy na trase vyskytne. Označme si symbolem v první takový vrchol. Tento vrchol rozdělí U na dvě části — část od u do v označme U_1 a část od v do u označme U_2. Část trasy od druhého příchodu robota do vrcholu u do jeho druhého příchodu

do v označme V. Naše trasa tedy vypadá následovně: Robot přijde do u, projde po řadě úseky U_1, U_2, V a poté zbytek trasy. V takovém případě však existuje i jiná trasa: Robot stejně jako předtím přijde do u, projde postupně úseky V, U_2, U_1 a zbytek trasy pak projde stejně jako v původní trase.



Z provedené úvahy vyplývá, že nemá-li existovat žádná jiná trasa, musí mít všechny vrcholy úseku U (kromě u) počet průchodů 1. Úsek U bude tedy tvořit *ucho* nad vrcholem u.



Jestliže hrany tvořící *ucho* z grafu odstraníme, nezměníme tím počet tras existujících v grafu. (Každá trasa v grafu totiž vypadá následovně: Robot nějakým způsobem přijde do vrcholu u, potom projde *ucho* a nakonec nějak projde zbytek grafu. Když hrany tvořící *ucho* odstraníme, ke každé trase v původním grafu najdeme odpovídající trasu v novém grafu tak, že z ní odstraníme hrany *ucha*.) Tím ale dostaneme graf s méně hranami, na němž můžeme tento postup zopakovat. Jestliže se nám takto podaří postupně odstranit všechny hrany z grafu, znamená to, že i původní graf měl jen jednu možnou trasu. Naopak, pokud v některém kroku zjistíme, že v právě zpracovávaném grafu existuje více tras, znamená to, že také náš původní graf obsahoval více tras.

Předchozí rozbor je již návodem, jak sestrojit algoritmus řešící tuto úlohu. Pro každý vrchol si budeme pamatovat počet průchodů. Má-li některý vrchol počet průchodů větší než 2, ohlásíme, že existuje jiná trasa a výpočet ihned ukončíme. V opačném případě začneme postupně číst ze vstupu zadanou trasu robota a budeme si pamatovat, přes které vrcholy jsme již prošli. Když se nám některý vrchol v zopakuje, podíváme se, zda některý vrchol mezi oběma příchody do v nemá počet průchodů 2. Pokud takový vrchol najdeme, ohlásíme, že existuje jiná trasa a ukončíme

výpočet. Pokud ne, odstraníme tyto vrcholy z trasy a pokračujeme dále. (Všimněte si, že si vůbec nepotřebujeme pamatovat graf a měnit ho.) Když výpočet ukončíme s prázdnou trasou, ohlásíme, že žádná jiná trasa neexistuje.

Správnost tohoto algoritmu vyplývá z výše uvedeného popisu. Zbývá odvodit jeho časovou složitost. Nechť má náš graf N vrcholů a M hran. Jestliže z některého vrcholu vycházejí alespoň tři hrany, jakmile první takový vrchol najdeme, můžeme výpočet ukončit. V tomto případě je časová složitost algoritmu $O(N)$. (Pokud bychom dočetli ze vstupu celý graf a až potom kontrolovali počty průchodů přes vrcholy, zhoršila by se časová složitost na $O(M+N)$.) V opačném případě z každého vrcholu vycházejí nejvýše dvě hrany, proto $M = O(N)$. (Tedy náš graf má jen lineárně mnoho hran v závislosti na počtu vrcholů.) Proto také počet hran trasy je $O(N)$, neboť v trase je každá hrana obsažena právě jednou. Každý vrchol na trase nejvýše jednou načteme, nejvýše jednou se během výpočtu algoritmu podíváme na počet průchodů přes něj a nejvýše jednou ho z trasy vyřadíme. To znamená, že (při vhodné implementaci) bude také v tomto případě časová složitost popsaného algoritmu $O(N)$.

P – II – 3

Jestliže se učitel otáčí ze svého základního směru k nějakému žákovi proti směru hodinových ručiček, řekneme, že tento žák je nalevo. Jestliže se otáčí po směru hodinových ručiček, žák je napravo.

Nejprve dokážeme, že vždy existuje řešení, v němž je učitel otočen směrem k nějakému žákovi. Představme si přímku, která prochází bodem $[0,0]$ a určuje základní směr. Otočme nyní tuto přímku kolem bodu $[0,0]$ doleva o malý úhel α tak, aby žádný žák, který byl nalevo, nepřešel na pravou stranu, a naopak. Takovýmto natočením přímky úhly otočení všech žáků nalevo klesnou o α a úhly otočení všech žáků napravo vzrostou o α. Pokud je tedy nalevo více žáků, průměrný úhel otočení se zmenší, pokud je nalevo méně žáků, průměrný úhel se zvětší a pokud je na obou stranách stejně žáků, průměrný úhel se nezmění. Jestliže přímka určující nejlepší základní směr neprochází žádným bodem, alespoň jedním směrem ji můžeme trochu natočit, aniž by průměrný úhel vzrostl. S otáčením přímky přestaneme, jakmile přímka narazí na první bod.

Zbývá vyřešit případ, že přímka sice prochází některým bodem, ale učitel se dívá opačným směrem, tj. je otočen zády k tomuto žákovi. Taková situace ale nikdy není optimálním řešením. Předpokládejme, že

napravo je alespoň tolik žáků jako nalevo. Když otočíme přímku kousek doprava, všem napravo a žákovi za zády učitele klesne úhel otočení, takže celkově se průměrný úhel otočení zlepší. Když je naopak nalevo více žáků, průměrný úhel se zlepší natočením přímky doleva.

Tím jsme dokázali, že stačí zkoumat jen n základních směrů, v nichž je učitel otočen směrem k některému žákovi. Pro každou z těchto n možností spočítáme průměrný úhel otočení a vybereme nejlepší. Jestliže budeme počítat průměrný úhel otočení pro každý směr zvlášť, dostaneme algoritmus s časovou složitostí $O(n^2)$. My si však ukážeme lepší algoritmus s časovou složitostí $O(n \log n)$.

Nechť α_i je úhel, který s osou x svírá polopřímka vedoucí z bodu $[0,0]$ k žákovi i (tj. úhel, který vypočítáme funkcí `uhel`(x_i, y_i)). Uvědomte si, jaký je vlastně úhel otočení mezi učitelem obráceným čelem k žákovi u a mezi žákem i. Musíme uvažovat čtyři případy:

- Žák i je nalevo a $\alpha_i > \alpha_U$. Úhel otočení je $\alpha_i - \alpha_U$.
- Žák i je nalevo a $\alpha_i < \alpha_U$. Úhel otočení je $\alpha_i - \alpha_U + 360°$.
- Žák i je napravo a $\alpha_i < \alpha_U$. Úhel otočení je $\alpha_U - \alpha_i$.
- Žák i je napravo a $\alpha_i > \alpha_U$. Úhel otočení je $\alpha_U - \alpha_i + 360°$.

Chceme-li určit průměrný úhel otočení, potřebujeme sečíst úhly otočení všech žáků. Sčítáním dostaneme následující výraz:

$$\sum_{\substack{\text{žák } i \\ \text{je nalevo}}} \alpha_i - \sum_{\substack{\text{žák } i \\ \text{je napravo}}} \alpha_i - L \cdot \alpha_U + P \cdot \alpha_U + X \cdot 360°,$$

kde L je počet žáků nalevo, P je počet žáků napravo a X je počet žáků, kteří jsou nalevo s úhlem menším než α_U nebo napravo s úhlem větším než α_U.

Součet úhlů otočení tedy umíme spočítat v konstantním čase, jestliže známe součet hodnot α_i pro žáky nalevo a napravo a jestliže známe počty L, P, X.

Náš algoritmus si nejprve utřídí body podle úhlu α_i (tj. proti směru hodinových ručiček). Potom si pro každé i spočítá součet úhlů α_j pro prvních i bodů v utříděném pořadí a tato čísla uloží do pole β (tj. $\beta_i = \alpha_1 + \alpha_2 + \ldots + \alpha_i$). Všimněte si, že $\beta_i = \alpha_i + \beta_{i-1}$, takže při průchodu setříděným polem s hodnotami úhlů α_i zleva doprava dokážeme spočítat všechny hodnoty β_i v lineárním čase. Když nyní budeme chtít určit součet úhlů v úseku od i-tého po j-tého žáka, stačí spočítat $\beta_j - \beta_{i-1}$.

Po této inicializaci budeme postupně zkoumat všechny možné základní směry v tom pořadí, v jakém se nacházejí v setříděném poli. Nechť

U je žák, který určuje základní směr. Pro každé U si najdeme posledního žáka l, který je nalevo. Všichni žáci mezi U a l (včetně l) jsou nalevo, ostatní žáci jsou napravo. Někdy máme $l < U$, v takovém případě jsou nalevo žáci $U+1, \ldots, n$ a $1, \ldots, l$. Podobně žáci napravo buď tvoří jeden, nebo dva souvislé úseky v utříděném poli. Známe-li U a l, můžeme použít pole β k tomu, abychom v konstantním čase zjistili součet úhlů α_i pro žáky nalevo a napravo, neboť to jsou součty jednoho nebo dvou souvislých úseků v poli α. Počty žáků nalevo a napravo (L a P) také lehce spočítáme. Určení hodnoty X se zdá trochu těžší, ale není — žák i má α_i menší než α_U tehdy, když je i menší než U (neboť pole je setříděné podle α). Pokud tedy známe indexy U a l, dokážeme snadno spočítat součet (a průměr) úhlů otočení v konstantním čase.

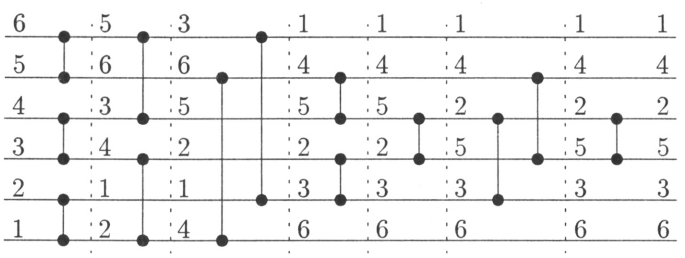
Zbývá už jen vyřešit problém, jak efektivně nalézt hodnotu l, tj. index posledního prvku ležícího vlevo od U. Pro $U = 1$ jednoduše začneme s $l = 1$ a budeme zvyšovat l, dokud nenajdeme poslední prvek, který je nalevo. Pro každou další hodnotu U využijeme fakt, že l z předcházejícího kroku je nyní určitě nalevo. Začneme proto od předcházející hodnoty l a budeme l zvyšovat, dokud nenajdeme první bod vpravo (když přijdeme na konec pole, pokračujeme v něm cyklicky zase od začátku). Index l tímto způsobem během celého výpočtu oběhne pole pouze dvakrát, takže celková složitost hledání posledního prvku vlevo je $O(n)$.

Výsledná časová složitost algoritmu je $O(n \log n)$, neboť třídění pracuje v čase $O(n \log n)$, hodnoty pole β dokážeme spočítat v čase $O(n)$, celkový čas otáčení indexu l je $O(n)$ a výpočet součtu úhlů otočení je konstantní pro jeden směr, tj. $O(n)$ pro všechny směry.

V ukázkovém programu jsme kvůli úspoře místa nahradili $O(n \log n)$ třídění kvadratickým. Použili jsme také pole délky $2n$, v němž se každý bod nachází dvakrát, což zjednodušuje výpočty (není třeba uvažovat, že poslední bod vlevo bude mít index menší než U — v případě potřeby jednoduše pokračujeme s hledáním indexu l dále za n a využíváme druhé kopie bodů v poli). Při implementaci je nutné dbát na to, aby program správně ošetřil různé okrajové případy, například, když všechny body leží nalevo nebo napravo od U.

P – II – 4

a) Správnou odpovědí je například vstup $6, 5, 4, 3, 2, 1$ (ale také mnoho jiných vstupů). Pro vstup $6, 5, 4, 3, 2, 1$ výpočet komparátorové sítě probíhá následovně:



b) Je třeba odstranit komparátor vyznačený šipkou na předcházejícím obrázku (jediné správné řešení). Dokážeme, že po odstranění tohoto komparátoru síť třídí. V prvních třech vrstvách se minimum dostane na první vodič a maximum na poslední vodič. To můžeme dokázat následujícím způsobem. Po první vrstvě je minimum na některém z vodičů 1, 3 nebo 5. Po druhé vrstvě je na vodiči 1 nebo 5 a po třetí vrstvě je určitě na vodiči 1. Podobně maximum je po první vrstvě na vodiči 2, 4 nebo 6, po druhé vrstvě na vodiči 2 nebo 6 a po třetí vrstvě musí být na vodiči 6.

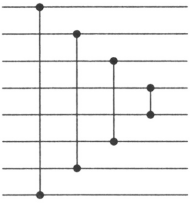
První a poslední vodič tedy po třetí vrstvě obsahují správné hodnoty. Další tři vrstvy setřídí prostřední čtyři vodiče. Čtvrtá vrstva umístí minimum z druhého a třetího vodiče na druhý vodič. Podobně pátý vodič obsahuje maximum ze čtvrtého a pátého vodiče. Pátá vrstva původní sítě se vynechává. V šesté vrstvě porovnáme mezi sebou maxima a minima z předcházejícího kroku, tj. po této vrstvě už druhý vodič obsahuje minimum a pátý vodič maximum ze zbývajících čtyř prvků. Zbývá dotřídit prostřední dva vodiče, což provede komparátor v poslední vrstvě.

c) Označme x_i číslo na vstupu i. Dokážeme nejprve následující pozorování: pro každé $i \leqq n$ jedno z čísel x_i a x_{2n-i+1} patří mezi n nejmenších čísel a druhé mezi n největších. Předpokládejme, že obě čísla x_i a x_{2n-i+1} patří mezi n nejmenších čísel. Potom čísla $x_1, x_2, \ldots, x_{i-1}$ patří také mezi n nejmenších čísel, neboť jsou menší než x_i. Podobně také čísla $x_{n+1}, x_{n+2}, \ldots, x_{2n-i}$ patří mezi n nejmenších čísel, neboť jsou menší než x_{2n-i+1}. To znamená, že jsme celkově nalezli $i + (2n - i + 1 - n) = n + 1$ čísel, která patří mezi n nejmenších, což je spor. Alespoň jedno z čísel x_i a x_{2n-i+1} musí tedy patřit mezi n největších čísel.

Předpokládejme nyní, že obě čísla x_i a x_{2n-i+1} patří mezi n největších čísel. Potom ale také čísla $x_{i+1}, x_{i+2}, \ldots, x_n$ a $x_{2n-i+2}, x_{2n-i+3}, \ldots, x_{2n}$ patří mezi n největších čísel, neboť jsou větší než x_i nebo x_{2n-i+1}. Celkem jsme tedy nalezli $(n - i + 1) + (2n - 2n + i) = n + 1$ čísel, která patří

mezi n největších, což je spor. Proto obě čísla nemohou zároveň patřit ani mezi n největších.

Dokázali jsme, že jedno z čísel x_i a x_{2n-i+1} patří do horní poloviny výstupů a druhé do dolní. Do horní patří to z nich, které je menší. Stačí je tedy porovnat jedním komparátorem a každé z nich se tím dostane do správné poloviny výstupů. Toto provedeme pro každou dvojici x_i a x_{2n-i+1} pro $i = 1, 2, \ldots, n$. Výsledná síť bude mít $n/2$ komparátorů v jediné vrstvě. Příklad takové sítě pro 8 vstupů ($n = 4$) ukazuje obrázek.



P – III – 1

Nejdříve si zavedeme několik pojmů, které nám usnadní vyjadřování při řešení této úlohy. Jestliže A je matice, pak A_i bude označovat její i-tý sloupec. *Signatura* prvku $a_{i,j}$ matice A je rovna hodnotě prvku $a_{i,j}$, pokud $a_{i,j}$ je číslo. Když $a_{i,j}$ je žolík, pak jeho signaturou rozumíme nejmenší číslo, které lze za žolík na pozici $a_{i,j}$ dosadit tak, aby i-tý řádek matice A tvořil neklesající posloupnost. Pokud tedy $a_{i,j}$ je žolík, pak jeho signatura je rovna největšímu z čísel mezi prvky $a_{i,1}, \ldots, a_{i,j-1}$; když mezi těmito prvky není žádné číslo, je signatura $a_{i,j}$ rovna nule. Signatura sloupce matice je *utříděná*, jestliže posloupnost signatur jeho prvků od prvního po poslední řádek je neklesající.

Po zbytek řešení je A pevně zadaná matice z naší úlohy. Předpokládejme na chvíli, že máme dány sloupce $A'_1, \ldots, A'_k$, které jsou přerovnáním prvků ve sloupcích $A_1, \ldots, A_k$ matice A. Matici A^* budeme nazývat *rozšířením* sloupců $A'_1, \ldots, A'_k$, pokud jsou splněny následující podmínky:

- ▷ matice A^* má stejné rozměry jako matice A,
- ▷ i-tý sloupec matice A^* je přerovnáním i-tého sloupce matice A,
- ▷ do matice A^* lze za žolíky doplnit celá čísla tak, aby každý její řádek tvořil neklesající posloupnost a
- ▷ prvních k sloupců matice A^* jsou sloupce $A'_1, \ldots, A'_k$.

Navrhneme algoritmus, o kterém později ukážeme, že řeší naši úlohu. Algoritmus bude postupně přerovnávat sloupce matice A od prvního po poslední. Předpokládejme, že už jsme přerovnali prvních k sloupců

a že jsme tedy již nalezli $A'_1, \ldots, A'_k$; navíc předpokládejme, že signatura každého ze sloupců $A'_1, \ldots, A'_k$ je utříděná. Popíšeme, jak přerovnáme další sloupec, tj. jak nalezneme A'_{k+1}. Prvky sloupce A'_{k+1} budeme volit v pořadí zdola nahoru: Za prvek $a'_{i,k+1}$ zvolíme největší dosud nepoužité číslo ze sloupce A_{k+1}, které je alespoň tak velké jako signatura prvku $a'_{i,k}$. Pokud takové číslo neexistuje, zvolíme jako prvek $a'_{i,k+1}$ žolík. Pokud již sloupec A_{k+1} neobsahuje žádné nepoužité žolíky, prohlásíme, že sloupce matice A nelze přerovnat. Povšimněte si, že když jsme úspěšně vytvořili sloupec A'_{k+1}, pak jeho signatura je utříděná.

DŮKAZ SPRÁVNOSTI. Je jasné, že pokud se našemu algoritmu podařilo přerovnat všechny sloupce matice A, potom výsledná matice je řešením úlohy. Zbývá ukázat, že když náš algoritmus nenalezl přerovnání sloupců matice A, pak žádné přerovnání sloupců matice A úlohu neřeší. K tomu si nejprve dokážeme dvě pomocná tvrzení:

Tvrzení 1. *Nechť $A'_1, \ldots, A'_k$ jsou přerovnání prvních k sloupců matice A, jejichž signatury jsou utříděné. Nechť A'_{k+1} je přerovnání $(k+1)$-ního sloupce matice A, které nalezl náš algoritmus. Pokud existuje rozšíření sloupců $A'_1, \ldots, A'_k$, pak existuje i rozšíření sloupců $A'_1, \ldots, A'_k, A'_{k+1}$.*

DŮKAZ. Předpokládejme, že existuje rozšíření A^* sloupců $A'_1, \ldots, A'_k$. Pokud se $(k+1)$-ní sloupec A^* shoduje s A'_{k+1}, pak tvrzení platí z triviálních důvodů. Předpokládejme tedy, že sloupce A^*_{k+1} a A'_{k+1} jsou různé.

Nechť j je číslo nejspodnějšího řádku, kde se tyto sloupce liší. Můžeme předpokládat, že A^* je mezi všemi rozšířeními sloupců $A'_1, \ldots, A'_k$ to, pro které je j nejmenší, tj. A^*_{k+1} se liší od A'_{k+1} co nejvýše. Mohou nastat dva případy:

- $a'_{j,k+1} = *$ ($a^*_{j,k+1}$ je tedy číslo). Potom ale náš algoritmus měl jako $a'_{j,k+1}$ zvolit některé číslo ze sloupce A_{k+1}: Číslo $a^*_{j,k+1}$ je totiž z triviálních důvodů větší nebo rovno signatuře prvku $a_{j,k}$ a tedy, když algoritmus volil prvek $a'_{j,k+1}$, existovalo ve sloupci A_{k+1} nepoužité číslo, které bylo větší nebo rovno signatuře prvku $a'_{j,k}$.
- $a'_{j,k+1}$ je číslo ($a^*_{j,k+1}$ je buď jiné číslo, nebo žolík). Sloupec A^*_{k+1} obsahuje číslo $a'_{j,k+1}$ na jiném než j-tém řádku; buď i číslo tohoto řádku. Platí tedy $a^*_{i,k+1} = a'_{j,k+1}$ a $i < j$.
  Vyměňme nyní řádky i a j ve sloupcích $k+1$ až n v matici A^*; označme A^{**} výslednou matici. Dokážeme, že A^{**} je také rozšíření $A'_1, \ldots, A'_k$. Toto je ale ve sporu s volbou A^* jako rozšíření sloupců $A'_1, \ldots, A'_k$ takového, že se sloupce A^*_{k+1} a A'_{k+1} shodovaly na co nejvíce pozicích zdola.

Když dosadíme za prvky matice A^* jejich signatury, budou všechny řádky neklesající. Dosaďme ty samé hodnoty za odpovídající žolíky v matici A^{**} (tj. hodnoty z i-tého řádku do j-tého a naopak). Jediné dvě dvojice prvků, kde by podmínka monotonie mohla být po výměně porušena, jsou prvky v k-tém a $(k+1)$-ním sloupci v i-tém nebo v j-tém řádku. Protože ale sloupec A'_k je utříděný, je signatura prvku $a'_{i,k}$ menší nebo rovna signatuře prvku $a'_{j,k}$ a tedy i-tý řádek je neklesající. Prvek $a^*_{i,k+1} = a'_{j,k+1}$ je číslo a signatura $a^*_{j,k} = a^{**}_{j,k}$ je menší nebo rovna číslu $a^*_{i,k+1} = a^{**}_{j,k+1}$. Tedy i j-tý řádek je neklesající.

Tvrzení 2. *Nechť jsou dána přerovnání $A'_1, A'_2, \ldots, A'_k$ prvních k sloupců matice A a nechť je signatura sloupce A'_k utříděná. Pokud se našemu algoritmu nepodaří vytvořit sloupec A'_{k+1}, pak neexistuje rozšíření sloupců $A'_1, A'_2, \ldots, A'_k$.*

Důkaz. Označme j řádek, na kterém se náš algoritmus zastavil. Mezi dosud nepoužitými prvky sloupce A_{k+1} tedy nejsou již žádné žolíky a všechna jeho čísla jsou menší než signatura $a'_{j,k}$. To ale znamená, že sloupec A_{k+1} obsahuje pouze $m-j$ (m je počet řádků matice A) žolíků a čísel, jež jsou větší nebo rovna signatuře prvku $a'_{j,k}$. Pokud ale existuje rozšíření A^* sloupců $A'_1, A'_2, \ldots, A'_k$, pak všechny prvky $a^*_{j,k+1}, \ldots, a^*_{m,k+1}$ jsou buď žolík, nebo čísla větší nebo rovna signatuře prvku $a'_{j,k}$ (číslo $a^*_{i,k+1}$ v i-tém řádku je větší nebo rovno signatuře $a'_{i,k}$ a sloupec A'_k je utříděný). Potom ale sloupec A_k obsahuje alespoň $m-j+1$ žolíků a čísel, která jsou větší nebo rovna signatuře prvku $a'_{j,k}$, což jak víme není pravda.

Pomocí těchto dvou tvrzení již snadno dokážeme správnost našeho algoritmu. Pokud pro 0 sloupců existuje rozšíření, pak podle Tvrzení 2 nalezne náš algoritmus sloupec A'_1 a pro něj též existuje rozšíření tentokrát podle Tvrzení 1. Nyní podle Tvrzení 2 nalezne náš algoritmus sloupec A'_2 a pro sloupce A'_1, A'_2 existuje rozšíření podle Tvrzení 1, atd. Pokud tedy existuje řešení, náš algoritmus ho nalezne.

Odhad časové a paměťové složitosti: Nechť vstupní matice A má n sloupců a m řádků. Na setřídění čísel v každém z n sloupců potřebujeme čas $O(m \log m)$. Zbylé operace již můžeme vykonat v čase $O(m)$ (pro jeden sloupec). Celková časová složitost algoritmu je tedy $O(nm \log m)$. Paměťová složitost je $O(mn)$.

P – III – 2

Úlohu budeme řešit pomocí dynamického programování. Trasu, která splňuje všechny podmínky kladené na vyhovující trasy s výjimkou podmínky, že musí končit v nejníže položeném orientačním bodě, budeme nazývat *částečná trasa*. Náš program bude počítat počet částečných tras, které končí v jednotlivých orientačních bodech.

Nejdříve si popíšeme algoritmus, který pracuje v čase $O(N^3)$, kde N je počet orientačních bodů. Utřídíme orientační body sestupně podle jejich nadmořské výšky a očíslujeme je v získaném pořadí od 1 do N. Pro i, j $(1 \leqq i < j \leqq N)$ bude hodnota $a[i,j]$ určovat počet částečných tras, které mají i jako předposlední orientační bod a končí v bodě j. Pro $i = 1$ bude $a[i,j] = 1$, protože trasa z orientačního bodu 1 do orientačního bodu j je právě jedna. Předpokládejme, že jsme již spočítali hodnoty $a[i',j']$ pro $j' < j$ a chceme spočítat hodnoty $a[i,j]$ pro $1 \leqq i \leqq N$. Pokud má uvažovaná částečná trasa končit úsekem i,j, musí do bodu i přijít z bodu k (s nadmořskou výškou větší než je výška bodu i) takového, že úhel otočení ze směru $\vec{ki}$ do směru $\vec{ij}$ je nejvýše 45°. Označme $S(i,j)$ množinu všech takových bodů k. Potom hodnota $a[i,j]$ je určena následujícím vzorečkem:

$$a[i,j] = \sum_{k \in S(i,j)} a[k,i].$$

Algoritmus pracující v kubickém čase lze nyní již snadno sestrojit. Úvodní setřídění orientačních bodů dle nadmořských výšek lze provést v čase $O(N \log N)$. Hodnoty $a[i,j]$ budeme počítat podle rostoucí hodnoty indexu i. Pro každou z $O(N^2)$ dvojic i a j existuje pouze $O(N)$ čísel k — otestování, zda $k \in S(i,j)$ provedeme pomocí funkce `uhel` ze zadání úlohy. Počet hledaných tras pak určíme jako součet hodnot $a[j,N]$ přes všechna j, $1 \leqq j < N$. Náš algoritmus zjevně pracuje v čase $O(N^3)$ a v prostoru $O(N^2)$.

Právě sestrojený algoritmus ještě dále zrychlíme pomocí podobného triku, jaký jsme použili již v minulých kolech. Pro bod i setřídíme body $k, k < i$, podle směrů $\vec{ki}$ ve směru hodinových ručiček, a též body $j, j > i$, podle směrů $\vec{ij}$. Pro každé $j > i$, je množina $S(i,j)$ tvořena souvislým úsekem bodů k, $k < i$, v právě zavedeném uspořádání — nechť $l(i,j)$ a $p(i,j)$ jsou krajní body tohoto souvislého úseku (předpokládejme, že je neprázdný). Pro pevný bod i budeme k $a[i,j]$ přičítat součet hodnot od $a[l(i,j),i]$ do $a[p(i,j),i]$, kde j $(j > i)$ budeme procházet v pořadí podle

výše uvedeného uspořádání. Jak se směr $\vec{ij}$ postupně natáčí, hodnoty $l(i,j)$ a $p(i,j)$ se postupně mění (ale stále stejným směrem). Hodnotu součtu prvků mezi $a[l(i,j),i]$ a $a[p(i,j),i]$ si můžeme pamatovat v pomocné proměnné a při změně $l(i,j)$ nebo $p(i,j)$ ji patřičně upravit. Na výpočet v pevném bodě i takto budeme potřebovat kromě úvodního třídění čas $O(N)$.

Třídění podle směrů (úhlů) spotřebuje v každém z N bodů čas $O(N \log N)$. Výsledná časová složitost našeho algoritmu tedy bude $O(N^2 \log N)$ a paměťová bude $O(N^2)$. Při implementaci algoritmu je třeba dávat pozor na několik okrajových případů, zejména na již výše zmiňovaný případ, že množina $S(i,j)$ je prázdná. Kvůli úspoře místa bylo ve vzorovém programu použito kvadratického třídícího algoritmu namísto optimálního algoritmu pracujícího v čase $O(N \log N)$.

P – III – 3

Nejdříve si předvedeme jednodušší řešení, které pro každou permutaci vytvoří síť s $O(\log n)$ vrstvami a $O(n \log n)$ komparátory. Později toto řešení vylepšíme, aby používalo pouze $O(n)$ komparátorů. Časová složitost obou algoritmů bude $O(n \log n)$.

Nechť permutace na vstupu je $A = a_1, a_2, \ldots, a_n$ a zvolme $k := \lfloor n/2 \rfloor$ ($\lfloor x \rfloor$ je dolní celá část čísla x). Po průchodu první vrstvou budou na prvních k vodičích čísla $1, 2, \ldots, k$ (ne nutně setříděná) a na zbylých $n-k$ vodičích čísla $k+1, k+2, \ldots, n$ (opět ne nutně setříděná). První vrstva bude vypadat následovně: Nechť S_1 je množina vodičů z horní poloviny, na kterých jsou čísla z dolní poloviny, tj. $S_1 = \{i : i \leqq k \,\&\, a_i > k\}$. Podobně S_2 bude množina vodičů patřících do dolní poloviny, na kterých jsou čísla z horní poloviny, tj. $S_2 = \{i : i > k \,\&\, a_i \leqq k\}$. Zřejmě $|S_1| = |S_2|$. Pomocí komparátorů spojíme dvojice vodičů, z nichž vždy jeden je z množiny S_1 a druhý S_2. Takto vytvoříme první vrstvu sítě a je zřejmé, že tato vrstva má požadovanou vlastnost.

Nyní rozdělíme n vodičů na dvě skupiny: horních k vodičů a dolních $n-k$ vodičů. Právě popsaný postup zopakujeme na každé z těchto dvou skupin zvlášť a takto vytvoříme druhou vrstvu. Na vodičích z každé čtvrtiny budou čísla z této čtvrtiny. Takto budeme pokračovat, dokud v každé části, na které máme vodiče rozděleny, nezbude jediný vodič. Počet vrstev takto vytvořené sítě bude $O(\log n)$ (velikosti částí se zmenší v každé vrstvě na polovinu) a počet komparátorů této sítě bude $O(n \log n)$.

Dále si popíšeme konstrukci s $O(n)$ komparátory. Ta funguje podobně, ale dvojice vodičů z množin S_1 a S_2 vybereme šikovnějším způsobem. Permutaci si můžeme představit jako orientovaný graf na vrcholech $1, \ldots, n$, kde z vrcholu i vede právě jedna hrana, a to do vrcholu a_i. Zjevně z každého vrcholu vychází právě jedna hrana a právě jedna hrana do něj vchází. Cykly v takto vytvořeném grafu budeme nazývat *cykly* permutace. Např. permutace $(7, 5, 4, 1, 2, 6, 8, 3)$ má 3 cykly: $(1 \to 7 \to 8 \to 3 \to 4 \to 1)$, $(2 \to 5 \to 2)$, $(6 \to 6)$.

V naší komparátorové síti bychom chtěli přesunout číslo z vodiče i na vodič a_i. Protože čísla i a a_i jsou ve stejném cyklu, je zbytečné pomocí komparátorů spojovat vodiče z různých cyklů permutace A. Na začátku v lineárním čase proto rozdělíme permutaci na její cykly a v každém z nich budeme řešit úlohu samostatně. Může se ale samozřejmě stát, že permutace je tvořena jen jedním cyklem a tímto dělením na menší úlohy si tedy nepomůžeme.

Zvolme jeden pevný cyklus $(x_1 \to x_2 \to \ldots \to x_l \to x_1)$ permutace A a nechť $k := \lfloor l/2 \rfloor$ (k bude mít opět roli „poloviny“ délky cyklu). Nejmenších k čísel tohoto cyklu budeme nadále nazývat *malá* čísla a zbylých $n - k$ čísel budeme nazývat *velká* čísla. Např. pro cyklus $(1 \to 7 \to 8 \to 3 \to 4 \to 1)$ je $k = 2$, čísla 1 a 3 jsou malá a čísla 4, 7 a 8 jsou velká.

Podívejme se na souvislé úseky velkých a malých čísel v našem cyklu; souvislý úsek může pokračovat i z x_l do x_1. Nechť x_i je poslední číslo v některém z úseků velkých čísel; za x_i tedy následuje úsek malých čísel a nechť x_j je poslední číslo v tomto úseku.

Vodič x_i obsahuje číslo a_{x_i}, které je malé, a naopak vodič x_j obsahuje číslo a_{x_j}, které je velké. Kdybychom spojili x_i-tý a x_j-tý vodič komparátorem, dojde k výměně, protože vodič s malým číslem x_i obsahuje velké číslo a_{x_i} a naopak vodič s velkým číslem x_j obsahuje malé číslo a_{x_j}.

Pro každý souvislý úsek velkých čísel takto vytvoříme komparátor, který spojí vodič odpovídající poslednímu číslu tohoto úseku s vodičem, který odpovídá poslednímu číslu následujícího úseku malých čísel. V našem příkladu $(1 \to 7 \to 8 \to 3 \to 4 \to 1)$ čísla 7 a 8 tvoří první souvislý úsek velkých čísel a 3 následující úsek malých čísel. Do sítě tedy přidáme komparátor mezi vodiče 8 a 3. Další souvislý úsek velkých čísel je tvořen číslem 4 a úsek malých čísel pouze číslem 1. Do naší sítě tedy přidáme komparátor mezi vodiče 1 a 4.

Takto vytvoříme komparátory první vrstvy. Po průchodu první vrstvou se původní velký cyklus rozpadne na několik menších. Každý souvislý

úsek malých čísel bude tvořit samostatný cyklus, naopak všechna velká čísla budou obsažena v jednom cyklu (nakreslete si obrázek!). Pokud jsme použili p komparátorů, vytvoříme z původního jednoho cyklu $p+1$ nových cyklů. Nyní obdobným postupem nalezneme cykly permutace po průchodu prvním cyklem a vytvoříme druhou vrstvu. Skončíme, když všechny cykly jsou jednoprvkové. Protože každý komparátor „přidá" jeden cyklus, bude mít výsledná síť nejvýše $n-1$ komparátorů. Každá vrstva zmenší velikost největšího cyklu zhruba na polovinu, počet vrstev je proto logaritmický v n.

Výše popsaný algoritmus může každou z $O(\log n)$ vrstev vytvořit v čase $O(n)$: Nejdříve v čase $O(n)$ rozložíme permutaci na cykly. To lze udělat tak, že si vytvoříme pomocné pole velikosti n, které na začátku vynulujeme. Vezmeme nejmenší číslo i takové, že i-tý prvek tohoto pole je stále nulový, a nalezneme jeho cyklus, tj. procházíme posloupnost i, a_i, a_{a_i} atd., dokud se nevrátíme zpět do i. Všem číslům této posloupnosti uložíme do pomocného pole číslo i. Na konci bude pomocné pole obsahovat pro každý prvek číslo nejmenšího prvku v jeho cyklu. Když jsme nalezly všechny cykly, spočítáme si jejich velikosti a rozdělíme čísla v nich na velká a malá. Tu je třeba postupovat opatrně — musíme totiž (rychle) určit hranici, která oddělí velká a malá čísla v jednom cyklu. Jednou z možností je použít lineární algoritmus na hledání mediánu. Jednodušší řešení je následující: Budeme procházet naše pomocné pole od nejmenšího indexu k největšímu — u každého cyklu budeme jeho čísla označovat jako malá, dokud nenavštívíme polovinu jeho prvků (pro každý cyklus máme zvláštní čítač, kolik jeho prvků jsme již navštívili) a pak prvky tohoto cyklu budeme označovat jako velká. Tento průchod, na který spotřebujeme čas $O(n)$, nám naráz rozdělí čísla ve všech cyklech na velká a malá. Po vytvoření nové vrstvy nesmíme zapomenout spočítat, jak se nám změnila naše permutace po průchodu touto vrstvou.

P – III – 4

Úlohu si nejdříve přeformulujeme do řeči teorie grafů. Uzly kanalizační sítě budeme nazývat *vrcholy*, potrubí vedoucí mezi nimi *hrany* a celá kanalizační síť pak pro nás bude *graf*. *Sledem* budeme rozumět posloupnost ne nutně různých vrcholů a hran $v_0, e_1, v_1, e_2, \ldots, v_{k-1} e_k, v_k$ takovou, že hrana e_i spojuje vrcholy v_{i-1} a v_i. Sled nazveme *tahem*, pokud se v něm žádná hrana nevyskytuje dvakrát.

Naše úloha tedy požaduje nalézt v zadaném grafu G nejmenší množinu tahů takovou, že každá hrana je obsažena v právě jednom z nich. Pokud si představíme náš graf G jako obrázek, chceme ho nakreslit co nejmenším počtem tahů.

Nejdříve si uvědomíme několik jednoduchých skutečností. Pokud je graf nesouvislý, můžeme úlohu řešit pro každou jeho komponentu souvislosti zvlášť. Připomeňme si, že komponenta souvislosti grafu je maximální množina vrcholů taková, že mezi každými dvěma z nich vede sled. Graf je souvislý, pokud je tvořen jedinou komponentou souvislosti. Nadále tedy stačí řešit případ, že zadaný graf je souvislý. Stupeň vrcholu grafu je počet hran, které z něho vycházejí. Součet stupňů všech vrcholů grafu je sudý, neboť každá hrana tento součet zvýší o dva (u každého ze svých dvou konců o jedna). Protože součet stupňů všech vrcholů je sudý, má každý graf sudý počet vrcholů lichého stupně. Nechť tedy náš graf má $2k$ vrcholů lichého stupně.

V každém z těchto $2k$ vrcholů alespoň jeden z hledaných tahů končí nebo začíná. Nechť w je vrchol lichého stupně, kde žádný tah ani nezačíná, ani nekončí. Každý tah musí obsahovat sudý (třeba i nulový) počet hran vycházejících z vrcholu w a protože všechny tahy dohromady obsahují každou hranu grafu právě jednou, musí být stupeň vrcholu w sudý, což není. Potom ale libovolné řešení musí obsahovat alespoň k tahů (pro $k = 0$ alespoň jeden tah).

Nejprve vyřešíme případ, že všechny vrcholy grafu mají sudé stupně, tj. $k = 0$. Obecný případ s nenulovým k vyřešíme později. Pro $k = 0$ dokážeme, že existuje uzavřený tah obsahující všechny hrany. Připomeňme si, že tah je uzavřený, pokud začíná a končí ve stejném vrcholu. Vezměme nejdelší tah v našem grafu, tj. ten, co obsahuje co nejvíce hran. Takový tah je zřejmě uzavřený: Kdyby nebyl, pak by z jeho posledního vrcholu, označme ho w, vycházel lichý počet hran obsažených v našem tahu, ale protože stupeň w je sudý, existovala by i nepoužitá hrana vycházející z vrcholu w a náš tah by šel prodloužit. Tedy nejdelší tah je nutně uzavřený. Pokud tento tah neobsahuje všechny hrany, pak existuje vrchol w, ze kterého vychází jak hrana, která je obsažena v našem tahu, tak i hrana, která není v našem tahu (uvědomme si, že graf je souvislý). Protože náš tah je uzavřený, můžeme předpokládat, že w je jeho první a zároveň poslední vrchol a tento tah lze potom prodloužit přidáním, nepoužité hrany, která vede z vrcholu w. Právě jsme tedy ukázali, že nejdelší tah v souvislém grafu, jehož všechny stupně jsou sudé, obsahuje každou hranu právě jednou (jinak by šel prodloužit).

Zpět k našemu problému: Jak nakreslit souvislý graf s $2k$ vrcholy lichého stupně pomocí k tahů? Přidejme do našeho grafu k hran, které budou spojovat dvojice vrcholů lichého stupně. Takto vytvořený graf je souvislý a všechny jeho vrcholy mají sudé stupně. Existuje v něm tedy tah, který obsahuje každou hranu právě jednou. Pokud z tohoto tahu vynecháme k přidaných hran, pak se nám rozpadne na k tahů — sestrojili jsme tedy hledaných k tahů.

Zbývá vyřešit jediný úkol — efektivně nalézt uzavřený tah v grafu jehož všechny stupně jsou sudé. Zbývající operace, tj. rozložit graf na komponenty souvislosti a přidat hrany mezi vrcholy lichého stupně, jistě zvládneme v lineárním čase v počtu vrcholů a hran. Algoritmus bude postupně přidávat hrany do vytvářeného tahu v grafu; jeho jádrem bude rekurzivní procedura `vytvor_tah`. Začneme vytvářet tah ve vrcholu v, dokud nepřijdeme do vrcholu, z něhož už nevede žádná nepoužitá hrana. Protože stupně všech vrcholů jsou sudé, musí být tímto vrcholem opět vrchol v. Máme tedy nějaký uzavřený tah. Pokud existuje vrchol u, jehož všechny hrany jsme ještě nepoužili, zavoláme pro něj proceduru `vytvor_tah`. Takto nalezneme tah, který prochází vrcholem u a obsahuje pouze dosud nepoužité hrany. Vrchol u nyní nemá žádné nepoužité hrany a nově nalezený tah spojíme s původním tahem. Takto pokračujeme, dokud existuje vrchol u, jehož některá hrana je nepoužitá.

Nyní k samotné implementaci výše popsaného algoritmu. V programu budeme mít pro každý vrchol uložen seznam jeho hran a ukazatel do tohoto seznamu na takovou hranu, že všechny hrany před ní jsou již použité. Pokud budeme potřebovat najít dosud nepoužitou hranu, budeme tento ukazatel posunovat v seznamu, dokud takovou hranu nenalezneme. Pokud jsou všechny hrany použité, ukazatel se posune až na konec seznamu. Takto u každého vrcholu spotřebujeme dohromady čas úměrný jeho stupni, a tedy celkově čas $O(m)$ pro všechny vrcholy dohromady. Samotné tahy budeme reprezentovat jako seznamy hran — to nám umožní tahy nejen rychle vytvářet, ale i spojovat. Náš algoritmus tedy nejdříve nalezne počáteční tah, např. z prvního vrcholu. Potom budeme procházet vrcholy na tomto tahu, dokud nenarazíme na první vrchol s nějakou dosud nepoužitou hranou. Zavoláme na tento vrchol proceduru `vytvor_tah` a spojíme tah, který tato procedura nalezne, s původním tahem. Všimněte si, že při vhodné implementaci procedury `vytvor_tah` nebude již třeba pro vrcholy na přidávaném tahu kontrolovat, zda jsou všechny jejich hrany použity (pokud to zkontroluje sama volaná procedura `vytvor_tah`). Cel-

ková časová a paměťová složitost našeho algoritmu tedy bude $O(m)$, kde m je počet hran.

Kvůli zjednodušení vzorového programu náš program nespojuje vrcholy lichého stupně jen v rámci jedné komponenty — to ale zřejmě nebude mít vliv na počet nalezených tahů.

P – III – 5

Myšlenka řešení této úlohy je velmi jednoduchá. Vygenerujeme postupně všechny pozice s nejvýše N kameny, ze kterých se dá cílová pozice dosáhnout. Budeme si pamatovat všechny již vygenerované pozice a kdykoliv vygenerujeme nějakou další, nejdříve si zkontrolujeme, zda již mezi dříve vytvořenými pozicemi náhodou není. Není-li tomu tak, zapamatujeme si ji a zvýšíme počítadlo vyhrávajících pozic s nejvýše N kameny o jedničku.

Pozice budeme generovat v pořadí od cílové pozice. Pokud posledním tahem byl skok doprava z políčka i na políčko $i+2$, pak jsou nyní políčka i a $i+1$ prázdná a naopak políčko $i+2$ obsahuje hrací kámen. Pozice, ze kterých se dá dostat do pozice p skokem doprava, jsou právě ty, které vzniknou záměnou řetězce 001 v popisu pozice p řetězcem 110 (představujeme si, že pozice je popsána posloupností nul a jedniček, kde nula reprezentuje prázdné políčko a jednička reprezentuje obsazené políčko). Podobně ty pozice, ze kterých se dá dostat do p skokem doleva, jsou právě ty, které vzniknou záměnou řetězce 100 řetězcem 011.

Výše popsaným postupem můžeme tedy pro pozici p vygenerovat všechny pozice, ze kterých se lze do p dostat jedním tahem. Popíšeme si proceduru `generuj`, která pro zadanou pozici p vygeneruje všechny pozice, ze kterých se dá do pozice p dostat. Výše uvedeným postupem tato procedura bude vytvářet pozice q, z nichž se dá přejít do p jedním tahem. Pokud q ještě nebyla nalezena, označíme q jako nalezenou a rekurzivně na ni zavoláme proceduru `generuj`. Potřebujeme umět rychle ověřovat, zda již pozice q byla někdy dříve vygenerována. Vhodných datových struktur je několik, např. hašovací tabulky, různé varianty vyhledávacích stromů atd. My použijeme datovou strukturu nazývanou trie.

Trie je datová struktura na uchovávání řetězců nad konečnou abecedou, v našem případě to budou řetězce nul a jedniček. Trie je tvořena stromem, kde každý vrchol má nejvýše tolik synů, jaká je velikost abecedy, tj. v našem případě nejvýše dva syny. Cesty od kořene stromu odpovídají řetězcům nad abecedou. V našem případě to bude tak, že

pokud i-tá hrana cesty vede do levého syna, je i-tý znak řetězce 0, a pokud vede do pravého syna je tento znak 1. Každý vrchol w ve stromě obsahuje ukazatele na své syny (pokud nějaké má) a jeden bit udávající, zda řetězec, kterému tento vrchol odpovídá, byl do trie vložen či nikoliv. Na začátku je trie tvořena pouze kořenem, jehož bit je nastaven na nulu. Slovo do trie vložíme tak, že jdeme po hranách odpovídajících znakům vkládaného slova a pokud už nemůžeme z nějakého vrcholu dále pokračovat, jednoduše vytvoříme nový vrchol, připojíme ho jako syna tohoto vrcholu a pokračujeme do něj. Až nalezneme vrchol odpovídající celému vkládanému slovu, nastavíme jeho bit na jedničku. Vyhledávání řetězce je stejně jednoduché: Z kořene se opět necháme navigovat pomocí znaků hledaného řetězce. Pokud už dále nemůžeme pokračovat, pak hledaný řetězec ve stromě není. Jinak jsme nalezli vrchol, který odpovídá vyhledávanému řetězci, a podle jeho bitu poznáme, zda je řetězec ve stromě obsažen nebo není. Poznamenejme, že existují i chytřejší implementace této datové struktury než ta, kterou jsme si zde právě popsali.

Časová a paměťová složitost našeho programu závisí na počtu P hledaných pozic. Pro každou z nich vytvoříme v našem stromě nejvýše K uzlů — prostorová složitost algoritmu je tedy nejvýše $O(PK)$. Navíc pro každou z nich musíme vygenerovat všech nejvýše $2k-4$ jejich přímých předchůdců a pro každého z těchto předchůdců otestovat v čase $O(K)$, zda je v trie již obsažen. Celková časová složitost algoritmu tedy bude $O(PK^2)$.