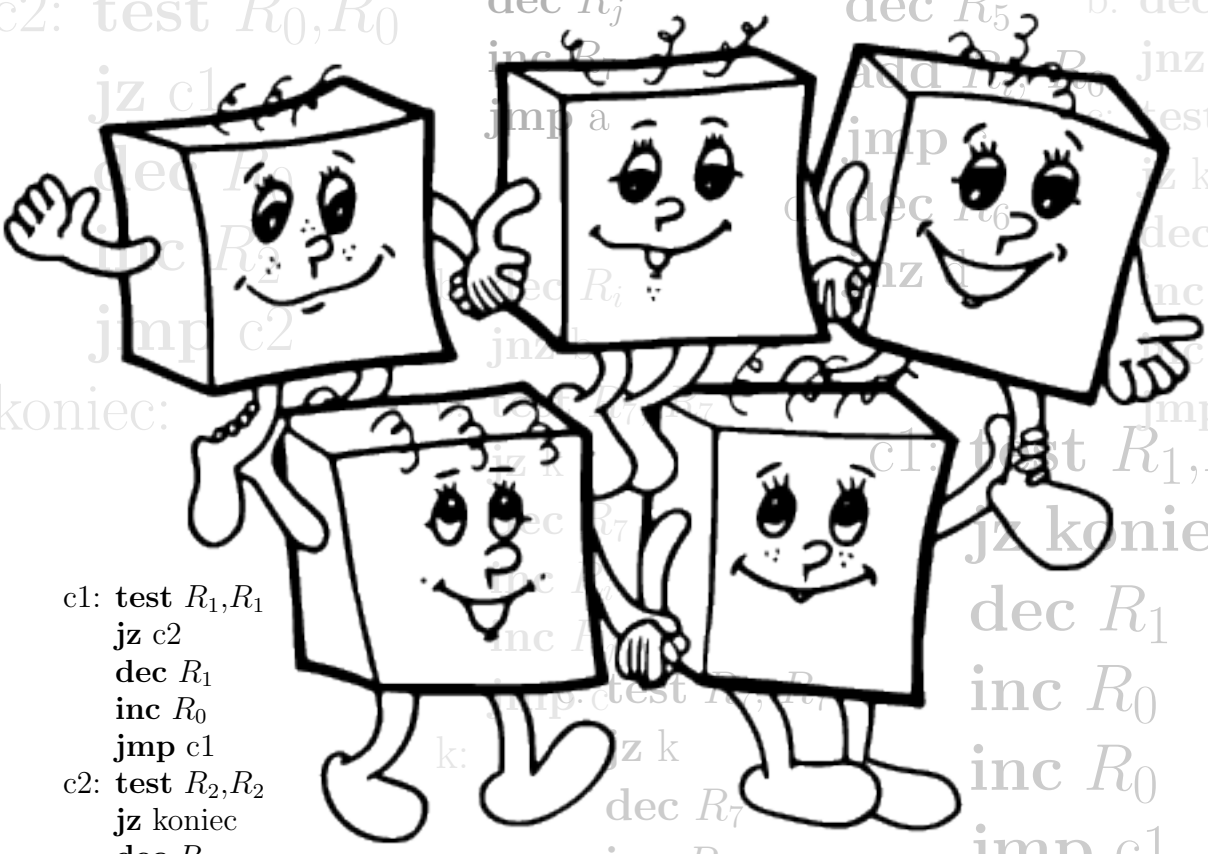
a: **sub** R_1, R_0
sub R_1, R_0
jz koniec
dec R_1
inc R_0
jmp a
koniec:

OLYMPIÁDA V INFORMATIKE



c1: **test** R_1,R_1
jz c2
dec R_1
inc R_0
jmp c1
c2: **test** R_2,R_2
jz koniec
dec R_2
inc R_0
jmp c2
koniec:

22. ročník 2006/2007

ISBN 80-8072-053-3

9 788080 720537

Dvadsiaty druhý ročník Olympiády v informatike

RNDr. Michal Forišek, Bc. Jakub Kováč, Slovenská komisia OI
Dvadsiaty druhý ročník Olympiády v informatike
ISBN 80-8072-053-3

Obsah

O priebehu 22. ročníka Olympiády v informatike 3
Zadania domáceho kola kategórie A 5
Zadania domáceho kola kategórie B 15
Zadania krajského kola kategórie A 21
Zadania krajského kola kategórie B 27
Zadania celoštátneho kola kategórie A 32
Riešenia domáceho kola kategórie A 44
Riešenia domáceho kola kategórie B 63
Riešenia krajského kola kategórie A 68
Riešenia krajského kola kategórie B 79
Riešenia celoštátneho kola kategórie A 90
Výsledky krajského kola kategórie A 113
Výsledky celoštátneho kola kategórie A 116
Výsledky Česko-Poľsko-Slovenského prípravného stretnutia 117
Výsledky Medzinárodnej olympiády v informatike 118
Výsledky Stredoeurópskej olympiády v informatike 119

O priebehu 22. ročníka Olympiády v informatike

Po dvadsaťjeden rokoch prežitých pod krídlami Matematickej olympiády sa jej programátorská kategória P v školskom roku 2006/07 osamostatnila. Predmetová olympiáda mladých programátorov tak po prvýkrát prebehla pod novou hlavičkou – Olympiáda v informatike (OI).

V uvedenom školskom roku postupne začínala vznikať organizačná štruktúra novej samostatnej súťaže. Mnoho vecí ešte prebiehalo s podporou Matematickej olympiády. Napriek rozdeleniu na dve samostatné súťaže (ktoré bolo len prirodzeným dôsledkom toho, že skutočne ide o dve rôzne predmetové olympiády) plánujú obe olympiády aj v budúcnosti úzko spolupracovať a minimálne raz ročne organizovať spoločné zasadnutie komisií oboch olympiád.

Na konci prvého samostatného ročníka OI ešte stále nebola oficiálne Ministerstvom školstva SR menovaná Slovenská komisia OI (SK OI), a súťaž teda fungovala len vďaka dobrej vôli jej zanietených organizátorov, ktorým za to patrí veľká vďaka. V čase vydania tejto ročenky už SK OI existuje a pôsobí v nasledujúcom zložení:

- doc. RNDr. Gabriela Andrejková, CSc.,
  predsedkyňa, ÚI PF UPJŠ, Košice
- RNDr. Michal Forišek, podpredseda, KI FMFI UK, Bratislava
- RNDr. Andrej Blaho, FMFI UK, Bratislava
- Mgr. Juliana Lipková, FMFI UK, Bratislava
- Bc. Ján Katrenič, PF UPJŠ, Bratislava
- Bc. Lukáš Poláček, FMFI UK, Bratislava
- PaedDr. Miloslava Sudolská, PhD.,
  KI FPV UMB, krajská predsedkyňa pre BB
- RNDr. Eva Hanulová, Gym. J. Hronca, krajská predsedkyňa pre BA
- RNDr. Rastislav Krivoš-Belluš,
  Ústav informatiky PF UPJŠ, krajský predseda pre KE
- prof. Ing. Veronika Stoffová, CSc.,
  KI PF UJS, krajská predsedkyňa pre NR

- Mgr. Mária Majherová, Ph.D.,
  Katedra matematiky FHPV PU, krajská predsedkyňa pre PO
- Ing. Andrea Julény, KI FMech TnUAD, krajská predsedkyňa pre TN
- Mgr. Blanka Thomková, Gym. Jána Hollého, krajská predsedkyňa pre TT
- RNDr. Peter Varša, Ph.D., KI FRI ŽU, krajský predseda pre ZA

Nový názov a nové logo však neboli jedinými zmenami v tomto ročníku OI. Najdôležitejšou novinkou bolo zavedenie novej kategórie B. Táto kategória je určená pre žiakov, ktorí ani v školskom roku súťaže, ani v nasledujúcom školskom roku ešte nematurujú. (T.j. na klasickom štvorročnom gymnáziu je kategória B určená pre prvé dva ročníky.) Motiváciou pre založenie tejto kategórie bola snaha Slovenskej komisie OI pomôcť mladším žiakom prekonať obrovskú vzdialenosť medzi úrovňou, na ktorej je výučba programovania na stredných školách a úrovňou, na ktorej sú úlohy kategórie A (a následných medzinárodných súťaží).

Kategória B voľne zodpovedá (napr. približnou náročnosťou úloh) začiatočníckej kategórii Z v Korešpondenčnom seminári z programovania (KSP) – korešpondenčnej súťaže, ktorej jedným z cieľov je práve dlhodobá príprava mladých olympionikov. Paralelne s dvadsiatym druhým ročníkom OI prebehol dvadsiaty štvrtý ročník KSP.

Víťazi celoštátneho kola kategórie A boli už tradične pozvaní na týždňové výberové sústredenie, na ktorom boli vybrané štvorice reprezentantov Slovenska na Medzinárodnej olympiáde v informatike (IOI) v chorvátskom Záhrebe a na Stredoeurópskej olympiáde v informatike (CEOI) v Brne. Šestica riešiteľov dostala tiež príležitosť zúčastniť sa týždňového Česko-poľsko-slovenského prípravného sústredenia (CPSPC), ktoré sa v tomto roku konalo v Prahe. Výsledky našich súťažiacich na týchto súťažiach nájdete na konci ročenky.

Priebeh Olympiády v informatike samozrejme zahŕňa obetavú prácu mnohých ľudí, ktorí sa na jej chode podieľajú v jednotlivých krajoch, na školách, na krajských školských úradoch a centrách voľného času... Bez týchto ľudí by naša olympiáda nemohla fungovať a aspoň touto cestou im všetkým ďakujeme a dúfame, že aspoň rovnako dobre bude naša spolupráca fungovať aj v budúcnosti.

Michal Forišek, podpredseda SK OI

Zadania domáceho kola kategórie A

A-I-1 Rozvoz pizze

Marcove dve životné lásky boli kulinárstvo a cyklistika. A ako to tak chodí, jedného dňa prišiel na to, že môže obe uplatniť na svoju obživu, a založil si firmu na výrobu a rozvoz pizze. Plánuje ju prevádzkovať tak, že vždy najskôr nazbiera objednávky, a keď ich už bude mať dosť, napečie pizzu, sadne na bicykel a pôjde ju rozvážať.

Marco zatiaľ vie piecť len jediný druh pizze. Keďže si ale uvedomil, že tým konkurenciu nepredbehne, rozhodol sa nalákať zákazníkov na to, že bude prijímať objednávky aj na šestinové časti pizze. Bude sa teda dať objednať napríklad 1/6, 4/6 alebo 15/6 pizze.

Navyše Marco vyhlásil, že pizzu nebude pred doručením krájať viac ako je nutné (aby si ju zákazník mohol nakrájať podľa svojho gusta). Presnejšie, zákazník dostane čo najviac celých pízz a jeden kus tvoriaci necelú časť jeho objednávky. Teda napríklad zákazník, ktorý si objedná 15/6 pizze, dostane dve celé pizze a ešte jeden kus veľkosti pol pizze.

Presne v okamihu, kedy mu priniesli z tlačiarne rozmnožené reklamné letáky, si Marco uvedomil, že nebude vôbec ľahké skombinovať objednávky tak, aby mu nezostala kopa kúskov pizze, ktoré nik nechce. Obrátil sa preto na vás, aby ste mu pomohli.

Súťažná úloha: Napíšte program, ktorý na vstupe dostane údaje o jednotlivých objednávkach a vypočíta, koľko najmenej pízz stačí Marcovi napiecť a vhodne rozkrájať, ak chce uspokojiť všetky objednávky.

Formát vstupu: Prvý riadok vstupu obsahuje jedno celé číslo N ($1 \leq N \leq 10\,000$) – počet objednávok.

Nasleduje N riadkov. Každý z nich popisuje jednu objednávku: obsahuje jedno celé číslo c_i ($1 \leq c_i \leq 100$) – počet objednaných šestín pizze.

Formát výstupu: Na výstup vypíšte jeden riadok a v ňom jedno celé číslo p – najmenší počet pízz, ktoré stačia na splnenie všetkých objednávok, keď ich Marco upečie a vhodne rozkrája. Nezabudnite na Marcov sľub zákazníkom, že doručené časti pizze nesmú byť rozkrojené.

Príklady:

Vstup

```
3
2
2
3
```

Výstup

```
2
```

Je viac možností, ako uvedené kusy vyrobiť. Napríklad z jednej pizze vyrežeme dva kusy po 2/6 a druhú rozrežeme napoly a jednu z polovíc doručíme.

Vstup

```
3
4
5
3
```

Výstup

```
3
```

V tomto prípade treba kvôli každému z kusov upiecť celú pizzu. Dve pizze síce majú dostatočnú plochu, ale nevieme ich vhodne nakrájať.

A-I-2 Zasypané mesto

Archeológ Alfréd Hrozný skúma nedávno nájdené zasypané mesto v púšti. Ako prvý krok sa rozhodol, že pomocou sonaru určí, koľko miestností mali spolu všetky domy v meste.

Kus púšte, kde mesto ležalo, si Alfréd pokryl štvorcovou sieťou s rozmermi $M \times N$. V ruke so sonarom postupne prešiel všetky riadky takto vytvorenej štvorcovej siete a svoje merania si zaznamenal. Pre jednoduchosť predpokladal, že pod každým poľom tejto siete sa nachádza buď kameň, alebo piesok. Na základe získaných dát by rád určil, koľko miestností (súvislých oblastí piesku) v zasypanom meste bolo.

Súťažná úloha: Na vstupe je daný popis zasypaného mesta, ktoré si predstavujeme ako štvorcovú sieť s rozmermi $M \times N$. Políčka štvorcovej siete sú popísané po riadkoch od horného k spodnému a v jednotlivých riadkoch postupne zľava doprava.

Alfréd si svoje merania zapísal v skrátenej podobe. Ak bolo v riadku za sebou x políčok rovnakého typu, zapísal si jednoducho len číslo x. Zo zápisu „3 2 7“ by však ešte nebolo jasné, či je na prvých troch políčkach kamenie alebo piesok. Preto Alfréd zapísal svoje merania nasledovne:

Zápis každého riadku tvorí niekoľko dvojíc **nezáporných** celých čísel. Prvé číslo z každej dvojice udáva počet políčok s pieskom, druhé počet nasledujúcich políčok s kameňmi.

Vašou úlohou je spočítať počet miestností v zasypanom meste. Miestnosť je súvislá oblasť piesku, ktorú už v žiadnom smere nie je možné zväčšiť. Dve políčka štvorcovej siete považujeme za susedné, ak majú spoločnú hranu (spoločný vrchol nestačí).

Formát vstupu: V prvom riadku vstupu sa nachádzajú tri nezáporné čísla M, N a K – počet riadkov a stĺpcov štvorcovej siete mesta a celkový počet dvojíc, ktoré popisujú jej obsah. Je známe, že M a N sú menšie než 50 000 a že K je menšie ako 1 000 000 000.

Každý z ďalších K riadkov obsahuje dve nezáporné celé čísla p a q, kde p je počet políčok zasypaných pieskom a q je počet nasledujúcich políčok, pod ktorými sú kamene. Políčka sú popísané po riadkoch začínajúc od horného riadku siete, v rámci jedného riadku vždy zľava doprava. Každá dvojica čísel popisuje úsek, ktorý celý leží v jednom riadku.

Pokiaľ sa Vám nepodarí vyriešiť úlohu s vyššie popísanými obmedzeniami na M, N a K, predpokladajte, že M a N sú najviac 500 a K je najviac 100 000.

Formát výstupu: Na výstup vypíšte jediný riadok a v ňom jediné nezáporné číslo – počet miestností v zasypanom meste. Všimnite si, že ak sú úplne všade kamene, tak je toto číslo rovné nule.

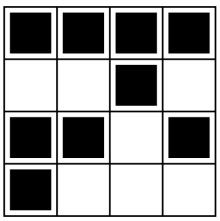
Príklad:

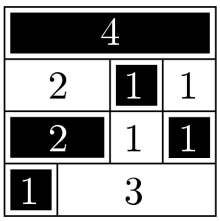
Vstup

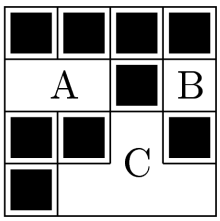
```
4 4 7
0 4
2 1
1 0
0 2
1 1
0 1
3 0
```

Výstup

```
3
```







Na prvom obrázku je znázornené mesto z príkladu. Na druhom sú uvedené čísla, ktoré si preň zapísal Alfréd (okrem núl). Na treťom obrázku sú písmenami označené miestnosti nachádzajúce sa v meste.

A-I-3 Okružná jazda

V mestečku Blatislava je neuveriteľne komplikovaný dopravný systém. Tvoria ho navzájom poprepájané križovatky. V každej križovatke sa môže stretávať ľubovoľne veľa ulíc. Nedávno sa miestny starosta rozhodol, že je čas na reformu dopravnej situácie.

Najskôr z každej ulice urobil jednosmerku, pričom si dal pozor na to, aby do každej križovatky aspoň jedna cesta vchádzala a aby z každej križovatky aspoň jedna cesta vychádzala. Následne v každej križovatke zakázal jednu možnosť odbočenia. (Teda ak do križovatky viedlo k ulíc a z nej ℓ ulíc, po zavedení zákazu sa dala prejsť $k\ell - 1$ spôsobmi.)

Reforma mala úspech, Blatislavčanom neuveriteľne sťažila pohyb po uliciach. Mnohí sa nevedeli dostať do práce, prípadne (tí menej šťastní) z práce domov. Aby starosta podobné tvrdenia vyvrátil, rozhodol sa nájsť *okružnú jazdu* – cyklickú postupnosť ulíc, ktoré nasledujú po sebe, všetky odbočenia v nej sú povolené, nikdy nejdeme v protismere a každá ulica v meste sa v tejto postupnosti vyskytuje práve raz.

Ak by starosta okružnú jazdu našiel, bolo by každému jasné, že z každej ulice mesta sa dá dostať na každú inú. Všimnite si ale, že takáto postupnosť nemusí existovať. Na to napríklad stačí, aby v meste bola križovatka, z ktorej vychádza menej ulíc ako do nej vchádza.

Súťažná úloha: Napíšte program, ktorý zistí, či v meste existuje okružná jazda, a ak áno, tak jednu ľubovoľnú nájde.

Formát vstupu: V prvom riadku vstupu sú dve prirodzené čísla N a M – počet križovatiek a počet jednosmeriek. Križovatky sú očíslované od 1 do N.

Nasleduje M riadkov, každý popisuje jednu ulicu: obsahuje dve čísla u a v ($1 \leq u, v \leq N$), ktoré hovoria, že z križovatky u vedie jednosmerka do križovatky v. Medzi každou dvojicou križovatiek vedie v každom smere najviac jedna ulica.

Nasleduje N riadkov, i-ty z nich popisuje zakázané odbočenie na križovatke s číslom i: obsahuje dve čísla u a v ($1 \leq u, v \leq N$), ktoré hovoria, že ak do križovatky i prichádzame z križovatky u, nesmieme odísť ulicou vedúcou ku v.

Formát výstupu: Ak existuje riešenie, vypíšte jeden riadok a v ňom postupnosť medzerami oddelených čísel $v_1, v_2, \ldots, v_m$ ($1 \leq v_i \leq N$ pre každé i)

takú, že:

- z v_i do v_{i+1} (pre $1 \leq i \leq M$) vedie ulica,
- ak ideme po ulici z v_i do v_{i+1}, je povolené odbočiť do ulice z v_{i+1} do v_{i+2} (pre $1 \leq i \leq M$),
- každá ulica sa v postupnosti nachádza, t. j. ak existuje ulica z u do v, tak existuje i také, že $v_i = u$ a $v_{i+1} = v$.

Indexy počítáme cyklicky, t. j. $v_{m+1} = v_1$ a $v_{m+2} = v_2$. Ak existuje viac riešení, vypíšte jedno ľubovoľné. Ak neexistuje žiadne riešenie, namiesto postupnosti vypíšte reťazec „`Okruzna jazda neexistuje.`“.

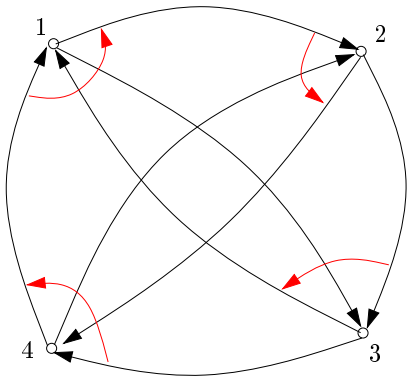
Príklady:

Vstup

```
4 8
1 2
2 3
3 1
3 4
1 3
4 2
2 4
4 1
4 2
1 4
2 1
3 1
```

Výstup

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



Na obrázku je cestná sieť z prvého príkladu. Šípky okolo križovatiek ukazujú zakázaný smer odbočenia. Vypísaný výstup je jeden z viacerých možných.

Vstup

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

Výstup

```
Okruzna jazda neexistuje.
```

A-I-4 Grafomat

V tomto ročníku olympiády sa budeme v teoretickej úlohe stretávať s grafomatmi. V študijnom texte uvedenom za zadaním tejto úlohy sú tieto stroje popísané.

Súťažná úloha: Napíšte program pre grafomat, ktorý v zadanom 3-grafe s vyznačenými dvoma vrcholmi nájde najkratšiu cestu medzi nimi a vyznačí vrcholy ležiace na tejto ceste. Môžete predpokladať, že cesta vždy existuje. Pokiaľ je najkratších ciest viac, vyberte si ľubovoľnú z nich.

V jednom zo zadaných vrcholov bude mať na začiatku premenná x hodnotu 1, v druhom 2 a vo všetkých ostatných vrcholoch bude mať hodnotu 0.

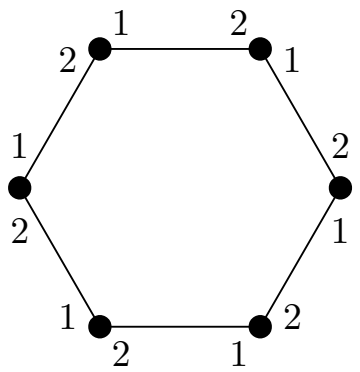
Na konci výpočtu by mala mať premenná y hodnotu 1 vo vrcholoch tvoriacich jednu najkratšiu cestu a hodnotu 0 v ostatných vrcholoch.

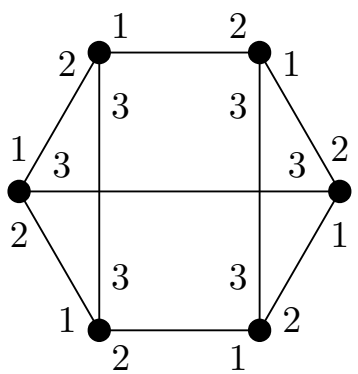
Pokúste sa napísať taký program, ktorého časová zložitosť bude závisieť len **na dĺžke zostrojenej cesty** a nie na veľkosti celého grafu.

Študijný text

Grafom nazývame ľubovoľnú konečnú množinu V *vrcholov* grafu spolu s množinou E *hrán*, čo sú neusporiadané dvojice vrcholov. Každá hrana teda predstavuje spojenie medzi dvoma vrcholmi. Žiadne dva vrcholy nie sú spojené viac ako jednou hranou, žiadna hrana nespája vrchol sám so sebou. Počet hrán grafu budeme označovať N a počet jeho hrán M.

K-graf budeme hovoriť takému grafu, v ktorom z každého vrcholu vedie práve K hrán a konce týchto hrán sú očíslované prirodzenými číslami od 1 po K. Konce jednej hrany môžu byť očíslované rôzne. Pokiaľ budeme hovoriť o hranách vychádzajúcich z nejakého vrcholu v, budeme spomínať *miestne* čísla hrán (čísla toho konca, ktorým je v) a *vzdialené* čísla (to sú tie na opačných koncoch hrán). Nasledujúci obrázok ukazuje príklad 2-grafu a 3-grafu.





Ohodnotením grafu nazveme priradenie prvkov nejakej konečnej množiny vrcholom grafu - teda napríklad rozdelenie vrcholov na čierne a biele, alebo označenie vrcholov číslami od 1 po 5.

Grafomat je zariadenie na automatické riešenie grafových úloh. Jeho vstupom je ľubovoľný K-graf G spolu s jeho ohodnotením. Výstupom je nejaké ďalšie ohodnotenie toho istého grafu. Samotný výpočet je vykonávaný *automatmi* umiestnenými v jednotlivých vrcholoch grafu. Každý automat má svoju pamäť a riadi sa programom. Programy všetkých automatov sú identické. Okrem toho, že automat môže ľubovoľne narábať so svojou pamäťou, môže aj nahliadať do pamäte svojich susedov.

Pamäť automatu si môžeme predstaviť ako pascalovské premenné typu interval. Teda každá premenná obsahuje jedno prirodzené číslo, ktoré je z nejakého pevne zvoleného rozsahu, ktorý nezávisí na veľkosti vstupu. Okrem toho je tiež možné používať pole takýchto premenných. Opäť, rozmery poľa musia byť dopredu známe a nesmú závisieť od veľkosti vstupu. Žiadne iné typy premenných (neobmedzene veľké čísla, smerníky, ...) sa nedajú používať.

Zvláštnu rolu hrajú premenné x a y. Premenná x na začiatku výpočtu obsahuje vstupné ohodnotenie toho vrcholu grafu, v ktorom program beží, hodnota premennej y na konci výpočtu určí výstupné ohodnotenie tohto vrcholu. Všetky premenné s výnimkou premennej x majú svoju počiatočnú hodnotu pevne určenú. Deklarácia premenných vyzerá napríklad takto:

```
var x: 1..5;          { číslo od 1 do 5, na začiatku je to vstup }
    y: 1..5 = 3;          { y je na začiatku 3, na konci výstup }
    z: array [1..2] of 3..4 = (3, 4);        { dvojprvkové pole }
```

Riadiaci program automatu si môžeme predstavit ako pascalovský program, v ktorom zakážeme používanie rekurzie a dovolíme manipulovať len s premennými v pamäti automatu a s premennými v pamäti susedných automatov. Na svoje vlastné premenné sa automat odkazuje ich menami, ako by to boli obyčajné pascalovské globálne premenné. Na to, aby sme vedeli odkazovať na premenné susedov, využijeme miestne čísla hrán. Presnejšie, nech `i` je celočíselný výraz s hodnotou z rozsahu $1 \ldots K$. Potom `S[i].p` je výraz predstavujúci premennú `p` u suseda, do ktorého od nás vedie hrana s miestnym číslom `i`. (Samozrejme, `i` môže byť ľubovoľný výraz a `p` ľubovoľné meno premennej.) Premenné susedov sa dajú len čítať.

Aby program mohol dávať do súvislosti svoje hrany s hranami svojich susedov, má k dispozící ešte premenne `P[1]` až `P[K]`, ktoré sú pevne nastavené tak, že `P[i]` obsahuje vzdialené číslo tej hrany, ktorá má miestne číslo `i`.

Použitie si ukážeme na výraze `S[i].S[P[i]].x`. Označme vrchol, v ktorom práve sme, písmenom v. Výrazy `S[i].volaco` sú premenné suseda, do ktorého sa dostaneme hranou s číslom `i`. Označme tohto suseda písmenom u. Premenná, ktorú chceme, je `S[nieco].x`. Takže chceme premennú `x` od niektorého suseda vrcholu u. Ale ktorého? Toho, do ktorého sa dostaneme hranou s číslom `P[i]`. To je ale práve vzdialené číslo hrany, ktorou sme do u prišli. Inými slovami, v u je to jeho miestne číslo hrany, ktorá vedie späť do v. Takže výraz `S[i].S[P[i]].x` predstavuje to isté ako výraz `x`.

Ak sa vám to zdá zložité, povieme si to ešte raz v dvoch krokoch. Najskôr sa vyhodnotí `nieco=P[i]`. (`P[i]` je naša lokálna premenná.) Teraz sa pozrieme na premennú `S[i].S[nieco].x`. (To prvé `S` je naše lokálne pole s premennými susedov, to druhé je podobné pole u suseda.) No a tento sused v premennej `S[nieco].x` vidí to, čo máme v našej premennej `x`.

Výpočet automatu prebieha v taktoch, a to nasledovne: V nultom takte sa premenné všetkých automatov nastavia na počiatočnú hodnotu a premenné x na vstupné ohodnotenie jednotlivých vrcholov. V každom ďalšom takte sa vždy znova spustí program každého automatu, pričom premenné svojich susedov vidí program v stave, v akom boli na začiatku taktu. Aj keď jednotlivé automaty bežia súčasne, nemôže sa teda stať, že by jeden čítal z premennej, do ktorej práve druhý zapisuje.

Výpočet pokračuje tak dlho, až kým v nejakom takte všetky automaty nevykonajú špeciálny príkaz `stop`. Potom sa výpočet zastaví a z premenných y grafomat prečíta výstupné ohodnotenie grafu. Pokiaľ príkaz `stop` urobia len niektoré automaty, výpočet pokračuje a to aj na tých automatoch, ktoré príkaz `stop` spravili. Štruktúra grafu a obsahy premenných `P` zostávajú po celú dobu výpočtu nezmenené.

Za *časovú zložitosť* výpočtu budeme považovať počet taktov, ktoré ubehnú po zastavenie programu. Nijako teda nezávisí na rýchlosti jednotlivých automatov. Podobne ako je to u časovej zložitosti klasických algoritmov, nebudeme hľadieť na multiplikatívne konštanty a bude nás zaujímať len asymptotické správanie zložitosti – teda či je lineárna, kvadratická, atď. Prípady, kedy výpočet neskončí, nebudeme pripúšťať, pre úplnosť ale dodajme, že vtedy sa hodnoty premenných musia nutne opakovať.

Príklad 1: Je zadaný 3-graf a v ňom vyznačený jeden vrchol v, a to tak, že jeho premenná x bude inicializovaná jednotkou a ostatné vrcholy budú mať nulu. Napíšte program pre grafomat, ktorý označí všetky vrcholy, do ktorých sa dá dostať z vrcholu v po hranách, a to tak, že ich premenná y bude mať na konci výpočtu hodnotu jedna, pre ostatné vrcholy bude mať hodnotu nula.

Riešenie: Inšpirujeme sa prehľadávaním do šírky. V každom takte sa každý vrchol pozrie, či je niektorý z jeho susedov už označený. Ak je, tak sa sám označí. Pokiaľ sa označenie nezmení, vrchol volaním `stop` súhlasí so zastavením. Priebeh výpočtu bude teda vyzerať tak, že po i-tom takte budú označené tie vrcholy, ktorých vzdialenosť od v je menšia alebo rovná i. Výpočet sa zastaví vtedy, keď sa hodnoty premenných prestanú meniť, čo znamená, že po najviac N taktoch. Preto je časová zložitosť nášho programu lineárna od počtu vrcholov (na rozdiel od klasického prehľadávania do šírky, ktoré závisí od počtu hrán).

Program vyzerá nasledovne:

```
var x: 0..1;                    { bol vrchol označený vo vstupe? }
    y: 0..1 = 0;                            { je označený teraz? }
    prev: 0..1 = 0;                        { predchádzajúci stav }
    i: 1..3;

begin
   prev := y;                { zapamätáme si, či už bol označený }
   if x=1 then y := 1;          { prenesieme označenie zo vstupu }
   for i := 1 to 3 do          { pozrieme sa na všetkých susedov }
      if S[i].y <> 0 then            { ak je i-ty sused označený }
         y := 1;                              { označ aj sám seba }
   if y = prev then stop;    { ak sa nič nemení, môžeme skončiť }
end.
```

Príklad 2: Majme 2-graf zložený z jedného cyklu párnej dĺžky, teda z vrcholov očíslovaných $0 \ldots N-1$, pričom vrchol i je spojený hranou označenou 1 s vrcholom $(i+1) \bmod N$ a hranou označenou 2 s vrcholom $(i-1) \bmod N$. (Príklad takého grafu pre $N = 6$ nájdete na obrázku na začiatku tohto textu.) V tomto grafe je vyznačený jeden vrchol v, rovnako ako v predchádzajúcom príklade. Napíšte program pre grafomat, ktorý označí vrchol protiľahlý k v, teda vrchol s číslom $(v + N/2) \bmod N$.

Riešenie: Vyšleme signál putujúci z vrcholu v v smere jednotkových hrán rýchlosťou 1 vrchol za takt. Zároveň vyšleme druhý signál putujúci rovnakou rýchlosťou opačným smerom. Akonáhle nejaký vrchol zistí, že do neho prišli oba signály, označí sa a signály už ďaľej neposiela.

```
var x: 0..1;                              { vstupná značka vrcholu }
    y: 0..1 = 0;                                   { výstupná značka }
    l, r: 0..1 = 0;                { už týmto vrcholom prešiel signál
```

```
                                                  doľava a doprava? }
begin
   if x=1 then                                  { začíname posielať }
      begin x := 0; l := 1; r := 1; end;
   else if (S[2].l=1) and (S[1].r=1) then  { signály sa stretli }
      begin y := 1; stop; end
   else if (S[2].l=1) and (l=0) then l := 1    { pošleme doľava }
   else if (S[1].r=1) and (r=0) then r := 1   { pošleme doprava }
   else stop;                      { nič sa nedeje, môžeme končiť }
end.
```

Zadania domáceho kola kategórie B

B-I-1 Pestovanie mrkvičiek

Janko je majiteľom veľmi úspešnej firmy, zaoberajúcej sa pestovaním mrkvičiek. Jeho pole, po ktorom sa rád prechádza, je veľká rovina, ktorá sa tiahne od vidím do nevidím celou obrovskou krajinou. Aby Janko presne vedel, kde sú mrkvičky nasadené, rozhodol sa ich nasadiť práve do bodov s celočíselnými súradnicami.

Teraz sa rozhodol svoje podnikanie rozšíriť a poskytovať mrkvičky aj cez internet. Objednal si preto pripojenie svojej farmy optickým káblom. Dozvedel sa ale hroznú novinu. Aby sa kábel dostal až ku jeho farme, musí prechádzať popod mrkvičkové pole. Na jednom mieste musí vystúpiť na povrch, aby sa vyhol zavlažovaciemu zariadeniu, a za ním opäť zostúpiť pod zem. Spoločnosť, ktorá mu má kábel inštalovať, mu oznámila súradnice, kde na jeho poli kábel vystúpi a kde sa vráti opäť pod zem. Janko teraz počíta, o koľko mrkvičiek kvôli káblu nad zemou príde. Skúste mu pomôcť.

Súťažná úloha: Napíšte program, ktorý pre zadané dva body s celočíselnými súradnicami zistí, koľko bodov s celočíselnými súradnicami leží na úsečke, ktorá zadané body spája.

Formát vstupu: Prvý riadok vstupu obsahuje súradnice prvého bodu (dve celé čísla x_1, y_1 oddelené medzerou). Druhý riadok obsahuje súradnice druhého bodu x_2, y_2. Môžete predpokladať, že $-10^9 \leq x_1, x_2, y_1, y_2 \leq 10^9$ a že sú to dva rôzne body.

Formát výstupu: Vypíšte jediný riadok a v ňom jediné číslo – počet bodov s celočíselnými súradnicami na úsečke spájajúcej body zo vstupu (vrátane jej koncových bodov).

Príklad:

Vstup	Výstup
1 1 5 3	3

V tomto príklade ide o body $[1, 1]$, $[3, 2]$ a $[5, 3]$.

B-I-2 Hladný Samo

Hladný Samo sa konečne dočkal. Jeho rodičia práve odišli z domu a on sa konečne môže poriadne najesť. Vkĺzol do bytu, otvoril chladničku a už už sa chcel pustiť do všetkého, čo v nej našiel. Potom sa však pozrel bližšie a pochopil, že keby sa všetky tieto veci skombinovali v jeho žalúdku, asi by to nemalo najlepšie následky. Určite by naraz nemal zjesť čokoládu s kečupom, vyprážané kurča s mliekom ani kapustu s lekvárom. Narýchlo si preto vytvoril dvojice jedál, ktoré teraz určite nechce naraz zjesť. Samo je však veľmi hladný a preto by toho chcel zjesť čo najviac.

Teraz však len smutne pozerá do chladničky a rozmýšľa, čo z toho má zjesť. Pretože ste jeho dobrý kamarát/ka, napíšte mu program, ktorý mu pomôže.

Súťažná úloha: Napíšte program, ktorý zistí, koľko najviac toho Samo môže zjesť bez toho, aby zjedol obe zo zakázanej dvojice.

Formát vstupu: Prvý riadok vstupu obsahuje počet potravín v chladničke N ($1 \leq N \leq 25$). Na nasledujúcich riadkoch sú dvojice čísel i, j ($1 \leq i, j \leq N$) reprezentujúce dvojice čísel potravín, ktoré sa nedajú zjesť spolu. Vstup je ukončený dvojicou čísel „0 0“.

Formát výstupu: Prvý riadok výstupu má obsahovať číslo M – maximálny počet potravín, ktoré môže Samo zjesť. V ďalšom riadku má byť M čísel týchto potravín. Ak existuje viacero možností s rovnakým počtom potravín, vypíšte ľubovoľnú jednu z nich.

Príklad:

Vstup

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

Výstup

```
4
1 3 4 5
```

B-I-3 Družstvá

Na ihrisku sa zišlo niekoľko detí, ktoré si medzi sebou chcú zahrať bližšie nešpecifikovanú loptovú hru. Ich jediný problém je rozdelenie sa do dvoch družstiev. Vlastne ich ani veľmi nezaujíma, aby proti sebe hrali družstvá s rovnakým počtom hráčov, dôležité je len, aby rovnako silní hráči boli zastúpení v oboch družstvách.

Preto medzi sebou našli dvojice rovnako silných hráčov, ktorí musia hrať proti sebe. Napr. nech na ihrisko prišli Janko, Ferko, Jožko, Peťko a Mirko. Navyše vieme, že Janko určite musí hrať proti Jožkovi a Ferko musí hrať proti Peťkovi. Potom jedno možné rozdelenie do družstiev je:

1. družstvo: Janko, Ferko, Mirko
2. družstvo: Jožko, Peťko

Súťažná úloha: Na vstupe je počet hráčov a zoznam dvojíc hráčov, ktorí nemôžu hrať v jednom družstve. Napíšte program, ktorý zistí, či je možné vytvoriť dve družstvá s požadovanými vlastnosťami. Ak áno, tak jedno také rozdelenie vypíše.

Formát vstupu: Prvý riadok vstupu obsahuje celé čísla N a M, kde N je počet hráčov a M je počet dvojíc hráčov, ktorí nemôžu hrať spolu ($N \leq 10\,000$). Hráči sú očíslovaní číslami od 1 do N. Na každom z nasledujúcich M riadkov sa nachádzajú dve čísla, určujúce dvojicu hráčov, ktorí nemôžu hrať spolu v jednom družstve.

Formát výstupu: Prvý riadok výstupu má obsahovať slovo „`ANO`“, ak aspoň jedno vhodné rozdelenie existuje, a slovo „`NIE`“ v opačnom prípade. V prípade, že nejaké rozdelenie existuje, v druhom riadku výstupu majú byť čísla hráčov hrajúcich v jednom z družstiev. Ak existuje viacero správnych rozdelení, vypíšte ľubovoľné jedno z nich.

Príklady:

Vstup

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

Výstup

```
ANO
1 2 5
```

Vstup

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

Výstup

```
NIE
```

B-I-4 Assembler

Mnoho mladých programátorov túži naučiť sa programovať v assembleri. Prichystali sme pre vás príklad, ktorý vám dáva možnosť si to vyskúšať.

Budeme uvažovať zjednodušený assembler. K dispozícií je 8 registrov („premenných“) označených $R_0, \ldots, R_7$. Okrem nich už nie je k dispozícii žiadna ďalšia pamäť. Registre vedia uchovávať ľubovoľne veľké nezáporné celé číslo. Na prácu s nimi máte 6 inštrukcií:

1. **inc** R_i (increment)

   Zvýši hodnotu registra R_i o 1.

2. **dec** R_i (decrement)

   Ak je $R_i > 0$, zníži hodnotu registra R_i o 1. Ak $R_i = 0$, tak nespraví nič. Ak je po vykonaní inštrukcie $R_i = 0$, tak sa nastaví príznak **Z** na 1. Inak sa **Z** nastaví na 0.

3. **jmp** návestie (jump)

   Skočí na inštrukciu napísanú za daným návestím (niečo ako **goto** v Pascale/C).

4. **test** R_i, R_j

   Vypočíta bitový súčin (bitwise and) registrov R_i a R_j. Výsledok však ignoruje a len nastaví príznak **Z**, podľa toho, či je daný súčin 0 alebo nie. Zvyčajne budete túto inštrukciu používať len na overenie, či je register R_i nulový a to príkazom **test** R_i, R_i, čo nastaví príznak **Z** na 1 ak $R_i = 0$, inak ho nastaví na 0.

5. **jz** návestie (jump if zero)

   V prípade, že je príznak **Z** nastavený na 1, skočí na dané navestie. V opačnom prípade pokračuje vo vykonávaní programu.

6. **jnz** návestie (jump if not zero)

   V prípade, že je príznak **Z** nastavený na 0, skočí na dané návestie.

Pre jednoduchosť budeme do jedného riadku programu písať len jednu inštrukciu. Pred ľubovoľnou inštrukciou môže byť napísané návestie (oddelené od inštrukcie dvojbodkou). Je to značka v mieste programu, na ktorú sa môžeme odvolávať inštrukciami **jmp**, **jz** a **jnz**.

Formálne vyzerá program nasledovne:

$$
\begin{aligned}
\langle program\rangle &::= \langle riadok\rangle \mid \langle riadok\rangle\ \langle program\rangle \\
\langle riadok\rangle &::= \langle instrukcia\rangle \mid \langle navestie\rangle\mathbf{:}\ \langle instrukcia\rangle \\
\langle instrukcia\rangle &::= \mathbf{inc}\ \langle register\rangle \mid \\
&\quad\ \mathbf{dec}\ \langle register\rangle \mid \\
&\quad\ \mathbf{jmp}\ \langle navestie\rangle \mid \\
&\quad\ \mathbf{test}\ \langle register\rangle\mathbf{,}\langle register\rangle \mid \\
&\quad\ \mathbf{jz}\ \langle navestie\rangle \mid \\
&\quad\ \mathbf{jnz}\ \langle navestie\rangle \\
\langle register\rangle &::= R_0 \mid R_1 \mid \ldots \mid R_7
\end{aligned}
$$

Pričom $\langle navestie\rangle$ je ľubovoľný reťazec neobsahujúci ':' (dvojbodku).

Inštrukcie programu sa pri jeho spustení vykonávajú zaradom, podľa toho, ako boli zapísané v programe, začínajúc od prvej inštrukcie. Jedinou výnimkou sú inštrukcie **jmp**, **jz** a **jnz** po ktorých môže program pokračovať na inom mieste.

Príklad: V registroch R_1 a R_2 sú zapísané nejaké čísla (ostatné registre sú vynulované). Napíšte program, ktorý do registra R_0 zapíše súčet čísel v registroch R_1 a R_2.

Riešenie: Najskôr budeme postupne register R_1 zmenšovať o 1 a súčasne zvyšovať hodnotu registra R_0, až kým R_1 nebude nula. Tým vlastne nastavíme R_0 na hodnotu R_1. Potom to isté spravíme aj s registrom R_2, čím k R_0 pripočítame hodnotu R_2.

```
c1: test R1,R1
    jz c2
    dec R1
    inc R0
    jmp c1
c2: test R2,R2
    jz koniec
    dec R2
    inc R0
    jmp c2
koniec:
```

Súťažná úloha:

a) Napíšte program, ktorý zo súboru `program.asm` načíta program, zo súboru `registre.in` načíta počiatočný obsah registrov a odsimuluje výpočet

programu. V prípade, že výpočet skončí, vypíše do súboru `registre.out` obsah registrov po ukončení výpočtu.

Príklad:

program.asm

```
c1: test R1,R1
    jz koniec
    dec R1
    inc R0
    inc R0
    jmp c1
koniec:
```

registre.in

```
0 47 0 0 0 0 0 0
```

registre.out

```
94 0 0 0 0 0 0 0
```

b) V registroch R_1 a R_2 sú zapísané nejaké čísla, pričom $R_2 \neq 0$ (ostatné registre sú vynulované). Napíšte assemblerovský program, ktorý do registra R_0 zapíše zvyšok po delení čísla R_1 číslom R_2.

Zadania krajského kola kategórie A

A-II-1 Stále ešte zasypané mesto

Archeológ Alfréd Hrozný už s vašou pomocou zmapoval zasypané mesto. Teraz sa rozhodol, že naženie mladých študentov, nech mu ho spod piesku vyhrabú. Z univerzity sa mu prihlásilo na pomoc N prvákov a N druhákov. Každý z nich študoval práve jedno z nasledujúcich zameraní: stredovekú históriu, staroveku históriu alebo archeológiu.

Alfréd by chcel, aby študenti pracovali vo dvojiciach. Nie však hocijakých. V každej dvojici musí byť druhák, lebo dvaja prváci by boli príliš pochabí. A navyše študenti v každej dvojici by mali mať rôzne zamerania, aby sa ich vedomosti vhodne dopĺňali.

Súťažná úloha: Napíšte program, ktorý na vstupe dostane údaje o jednotlivých študentoch a zistí, či (a ak áno, ako) sa dajú popárovať tak, aby boli splnené Alfrédove požiadavky.

Snažte sa nájsť program, pri ktorom by nájdenie priradenia bolo čo najefektívnejšie. (Odhliadneme teda od času potrebného na načítanie vstupu a výpis výstupu.) Dajte si záležať na dôkaze, že váš algoritmus naozaj pre všetky možné vstupy funguje.

Formát vstupu: Prvý riadok vstupu obsahuje jedno celé číslo N, ktoré udáva počet dvojíc študentov, ktoré budeme vyrábať. Nasleduje $2N$ riadkov, každý popisuje jedného študenta: obsahuje jeho ročník (číslo 1 alebo 2) a jeho zameranie (reťazec „`stredovek`“, „`starovek`“ alebo „`archeologia`“). Môžete predpokladať, že prvákov je rovnako ako druhákov.

Formát výstupu: Ak sa študenti nedajú rozdeliť do vhodných dvojíc, vypíšte o tom správu. V opačnom prípade vypíšte N riadkov, v každom najskôr zameranie prváka a potom zameranie druháka, ktorý s ním bude vo dvojici.

Príklad:

Vstup

```
3
1 starovek
1 starovek
2 archeologia
1 stredovek
2 stredovek
2 starovek
```

Výstup

```
starovek archeologia
starovek stredovek
stredovek starovek
```

A-II-2 Nová okružná jazda

Z domáceho kola si isto pamätáte mestečko Blatislava a jeho dopravné ťažkosti. Napriek tomu si aspoň stručne celú situáciu popíšeme. Blatislavskú dopravnú sieť tvoria križovatky poprepájané ulicami. Do jednej križovatky môže vchádzať ľubovoľný počet ulíc.

Na začiatku tohoto ročníka olympiády sa starosta rozhodol spraviť z každej ulice jednosmerku (tak, aby do každej križovatky vchádzalo rovnako veľa ulíc ako z nej vychádzalo) a zakázať v každej križovatke jeden spôsob prechodu. Dnes na magistrát dostali prvé výsledky tejto dopravnej reformy: Počet nehôd stúpol o 47 percent.

Zasadol krízový štáb, a ten sa rozhodol situáciu riešiť nasledovne: V prvom rade sa zrušia jednosmerky, teda každou ulicou sa bude dať chodiť oboma smermi. Zato sa ale na každej križovatke zakážu až tri spôsoby odbočenia. Tieto zákazy budú tiež obojsmerné, t.j. zakáz odbočenia z ulice a na ulicu b zároveň zakazuje odbočiť z ulice b na ulicu a.

Majme teda križovatku $t > 2$ ulíc. Vzhľadom na históriu Blatislavy vieme, že t je párne, a teda nutne $t \geq 4$. Takúto križovatku sa po zavedení zákazov odbočenia bude dať prejsť $t(t-1) - 6$ spôsobmi.

Opäť ale bude treba presvedčiť občanov, že je nová reforma dobrá.

Súťažná úloha: Napíšte program, ktorý dostane na vstupe popis cestnej siete v Blatislave a nájde v nej okružnú cestu – cyklickú postupnosť na seba nadväzujúcich ulíc takú, že všetky odbočenia v nej sú povolené a obsahuje každú ulicu práve raz.

Formát vstupu: V prvom riadku vstupu sa nachádzajú dve prirodzené čísla N a M – počet križovatiek a počet ulíc. Križovatky sú očíslované od 1 do N.

Nasledujúcich M riadkov popisuje jednotlivé ulice. V každom z nich sú dve čísla križovatiek, ktoré daná ulica spája. Môžete predpokladať, že každá ulica spája dve rôzne križovatky a že medzi každou dvojicou križovatiek vedie najviac jedna ulica.

Ďalej nasleduje N riadkov, ktoré popisujú zákazy odbočenia. V i-tom z nich je 6 čísel: $u_{i,1}$, $v_{i,1}$, $u_{i,2}$, $v_{i,2}$, $u_{i,3}$ a $v_{i,3}$. Tieto čísla hovoria, že v križovatke s číslom i sú zakázané nasledujúce spôsoby odbočenia: Ak idem z križovatky $u_{i,x}$ (pre nejaké $x \in \{1, 2, 3\}$) do križovatky i, nesmiem odbočiť do ulice vedúcej na križovatku $v_{i,x}$. A pochopiteľne aj naopak, idúc od $v_{i,x}$ nesmiem odbočiť smerom na $u_{i,x}$.

Formát výstupu: Ak okružná jazda neexistuje, podajte o tom správu. V

opačnom poradí vypíšte postupnosť M čísel: čísla križovatiek v poradí, v akom ich prechádza jedna možná okružná jazda.

Formálne, vypíšte postupnosť $v_1, \ldots, v_M$ takú, že:

- Pre každé i $(1 \leq i \leq M)$ platí, že medzi križovatkami v_i a v_{i+1} naozaj vedie ulica.
- Pre každé i $(1 \leq i \leq M)$ platí, že ak ideme ulicou z križovatky v_i na križovatku v_{i+1}, je dovolené odbočiť smerom na križovatku v_{i+2}.
- Každou ulicou ideme práve raz, teda pre každú ulicu existuje i také, že v_i a v_{i+1} sú vrcholy, ktoré spája.

Indexy počítame cyklicky, teda $v_{m+1} = v_1$ a $v_{m+2} = v_2$.

Príklad:

Vstup

```
5 10
1 2
2 3
3 4
4 5
5 1
1 3
2 4
3 5
4 1
5 2
5 2 2 3 4 5
1 3 1 5 3 4
2 4 2 1 4 5
3 5 2 3 1 5
4 1 1 2 3 4
```

Výstup

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

A-II-3 Rozvoz pizze vo väčšom

Po tom, ako ste Marcovi v domácom kole pomohli, slávi jeho firma úspechy. Marco sa teda rozhodol, že firmu rozšíri a otvorí niekoľko pobočiek na hranici mesta (kde ešte nie je taká silná konkurencia). Potrebuje ale vašu pomoc, aby zistil, kde presne tieto pobočky otvoriť.

Pre jednoduchosť si predstavme, že hranica mesta je kružnica, ktorú si rozdelíme na N úsekov (mestské časti). Je nutné, aby každý úsek bol celý obsluhovaný jednou pizzériou, aby obyvatelia nemali zmätok a jasne vedeli, do ktorej pizzérie majú ísť. Navyše každá pizzéria musí pochopiteľne obsluhovať súvislý kus hranice mesta, teda niekoľko po sebe nasledujúcich úsekov.

Posledný problém je veľmi jednoduchý: jedna pizzéria dokáže denne obslúžiť najviac K zákazníkov. Pre každý úsek hranice mesta vie Marco povedať, koľko zákazníkov tam denne bude mať. Pizzérie teda treba rozmiestniť tak, aby žiadna z nich nemala v obsluhovaných úsekoch dokopy viac ako K zákazníkov denne.

Súťažná úloha: Napíšte program, ktorý načíta kapacitu jednej pizzérie, počet úsekov a údaje o počtoch zákazníkov v nich, a spočíta, koľko najmenej pizzérií musí Marco postaviť, aby obslúžil zákazníkov zo všetkých úsekov, a ako im prideliť úseky.

Formát vstupu: V prvom riadku vstupu sú dve prirodzené čísla N a K – počet úsekov kružnice a počet zákazníkov, ktorých denne zvládne obslúžiť jedna pizzéria.

Nasleduje N riadkov, každý popisuje jeden úsek kružnice v poradí, v akom na nej ležia. Pritom i-ty z týchto riadkov obsahuje jedno celé číslo A_i – počet zákazníkov, ktorých treba v tomto úseku denne obslúžiť.

Môžete predpokladať, že žiadne A_i nie je väčšie ako K.

Formát výstupu: V prvom riadku vypíšte minimálny počet pizzérií M.

V nasledujúcich M riadkoch vypíšte popis jedného optimálneho riešenia. Presnejšie, v i-tom riadku vypíšte číslo S_i prvého a číslo T_i posledného úseku, ktorý bude obsluhovať i-ta pizzéria.

(Ak $T_i < S_i$, znamená to, že táto pizzéria obsluhuje úseky $S_i, \ldots, N, 1, \ldots, T_i$, inak pizzéria obsluhuje úseky $S_i, \ldots, T_i$.)

Pripomíname, že v optimálnom riešení musí byť každý úsek obsluhovaný práve jednou pizzériou, a navyše pre každú pizzériu musí byť súčet počtov zákazníkov v jej úsekoch nanajvýš rovný K.

Príklad:

Vstup

```
7 11
4
4
7
6
6
4
2
```

Výstup

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

Prvá pizzéria obsluhuje úseky 6, 7 a 1, v tých je dokopy $4 + 2 + 4 = 10$ zákazníkov. Druhá bude mať $4 + 7 = 11$ zákazníkov, tretia a štvrtá po 6.

A-II-4 Grafomat loví zver

Študijný text je identický s tým zo zadaní domáceho kola (úloha A-I-4).

Súťažná úloha: Kráľ Miesiželezo XXXXVII. veľmi rád organizoval lovy. Priečilo sa mu na nich ale strieľanie do všetkého živého, či už to boli zvieratá, alebo zablúdení poľovníci. Preto si dal vyrobiť mechanickú zver a mechanických lovcov, a mohlo sa ísť loviť.

Mechanickí lovci vždy zahnali zviera do nory a utvorili okolo nej kruhovú formáciu. Každý lovec sa pripojil ku svojim susedom a všetci svorne čakali. Len čo si niekto všimol, že zviera vystrčilo z nory anténky, poslal správu ostatným a všetci naraz vystrelili. (Smerom do nory, nie na seba!)

Vašou úlohou je napísať program, ktorý bude lovcov riadiť.

Majme 2-graf, tvorený jedným cyklom párnej dĺžky. Vrcholy sú očíslované od 0 do $N-1$, pričom vrchol i je spojený (hranou číslo 1) s vrcholom $((i+1) \bmod N)$ a (hranou číslo 2) s vrcholom $((i-1) \bmod N)$. Príklad takéhoto 2-grafu je na prvom obrázku v študijnom texte.

Váš program sa má správať nasledovne: Ak je pri spustení vo všetkých vrcholoch $x = 0$, skončí s $y = 0$ v každom vrchole. Ak dostane v jednom vrchole $x = 1$ a v ostatných $x = 0$, v konečnom čase skončí s tým, že vo všetkých vrcholoch bude $y = 1$, pričom **po každom z predchádzajúcich krokov** muselo všade byť $y = 0$.

Slovne: Ak lovci nič nevidia, nestrieľajú. Ak práve jeden zazrie zver, po konečnom čase všetci naraz vystrelia. Môžete predpokladať, že pri spustení bude

nanajvýš v jednom vrchole $x = 1$. Ak sa vám to hodí, môžete navyše predpokladať, že N je vhodného tvaru (napr. štvorec, mocnina dvoch, a pod.).

Zadania krajského kola kategórie B

B-II-1 Kerfúr ten je lacnejší

Podobné tvrdenia počúvame denne od marketingových oddelení supermarketov. Je však možné, aby všetci hovorili pravdu? To by veľmi zaujímalo úrad pre ochranu spotrebiteľa. Ten pozorne sledoval reklamy a zapisoval si z nich tvrdenia typu „Supermarket A je lacnejší ako supermarket B“. Teraz by chceli zistiť, či to môže byť naozaj všetko pravda.

Súťažná úloha: Napíšte program, ktorý dostane na vstupe reklamy a zistí, či takáto situácia môže nastať. Pre jednoduchosť predpokladajme, že všetky supermarkety predávajú rovnaké výrobky a ak reklama hovorí, že supermarket A je lacnejši ako B, znamená to, že všetky výrobky majú v A ostro menšiu cenu ako v B. Zároveň predpokladáme, že počas doby, kedy úrad sledoval reklamy, sa ceny v žiadnom supermarkete nemenili.

Formát vstupu: Prvý riadok vstupu obsahuje celé čísla N a M, kde N je počet skúmaných supermarketov a M je počet reklám. Na nasledujúcich M riadkoch sa nachádzajú dvojice celých čísel i, j ($1 \leq i, j \leq N$) s významom „supermarket číslo i je lacnejší ako supermarket číslo j“.

Formát výstupu: Jediný riadok výstupu má obsahovať slovo `ANO` ak môžu byť pravdivé úplne všetky reklamy zo vstupu, prípadne `NIE`, ak to nie je možné.

Príklady:

Vstup

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

Výstup

```
NIE
```

Vstup

```
7 4
1 4
4 7
4 6
6 7
```

Výstup

```
ANO
```

B-II-2 Televízna súťaž

V istej televíznej súťaži je finalista postavený pred mnoho dverí, pričom len za jednými z nich je skrytá hlavná cena, za ostatnými nie je nič. To, za ktorými z nich je hlavná cena, rozhoduje náhoda. Pred začiatkom súťaže sa vyrobí niekoľko lístočkov a na každý z nich sa napíše jedno číslo dverí. (To isté číslo môže byť napísané na viacerých lístočkoch, a niektoré čísla dokonca nemusia byť použité vôbec.) Potom tesne pred finále vyžrebuje moderátor jeden z lístočkov a za dvere, ktorých číslo je na lístočku, sa uloží hlavná cena.

Hráč má možnosť vidieť lístočky dopredu, aby vedel, že usporiadateľ nepodvádza. Aké číslo dverí si má vybrať, aby pravdepodobnosť, že vyhrá, bola najväčšia?

Súťažná úloha: Napíšte program, ktorý zistí ktoré zo zadaných čísel sa vo vstupe vyskytuje najčastejšie.

Formát vstupu: Prvý riadok vstupu obsahuje počet lístkov N ($1 \leq N \leq 10\,000$). V nasledujúcom riadku je N celých čisel oddelených medzerami. Tieto čísla sú v rozmedzí od 1 do 10^9.

Formát výstupu: Prvý riadok výstupu má obsahovať jediné číslo k, ktoré sa medzi vstupnými číslami vyskytovalo najčastejšie.

Ak je viacero možných odpovedí, môžete vypísať ľubovoľnú jednu z nich.

Príklad:

Vstup	Výstup
5 100 23 4321 100 47	100

B-II-3 Kartári

Miško a Lacko sú veľkí kartári. Skoro nikdy ich neuvidíte robiť niečo iné ako hrať karty. Najradšej hrajú hru Rah, ktorú si sami vymysleli. Nebudeme tu vysvetľovať jej presné pravidlá, sú totiž dosť komplikované.

Dôležité je, že táto hra sa hrá s klasickými žolíkovými kartami (bez žolíkov). V balíku je teda 8 kusov každej možnej hodnoty. Farby kariet pre nás nebudú dôležité.

Hodnoty kariet označíme znakmi nasledovne: dvojke až deviatke budú zodpovedať znaky ‘2’ až ‘9’. Desiatka bude ‘T’ (z anglického *ten*), a ďalej to už pôjde podľa písmen na kartách, teda jano bude ‘J’, dáma ‘Q’, kráľ ‘K’ a eso ‘A’.

Obidvaja hráči majú svoju kôpku kariet (na začiatku hry prázdnu). Potom sa striedajú v ťahoch. V jednom ťahu si hráč buď môže zobrať nejakú kartu z balíka a dať si ju na spodok svojej kôpky, alebo zahrať kartou, ktorú má na vrchu kôpky.

Miško však začal Lacka podozrievať, že podvádza. Preto by potreboval program, ktorý by mu presne vedel povedať, akou kartou má kedy Lacko zahrať, ak vie, aké karty si Lacko vyberal z balíka.

Súťažná úloha: Napíšte program, ktorý bude načítavať kroky, ktoré Lacko vykonáva. V prípade, že Lacko chce zahrať kartou z vrchu svojej kôpky, vypíšte kartu, ktorou zahral. (Pozor, partia môže byť ľubovoľne dlhá, Lacko môže opäť zobrať do kôpky karty, ktoré už predtým zahral.)

Formát vstupu: Každý riadok vstupu obsahuje jeden príkaz. Príkazy môžu byť nasledovných typov:

- `1 c` – Lacko zobral z balíka kartu `c` (`c` je jeden zo znakov `23456789TJQKA`)
- `2` – Lacko sa rozhodol zahrať kartou z vrchu svojej kôpky
- `0` – koniec vstupu

Formát výstupu: Pre každý riadok vstupu s príkazom 2 má výstup obsahovať jeden riadok a v ňom jeden znak, označujúci kartu, ktorou Lacko zahral. Môžete predpokladať, že vždy, keď má zahrať kartu, nejakú v kôpke má.

Príklad:

Vstup

```
1 J
1 A
2
1 3
1 J
2
2
0
```

Výstup

```
J
A
3
```

B-II-4 Assembler II

a) *(3 body)* Definujte pomocou existujúcich inštrukcií nové inštrukcie:
   – „**mov** R_i, R_j“, ktorá priradí do registra R_i hodnotu registra R_j
   – „**mul** R_i, R_j“, ktorá priradí do registra R_i súčin hodnôt R_i a R_j.

b) *(7 bodov)* V registri R_1 je uložené nezáporné celé číslo, ostatné registre sú vynulované. Napíšte program v našom assembleri, ktorý do registra R_0 zapíše dolnú celú časť odmocniny z R_1 (teda najväčšie celé k také, že k^2 je menšie alebo rovné ako hodnota v R_1).
Snažte sa, aby váš program bol efektívny, t.j. potreboval čo najmenej inštrukcií v závislosti od veľkosti čísla v R_1.

Inštrukcie **mov** a **mul** majú nechať obsah registra R_j nezmenený. Môžete predpokladať, že $i \neq j$, že $i, j < 5$ a že registre R_5, R_6 a R_7 sú pred vykonaním vašej inštrukcie vynulované. Musia byť nulové aj po vykonaní vašej inštrukcie, ale počas vykonávania ich môžete použiť ako pomocné premenné.

Pri riešení súťažných úloh si môžete na skrátenie zápisu programu podobne definovať aj iné nové inštrukcie.

Študijný text

Budeme uvažovať zjednodušený assembler. K dispozícií je 8 registrov („premenných“) označených $R_0, \ldots, R_7$. Okrem nich už nie je k dispozícii žiadna ďalšia pamäť. Registre vedia uchovávať ľubovoľne veľké nezáporné celé číslo. Na prácu s nimi máte 6 inštrukcií:

1. **inc** R_i (increment)

   Zvýši hodnotu registra R_i o 1.

2. **dec** R_i (decrement)

   Ak je $R_i > 0$, zníži hodnotu registra R_i o 1. Ak $R_i = 0$, tak nespraví nič. Ak je po vykonaní inštrukcie $R_i = 0$, tak sa nastaví príznak **Z** na 1. Inak sa **Z** nastaví na 0.

3. **jmp** návestie (jump)

   Skočí na inštrukciu napísanú za daným návestím (ako **goto** v Pascale/C).

4. **test** R_i, R_j

   Vypočíta bitový súčin (bitwise and) registrov R_i a R_j. Výsledok však ignoruje a len nastaví príznak **Z**, podľa toho, či je daný súčin 0 alebo nie. Zvyčajne budete túto inštrukciu používať len na overenie, či je register R_i nulový a to príkazom **test** R_i, R_i, čo nastaví príznak **Z** na 1 ak $R_i = 0$, inak ho nastaví na 0.

5. **jz** návestie (jump if zero)

   V prípade, že je príznak **Z** nastavený na 1, skočí na dané navestie. V opačnom prípade pokračuje vo vykonávaní programu.
6. **jnz** návestie (jump if not zero)

   V prípade, že je príznak **Z** nastavený na 0, skočí na dané návestie.

Pre jednoduchosť budeme do jedného riadku programu písať len jednu inštrukciu. Pred ľubovoľnou inštrukciou môže byť napísané návestie (oddelené od inštrukcie dvojbodkou). Je to značka v mieste programu, na ktorú sa môžeme odvolávať inštrukciami **jmp**, **jz** a **jnz**.

Formálne vyzerá program nasledovne:

$$
\begin{aligned}
\langle \textit{program} \rangle &::= \langle \textit{riadok} \rangle \mid \langle \textit{riadok} \rangle\ \langle \textit{program} \rangle \\
\langle \textit{riadok} \rangle &::= \langle \textit{instrukcia} \rangle \mid \langle \textit{navestie} \rangle\textbf{:}\ \langle \textit{instrukcia} \rangle \\
\langle \textit{instrukcia} \rangle &::= \textbf{inc}\ \langle \textit{register} \rangle \mid \textbf{dec}\ \langle \textit{register} \rangle \mid \\
&\quad\ \ \textbf{jmp}\ \langle \textit{navestie} \rangle \mid \textbf{test}\ \langle \textit{register} \rangle\textbf{,}\langle \textit{register} \rangle \mid \\
&\quad\ \ \textbf{jz}\ \langle \textit{navestie} \rangle \mid \textbf{jnz}\ \langle \textit{navestie} \rangle \\
\langle \textit{register} \rangle &::= R_0 \mid R_1 \mid \ldots \mid R_7
\end{aligned}
$$

Pričom $\langle \textit{navestie} \rangle$ je ľubovoľný reťazec neobsahujúci ':' (dvojbodku).

Inštrukcie programu sa pri jeho spustení vykonávajú zaradom, podľa toho, ako boli zapísané v programe, začínajúc od prvej inštrukcie. Výnimkou sú inštrukcie **jmp**, **jz** a **jnz** po ktorých môže program pokračovať na inom mieste.

Príklad: V registroch R_1 a R_2 sú zapísané nejaké čísla (ostatné registre sú vynulované). Napíšte program, ktorý do registra R_0 zapíše súčet čísel v registroch R_1 a R_2.

Riešenie: Najskôr budeme postupne register R_1 zmenšovať o 1 a súčasne zvyšovať hodnotu registra R_0, až kým R_1 nebude nula. Tým vlastne nastavíme R_0 na hodnotu R_1. Potom to isté spravíme aj s registrom R_2, čím k R_0 pripočítame hodnotu R_2.

```
c1: test R1,R1
    jz c2
    dec R1
    inc R0
    jmp c1
c2: test R2,R2
    jz koniec
    dec R2
    inc R0
    jmp c2
koniec:
```

Zadania celoštátneho kola kategórie A

A-III-1 Pizza vracia úder

Marec, mesiac daňových priznaní, zbrzdil rozmach Marcovej firmy. Počas rozvážania pizze a zakladania nových pobočiek nemal Marco čas zaoberať sa účtovníctvom, a teraz s hrôzou zistil, že si do svojich poznámok zabudol zapísať, čo sú príjmy a čo výdaje. Napriek tomu nespanikáril, lebo si spomenul na príbehy, ktoré mu rozprával jeho dedo (bývalá hlava talianskej mafie) a rozhodol sa doplniť (čítaj: sfalšovať) účtovné záznamy.

Aby výsledok vyzeral čo najdôveryhodnejšie, rozhodol sa použiť čísla zo svojich poznámok a vybrať si pri každom čísle, či je to výdaj alebo príjem. Naviac sa rozhodol, že keď už falšovať, tak poriadne. Rád by vyzeral ako úspešný obchodník, a teda neskončil v strate. Na druhej strane by chcel platiť čo najmenšie dane, teda jeho zisk by mal byť čo najmenší. Pri riešení tejto neľahkej úlohy by mu mohlo pomôcť to, že veľkosť čísel v jeho poznámkach je podstatne menšia ako ich počet.

Súťažná úloha: Je daných N prirodzených čísel $a_1, \ldots, a_N$ z rozsahu 1 až K. Hodnota K je väčšinou rádovo menšia ako N. Vašou úlohou je nájsť čísla $s_i \in \{-1, +1\}$ také, že súčet $z = s_1a_1 + s_2a_2 + \ldots + s_Na_N$ je nezáporný a zároveň najmenší možný.

Formát vstupu: záznamov.

Druhý riadok obsahuje N hodnôt $a_1, \ldots, a_N$ – jednotlivé účtovné záznamy.

Formát výstupu: i-tu položku ako príjem a - ak ju má brať ako výdaj.

Príklady:

Vstup

```
4
1 2 3 4
```

Výstup

```
+
-
-
+
```

Platí $+a_1 - a_2 - a_3 + a_4 = 0$ *a lepšie to zjavne nejde.*

Vstup

```
1000
2 3 3 3 3 3 ... 3 3 3 3 3 3
```

Výstup

```
-
-
-
...
+
+
+
```

Veľkú časť vstupu aj výstupu sme pre prehľadnosť vynechali.
Vo výstupe je najskôr 500 *znakov* `-`*, potom* 500 *znakov* `+`*. Výsledný zisk je 1. Zisk 0 sa zjavne dosiahnuť nedá, preto je toto riešenie optimálne.*

A-III-2 Obdĺžnik

Súťažná úloha: V rovine je daných N navzájom rôznych bodov $B_1, \ldots, B_N$.

Vašou úlohou je navrhnúť algoritmus, ktorý nájde obdĺžnik $A_1A_2A_3A_4$, ktorého vrcholy sú niektoré zo zadaných bodov (t. j. $A_1 = B_i$ pre nejaké i, $1 \leq i \leq N$, analogicky pre A_2, A_3 a A_4). Hrany obdĺžnika $A_1A_2A_3A_4$ musia byť rovnobežné s osami, teda x-ové súradnice vrcholov A_1 a A_4 a vrcholov A_2 a A_3 musia byť rovnaké a rovnako y-ové súradnice vrcholov A_1 a A_2 a vrcholov A_3 a A_4 musia byť rovnaké.

Navyše počet bodov B_i, ktoré ležia vo vnútri alebo na obvode vami nájdeného obdĺžnika $A_1A_2A_3A_4$, musí byť maximálny možný.

Formát vstupu: Nasleduje N riadkov, pričom i-ty riadok obsahuje súradnice i-teho bodu.

Formát výstupu: podmienky zo zadania. Ak sa zo zadaných bodov nedá vytvoriť žiadny vhodný obdĺžnik, vypíšte namiesto toho správu „`Neexistuje vhodný obdĺžnik`“.

Príklady:

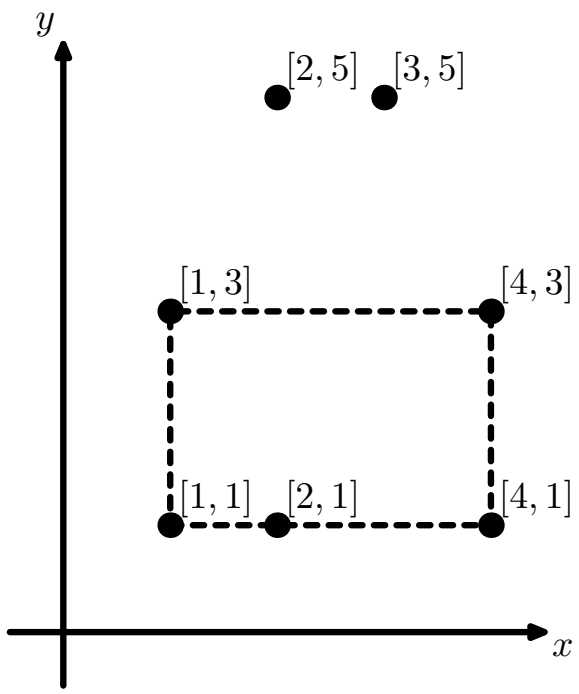
Vstup

```
7
1 1
2 1
4 1
1 3
4 3
2 5
3 5
```

Výstup

```
5
```

Takto vyzerá situácia popisovaná týmto vstupom a jediné optimálne riešenie:



Vstup

```
4
1 1
2 1
1 2
4 7
```

Výstup

```
Neexistuje vhodný obdĺžnik
```

A-III-3 Grafomat a šamani

V osadách indiánskeho kmeňa Apačov-Grafomatérov sa zajtra koná najväčšia slávnosť roka, pri ktorej majú šamani kmeňu zabezpečiť náklonnosť veľkého Manitú pri tohtoročnom love bizónov.

Rituál vyžaduje, aby sa indiáni v niektorých osadách pomaľovali na červeno a v ostatných osadách na modro. No a šamani vedia, že rituál bude úspešný len vtedy, ak bude „červených“ a „modrých“ osád rovnako. A tak teraz v každej osade šaman stojí pri svojom signálnom ohni a snaží sa so susednými osadami dohodnúť, akou farbou sa vlastne kde majú maľovať.

Súťažná úloha: Napíšte program pre grafomat, ktorý v zadanom 3-grafe s jedným označeným vrcholom označí presne polovicu vrcholov grafu a skončí.

Presnejšie, váš program sa má správať nasledovne: Na začiatku je v jednom vrchole grafu $x = 1$ a vo všetkých ostatných $x = 0$. Na konci výpočtu má byť v práve polovici vrcholov $y = 1$ (tieto vrcholy predstavujú „červené“ dediny) a v ostatných vrcholoch má byť $y = 0$.

Môžete predpokladať, že graf je súvislý a že má párny počet vrcholov, takže hľadané rozdelenie vždy existuje. (Dá sa dokázať, že dokonca každý 3-graf má párny počet vrcholov, ale pre riešenie úlohy to nie je podstatné.)

Rady a upozornenia: Ak si s úlohou neviete poradiť, rozmyslite si najskôr, ako by ste ju riešili, ak by použitým grafom nebol 3-graf ale strom (súvislý graf bez cyklov).

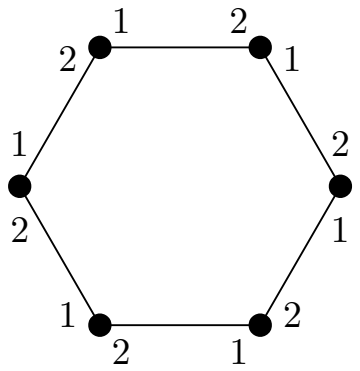
Dajte si pozor na to, že počet stavov každého automatu musí byť konečný, teda **nemôžete** používať premenné, ktorých veľkosť by závisela od veľkosti grafu.

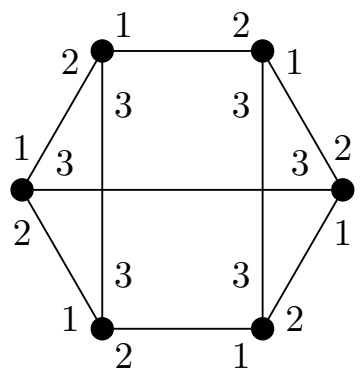
Študijný text

Študijný text je identický s tým z domáceho kola až na definíciu ukončenia výpočtu a príslušnú úpravu riešenia príkladov.

Grafom nazývame ľubovoľnú konečnú množinu V *vrcholov* grafu spolu s množinou E *hrán*, čo sú neusporiadané dvojice vrcholov. Každá hrana teda predstavuje spojenie medzi dvoma vrcholmi. Žiadne dva vrcholy nie sú spojené viac ako jednou hranou, žiadna hrana nespája vrchol sám so sebou. Počet hrán grafu budeme označovať N a počet jeho hrán M.

K-graf budeme hovoriť takému grafu, v ktorom z každého vrcholu vedie práve K hrán a konce týchto hrán sú očíslované prirodzenými číslami od 1 po K. Konce jednej hrany môžu byť očíslované rôzne. Pokiaľ budeme hovoriť o hranách vychádzajúcich z nejakého vrcholu v, budeme spomínať *miestne* čísla hrán (čísla toho konca, ktorým je v) a *vzdialené* čísla (to sú tie na opačných koncoch hrán). Nasledujúci obrázok ukazuje príklad 2-grafu a 3-grafu.





Ohodnotením grafu nazveme priradenie prvkov nejakej konečnej množiny vrcholom grafu - teda napríklad rozdelenie vrcholov na čierne a biele, alebo označenie vrcholov číslami od 1 po 5.

Grafomat je zariadenie na automatické riešenie grafových úloh. Jeho vstupom je ľubovoľný K-graf G spolu s jeho ohodnotením. Výstupom je nejaké ďalšie ohodnotenie toho istého grafu. Samotný výpočet je vykonávaný *automatmi* umiestnenými v jednotlivých vrcholoch grafu. Každý automat má svoju pamäť a riadi sa programom. Programy všetkých automatov sú identické. Okrem toho, že automat môže ľubovoľne narábať so svojou pamäťou, môže aj nahliadať do pamäte svojich susedov.

Pamäť automatu si môžeme predstaviť ako pascalovské premenné typu interval. Teda každá premenná obsahuje jedno prirodzené číslo, ktoré je z nejakého pevne zvoleného rozsahu, ktorý nezávisí na veľkosti vstupu. Okrem toho je tiež možné používať pole takýchto premenných. Opäť, rozmery poľa musia byť dopredu známe a nesmú závisieť od veľkosti vstupu. Žiadne iné typy premenných (neobmedzene veľké čísla, smerníky, ...) sa nedajú používať.

Zvláštnu rolu hrajú premenné x a y. Premenná x na začiatku výpočtu obsahuje vstupné ohodnotenie toho vrcholu grafu, v ktorom program beží, hodnota premennej y na konci výpočtu určí výstupné ohodnotenie tohto vrcholu. Všetky premenné s výnimkou premennej x majú svoju počiatočnú hodnotu pevne určenú. Deklarácia premenných vyzerá napríklad takto:

```
var x: 1..5;          { číslo od 1 do 5, na začiatku je to vstup }
    y: 1..5 = 3;          { y je na začiatku 3, na konci výstup }
    z: array [1..2] of 3..4 = (3, 4);        { dvojprvkové pole }
```

Riadiaci program automatu si môžeme predstavit ako pascalovský program, v ktorom zakážeme používanie rekurzie a dovolíme manipulovať len s premennými v pamäti automatu a s premennými v pamäti susedných automatov. Na

svoje vlastné premenné sa automat odkazuje ich menami, ako by to boli obyčajné pascalovské globálne premenné. Na to, aby sme vedeli odkazovať na premenné susedov, využijeme miestne čísla hrán. Presnejšie, nech `i` je celočíselný výraz s hodnotou z rozsahu $1 \ldots K$. Potom `S[i].p` je výraz predstavujúci premennú `p` u suseda, do ktorého od nás vedie hrana s miestnym číslom `i`. (Samozrejme, `i` môže byť ľubovoľný výraz a `p` ľubovoľné meno premennej.) Premenné susedov sa dajú len čítať.

Aby program mohol dávať do súvislosti svoje hrany s hranami svojich susedov, má k dispozící ešte premenne `P[1]` až `P[K]`, ktoré sú pevne nastavené tak, že `P[i]` obsahuje vzdialené číslo tej hrany, ktorá má miestne číslo `i`.

Použitie si ukážeme na výraze `S[i].S[P[i]].x`. Označme vrchol, v ktorom práve sme, písmenom v. Výrazy `S[i].volaco` sú premenné suseda, do ktorého sa dostaneme hranou s číslom `i`. Označme tohto suseda písmenom u. Premenná, ktorú chceme, je `S[nieco].x`. Takže chceme premennú `x` od niektorého suseda vrcholu u. Ale ktorého? Toho, do ktorého sa dostaneme hranou s číslom `P[i]`. To je ale práve vzdialené číslo hrany, ktorou sme do u prišli. Inými slovami, v u je to jeho miestne číslo hrany, ktorá vedie späť do v. Takže výraz `S[i].S[P[i]].x` predstavuje to isté ako výraz `x`.

Ak sa vám to zdá zložité, povieme si to ešte raz v dvoch krokoch. Najskôr sa vyhodnotí `nieco=P[i]`. (`P[i]` je naša lokálna premenná.) Teraz sa pozrieme na premennú `S[i].S[nieco].x`. (To prvé `S` je naše lokálne pole s premennými susedov, to druhé je podobné pole u suseda.) No a tento sused v premennej `S[nieco].x` vidí to, čo máme v našej premennej `x`.

Výpočet automatu prebieha v taktoch, a to nasledovne: V nultom takte sa premenné všetkých automatov nastavia na počiatočnú hodnotu a premenné x na vstupné ohodnotenie jednotlivých vrcholov. V každom ďalšom takte sa vždy znova spustí program každého automatu, pričom premenné svojich susedov vidí program v stave, v akom boli na začiatku taktu. Aj keď jednotlivé automaty bežia súčasne, nemôže sa teda stať, že by jeden čítal z premennej, do ktorej práve druhý zapisuje.

Zmena oproti krajskému kolu:
Výpočet pokračuje tak dlho, až kým po nejakom takte neplatí, že všetky premenné vo všetkých automatoch majú rovnakú hodnotu ako mali po predchádzajúcom takte. Inými slovami, výpočet končí, akonáhle sa v nejakom takte už nič nové nestane. Vtedy grafomat prečíta z premenných y výstupné ohodnotenie grafu.

Štruktúra grafu a obsahy premenných `P` zostávajú po celú dobu výpočtu

nezmenené.

Za *časovú zložitosť* výpočtu budeme považovať počet taktov, ktoré ubehnú po zastavenie programu. Nijako teda nezávisí na rýchlosti jednotlivých automatov. Podobne ako je to u časovej zložitosti klasických algoritmov, nebudeme hľadieť na multiplikatívne konštanty a bude nás zaujímať len asymptotické správanie zložitosti – teda či je lineárna, kvadratická, atď. Prípady, kedy výpočet neskončí, nebudeme pripúšťať, pre úplnosť ale dodajme, že vtedy sa hodnoty premenných musia nutne opakovať.

Príklad 1: Je zadaný 3-graf a v ňom vyznačený jeden vrchol v, a to tak, že jeho premenná x bude inicializovaná jednotkou a ostatné vrcholy budú mať nulu. Napíšte program pre grafomat, ktorý označí všetky vrcholy, do ktorých sa dá dostať z vrcholu v po hranách, a to tak, že ich premenná y bude mať na konci výpočtu hodnotu jedna, pre ostatné vrcholy bude mať hodnotu nula.

Riešenie: Inšpirujeme sa prehľadávaním do šírky. V každom takte sa každý vrchol pozrie, či je niektorý z jeho susedov už označený. Ak je, tak sa sám označí. Pokiaľ sa označenie vrcholu nezmení, vrchol tým vlastne hovorí, že súhlasí so zastavením. Priebeh výpočtu bude teda vyzerať tak, že po i-tom takte budú označené tie vrcholy, ktorých vzdialenosť od v je menšia alebo rovná i. Výpočet sa zastaví vtedy, keď sa hodnoty premenných prestanú meniť, čo znamená, že po najviac N taktoch. Preto je časová zložitosť nášho programu lineárna od počtu vrcholov (na rozdiel od klasického prehľadávania do šírky, ktoré závisí od počtu hrán).

Program vyzerá nasledovne:

```
var x: 0..1;                    { bol vrchol označený vo vstupe? }
    y: 0..1 = 0;                            { je označený teraz? }
    i: 1..3;
begin
   if x=1 then y := 1;          { prenesieme označenie zo vstupu }
   for i := 1 to 3 do          { pozrieme sa na všetkých susedov }
      if S[i].y <> 0 then           { ak je i-ty sused označený }
         y := 1;                             { označ aj sám seba }
end.
```

Príklad 2: Majme 2-graf zložený z jedného cyklu párnej dĺžky, teda z vrcholov očíslovaných $0 \ldots N-1$, pričom vrchol i je spojený hranou označenou 1 s vrcholom $(i+1) \bmod N$ a hranou označenou 2 s vrcholom $(i-1) \bmod N$. (Príklad takého grafu pre $N = 6$ nájdete na obrázku na začiatku tohto textu.) V tomto grafe je vyznačený jeden vrchol v, rovnako ako v predchádzajúcom

príklade. Napíšte program pre grafomat, ktorý označí vrchol protiľahlý k v, teda vrchol s číslom $(v + N/2) \bmod N$.

Riešenie: Vyšleme signál putujúci z vrcholu v v smere jednotkových hrán rýchlosťou 1 vrchol za takt. Zároveň vyšleme druhý signál putujúci rovnakou rýchlosťou opačným smerom. Akonáhle nejaký vrchol zistí, že do neho prišli oba signály, označí sa a signály už ďalej neposiela.

```
var x: 0..1;                              { vstupná značka vrcholu }
    y: 0..1 = 0;                                { výstupná značka }
    l, r: 0..1 = 0;          { už týmto vrcholom prešiel signál
begin                                           doľava a doprava? }
   if x=1 then begin
      x := 0; l := 1; r := 1;                { začíname posielať }
   end else if (S[2].l=1) and (S[1].r=1) then
      y := 1                                { signály sa stretli }
   else if (S[2].l=1) and (l=0) then
      l := 1                               { prišiel signál zľava }
   else if (S[1].r=1) and (r=0) then
      r := 1;                             { prišiel signál sprava }
end.                  { inak sa v tomto vrchole teraz nič nedeje }
```

Zjavne výpočet tohto grafomatu skončí presne po $(N/2) + 2$ taktoch. Po $(N/2) + 1$ taktoch budeme v situácii, kedy:

- začiatočný vrchol bude mať $x = 0$, $l = r = 1$,
- $N - 1$ vrcholov naľavo od neho bude mať $l = 1$,
- $N - 1$ vrcholov napravo bude mať $r = 1$,
- no a vrchol oproti bude mať $l = r = 0$ a $y = 1$.

V nasledujúcom takte sa už žiadna premenná v žiadnom vrchole nezmení, a teda výpočet končí.

A-III-4 Polícia zasahuje

V meste Blatislava sa tentokrát usídlila mafia. Keď už jej výčiny prekročili únosnú medzu, bola miestna polícia poverená zatrhnúť jej ich.

Keď však mafiáni zistili, že ich sleduje polícia, vymysleli si fintu. Začali medzi svojimi domami chodiť cez mestskú kanalizáciu. Polícia to práve včera odhalila a rozhodla sa, že postaví do kanalizácie pod niektoré domy hliadky tak, aby sa už žiadni dvaja mafiáni k sebe nedostali.

V meste je N domov. Niektoré z nich patria mafii, niektoré bežným občanom. Z každého domu sa dá dostať do kanalizácie. Kanalizačný systém vyzerá tak, že medzi každými dvoma domami sa ním dá prejsť práve jedným spôsobom (ak, pravda, nechceme ísť žiadnym miestom dvakrát). Kanalizácia je teda súvislá a neobsahuje žiadne cykly. Príklad toho, ako môže kanalizačná sieť vyzerať, nájdete pri príklade vstupu a výstupu.

Služba v kanalizácii však nikoho príliš neteší, a tak by bolo dobre zistiť, koľko najmenej strážcov zákona na túto službu treba.

Súťažná úloha: Napíšte program, ktorý načíta mapu kanalizačnej siete a popis toho, kde stoja mafiánske domy, a spočíta, koľko najmenej policajných hliadok stačí rozmiestniť, aby sa žiadni dvaja mafiáni k sebe cez kanalizáciu nedostali.

Pripomíname, že hliadky musia byť umiestnené pod niektorými z domov (je jedno, či mafiánskych alebo nie), teda nie je dovolené umiestniť hliadku do stoky medzi dva domy.

Ak umiestnime nejakú hliadku priamo pod dom mafiána, tento mafián sa už cez kanalizáciu nedostane nikam.

Formát vstupu: Prvý riadok vstupu obsahuje dve celé čísla N ($3 \leq N \leq 100\,000$) a P ($2 \leq P < N$). Číslo N je počet domov, a zároveň teda počet významných miest v kanalizácii. Kanalizačný systém je tvorený presne $N - 1$ stokami, pričom každá stoka priamo spája niektoré dve významné miesta (a teda dva domy). Domy sú očíslované číslami od 1 do N. Číslo P je počet domov, ktoré patria mafii.

Nasleduje $N - 1$ riadkov, každý z nich obsahuje čísla jednej dvojice domov, ktorá je spojená stokou.

Každý z posledných P riadkov obsahuje jedno číslo domu patriaceho mafii.

Formát výstupu: Vypíšte jeden riadok a v ňom jedno celé číslo – najmenší počet hliadok, ktorý stačí umiestniť pod niektoré domy tak, aby sa medzi žiadnymi dvoma mafiánskymi domami nedalo cez kanalizáciu prejsť.

Príklad:

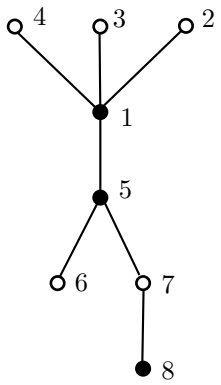
Vstup

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

Výstup

```
2
```

Jedna možnosť ako umiestniť dve hliadky je pod domy s číslami 1 a 5. Na obrázku prázdne krúžky predstavujú domy mafiánov.



A-III-5 Rybka Julka

Za horúcich letných dní voda v rybníku Blaťáku občas skoro vrie, a tak sa malé rybky vždy utekajú ukryť do hlbokých chladných tôní. Len rybka Julka sa dnes akosi pozabudla, a už začína byť v rybníku nepríjemne horúco. Pomôžte jej dostať sa čo najrýchlejšie do úkrytu!

Rybník Blaťák má tvar obdĺžnika a je rozdelený na $R \times S$ rovnako veľkých štvorcových políčok. V tomto okamihu má políčko so súradnicami $[r, s]$ teplotu $T_{r,s}$ rybích stupňov.

Všetky časti rybníka sa ohrievajú rovnakým tempom: každú sekundu teplota stúpne o jeden rybí stupeň.

Julka znesie rozsah teplôt od t_1 po t_2 rybích stupňov (vrátane oboch hraníc). V tomto okamihu sa nachádza na súradniciach $[J_r, J_s]$ a potrebovala by sa dostať do svojho úkrytu, ktorý má súradnice $[C_r, C_s]$. Počas každej sekundy sa vie Julka presunúť na jedno zo štyroch stranou susediacich políčok, prípadne sa môže rozhodnúť počkať na mieste. Vždy po tom, ako Julka vykoná jednu akciu, sa teplota rybníka zvýši.

Julka sa nesmie cestou do cieľa opustiť rybník, nesmie sa prehriať a nesmie ani prechladnúť. Môže sa teda pohybovať len po políčkach, ktorých aktuálna

teplota T je z jej rozsahu, teda platí $t_1 \leq T \leq t_2$. Táto podmienka musí byť splnená od okamihu, kedy Julka na políčko pripláva, až do okamihu, kým z neho odpláva (medzitým teplota určite stúpla aspoň o stupeň). Výnimkou je cieľové políčko, kam môže vplávať bez ohľadu na jeho teplotu.

Súťažná úloha: Napíšte program, ktorý načíta popis celej situácie, zistí, v akom najkratšom čase sa vie Julka dostať do úkrytu a poradí jej jednu možnú optimálnu cestu.

Formát vstupu: V prvom riadku vstupu sú štyri celé čísla R, S, t_1 a t_2 oddelené medzerami.

Číslo R udáva počet „riadkov“ a S počet „stĺpcov“, na ktoré je rozdelený rybník. Platí $1 \leq R, S \leq 1000$. Čísla t_1 a t_2 udávajú Julkin teplotný interval, platí $0 \leq t_1 < t_2 \leq 10^6$.

V druhom riadku vstupu sú štyri celé čísla J_r, J_s, C_r a C_s – súradnice Julkinho štartu a cieľa. Platí $1 \leq J_r, C_r \leq R$ a $1 \leq J_s, C_s \leq S$.

Môžete predpokladať, že cieľové políčko je rôzne od začiatočného.

Rybník Blaťák je orientovaný tak, že na sever od políčka so súradnicami $[r, s]$ je políčko so súradnicami $[r-1, s]$, a na západ od $[r, s]$ je $[r, s-1]$.

Posledných R riadkov vstupu popisuje začiatočné teploty políčok rybníka. Presnejšie, v riadku $r+2$ je S medzerami oddelených celých čísel: teploty $T_{r,1}$, $T_{r,2}$, …, $T_{r,s}$. Pre každé r, s platí $0 \leq T_{r,s} \leq 10^6$.

Môžete predpokladať, že na začiatku je teplota políčka, kde sa Julka práve nachádza, pre ňu prijateľná.

Formát výstupu: Do výstupného súboru vypíšte jeden riadok a v ňom popis jednej optimálnej (teda najrýchlejšej) Julkinej cesty.

Popis cesty sa skladá z niekoľkých pokynov, ktoré Julka postupne vykoná. Pokyn je buď jeden znak `S`, `J`, `V`, `Z`, ktorý hovorí, že Julka sa má pohnúť smerom do danej svetovej strany, alebo kladné celé číslo, ktoré hovorí, koľko sekúnd má Julka zostať čakať na aktuálnom políčku.

Pokyny oddeľujte od seba jednou medzerou. Posledný pokyn musí byť pohyb. Nikdy nesmú po sebe nasledovať dva pokyny na čakanie.

Ak existuje viac rôznych optimálnych ciest, môžete si vybrať a vyššie uvedeným spôsobom popísať ľubovoľnú z nich.

Ak neexistuje žiadny spôsob ako Julku zachrániť, namiesto popisu cesty vypíšte vetu „`Chudinka Julka!`“ (bez uvodzoviek).

Príklady:

Vstup

```
4 3 10 20
2 2 4 3
9 7 13
0 18 15
1 15 19
2 3 0
```

Výstup

```
V S 1 Z Z 5 J J J V V
```

Toto je jedna z viacerých možných najrýchlejších ciest. Ľubovoľná z nich by bola správnym riešením, na počte príkazov nezáleží.

Vstup

```
2 3 20 30
1 1 1 3
25 13 25
27 24 0
```

Výstup

```
Chudinka Julka!
```

Skôr ako sa niektoré z políčok, ktoré majú na začiatku teplotu 0 a 13, zohreje na dostatočnú teplotu, ostatné prekročia Julkinu maximálnu teplotu. Najneskôr po siedmich sekundách chudinka Julka zahynie.

Riešenia domáceho kola kategórie A

A-I-1 Rozvoz pizze

Úlohu budeme riešiť „pažravo“ (greedy). Najskôr si spočítame, koľko kusov akej veľkosti potrebujeme upiecť – teda koľko celých pízz, koľko kusov veľkosti 5/6, atď. Toto vieme ľahko spraviť priamo pri načítaní vstupu.

Celé pizze nerobia žiadne problémy. Podobne s kusmi veľkosti 5/6 je to jednoduché – kvôli každému musíme upiecť jednu pizzu. Z tej nám 1/6 zvýši, a zjavne nič nepokazíme, ak tieto kúsky použijeme na splnenie ďalších objednávok. Ak už objednávky na 1/6 nemáme, zvyšné šestiny musíme zahodiť.

Pre kusy veľkosti 4/6 je situácia podobná. Na každý kus treba upiecť jednu celú pizzu, zostane nám kus veľkosti 2/6. Ak máme málo objednávok na takto veľké kusy, môžeme ho ešte rozkrojiť napoly a použiť na objednávky 1/6 pizze. (Zjavne je lepšie najskôr uspokojiť objednávky na 2/6 pizze.)

Doteraz bolo všetko takmer jednoznačne určené, a teda nami navrhnuté krájanie je zatiaľ optimálne. Zostáva nám vyriešiť situáciu, kedy všetky ešte neuspokojené objednávky sú na 3/6, 2/6 alebo 1/6 pizze.

♡: Objednávky 3/6 pizze vyriešime tak, že upečieme niekoľko pízz a každú rozkrojíme napoly. (Neskôr ukážeme, prečo je tento spôsob optimálny.) Ak bol počet objednávok 3/6 pizze nepárny, zostala nám polovica pizze. Ak ešte máme objednávku na 2/6 pizze, odkrojíme z tejto polovice 2/6, zvyšok potom (ak treba) použijeme na objednávku 1/6 pizze. V opačnom prípade rozrežeme zostávajúcu polovicu na tri kúsky po 1/6 pizze.

♠: Už teda mohli zostať objednávky len na 2/6 a 1/6 pizze. Aby sme tie uspokojili, budeme piecť pizze a krájať ich na tretiny, až kým sa nám neminú objednávky na 2/6 pizze. Zostávajúce objednávky na 1/6 pizze už vyriešime ľahko.

Prečo je tento postup optimálny?

Dokážeme, že pri uspokojovaní objednávok na 3/6, 2/6 a 1/6 pizze nám dokopy zvýšilo menej ako jedna celá pizza: Sú nasledujúce možnosti:

- Ak nám z časti ♡ nezostali žiadne zvyšky (teda počet objednaných polovíc bol párny, alebo bolo dosť vhodných objednávok na 2/6 a 1/6), tak všetky zvyšné kusy vzniknú v časti ♠. Tam ale zostane nevyužitá len časť poslednej upečenej pizze.

- Ak nám z časti ♡ zostala 1/6 pizze, znamená to, že už nezostali žiadne objednávky na 1/6 pizze. Potom ale v časti ♠ vyrábame len časti veľké 2/6, a teda nám zvýšia nanajvýš 4/6 pizze. (Z poslednej pizze sme vykrojili aspoň jednu časť veľkosti 2/6.) Dokopy teda zvýši najviac 5/6 pizze.

- Ak nám z časti ♡ zostali 2/6 alebo 3/6 pizze, už nemáme žiadne objednávky, a teda toto je celý zvyšok.

Pre objednávky na viac ako 3/6 pizze bolo optimálne delenie jednoznačne určené. Ak sa to dalo, zvyšky sme použili na uspokojenie menších objednávok. Táto časť riešenia je teda zjavne optimálna. Zvyšné objednávky nebolo možné uspokojiť upečením menšieho počtu pízz ako v našom riešení. (Ak by to šlo, muselo by pri našom riešení zvýšiť dosť odrezkov na celú pizzu, to ale nie je pravda.)

Riešenie má lienárnu časovú a konštantnú pamäťovú zložitosť.

Listing programu:

```
program pizza;
var
  n : word; { Počet objednávek }
  pozadavek : array[1..6] of longint; { Kolik dílů jaké velikosti potřebuji? }
  napect : longint; { Kolik pizz je třeba upéct? }

procedure nacti;
var
  F : Text;
  i, c : Integer;
begin
  for i := 1 to 6 do pozadavek[i] := 0;
  assign(F, 'pizza.in');
  reset(F);
  readln(F, n);
  for i := 1 to n do begin
    readln(F, c);
    { Vyřídíme nejdříve počet celých pizz }
    pozadavek[6] := pozadavek[6] + c div 6;
    { Nyní zvýšíme počet dílů příslušné velikosti }
    c := c mod 6;
    if c > 0 then inc(pozadavek[c]);
  end;
  close(F);
end;

procedure vypis;
var F : text;
begin
```

```
  assign(F, 'pizza.out');
  rewrite(F);
  writeln(F, napect);
  close(F);
end;

function min(a,b : longint) : longint;
begin if a < b then min := a else min := b; end;

procedure spocti;
begin
  { Určitě potřebuji tolik pizz, kolik bylo objednávek na celé pizzy }
  napect := pozadavek[6];

  { Pro každý díl velký 5/6 potřebuji jednu pizzu }
  napect := napect + pozadavek[5];

  { Díly velké 5/6 lze doplnit pouze díly velkými 1/6 }
  pozadavek[1] := pozadavek[1] - min(pozadavek[5], pozadavek[1]);

  { Pro každý díl velký 4/6 potřebuji jednu pizzu }
  napect := napect + pozadavek[4];

  { Doplníme díly velkými 2/6 a 1/6 }
  if pozadavek[4] > pozadavek[2] then begin
    pozadavek[4] := pozadavek[4] - pozadavek[2];
    pozadavek[2] := 0;
    pozadavek[1] := pozadavek[1] - min(2*pozadavek[4], pozadavek[1]);
  end else
    pozadavek[2] := pozadavek[2] - pozadavek[4];

  { Díly velké 3/6=1/2 jdou kombinovat spolu }
  napect := napect + (pozadavek[3]+1) div 2;

  { Pokud je počet polovin lichý, doplníme je z 2/6 a 1/6 }
  if pozadavek[3] mod 2 > 0 then begin
    if pozadavek[2] > 0 then begin
      dec(pozadavek[2]);
      if pozadavek[1] > 0 then
        dec(pozadavek[1]);
    end else
      pozadavek[1] := pozadavek[1] - min(pozadavek[1], 3);
  end;

  { Nyní pro zbylé díly velké 2/6 = 1/3 }
  napect := napect + (pozadavek[2]+2) div 3;

  { Doplníme případnou 1 necelou pizzu z dílu 1/6 }
  if pozadavek[2] mod 3 > 0 then
    pozadavek[1] := pozadavek[1] -
```

```
      min(pozadavek[1], 2*(3 - pozadavek[2] mod 3));

  { Nyní díly velké 1/6 }
  napect := napect + (pozadavek[1]+5) div 6;
end;

begin
  nacti;
  spocti;
  vypis;
end.
```

A-I-2 Zasypané mesto

Najprv si popíšeme jednoduchšie riešenie, ktoré bude časovo a pamäťovo náročnejšie. V tomto riešení si budeme celú štvorcovú sieť uchovávať v pamäti ako dvojrozmerné pole. Zaznamenáme do nej, pod ktorými políčkami je piesok a pod ktorými je kamenie. Naviac označíme všetky políčka ako *nenavštívené* (nenavštívené políčka budú zodpovedať políčkam miestností, ktoré sme doteraz nenašli pri prechode sieťou).

Štvorcovou sieťou budeme prechádzať postupne po riadkoch a hľadať políčko s pieskom, ktoré je zatiaľ *nenavštívené*. Keď ho nájdeme, vieme, že sme práve objavili novú zasypanú miestnosť. Teraz prestaneme spracovávať políčka postupne po riadkoch a namiesto toho teraz *navštívíme* všetky políčka novo nájdenej miestnosti.

Potom, čo navštívíme a spracujeme všetky políčka novej miestnosti, budeme pokračovať v prechádzaní štvorcovej siete po riadkoch tam, kde sme skončili. Skončíme, až prejdeme celú sieť.

Vynechali sme zatiaľ jednu dôležitú časť: Ako navštíviť všetky políčka práve objavenej miestnosti? To urobíme pomocou *prehľadávánia do šírky*. Začneme v novo nájdenom políčku, ktoré označíme ako *navštívené*. Potom budeme prechádzať jeho susedné políčka a každé také políčko, pod ktorým je piesok a ktoré je *nenavštívené*, označíme ako *navštívené* a spracujeme. Spracovaním políčka rozumieme to, že skontrolujeme, či nemá nejakého *nenavštíveného* suseda a ak áno, tak tohoto suseda označíme ako *navštíveného* a pridáme ho na koniec zoznamu políčok, ktoré musíme spracovať. Políčka sú teda spracovávané v poradí, v ktorom sme ich prvýkrát navštívili.

Takéto riešenie má časovú i pamäťovú zložitost lineárne závislú od počtu políčok siete, teda $O(NM)$. Celá sieť sa nám ale vzhľadom k obmedzeniam zo

zadania úlohy do pamäte nevojde, a tak musíme navrhnúť pamäťovo úspornejšie riešenie.

Zadanú sieť budeme spracovávať po riadkoch. V jednom kroku spracujeme vždy jeden riadok, pričom si budeme pamätať predchádzajúci riadok. Potom, čo riadok spracujeme, stane sa z neho predchádzajúci riadok, načítame nový a budeme pokračovať, až kým nespracujeme všetky políčka siete.

Teraz si popíšeme, ako vlastne budeme riadky spracovávať. Naším cieľom je, aby všetky políčka s pieskom na spracovávanom riadku boli očíslované (ofarbené) číslami 1 až b. Musia byť naviac očíslované tak, aby platilo: Dve políčka s pieskom majú tú istú farbu **práve vtedy**, ak z doteraz spracovaných riadkov (vrátane aktuálneho) o nich vieme, že patria do tej istej miestnosti.

Celý algoritmus teda bude fungovať nasledovne. Na začiatku pridáme pred celú sieť riadok políčok s kamením (ten je už spracovaný, keďže na ňom žiadne políčka s pieskom nie sú). Potom načítame ďalší riadok a pomocou predchádzajúceho riadku ho spracujeme – teda ofarbíme ho tak, aby sme dodržali vyššie stanovenú podmienku. (Podrobnejšie si tento krok popíšeme neskôr.)

V priebehu spracúvania ďalšieho riadku spočítame, koľko na predchádzajúcom riadku skončilo miestností (je to počet farieb, ktoré nesusedia so žiadnym pieskovým políčkom na aktuálnom riadku) a tento počet pripočítame k celkovému počtu objavených miestností.

Takýto algoritmus je určite správny: každá miestnosť niekedy skončí, takže ju určite započítame aspoň raz. Na druhej strane, kvôli tomu, ako ofarbujeme spracovávané riadky, nikdy nemôžeme žiadnu miestnosť započítať viackrát než raz. Zložitosť algoritmu bude M krát zložitosť spracovania jedného riadku.

Ešte teda musíme vyriešiť spracovanie načítaného riadku. Predchádzajúci riadok už máme ofarbený farbami 1 až b. Aktuálny riadok najprv ofarbíme farbami od $b+1$ do c tak, že susedné pieskové políčka tohto riadku dostanú rovnakú farbu. Potom využijeme informácie z predchádzajúceho riadku: pokiaľ pieskové políčko spracovávaného riadku susedí s pieskovým políčkom predchádzajúceho riadku, musíme „zlúčiť“ ich farby.

Pre každú farbu F si spravíme zoznam tých farieb, ktoré sa vyskytujú v nejakej miestnosti, v ktorej je použitá aj farba F. Tieto zoznamy zostrojíme jednoducho tak, že prejdeme celý riadok a vždy, keď nájdeme dvojicu farieb, ktoré treba zlúčiť, do zoznamu jednej zlučovanej farby pridáme druhú a naopak.

Všimnite si ale, že takto zostrojené zoznamy nemusia priamo obsahovať celé množiny „ekvivalentných“ farieb. Ak sme napríklad zaznamenali, že treba zlúčiť farby 1 a 2 aj farby 2 a 3, znamená to, že všetky tri farby 1, 2 aj 3 treba nahradiť jednou novou. Naším cieľom bude teda pre jednu farbu nájsť všetky

s ňou ekvivalentné farby. Všetky tieto farby potom prefarbíme na jednu novú farbu.

Na to použijeme vyššie popísaný postup prehľadávania do šírky: začneme s počiatočnou farbou a označíme ju ako použitú (ostatné farby sú na začiatku označené ako nepoužité). Potom budeme prechádzať jej zoznam farieb a ak narazíme na farbu, ktorá je zatiaľ nepoužitá, pridáme ju do zoznamu farieb, ktoré musíme spracovať. Až celý proces skončí, budeme poznať všetky farby, ktoré majú so zadanou farbou spoločnú miestnosť. Takéto riešenie spracuje jeden riadok v čase $O(N)$, a teda má časovú zložitosť $O(MN)$ a pamäťovú $O(N)$.

Všimnime si teraz, že na spracovanie riadku si stačí pamätať intervaly tvorené pieskom a ich farby. Teda nemusíme pracovať ani len s poľom dĺžky N, ktoré by reprezentovalo predchádzajúci a nový riadok štvorcovej siete. Pokiaľ predchádzajúci riadok obsahuje K_1 intervalov tvorených pieskom a nový K_2 takých intervalov, bude nám spracovanie týchto dvoch riadkov trvať $O(K_1+K_2)$, pretože počet dvojíc pretínajúcich sa intervalov, teda tých, pre ktoré musíme zlúčiť nejakú dvojicu farieb, je najviac $K_1 + K_2 - 1$. Samotné zlučovanie farieb (prehľadávanie do šírky) je potom lineárne od počtu dvojíc farieb, ktoré je treba zlúčiť. Celková časová zložitosť nášho algoritmu bude teda súčet hodnôt K_1 a K_2 cez všetky riadky, tj. $O(K + M)$ (do odhadu časovej zložitosti musíme započítať aj počet riadkov pre prípad, že by štvorcová sieť obsahovala veľa prázdnych riadkov, tj. K by bylo oveľa menšie než M). Pamäťová zložitosť je najviac lineárna od počtu políčok na jednom riadku, tj. $O(N)$.

Listing programu:

```
program mesto;
type usek = record              { datový typ pro načtený úsek písku }
              barva, zacatek, konec: word;
            end;
const maxN = 50000;

type radek = record             { datový typ pro řádek mapy }
               pocet_useku, pocet_barev: word;
               useky: array [1..maxN] of usek;
             end;

var vstup, vystup: text;        { vstupní a výstupní soubor }
    M, N: word;                 { rozměry mapy }
    L: longint;                 { počet dosud nalezených místností }
    predchozi_radek: radek;     { řádek, který byl už zpracován }
    novy_radek: radek;          { řádek, který je právě zpracováván }

procedure inicializuj;
begin
```

```
  assign(vstup, 'mesto.in');
  reset(vstup);
  readln(vstup, M, N, L);
  L := 0;
  predchozi_radek.pocet_useku := 0;
  predchozi_radek.pocet_barev := 0;
end;

type barva=record       { datový typ pro seznam barev, které mají být sloučeny }
             cislo: word;
             dalsi: ^barva;
           end;

procedure sluc_barvy;
  { provede sloučení místností uložených v predchozi_radek a novy_radek;
    zvýší L o jedna za každou místnost, která už nepokračuje na novém řádku }
var ma_byt_slouceno: array[1..2*maxN] of ^barva;
      { seznam barev, které mají být sloučeny }
    vysledna_barva: array[1..2*maxN] of word;
      { barva, na kterou má být přebarveno }
    novy_pocet_barev: word;             { nový počet barev }
    seznam: array[1..2*maxN] of word;   { seznam pro prohledávání při slučování }
    prvku_v_seznamu: word;              { počet prvků v seznamu }
    i, j: word;                         { několik pomocných proměnných }
    b:^barva;
begin
  for i := 1 to novy_radek.pocet_barev do ma_byt_slouceno[i] := nil;
  i := 1; j := 1;
  while (i <= predchozi_radek.pocet_useku) and (j <= novy_radek.pocet_useku) do
    begin
      if (predchozi_radek.useky[i].zacatek < novy_radek.useky[j].konec) and
         (novy_radek.useky[j].zacatek < predchozi_radek.useky[i].konec) then
        begin
          new(b);
          b^.cislo := predchozi_radek.useky[i].barva;
          b^.dalsi := ma_byt_slouceno[novy_radek.useky[j].barva];
          ma_byt_slouceno[novy_radek.useky[j].barva] := b;
          new(b);
          b^.cislo := novy_radek.useky[j].barva;
          b^.dalsi := ma_byt_slouceno[predchozi_radek.useky[i].barva];
          ma_byt_slouceno[predchozi_radek.useky[i].barva] := b;
        end;
      if predchozi_radek.useky[i].konec < novy_radek.useky[j].konec
        then inc(i) else inc(j);
    end;
  novy_pocet_barev := 0;
  for i := 1 to novy_radek.pocet_barev do vysledna_barva[i] := 0;
  for i := predchozi_radek.pocet_barev+1 to novy_radek.pocet_barev do
    if vysledna_barva[i] = 0 then
      begin
```

```
        inc(novy_pocet_barev);
        vysledna_barva[i] := novy_pocet_barev;
        seznam[1] := i;
        prvku_v_seznamu := 1;
        j := 0;
        while j < prvku_v_seznamu do
          begin
            inc(j);
            b := ma_byt_slouceno[seznam[j]];
            while b<>nil do
              begin
                if vysledna_barva[b^.cislo] = 0 then
                  begin
                    vysledna_barva[b^.cislo] := novy_pocet_barev;
                    inc(prvku_v_seznamu);
                    seznam[prvku_v_seznamu] := b^.cislo;
                  end;
                b := b^.dalsi;
              end;
          end;
      end;
  for i := 1 to predchozi_radek.pocet_barev do
    begin
      if vysledna_barva[i] = 0 then inc(L);
      while ma_byt_slouceno[i] <> nil do
        begin
          b := ma_byt_slouceno[i]^.dalsi;
          dispose(ma_byt_slouceno[i]);
          ma_byt_slouceno[i] := b;
        end;
    end;
  novy_radek.pocet_barev := novy_pocet_barev;
  for i := 1 to novy_radek.pocet_useku do
    novy_radek.useky[i].barva := vysledna_barva[novy_radek.useky[i].barva];
  predchozi_radek := novy_radek;
end;

procedure zpracuj_radek;        { načte řádek a zavolá proceduru sluc_barvy }
var pisek, kameni: word;        { načtená dvojice čísel }
    sloupec: word;              { pozice na načítaném řádku }
begin
  sloupec := 1;
  novy_radek.pocet_useku := 0;
  novy_radek.pocet_barev := predchozi_radek.pocet_barev;
  while sloupec <= N do
    begin
      readln(vstup, pisek, kameni);
      if pisek<>0 then
        begin
          if (novy_radek.pocet_useku > 0) and
```

```
            (sloupec = novy_radek.useky[novy_radek.pocet_useku].konec+1) then
          novy_radek.useky[novy_radek.pocet_useku].konec :=
            novy_radek.useky[novy_radek.pocet_useku].konec + pisek
        else
          begin
            inc(novy_radek.pocet_useku);
            inc(novy_radek.pocet_barev);
            novy_radek.useky[novy_radek.pocet_useku].zacatek := sloupec;
            novy_radek.useky[novy_radek.pocet_useku].konec := sloupec+pisek-1;
            novy_radek.useky[novy_radek.pocet_useku].barva :=
              novy_radek.pocet_barev;
          end;
      end;
    sloupec := sloupec + pisek + kameni;
  end;
  sluc_barvy;
end;
procedure ukonci;  { odsimuluje poslední řádek jako řádek jen s kamením }
begin
  novy_radek.pocet_useku := 0;
  sluc_barvy;
end;

procedure vypis;   { vypíše počet místností do výstupního souboru }
begin
  assign(vystup, 'mesto.out');
  rewrite(vystup);
  writeln(vystup, L);
  close(vystup);
end;

var i: word;
begin
  inicializuj;
  for i:=1 to M do zpracuj_radek;
  ukonci;
  vypis;
end.
```

A-I-3 Okružná jazda

Na úvod zavedieme označenie, ktoré sa štandardne používa v teórii grafov. Mapu Blatislavy budeme volať *graf*, križovatkám budeme hovoriť *vrcholy* a uliciam *hrany*. Hranu, ktorá vedie z vrcholu u do vrcholu v, budeme značiť uv. Počet vrcholov grafu označíme N a počet jeho hrán M.

Vstupný stupeň vrcholu je počet hrán, ktoré do neho vchádzajú, teda počet hrán, ktoré do neho vchádzajú. Podobne *výstupný stupeň* je počet hrán, ktoré z vrcholu vychádzajú. Zjavne nutnou podmienkou existencie riešenia našej úlohy je, aby sa v každom vrchole vstupný a výstupný stupeň rovnali. To isté inými slovami: Ak máme nejaký vrchol, ktorý má rôzny vstupný a výstupný stupeň, môžeme si byť istí, že okružná cesta neexistuje.

Hrany e a f na seba *nadväzujú*, ak e vchádza do vrcholu, z ktorého f vychádza. Každú takúto dvojicu hrán budeme v tomto riešení označovať slovom *prechod*. Postupnosť **navzájom rôznych** hrán $e_1, \ldots, e_k$ takú, že pre všetky i hrana e_{i+1} nadväzuje na hranu e_i, voláme *ťah*. Ťah, v ktorom navyše aj e_1 nadväzuje na e_k, voláme *uzavretý*. Ťah, ktorý obsahuje všetky hrany grafu, voláme *eulerovský*.

Našu úlohu (nájsť okružnú cestu) teraz môžeme formulovať takto: treba nájsť uzavretý eulerovský ťah, ktorý neobsahuje žiadny zo zakázaných prechodov.

Naše riešenie bude pozostávať z dvoch krokov. Najskôr nájdeme ľubovoľný eulerovský ťah, a ten potom upravíme tak, aby neobsahoval zakázané prechody.

Ako zostrojiť eulerovský ťah:

Ako začiatok si zjavne môžeme vybrať ľubovoľný vrchol. Teraz sa budeme ľubovoľným spôsobom prechádzať po grafe, pričom farbíme hrany, ktorými sme už išli a nejdeme tou istou hranou dvakrát. Časom sa dostaneme do situácie, že sa už nedá ísť ďalej. Keďže vždy pri prechode vrcholom ofarbíme jednu hranu, ktorá doň vchádza, a jednu, ktorá z neho vychádza, je len jedna možnosť, kde skončíme: vo vrchole, v ktorom sme začali.

Ak náš ťah prechádza všetkými hranami grafu, skončili sme. V opačnom prípade ho ideme zlepšovať. Pozrime sa na graf tvorený len neofarbenými hranami. Opäť platí, že každý vrchol v ňom má rovnaký vstupný a výstupný stupeň. Nájdeme prvý vrchol v v tomto grafe, ktorý nie je izolovaný a leží na už zostrojenom ťahu. (Ak takýto vrchol neexistuje, znamená to, že zadaný graf nie je *silne súvislý* a okružná cesta neexistuje.) Opäť ľubovoľným prechádzaním sa po neofarbených hranách nájdeme nejaký ťah začínajúci aj končiaci vo v. Pôvodne zostrojený ťah vieme teraz predĺžiť: vo v ho rozpojíme a vložíme doň tento nový. Takto pokračujeme, kým nemáme ťah prechádzajúci všetkými hranami, alebo nezistíme, že neexistuje.

Tento algoritmus sa dá implementovať tak, aby mal časovú aj pamäťovú zložitosť $O(M + N)$, teda lineárnu od veľkosti grafu.

Ako sa zbaviť zakázaných prechodov:

Najskôr vyriešime vrcholy, ktoré majú vstupný stupeň 1 a 2.

Ak má nejaký vrchol vstupný stupeň 1, zakázaný prechod spôsobí, že sa týmto vrcholom nedá prejsť, a teda okružná cesta neexistuje.

Ak má nejaký vrchol vstupný stupeň 2, zakázaný prechod spôsobí, že je práve jeden spôsob, ako cez tento vrchol prejsť. Ak teda do nášho vrcholu v vchádzajú hrany u_1v a u_2v, vychádzajú hrany vw_1 a vw_2, a prechod z u_1v na vw_2 je zakázaný, môžeme vrchol v a všetky štyri hrany z grafu vyhodiť a namiesto nich dať dve nové hrany u_1w_1 a u_2w_2.

(V našej implementácii sme graf nemenili, len sme si dali pozor, aby sme nepoužili zakázaný prechod: Ak pri zostrojovaní ťahu prídeme do v, dáme si pozor, aby sme odišli správnou hranou. Vo v nezačíname hľadanie nového ťahu.)

Predpokladajme teda, že náš graf obsahuje len vrcholy so vstupným stupňom 3 a viac. Ak vyššie uvedený algoritmus zistil, že neexistuje žiaden eulerovský ťah, tým skôr neexistuje okružná cesta. Zostáva vyriešiť prípad, keď sme nejaký eulerovský ťah našli. Ukážeme, že za daných predpokladov tento ťah vždy vieme upraviť tak, aby nepoužíval zakázané prechody.

Nech $e_1, \ldots, e_m$ je uzavretý eulerovský ťah. Bez ujmy na všeobecnosti predpokladajme, že prechod $e_m e_1$ cez vrchol v je zakázaný. Keďže v má stupeň aspoň 3, existujú ďalšie dve hrany e_a a e_b $(a < b)$, ktoré do neho vchádzajú. Potom ťah:

$$e_1, e_2, \ldots, e_a, e_{b+1}, e_{b+2}, \ldots, e_m, e_{a+1}, e_{a+2}, \ldots, e_b$$

je tiež eulerovský, a navyše zakázaný prechod cez v nepoužíva.

Opakovaním tohoto postupu postupne odstránime všetky zakázané prechody. Takéto riešenie má časovú zložitosť $O(ZM)$, kde Z je počet zakázaných prechodov v ťahu nájdenom na začiatku. V najhoršom prípade teda máme časovú zložitosť $O(NM)$.

Vzorové riešenie:

Namiesto toho, aby sme sa zlých prechodov zbavovali, budeme už prvý eulerovský ťah zostrojovať tak, aby sme sa im vyhli. To dosiahneme tak, že nebudeme voliť hranu, ktorou vrchol opustíme, úplne náhodne.

Náš algoritmus zaruči, že po každom predĺžení ťahu budú platiť nasledujúce podmienky:

(1) Aktuálny ťah neobsahuje zakázané prechody.

(2) Ak aktuálny ťah aspoň raz prechádza cez vrchol v a zakázaný prechod cez v je ef, tak aspoň jedna z hrán e a f je už použitá.

Ako predĺžiť ťah a neporušiť pri tom tieto podmienky?

Keď prechádzame nejakým vrcholom, sú tri možnosti: Ak sme ním už išli, zakázaný prechod už nevyrobíme. Ak sme ním ešte nešli a prichádzame prvou hranou jeho zakázaného prechodu, odídeme **inou hranou** ako druhou hranou jeho zakázaného prechodu. (Taká určite existuje.) Ak prichádzame ľubovoľnou inou hranou, odídeme druhou hranou zakázaného prechodu. V každom prípade sa nám podarilo zakázaný prechod „rozbiť“.

Zostáva doriešiť, v ktorom vrchole a ako začať ťah predlžovať. Začneme vo vrchole v, ktorý má vstupný stupeň aspoň 3, leží na existujúcom ťahu a vedie z neho aspoň jedna nepoužitá hrana. Tam stačí použiť pravidlo takmer identické tomu vyššie: ak hrana, ktorou v pôvodnom ťahu prídeme do v, je v jeho zakázanom prechode, odídeme tak, aby sme zakázaný prechod nevyrobili, inak odídeme prednostne tou, ktorá je v jeho zakázanom prechode. Rozmyslite si, že takto vieme zaručiť, že ani na začiatku ani na konci vloženej časti ťahu zakázaný prechod nevznikne.

Tento postup sa dá implementovať s časovou zložitosťou $O(M + N)$.

Listing programu:

```
program eulerovsky_tah;

const MAXN = 100;

type phrana = ^hrana;
     hrana = record
               z, k: integer;         { hrana z vrcholu z do vrcholu k }
               pr, dal: phrana;       { předchozí a následující hrana v seznamu }
               tah: integer;          { číslo tahu, do nějž byla hrana přidána }
               tah_p, tah_d: phrana;  { předchozí a následující hrana v tahu }
             end;

     vrchol = record
                stupen: integer;       { stupeň vrcholu }
                pouzite: hrana;        { hlava seznamu použitých hran z vrcholu }
                nepouzite: hrana;      { hlava seznamu nepoužitých hran }
                zak_z, zak_do: phrana; { přechod ze zak_z na zak_do je zakázaný }
              end;

var n: integer;                        { počet vrcholů grafu }
    a: integer;                        { číslo aktuálního tahu }
    v: integer;                        { aktuální vrchol }
    graf: array[1..MAXN] of vrchol;
    nr: integer;
    rozpracovane: array[1..MAXN] of integer; { seznam rozpracovaných vrcholů }
    nh: integer;                       { počet hotových vrcholu }
```

```
{ odebere hranu ze seznamu }
procedure odeber (hrana: phrana);
begin
  hrana^.dal^.pr := hrana^.pr;
  hrana^.pr^.dal := hrana^.dal;
  hrana^.dal := nil;
  hrana^.pr := nil;
end;

{ přidá hranu na začátek seznamu }
procedure pridej (var seznam: hrana; hrana: phrana);
begin
  hrana^.dal := seznam.dal;
  hrana^.pr := @seznam;
  seznam.dal := hrana;
  hrana^.dal^.pr := hrana;
end;

{ vrátí true, pokud je seznam prázdný }
function prazdny (var seznam: hrana): boolean;
begin prazdny := (seznam.dal = @seznam); end;

{ označí hranu daným číslem, přeřadí ji do seznamu použitých hran, a případně
  přiřadí vrchol, z nejž hrana vede, do seznamu rozpracovaných hran }
procedure oznac_hranu (hrana: phrana; cislo: integer);
var vrchol: integer;
begin
  vrchol := hrana^.z;
  hrana^.tah := cislo;

  odeber (hrana);
  if prazdny (graf[vrchol].nepouzite) then inc (nh);

  if prazdny (graf[vrchol].pouzite) and (graf[vrchol].stupen <> 2) then
    begin
      inc (nr);
      rozpracovane[nr] := vrchol;
    end;
  pridej (graf[vrchol].pouzite, hrana);
end;

{ vrátí true pokud je přechod z hrany Z na hranu K zakázaný }
function zakazano (z, k: phrana): boolean;
var vrchol: integer;
begin
  if z = nil then
    zakazano := false
  else
    begin
      vrchol := z^.k;
```

```
      if k^.z <> vrchol then
        writeln('Chyba');
      zakazano := (graf[vrchol].zak_z = z) and (graf[vrchol].zak_do = k);
    end;
end;

{ vybere nepoužitou hranu z vrcholu, na kterou je povoleno přejít z dané hrany }
function vyber_hranu (vrchol: integer; hrana: phrana): phrana;
var ah: phrana;
begin
  ah := graf[vrchol].nepouzite.dal;
  while (ah <> @graf[vrchol].nepouzite) and zakazano (hrana, ah) do
    ah := ah^.dal;

  if ah = @graf[vrchol].nepouzite then vyber_hranu := nil
    else vyber_hranu := ah;
end;

{ označuje tah z nepoužitých hran, začínající hranou PRVNI, číslem CISLO.
  Pokud PRVNI_TAH je true, tah může skončit, jakmile dorazí na hranu, která
  naváže na hranu PRVNI; jinak musí využít všechny hrany z vrcholu PRVNI^.z }
procedure oznac_tah (prvni: phrana; cislo: integer; prvni_tah: boolean);
var ahrana, prhrana: phrana;
    avrchol: integer;
begin
  prhrana := nil;
  ahrana := prvni;

  repeat
    oznac_hranu (ahrana, cislo);
    if prhrana <> nil then
      prhrana^.tah_d := ahrana;
    ahrana^.tah_p := prhrana;

    prhrana := ahrana;
    avrchol := ahrana^.k;
    ahrana := vyber_hranu (avrchol, ahrana);
  until (ahrana = nil)
        or (prvni_tah and (prhrana^.k = prvni^.z)
            and not zakazano (prhrana, prvni));

  prhrana^.tah_d := prvni;
  prvni^.tah_p := prhrana;
  if prvni^.z <> prhrana^.k then writeln('Chyba');
end;

{ pokud spojení tahu přes hrany H1 a H2 nevytvoří zakázaný přechod,
  spojí je a vrátí true, jinak vrátí false }
function spoj (h1, h2: phrana): boolean;
var p1, p2: phrana;
```

```
begin
  p1 := h1^.tah_p;
  p2 := h2^.tah_p;
  if zakazano (p1, h2) or zakazano (p2, h1) then
    spoj := false
  else
    begin
      p1^.tah_d := h2;
      h2^.tah_p := p1;
      p2^.tah_d := h1;
      h1^.tah_p := p2;
      spoj := true;
    end;
end;

{ spojí A-tý tah v daném vrcholu s předchozími }
procedure spoj_tahy (vrchol: integer);
var stara, nova, navic, ah: phrana;
    t: boolean;
begin
  ah := graf[vrchol].pouzite.dal;
  stara := nil;
  nova := nil;
  navic := nil;
  while ah <> @graf[vrchol].pouzite do
    begin
      if (ah^.tah = a) and (nova = nil) then
        nova := ah
      else if (ah^.tah < a) and (stara = nil) then
        stara := ah
      else
        navic := ah;
      ah := ah^.dal;
    end;
  if not spoj (stara, nova) then
    begin
      if navic^.tah = a then
        t := spoj (stara, navic)
      else
        t := spoj (nova, navic);
      if not t then
        writeln ('Chyba');
    end;
end;

{ najde tah z nepoužitých hran začínající v daném vrcholu,
  a označí jeho hrany daným číslem }
procedure najdi_tah (cislo, vrchol: integer);
var prvni: phrana;
begin
```

```
  prvni := vyber_hranu (vrchol, nil);
  oznac_tah (prvni, cislo, cislo = 1);
end;

{ najde hranu z vrcholu U do vrcholu V }
function najdi_hranu (u, v: integer): phrana;
var ah: phrana;
begin
  ah := graf[u].nepouzite.dal;
  while ah^.k <> v do ah := ah^.dal;
  najdi_hranu := ah;
end;

{ načte graf ze souboru fin. Pokud graf obsahuje vrcholy stupně 1, vrátí false }
function nacti_graf (var fin: text): boolean;
var m, i, u, v: integer;
    h, d: phrana;
begin
  readln (fin, n, m);

  for i := 1 to n do
    with graf[i] do
      begin
        stupen := 0;
        pouzite.dal := @pouzite;
        pouzite.pr := @pouzite;
        nepouzite.dal := @nepouzite;
        nepouzite.pr := @nepouzite;
      end;

  for i := 1 to m do
    begin
      readln (fin, u, v);
      new (h);
      h^.z := u;
      h^.k := v;
      h^.pr := nil;
      h^.dal := nil;
      h^.tah := 0;
      h^.tah_p := nil;
      h^.tah_d := nil;
      inc (graf[u].stupen);
      pridej (graf[u].nepouzite, h);
    end;

  for i := 1 to n do
    begin
      readln (fin, u, v);
      h := najdi_hranu (u, i);
      d := najdi_hranu (i, v);
```

```
      graf[i].zak_z := h;
      graf[i].zak_do := d;
      { hranu iv přesuneme na začátek seznamu, bude vždy první vybrána do tahu }
      odeber (d);
      pridej (graf[i].nepouzite, d);
    end;

  nacti_graf := true;
  for i := 1 to n do if graf[i].stupen < 2 then nacti_graf := false;
end;

{ vypíše eulerovský tah v grafu do souboru FOUT }
procedure vypis_tah (var fout: text);
var prvni, ah: phrana;
begin
  prvni := graf[1].pouzite.dal;
  ah := prvni^.tah_d;
  write (fout, prvni^.z);
  while ah <> prvni do begin write (fout, ' ', ah^.z);   ah := ah^.tah_d; end;
end;

var fin, fout: text;

begin
  assign (fin, 'okruh.in');
  reset (fin);
  assign (fout, 'okruh.out');
  rewrite (fout);

  if not nacti_graf (fin) then
    begin
      writeln (fout, 'Okruzni jizda neexistuje.');
      exit;
    end;

  a := 1;
  najdi_tah(a, 1);

  while nr <> 0 do begin
    v := rozpracovane[nr];
    dec (nr);
    inc (a);
    najdi_tah (a, v);
    spoj_tahy (v);
  end;

  if nh = n then vypis_tah (fout)
    else writeln (fout, 'Okruzni jizda neexistuje.');
end.
```

A-I-4 Grafomat

Na nájdenie najkratšej cesty medzi vrcholmi v a w použijeme podobne ako v Príklade 1 prehľadávanie do šírky. Začneme vo vrchole v. Aby sme vedeli „zostrojiť“ cestu, bude si každý označený vrchol navyše pamätať hranu, po ktorej doň značka prišla.

Na rozdiel oproti klasickému prehľadávaniu sa nám môže stať, že vrchol dostane naraz značku po viacerých hranách. V takom prípade si zapamätá ľubovoľnú jednu z nich.

Akonáhle sa pri prehľadávaní dostaneme do cieľového vrcholu, budeme postupovať späť po zapamätaných hranách a pritom značiť nájdenú cestu.

Takýto algoritmus už síce funguje, ale (obzvlášť ak je hľadaná cesta pomerne krátka v porovnaní s veľkosťou grafu) trávi zbytočne veľa času tým, že aj po tom, ako bol nájdený vrchol w a zostrojená cesta späť do v, prehľadávanie pokračuje vo zvyšku grafu ďalej, až kým nenavštívi všetky dosiahnuteľné vrcholy.

Potrebovali by sme vymyslieť spôsob, ako prehľadávanie zastaviť v okamihu, keď sme už našli hľadanú cestu. To sa dá spraviť takto:

Akonáhle prídeme pri označovaní cesty do vrcholu v, pošleme z neho ešte jednu „vlnu“ značiek. Tá bude mať za úlohu dobehnúť a zastaviť prvú vlnu. Pochopiteľne, aby to fungovalo, potrebovali by sme, aby naša druhá vlna postupovala rýchlejšie ako tá prvá, ktorá robí samotné prehľadávanie. My síce nevieme poslať druhú vlnu rýchlejšie ako vrchol za takt, tu je ale ľahká pomoc – stačí všetky ostatné časti programu pustiť dvakrát pomalšie.

Ako to bude vyzerať s časovou zložitosťou? Ak má hľadaná cesta l hrán, prvá vlna príde do w po $2l$ krokoch. Po nasledujúcich $2l$ krokoch príde označovanie cesty späť do v. V tomto okamihu pustíme druhú vlnu. Tá teraz na prvú stráca $4l$ taktov. Za každé nasledujúce 2 takty jeden takt straty dobehne, teda o $8l$ taktov sa vlny stretnú a navzájom zrušia. Celkovo teda výpočet trvá $12l$ taktov, čiže je lineárny od dĺžky hľadanej cesty.

Listing programu:

```
var x: 0..2;        { vstupní značky }
    y: 0..1 = 0;    { výstupní značky }
    s: 0..2 = 0;    { stav prohledávání: 1=dorazila první vlna, 2=už i druhá }
    b: 0..3 = 0;    { kterou hranou jsme značku přijali, je-li s>0 }
    p: 0..1 = 0;    { počítadlo pro zpomalování }
    i: 1..3 = 1;    { pomocná proměnná }
```

```
begin
   p := 1-p;
   if (s=1) and ((S[1].s=2) or (S[2].s=2) or (S[3].s=2)) then
      s := 2                          { druhá vlna běží plnou rychlostí }
   else if p=1 then                   { ostatní části programu běží zpomaleně }
      case s of
         0: begin
               if x=1 then s := 1;    { začátek první vlny }
               for i := 1 to 3 do     { pokračování první vlny }
                  if S[i].s = 1 then begin s := 1; b := i; end;
               if s=0 then stop;      { stále se nic neděje }
            end;
         1: begin
               if x=2 then y := 1     { první vlna dorazila do cíle, jdeme zpět }
               for i := 1 to 3 do     { zpětný průchod }
                  if (S[i].y = 1) and (S[i].b = P[i]) then y := 1;
               if (y=1) and (x=1) then s := 2;
                  { zpětný průchod dorazil do počátku }
            end;
         2: stop;
      end;
end.
```

Riešenia domáceho kola kategórie B

B-I-1 Pestovanie mrkvičiek

Nech $A = [x_1, y_1]$ a $B = [x_2, y_2]$ sú body zo vstupu. Ak $x_1 = x_2$ alebo $y_1 = y_2$, úloha je ľahká.

Teraz bez ujmy na všeobecnosti predpokladajme, že $x_1 < x_2$. Nech $\delta_x = x_2 - x_1$ a $\delta_y = |y_1 - y_2|$. Nech $C = [x_3, y_3]$ je bod, ktorý leží na úsečke spájajúcej body $[x_1, y_1]$ a $[x_2, y_2]$ (rôzny od A). Označme $\delta'_x = x_3 - x_1$ a $\delta'_y = |y_1 - y_3|$. Potom z podobnosti trojuholníkov ABB' a ACC' (kde $B' = [x_2, y_1]$ a $C' = [x_3, y_1]$) plynie:

$$\frac{\delta_x}{\delta_y} = \frac{\delta'_x}{\delta'_y}$$

Nás zaujímajú také body C, pre ktoré δ'_x a δ'_y sú celé čísla. Chceme teda vedieť, koľko existuje takých celočíselných dvojíc a, b ($a \leq \delta_x$, $b \leq \delta_y$), že $\frac{a}{b} = \frac{\delta_x}{\delta_y}$. Dva zlomky sa rovnajú práve vtedy, ak sa po ich úprave na základný tvar rovnajú čitatele a menovatele. Ak d je najväčší spoločný deliteľ čísel δ_x a δ_y, tak zlomok $\frac{\delta_x}{\delta_y}$ má základný tvar $\frac{\delta_x/d}{\delta_y/d}$.

Hľadáme teda také celé čísla a, b, ktoré sa po vydelení nejakým celým číslom k rovnajú δ_x/d a δ_y/d. To sú práve čísla $(\delta_x/d, \delta_y/d)$, $(2\delta_x/d, 2\delta_y/d)$, $\ldots$, $((d-1)\delta_x/d, (d-1)\delta_y/d)$, (δ_x, δ_y), čo je dokopy d dvojíc. Spolu s bodom A (ktorý sme na začiatku úvahy vylúčili) máme $d+1$ bodov s celočíselnými súradnicami.

Číslo d vypočítame štandardným Euklidovým algoritmom. Všimnite si, že v našej implementácií, ak jeden zo vstupných parametrov pre funkciu `nsd` je 0, tak výsledkom tejto funkcie je to druhé číslo, teda prípad $x_1 = x_2$, resp. $y_1 = y_2$ nemusíme uvažovať ako špeciálny.

Listing programu:

```
var x1,y1:longint;
    x2,y2:longint;

function nsd(a,b:longint):longint;
var c:longint;
begin
  while (b<>0) do begin
    c:=a mod b; a:=b; b:=c;
  end;
```

```
  nsd:=a;
end;

begin
  readln(x1,y1);
  readln(x2,y2);
  writeln(nsd(abs(x2-x1),abs(y2-y1))+1);
end.
```

B-I-2 Hladný Samo

Táto úloha patri medzi tzv. ťažké úlohy v tom zmysle, že pre ňu ešte nikto nevymyslel efektívny algoritmus. Náše riešenie teda bude len skúšať všetky možnosti.

Ide teda o to, ako vygenerovať všetky možné podmnožiny potravín, pre jednu podmnožinu overiť, či v nej nie sú 2 také potraviny, ktoré sa nemôžu jesť spolu a vybrať nejakú s najväčším počtom prvkov. Generovať všetky možné podmnožiny sa dá rôzne. Pomerne jednoduché riešenie je postupne prechádzať čísla 0, 1, ..., $2^N - 1$, kde N je počet potravín, a reprezentovať ich binárny zápis ako podmnožinu (i-ty bit hovorí, či tam i-ta potravina je alebo nie).

Prehľadnejšie a efektívnejšie je generovať podmnožiny rekurzívne. Pre každú potravinu sa rozhodneme, či ju zjeme alebo nie a pre obe možnosti sa rekurzívne spustíme na nasledujúcu potravinu. Výhoda je tá, že môžeme hneď overiť, či danú potravinu môžeme v danom momente zjesť (zisťujeme, či potravina, ktorú chceme zjesť sa môže jesť spolu s doteraz zjedenými) a teda nemusíme to kontrolovať, až keď máme vygenerovanú celú podmnožinu.

Na konci už len zistíme, či sme našli väčšiu podmnožinu ako doteraz nájdenú. Časová zložitosť tohto algoritmu je $O(N2^N)$.

Listing programu:

```
const MAXN=25;

var N,M,a,b,i,j:integer;
    mnoz:array[1..MAXN] of integer;
    maxmnoz:array[1..MAXN] of integer;
    dvojice:array [1..MAXN,1..MAXN] of boolean;
    poc,max:integer;

procedure skus(k:integer);
var i:integer;
    ok:boolean;
begin
```

```
  if (k=N+1) then begin
    if (poc>max) then begin
      max:=poc;
      for i:=1 to poc do maxmnoz[i]:=mnoz[i];
    end;
  end else begin
    skus(k+1);
    ok:=true;
    for i:=1 to poc do
      if dvojice[k,mnoz[i]] then
        ok:=false;
    if ok then begin
      inc(poc);
      mnoz[poc]:=k;
      skus(k+1);
      dec(poc);
    end;
  end;
end;

begin
  readln(N,M);
  for i:=1 to N do for j:=1 to N do dvojice[i,j]:=false;

  for i:=1 to M do begin
    readln(a,b);
    dvojice[a,b]:=true; dvojice[b,a]:=true;
  end;

  skus(1);
  writeln(max);
  for i:=1 to max do write(maxmnoz[i],' ');
  writeln;
end.
```

B-I-3 Družstvá

Vezmime prvého hráča a bez ujmy na všeobecnosti ho priraďme do 1. družstva. Všetci hráči, ktorí nemôžu s ním hrať v družstve musia automaticky hrať v 2. družstve. Všetci, ktorí nemôžu hrať s týmito, musia hrať v 1. družstve, atď.

Teraz, buď sa nám podarí hráčov rozdeliť (ostane nám ešte rozdeliť hráčov, ktorí sú nezávislí od týchto), alebo budeme chcieť nejakého hráča dať do oboch družstiev. V takom prípade rozdelenie hráčov neexistuje. Funguje to len kvôli tomu, že máme len 2 družstvá. Ak by sme chceli zistiť, či sa dajú hráči rozdeliť do 3 družstiev, už to nie je také jednoduché.

Rýchlosť algoritmu závisí od zvolenej reprezentácie vzťahov medzi hráčmi a od použitého systému priradzovania. Vzťahy si môžeme zapisovať do booleovského poľa $N \times N$, kde $A[i,j]$ je `true` práve vtedy keď i a j nemôžu hrať spolu. Pri takejto reprezentácií je časová a pamäťová zložitosť $O(N^2)$.

Listing programu:

```
const MAXN=100;

var N,M,a,b,i,j:integer;
    dvojice:array [1..MAXN,1..MAXN] of boolean;
    druzstvo:array [1..MAXN] of integer;
    nedasa:boolean;

procedure rozdel(v:integer);
var i:integer;
begin
  for i:=1 to N do
    if dvojice[v][i] then begin
      if druzstvo[i]=0 then begin
        druzstvo[i]:=3-druzstvo[v];
        rozdel(i);
      end else if druzstvo[i]=druzstvo[v] then
        nedasa:=true;
    end;
end;

begin
  readln(N,M);
  for i:=1 to N do for j:=1 to N do dvojice[i,j]:=false;
  for i:=1 to N do druzstvo[i]:=0;

  for i:=1 to M do begin
    readln(a,b);
    dvojice[a,b]:=true; dvojice[b,a]:=true;
  end;

  nedasa:=false;
  for i:=1 to N do
    if druzstvo[i]=0 then begin
      druzstvo[i]:=1;
      rozdel(i);
    end;

  if nedasa then
    writeln('Nie')
  else begin
    writeln('Ano');
    for i:=1 to N do
      if druzstvo[i]=1 then write(i,' ');
```

```
    writeln;
  end;
end.
```

B-I-4 Assembler

a) Vstup načítame na dvakrát. Pri prvom prechode si riadky očíslujeme a čísla riadkov, na ktorých sa vyskytujú návestia si zapamätáme. Pri druhom prechode programom si zapisujeme inštrukcie, pričom návestia nahradzujeme číslami riadkov, kde sa vyskytuje ich definícia.

   Teraz už len odsimulujeme program na danom vstupe. Budeme si pamätať tzv. Instruction Pointer (IP), čo je číslo inštrukcie (v našom prípade riadku), kde bude vykonávanie programu pokračovať. V prípade jednoduchej inštrukcie len príslušne upravíme registre (a príznak **Z**). V prípade inštrukcií **jmp**, **jz** a **jnz** len upravíme IP na príslušné číslo riadku.

b) Postupne budeme od registra R_1 odčítavať hodnotu R_2. To budeme robiť tak, že vždy od oboch odčítame 1 a ku R_0 pričítame 1. Ak sa nám podarí odčítať celé R_2, tak v R_0 máme pôvodné R_2, takže to len prelejeme do R_2 a pokračujeme znovu. Ak sa nám to nepodarí (tzn. ak budeme chcieť od R_1 odčítať 1, ale už nebude z čoho), tak v R_0 je zapísané, koľko sa nám podarilo odčítať, čo je akurát zvyšok po delení.

   Na zisťovanie, či je už v R_2 nula sme využili to, že inštrukcia **dec** tiež nastavuje príznak **Z**, ak po jej vykonaní je $R_2 = 0$.

   c1: **test** R_1,R_1
   jz koniec
   dec R_1
   inc R_0
   dec R_2
   jz c2
   jmp c1
   c2: **test** R_0,R_0
   jz c1
   dec R_0
   inc R_2
   jmp c2
   koniec:

Riešenia krajského kola kategórie A

A-II-1 Stále ešte zasypané mesto

Očíslujme si možné zamerania: stredovek bude 1, starovek 2 a archeológia 3. Nech a_1, a_2 a a_3 sú počty prvákov s jednotlivými zameraniami, podobne nech b_1, b_2, b_3 sú počty druhákov. Tieto počty si vieme spočítať pri načítavaní vstupu.

Začnime tým, že sa pozrieme na nejaké situácie, kedy riešenie nebude existovať. Všimnime si napríklad prvákov so zameraním 1. Každý z nich musí mať partnera druháka so zameraním 2 alebo 3. Ak možných partnerov nemáme dosť (teda platí $a_1 > b_2+b_3$), riešenie určite neexistuje. Ďalšie podmienky dostaneme cyklickou obmenou tejto úvahy.

Keďže platí $a_1 + a_2 + a_3 = b_1 + b_2 + b_3 = N$, vieme získané podmienky upraviť do ekvivalentného tvaru: Aby existovalo riešenie, musí nutne platiť ($\heartsuit$) $a_1 + b_1 \leq N$, $a_2 + b_2 \leq N$ a $a_3 + b_3 \leq N$. V ďalšom texte ukážeme, že ak sú tieto podmienky splnené, riešenie vždy existuje.

Bez ujmy na všeobecnosti predpokladajme ($\star$), že $a_1 \geq a_2 \geq a_3$ a $a_3 \leq \min\{b_1, b_2, b_3\}$. (Inými slovami, vždy vieme dosiahnuť, aby tieto nerovnosti platili, stačí vhodne očíslovať zamerania a prípadne vymeniť prvákov s druhákmi.) Rozoberieme teraz dva prípady.

Nech platí $a_2 \leq b_1$. Potom vytvoríme dvojice nasledovne: Bude a_2 dvojíc (prvák so zameraním 2)+(druhák so zameraním 1). Zostalo nám $b_1 - a_2$ druhákov so zameraním 1. Keďže vieme zo $\heartsuit$, že $b_1 \leq a_2 + a_3$, je $b_1 - a_2 \leq a_3$. Zvyšných druhákov so zameraním 1 teda popárujeme s niektorými prvákmi so zameraním 3. Zostalo nám niekoľko (presnejšie $a_3 - b_1 + a_2$) prvákov so zameraním 3. Keďže podľa našich predpokladov $\star$ je $a_3 \leq b_2$, zjavne môžeme všetkých týchto prvákov popárovať s druhákmi so zameraním 2. A už sme vyhrali – ostali nám len prváci so zameraním 1 a druháci so zameraniami 2 a 3.

Zostáva nám vyriešiť prípad keď $a_2 > b_1$. Opäť začneme dvojicami (prvák so zameraním 2)+(druhák so zameraním 1), tentokrát ich bude b_1 a zostanú nám nejakí nepriradení prváci so zameraním 2. Tých priradíme druhákom so zameraním 3 (podľa $\heartsuit$ ich máme dosť). Podľa $\star$ máme aspoň toľko druhákov so zameraním 2 ako prvákov so zameraním 3, popárujeme ich, koľko ide. A opäť sme už vyhrali – ostali nám len prváci so zameraním 1 a druháci so zameraniami 2 a 3.

Takto sme dostali algoritmus, ktorého časová zložitosť (až na načítanie vstupu a výpis výstupu) aj pamäťová zložitosť je konštantná.

Listing programu:

```
program mesto;
var N:longint;
    a,b:array[1..3] of longint;
const spec:array[1..3] of string = ('starovek','stredovek','archeologia');
var rocnik_vymeneny:boolean; { pokiaľ a[3]<b[3], vymeníme ročníky }
    subor:text;

procedure nacitaj;
var subor:text;
    i,rocnik:longint;
    s:string;
begin
  assign(subor,'mesto.in');
  reset(subor);
  readln(subor,N);
  for i:=1 to 3 do begin a[i]:=0; b[i]:=0; end;
  for i:=1 to 2*N do begin
    readln(subor,rocnik,s);
    if rocnik=1 then for i:=1 to 3 do if (s=' '+spec[i]) then inc(a[i]);
    if rocnik=2 then for i:=1 to 3 do if (s=' '+spec[i]) then inc(b[i]);
  end;
  rocnik_vymeneny:=false;
end;

procedure zorad;
var x,i,j:longint;
    s:string;
begin
  for j:=1 to 3 do for i:=1 to j-1 do if a[i]<a[j] then begin
     x:=a[i]; a[i]:=a[j]; a[j]:=x;
     x:=b[i]; b[i]:=b[j]; b[j]:=x;
     s:=spec[i]; spec[i]:=spec[j]; spec[j]:=s;
  end;
  if b[3]<a[3] then begin
    for i:=1 to 3 do begin x:=a[i]; a[i]:=b[i]; b[i]:=x; end;
    rocnik_vymeneny:=true;
    zorad;
  end;
end;

procedure vypis(pocet: longint; spec1,spec2: integer);
  { vypíše "pocet" dvojíc študentov so špecializáciami spec1 a spec2 }
var x:integer;
begin
  if rocnik_vymeneny then begin x:=spec1; spec1:=spec2; spec2:=x end;
  for x:=1 to pocet do writeln(subor,spec[spec1],' ',spec[spec2]);
```

```
end;

begin
  nacitaj;
  zorad;
  assign(subor,'mesto.out');
  rewrite(subor);
  if (a[1]+b[1]>N) or (a[2]+b[2]>N) or (a[3]+b[3]>N) then begin
    writeln(subor,'Študenti sa nedajú rozdeliť do dvojíc.');
  end else
    if (a[2]<=b[1]) then begin
      vypis(a[2],2,1);
      vypis(b[1]-a[2],3,1);
      vypis(a[3]-(b[1]-a[2]),3,2);
      vypis(a[1]-b[3],1,2);
      vypis(b[3],1,3);
    end else begin
      vypis(b[1],2,1);
      vypis(a[2]-b[1],2,3);
      vypis(a[3],3,2);
      vypis(b[2]-a[3],1,2);
      vypis(b[3]-(a[2]-b[1]),1,3);
    end;
  close(subor);
end.
```

A-II-2 Nová okružná jazda

Na úvod si zopakujme terminológiu z riešenia úlohy domáceho kola. Mapu Blatislavy budeme volať graf, križovatky sú jeho vrcholy a ulice tvoria jeho hrany. Počet vrcholov je N, počet hrán je M.

Hranu medzi vrcholmi u a v značíme uv. Stupeň vrcholu je počet hrán, ktoré z neho vychádzajú. Podľa zadania má náš graf všetky stupne vrcholov párne a väčšie ako 2. Hovoríme, že hrany e a f na seba nadväzujú, ak majú spoločný vrchol.

Ťah je postupnosť vrcholov taká, že každé dva po sebe nasledujúce vrcholy sú spojené hranou, pričom žiadna hrana nie je použitá viac ako jedenkrát. Uzavretý ťah je ťah, ktorý začína a končí v tom istom vrchole. Eulerovský ťah je ťah, v ktorom je každá hrana grafu použitá práve raz. Našou úlohou je teda nájsť v danom grafe vhodný uzavretý eulerovský ťah.

Pre jednoduchšie vysvetlenie algoritmu špeciálne ošetríme vrcholy stupňa 4. Ľahko overíme, že keď zakážeme 3 prechody cez takýto vrchol, buď sa cez neho

nebude dať dvakrát prejsť, alebo bude jednoznačne určené, ako tieto prechody vyzerajú. V prvom prípade hľadaný ťah neexistuje, v druhom môžeme prechody cez tento vrchol nahradiť dvoma hranami.

Takto sa zbavíme všetkých vrcholov stupňa 4. Môžeme teda predpokladať, že každý vrchol v zadanom grafe má stupeň aspoň 6. Ukážeme, že ak je takýto graf súvislý, tak v ňom hľadaný ťah existuje.

Začneme tým, že graf rozložíme na niekoľko uzavretých ťahov, z ktorých žiaden nebude obsahovať zakázaný prechod: V každom vrchole popárujeme hrany do dvojíc tak, aby žiadna dvojica netvorila zakázaný prechod. Keďže vrcholy sú stupňa 6 a viac, toto zjavne vždy vieme spraviť. Tým už sme jednoznačne určili nejakú množinu uzavretých ťahov, ktoré dokopy obsahujú každú hranu grafu práve raz. (Začneme ľubovoľnou hranou, a vždy, keď prídeme do vrcholu, odídeme hranou, ktorá s ňou tam bola v páre, až kým sa nevrátime ku hrane, kde sme začali.)

Ak sa nám takto graf rozpadol na viac ako jeden ťah, musíme teraz ťahy pospájať do jedného. Zvolíme si ľubovoľný jeden ťah, nazveme ho hlavným ťahom a ostatné k nemu budeme pripájať.

Ak sa niekedy ocitneme v situácii, že ešte máme nepripojené ťahy, ale žiadny z nich nemá spoločný vrchol s hlavným ťahom, znamená to, že náš graf nie je súvislý, a teda v ňom eulerovský ťah neexistuje. Nech teraz v je vrchol, ktorým okrem hlavného ťahu prechádza ešte nejaký iný ťah. Ukážeme, ako tieto dva ťahy spojiť do jedného ťahu bez zakázaných prechodov.

Nech vo v na seba (okrem iného) nadväzujú hrany e_1 a e_2 v hlavnom ťahu a hrany e_3 a e_4 v nejakom inom ťahu. Ak sú prechody e_1e_3 a e_2e_4 povolené, môžeme tieto dva ťahy spojiť do jedného – nebude nadväzovať (e_1 na e_2) a (e_3 na e_4), ale (e_1 na e_3) a (e_2 na e_4). Ukážeme teraz, že takéto dve dvojice hrán určite vieme nájsť. Rozoberieme dva prípady:

- Nech hlavný ťah prechádza cez v len raz, cez hrany e_1 a e_2. Potom vo v sú aspoň dva páry hrán, ktoré do hlavného ťahu nepatria. Označme dva z nich e_3e_4 a e_5e_6. Všimnime si teraz nasledovné 4 dvojice prechodov: (e_1e_3 a e_2e_4), (e_1e_4 a e_2e_3), (e_1e_5 a e_2e_6) a (e_1e_6 a e_2e_5). Keďže zakázané sú len 3 prechody cez v, v niektorej z týchto 4 možností musia byť oba prechody povolené. Táto možnosť teda hovorí, ako môžeme nejaký ďalší ťah pripojiť ku hlavnému.

- Nech teda hlavný ťah prechádza cez v aspoň dvakrát. Zoberme dva takéto prechody a označme ich e_3e_4 a e_5e_6. Zoberme jeden prechod iného ťahu cez

v a označme ho e_1e_2. Rovnakou úvahou ako v predchádzajúcom prípade zistíme, že niektorý zo štyroch spôsobov ako tieto dva ťahy spojiť nie je zakázaný.

Posledná otázka: S akou časovou zložitosťou vieme tento algoritmus implementovať? V každom vrchole si budeme pamätať, ktoré hrany na seba nadväzujú a ktoré patria do hlavného ťahu. Ďalej budeme mať v každom vrchole zoznam hrán, ktoré do hlavného ťahu nepatria a nanajvýš dva prechody, ktoré do hlavného ťahu patria. Okrem toho budeme mať jeden globálny zoznam W vrcholov, cez ktoré ide hlavný ťah, ale ktoré ním ešte nie sú celé pokryté.

Náš program bude fungovať nasledovne: Ak je W prázdny, skončili sme. V opačnom prípade nech v je ľubovoľný vrchol z W. Pomocou pamätaných údajov vieme v konštantnom čase predĺžiť hlavný ťah vo vrchole v, a následne prejsť po pridanej časti ťahu a upraviť informácie vo vrcholoch, cez ktoré prechádza. Tento prechod má časovú zložitosť lineárnu od veľkosti pridanej časti. Celý algoritmus má teda časovú aj pamäťovú zložitosť lineárnu od veľkosti zadaného grafu.

(Listing programu z priestorových dôvodov neuvádzame, v prípade záujmu ho nájdete vo verzii, ktorá bude po skončení súťaže uverejnená na webe OI.)

A-II-3 Rozvoz pizze vo väčšom

Úlohu budeme riešiť metódou dynamického programovania. Postupne pre každé i spočítame nasledujúce údaje:

- Počet pizzérií P_i, ktoré treba na pokrytie úsekov i až N.
- Najväčší index Z_i taký, že platí: Ak máme P_i pizzérií, tak nimi okrem úsekov i až N vieme ešte pokryť aj úseky 1 až Z_i.
- Ako pomocný údaj si ešte budeme pamätať počet úsekov U_i a súčet počtov zákazníkov S_i, ktorých obsluhuje pri vyššie popísanom riešení „prvá“ pizzéria, teda pizzéria obsluhujúca úseky od i ďalej.

Na začiatku vieme povedať, že $P_N = 1$. V lineárnom čase nájdeme hodnotu Z_N (pridávame úseky, kým by súčet počtov zákazníkov nemal prekročiť K). U_N zatiaľ nebude zaujímavé, ale pre poriadok ho môžeme nastaviť na $Z_N + 1$, S_N bude práve zistený súčet počtov zákazníkov.

Teraz ideme nájsť všetky i, pre ktoré ešte platí, že $P_i = 1$. Nech teda už poznáme hodnoty $P_{i+1} = 1$, Z_{i+1} a S_{i+1}. Ak $A_i + S_{i+1} \leq K$, môžeme našej

zatiaľ jedinej pizzérii jednoducho pridať nový úsek a nič sa nemení. Bude teda $P_i = 1$, $Z_i = Z_{i+1}$, $U_i = (N + 1 - i) + Z_i$ a $S_i = A_i + S_{i+1}$. V opačnom prípade už nebudeme vedieť pokryť toľko úsekov „na začiatku“, teda musíme postupne zmenšovať Z_i (a adekvátne upravovať U_i a S_i), až kým nebude $S_i \leq K$.

Takto pokračujeme, kým $Z_i \geq 0$. Akonáhle sa dostaneme do situácie, že je $Z_i = 0$, a napriek tomu už je S_i priveľké, znamená to, že budeme potrebovať druhú pizzériu. Od tohoto okamihu budeme nové hodnoty počítať nasledovne:

Položíme $U_i = U_{i+1} + 1$ a $S_i = A_i + S_{i+1}$, teda pridáme „prvej“ pizzérii nový úsek. Kým $S_i > K$, zmenšujeme U_i o 1 a S_i o A_{i+U_i}. Takto vlastne „prvej“ pizzérii priradíme najväčší možný počet úsekov začínajúci i-tym. A čo ďalej? Zostáva nám pokryť úseky od $i + U_i$ po N. Toto sme už ale riešili a optimálne riešenie máme zapísané. Bude teda $P_i = 1 + P_{i+U_i}$ a $Z_i = Z_{i+U_i}$.

Spočítali sme teda hodnoty P_i a Z_i. Potrebujeme ešte ukázať dve veci: akú má ich výpočet časovú zložitosť a ako teraz zistíme optimálne riešenie pre celý kruh.

Tvrdíme, že náš výpočet má časovú zložitosť $O(N)$, teda lineárnu od počtu úsekov. Prečo je tomu tak? Môže sa síce stať, že pri výpočte niektorej konkrétnej hodnoty U_i budeme musieť prvej pizzérii postupne zobrať veľa úsekov, všimnime si ale, že dokopy za celý výpočet každý úsek prvej pizzérii zoberieme najviac jedenkrát. Preto na výpočet všetkých hodnôt U_i a S_i dokopy nám stačí čas $O(N)$. Výpočet hodnôt P_i a Z_i nám zjavne zložitosť nepokazí.

A ako teda nájsť optimálne riešenie pôvodnej úlohy? Predstavme si takéto optimálne riešenie. Ak nejaká pizzéria obsluhuje úseky 1 aj N, na chvíľu ju zrušíme. Teraz nech x je číslo najmenšieho obsluhovaného úseku. Potom zjavne P_x je počet pizzérií v optimálnom riešení. A môžeme si všimnúť, že $Z_x \geq x - 1$, lebo posledná použitá pizzéria musí pokryť aj prvých $x - 1$ úsekov.

My ale hodnotu x nepoznáme. Nič nám však nebráni v čase $O(N)$ vyskúšať všetky možné x, pre ktoré $Z_x \geq x - 1$, a zobrať minimum z hodnôt P_x.

Listing programu:

```
program pizza;
const maxN=100000;
var A: array[1..maxN] of longint;
    N, K: longint;
    Pocet, Kam, Dalsi: array[1..maxN] of longint;
    { Pocet[i] je minimální počet intervalů nutných k pokrytí úseků od i do N;
      tyto intervaly navíc pokryjí všechny úseky od i do Kam[i];
      v optimálním pokrytí začíná po intervalu se začátkem i
      další interval na Dalsi[i] }
    soubor: text;
```

```
    i, j: longint;
    soucet, optimum: longint;

begin
  assign(soubor,'pizza.in');
  reset(soubor);
  readln(soubor,N,K);
  for i:=1 to N do readln(soubor,A[i]);
  close(soubor);
  { nejdříve inicializujeme konec pole }
  { j bude největší číslo takové, že součet A[1] až A[j] je menší nebo roven K }
  j:=1; soucet:=A[1];
  while (j<=N) and (soucet+A[j]<=K) do begin
    soucet:=soucet+A[j];
    inc(j);
  end;
  if j=N+1 then begin
    { stačí jedna pizzerie }
    assign(soubor,'pizza.out');
    rewrite(soubor);
    writeln(soubor,1);
    writeln(soubor,1,' ',N);
    close(soubor);
    exit
  end;
  { nyní spočítáme hodnoty na konci pole }
  i:=N;
  { i bude index prvku, jehož hodnoty nyní počítáme }
  while j>0 do begin
    if soucet+A[i]>K then begin
      soucet:=soucet-A[j];
      dec(j);
      continue;
    end;
    Pocet[i]:=1;
    Kam[i]:=j;
    Dalsi[i]:=j+1;
    soucet:=soucet+A[i];
    dec(i);
  end;
  while soucet+A[i]<=K do begin
    Pocet[i]:=1;
    Kam[i]:=0;
    Dalsi[i]:=N+1;
    soucet:=soucet+A[i];
    dec(i);
  end;
  { i je stále index počítaného prvku a j je maximální
    index takový, že součet A[i+1] až A[j] je menší nebo roven K }
  j:=N;
```

```
  while i>0 do begin
    if soucet+A[i]>K then begin
      soucet:=soucet-A[j];
      dec(j);
      continue;
    end;
    Pocet[i]:=Pocet[j+1]+1;
    Kam[i]:=Kam[j+1];
    Dalsi[i]:=j+1;
    soucet:=soucet+A[i];
    dec(i);
  end;
  { zbývá nalézt optimum }
  optimum:=Pocet[1]; j:=1;
  for i:=1 to N do
    if (Kam[i]+1>=i) and (optimum>Pocet[i]) then begin
      optimum:=Pocet[i];
      j:=i;
    end;
  assign(soubor,'pizza.out');
  rewrite(soubor);
  writeln(soubor,optimum);
  i:=j;
  while Pocet[j]>1 do begin
    writeln(soubor,j,' ',Dalsi[j]-1);
    j:=Dalsi[j];
  end;
  if i=1 then
    writeln(soubor,j,' ',N)
  else
    writeln(soubor,j,' ',i-1);
  close(soubor);
end.
```

A-II-4 Grafomat loví zver

Zaoberajme sa najskôr jednoduchším problémom. Majme v grafe dva vrcholy l a p, ktoré sú spojené cestou. Chceme ich zosynchronizovať, teda spôsobiť, aby nejaká udalosť („zazretie zvieraťa“) v p spôsobila, že časom tá istá udalosť („výstrel“) nastane v oboch naraz.

Toto vieme dosiahnuť napr. trikom s rôzne rýchlymi signálmi, ktorý sme si ukázali v riešeniach domáceho kola. Vyšleme z p do l dva signály, z ktorých jeden ide polovičnou rýchlosťou ako druhý. Keď rýchlejší príde do l, odrazí sa a putuje naspäť. V okamihu, keď sa rýchlejší signál vráti do p, príde do l pomalší signál. V programe to vyzerá takto:

Listing programu:

```
var x: 0..3;                { 1=l, 2=p, 3=p během výpočtu, 0=ostatní }
    y: 0..1 = 0;            { výstup }
    a, aa, b: 0..3 = 0;     { signály a kolik taktů zbývá do jejich předání dál }
begin
   { Na počátku vyšleme signály A a B. }
   if x=2 then begin x:=3; a:=2; b:=3; end;

   { Signál A putuje vlevo rychlostí 1, vlevo se odrazí a je z něj A'. }
   if a>0 then dec(a);
   if S[1].a=1 then a:=1;
   if (x=1) and (a>0) then begin a:=0; aa:=2; end;

   { Signál A' putuje vpravo rychlostí 1. }
   if aa>0 then dec(aa);
   if S[2].aa=1 then aa:=1;

   { Signál B putuje vlevo rychlostí 1/2. }
   if b>0 then dec(b);
   if S[1].b=1 then b:=2;

   { Když A' a B doputují na protilehlý konec, vypálíme. }
   if (x=1) and (b>0) then begin b:=0; y:=1; end;
   if (x=3) and (aa>0) then begin aa:=0; y:=1; end;

   { Není-li co dělat, končíme. }
   if (a=0) and (aa=0) and (b=0) then stop;
end.
```

Keď vieme synchronizovať dvojice vrcholov, môžeme väčší počet vrcholov zosynchronizovať použitím metódy rozdeľ a panuj. Predstavme si, že máme $2^k + 1$ vrcholov v rade, pričom najľavejší z nich je na začiatku označený ako začiatok intervalu a najpravejší ako zarážka za koncom intervalu. Ako zosynchronizovať všetky vrcholy okrem najpravejšieho?

Pre $k = 1$ máme zosynchronizovať dva vrcholy, a to už vieme spraviť. Pre väčšie k rozdelíme náš interval na dve rovnako veľké polovice, zosynchronizujeme ich najľavejšie vrcholy, a potom paralelne zosynchronizujeme každú polovicu zvlášť. Ako rozdeliť interval na polovicu, keď nepoznáme jeho dĺžku? Opäť použijeme dva signály, tentokrát pomalší pôjde tretinovou rýchlosťou. Kým rýchlejší signál prejde na koniec, odrazí sa a vráti sa do polovice, pomalší práve dorazí do polovice intervalu. Keď sa teda signály stretnú, našli sme stred intervalu.

Majme teraz $N = 2^k$ vrcholov na kružnici. Tam to bude fungovať úplne rovnako, len vrchol, ktorý „zazrel zver“, bude v prvom kole slúžiť aj ako ľavý okraj intervalu, aj ako zarážka za pravým okrajom.

Každá iterácia tohoto algoritmu trvá lineárne dlho od dĺžky intervalov, ktoré práve synchronizujeme. Celková časová zložitosť teda je $O(N + N/2 + N/4 + \cdots + 1) = O(N)$.

Listing programu:

```
var x: 0..3;                        { 1=počátek, 2=okraj, 3=střed intervalu }
    y: 0..1 = 0;                    { výstřel }
    a, aa, b, c, cc, d: 0..3 = 0; { signály a kolik taktů zbývá do předání dál }

begin
   { Triviální vstup => vypálíme hned. }
   if (x=1) and (S[1].x=1) then begin y:=1; stop; end;

   { Na počátku vyšleme signály C a D. }
   if x=1 then begin x:=2; d:=3; end;
   if S[2].x=1 then c:=2;

   { Signál C putuje vpravo rychlostí 1, vpravo se odrazí jako C'. }
   if c>0 then dec(c);
   if S[2].c=1 then c:=1;
   if (x=2) and (c>0) then begin c:=0; cc:=2; end;

   { Signál C' putuje vlevo rychlostí 1. }
   if cc>0 then dec(cc);
   if S[1].cc=1 then cc:=1;

   { Signál D putuje vpravo rychlostí 1/3. }
   if d>0 then dec(d);
   if S[2].d=1 then d:=3;

   { Když se C' a D potkají, máme střed. Vyšleme z něj A a B. }
   if (x=0) and (cc>0) and (d>0) then begin cc:=0; d:=0; x:=3; a:=2; b:=3; end;

   { Signál A putuje vlevo rychlostí 1, vlevo se odrazí a je z něj A'. }
   if a>0 then dec(a);
   if S[1].a=1 then a:=1;
   if (x=2) and (a>0) then begin a:=0; aa:=2; end;

   { Signál A' putuje vpravo rychlostí 1. }
   if aa>0 then dec(aa);
   if S[2].aa=1 then aa:=1;

   { Signál B putuje vlevo rychlostí 1/2. }
   if b>0 then dec(b);
   if S[1].b=1 then b:=2;

   { Když A' doputuje zpět do středu a B doleva, spustíme se v obou polovinách. }
   if (x=2) and (b>0) then begin b:=0; x:=1; end;
   if (x=3) and (aa>0) then begin aa:=0; x:=1; end;
```

```
   { Není-li co dělat, končíme. }
   if (a=0) and (aa=0) and (b=0) and (c=0) and (cc=0) and (d=0) and (x<>1)
      then stop;
end.
```

Poznámka: Riešenie pre všeobecnú kružnicu nie je o toľko ťažšie. Keď delíme na polovicu interval nepárnej dĺžky, to, že je nepárnej dĺžky, zistíme tak, že sa nám signály v strede nestretnú presne. V takomto prípade stredný vrchol vynecháme a nebudeme ho synchronizovať. Skončíme s tým, že zosynchronizujeme nejakú množinu vrcholov, pričom žiadne dva nezosynchronizované spolu nesusedia. Stačí teda, aby si namiesto „strieľaj“ zosynchronizované vrcholy nastavili novú značku „zamier“. A v nasledujúcom takte vystrelí každý, kto vidí značku „zamier“ u seba alebo niektorého suseda.

Riešenia krajského kola kategórie B

B-II-1 Kerfúr ten je lacnejší

Pokúsime sa zostrojiť nejaké poradie supermarketov, ktoré by zodpovedalo všetkým reklamám. Ak sa nám to podarí, bude výstup `ANO`, naopak, ak zistíme, že sa to nedá, bude výstup `NIE`.

Majme teda niekoľko supermarketov. Rozdeľme si ich na dve kôpky. Na prvú dáme tie, o ktorých nik iný netvrdí, že je od nich lacnejší. Na druhej teda samozrejme zostanú tie, od ktorých poznáme aspoň jeden lacnejší.

Pokiaľ je prvá kôpka prázdna, hľadané usporiadanie zjavne neexistuje – žiaden zo supermarketov nemôže byť najlacnejší.

(Rozmyslite si, že ak táto situácia nastala, znamená to, že na vstupe sme dostali *cyklus* – postupnosť supermarketov $s_1, \ldots, s_k$ takú, že pre každé i je s_i lacnejší ako s_{i+1}, a navyše aj s_k má byť lacnejší ako s_1. Keby bolo treba, takýto cyklus nájdeme ľahko: začneme ľubovoľným supermarketom a vždy prejdeme na nejaký, ktorý má od práve spracúvaného byť lacnejší, až kým sa nám nejaký supermarket v takto získanej postupnosti nezopakuje.)

Pokiaľ je na prvej kôpke aspoň jeden supermarket, môžeme hociktorý z týchto supermarketov vyhlásiť za najlacnejší. Rozmyslite si, že tým naozaj nič nemôžeme pokaziť.

Vyberieme si teda ľubovoľný supermarket z prvej kôpky, prehlásime ho za najlacnejší a úplne naň zabudneme. A čo sme takto dostali? Tú istú úlohu, len máme o jeden supermarket menej. Rovnakým postupom teda pokračujeme ďalej, a sú len dve možnosti: buď časom narazíme na cyklus (a zistíme, že riešenie neexistuje), alebo sa nám časom všetky supermarkety minú a máme hotové celé poradie.

Ako to šikovne naprogramovať? Pre každý supermarket si budeme pamätať, koľko supermarketov má byť od neho lacnejších a tiež zoznam všetkých, ktoré majú byť od neho drahšie. Okrem toho si budeme v jednom poli pamätať čísla tých supermarketov, ktoré už od seba nemajú mať nikoho lacnejšieho.

Výber jedného supermarketu bude prebiehať nasledovne: Pozrieme sa do poľa s kandidátmi a vyberieme posledného z nich. Teraz ho potrebujeme odstrániť. Prejdeme všetky supermarkety, ktoré majú od neho byť väčšie. Každému z nich zmenšíme počítadlo menších o jedna, a ak kleslo na nulu, pridáme ho na koniec poľa kandidátov.

Spracovanie jedného supermarketu teda trvá lineárne dlho v závislosti od počtu reklám, ktoré ho chvália. Spracovanie všetkých supermarketov dokopy bude teda trvať lineárne dlho od počtu všetkých reklám. Lepšie to ani nejde, lebo rádovo rovnaký čas nám zoberie už len načítanie vstupu.

Listing programu:

```
program reklamy;
var mensich, vacsich, kandidati : array[1..1000] of longint;
    vacsie : array[1..1000,1..1000] of longint;
    N, M, i, j, x, y, kandidatov, teraz : longint;

begin
   readln(N,M);
   for i:=1 to N do begin mensich[i]:=0; vacsich[i]:=0; end;
   for i:=1 to M do begin
      readln(x,y);
      inc(vacsich[x]);  inc(mensich[y]);
      vacsie[ x ][ vacsich[x] ] := y;
   end;
   { inicializujeme pole kandidatov }
   kandidatov := 0;
   for i:=1 to N do if mensich[i]=0 then begin
      inc(kandidatov);
      kandidati[ kandidatov ]:=i;
   end;
   { N-krat skusime vybrat najlacnejsi supermarket }
   for i:=1 to N do begin
      { ak nemame ziadneho kandidata, zle je }
      if kandidatov=0 then begin writeln('NIE'); halt; end;
      { ak mame, odstranime ho }
      teraz := kandidati[ kandidatov ];
      dec(kandidatov);
      for j:=1 to vacsich[teraz] do begin
         x := vacsie[teraz][j];
         dec(mensich[x]);
         if mensich[x]=0 then begin { mame noveho kandidata }
            inc(kandidatov);
            kandidati[kandidatov] := x;
         end;
      end;
   end;
   writeln('ANO');
end.
```

Poznámka. Úloha sa dá preformulovať do jazyka teórie grafov. Jednotlivé supermarkety budú predstavovať vrcholy grafu, informáciu „je lacnejší“ si zaznačíme ako orientovanú hranu. Hľadané usporiadanie supermarketov sa volá

topologické usporiadanie vrcholov grafu. Ukázali sme, že takéto usporiadanie existuje práve vtedy, ak je graf zodpovedajúci zadaniu acyklický.

B-II-2 Televízna súťaž

Keďže čísla sú z príliš veľkého rozsahu na to, aby sme si mohli značiť v poli, ktoré sme koľkokrát videli, budeme si musieť pomôcť ináč.

Jedným možným riešením je čísla, ktoré dostaneme zadané na vstupe, uložiť do poľa a toto pole utriediť. Ak to spravíme použitím šikovného triediaceho algoritmu (napríklad QuickSort), budeme na utriedenie čísel potrebovať rádovo $N \log N$ operácií. Keď už máme čísla utriedené, stačí raz prejsť poľom a spočítať si, ktorá hodnota sa v ňom koľkokrát vyskytuje. (Vlimnite si, že po utriedení budú všetky výskyty konkrétneho čísla tvoriť v poli súvislý úsek.)

Listing programu:

```
program sutaz1;
var a : array[1..10047] of longint;
    n,i,kandidat,pkand,pteraz : longint;

procedure QuickSort(zaciatok, koniec: longint);
   { utriedi cisla, ktore su v poli A na poziciach zaciatok .. koniec }
var pivot, malych, strednych, i, pom : longint;
begin
   if (koniec <= zaciatok) then exit;        { ak je usek kratky, skonci }
   pivot := A[zaciatok];               { vyber jeden z prvkov ako pivota }
   malych := 0;                    { presunieme "male" prvky na zaciatok }
   for i:=zaciatok to koniec do
      if (A[i] < pivot) then begin
         pom:=A[i];  A[i]:=A[zaciatok+malych];  A[zaciatok+malych]:=pom;
         Inc(malych);
      end;
        { a mame "male" prvky na poziciach zaciatok .. zaciatok+malych-1 }
   strednych := malych;            { za ne presunieme prvky rovne pivotu }
   for i:=zaciatok+malych to koniec do
      if (A[i] = pivot) then begin
         pom:=A[i];  A[i]:=A[zaciatok+strednych];  A[zaciatok+strednych]:=pom;
         Inc(strednych);
      end;                       { a mame prvky rovne pivotu na poziciach
                                  zaciatok+malych .. zaciatok+strednych-1 }
                  { teraz samostatne utriedime usek "malych" prvkov ... }
   QuickSort(zaciatok, zaciatok+malych-1);
   QuickSort(zaciatok+strednych, koniec);  { ... a usek "velkych" prvkov }
end;

begin
```

```
   read(n);
   for i:=1 to n do read(a[i]);
   QuickSort(1,n);
    { v premennej "kandidat" je doteraz najcastejsi prvok, v "pkand" je
      pocet jeho vyskytov, v "pteraz" je pocet vyskytov aktualneho prvku }
   kandidat:=-1; pkand:=0; pteraz:=0;
   for i:=1 to n do begin
      if (i=1) or (a[i-1]=a[i]) then inc(pteraz) else pteraz:=1;
      if (pteraz > pkand) then begin kandidat:=a[i]; pkand:=pteraz; end;
   end;
   writeln(kandidat);
end.
```

Existujú však ešte lepšie riešenia, založené na metóde *hašovania*. Predstavme si, že nám z neba spadla funkcia f, ktorá každému číslu zo vstupu priradí číslo povedzme z intervalu od 0 do $N-1$. Navyše chceme, aby výstupné hodnoty f boli približne rovnako pravdepodobné. (Teda ak vypočítame hodnotu f pre veľa náhodných vstupov, dostaneme každý výsledok približne rovnako veľa krát.)

Ako by nám takáto funkcia pomohla pri riešení úlohy? Predstavme si, že máme N vedier označených číslami od 0 do $N-1$. Pre každé číslo x zo vstupu spočítame hodnotu $f(x)$ a do toho vedra x hodíme. Potom stačí každé vedro spracovať samostatne.

Získali sme tým niečo? Na vstupe bolo najviac N rôznych hodnôt, preto môžeme očakávať, že v priemere bude v každom vedre len jedna hodnota. Vedrá, v ktorých skončila len jedna hodnota, ľahko spracujeme v konštantnom čase. Pre každé z ostatných vedier môžeme napríklad použiť vyššie popísaný algoritmus – čísla utriedime a raz prejdeme.

Samozrejme, najhorší možný prípad je, že na vstupe je N rôznych čísel, ale všetky skončia v tom istom vedre. Takýto prípad je ale veľmi nepravdepodobný. V priemernom prípade má náš algoritmus časovú zložitosť lineárnu od počtu čísel na vstupe, v najhoršom prípade bude jeho časová zložitosť naďalej úmerná časovej zložitosti triedenia, teda $O(N \log N)$.

Posledná otázka: odkiaľ vziať funkciu f? Môžeme pre jednoduchosť uvažovať funkciu $f(x) = x \bmod N$. (V praxi sa používajú zložitejšie hašovacie funkcie.)

Listing programu:

```
program sutaz2;
type pzaznam = ^tzaznam;
     tzaznam = record next : pzaznam; hodnota : longint; end;
var vedra : array[0..10047] of pzaznam;
    prve : array[0..10047] of longint;
    pocet : array[0..10047] of longint;
    N,i,x : longint;
```

```
    kandidat,pkand : longint;

{ pridaj(kam,co) prida hodnotu "co" na zaciatok zoznamu "kam" }
procedure pridaj(kam : pzaznam; co : longint);
var tmp : pzaznam;
begin getmem(tmp,sizeof(tzaznam)); tmp^.hodnota:=co; tmp^.next:=kam; end;

{ hashovacia funkcia f }
function f(x : longint) : longint; begin f:=x mod N; end;

begin
   read(N);
   for i:=0 to n-1 do begin pocet[i]:=0; vedra[i]:=nil; end;
   for i:=1 to n do begin
      read(x);
      if pocet[f(x)]=0 then begin
         prve[f(x)]:=x; pocet[f(x)]:=1;
      end else begin
         if prve[f(x)]=x then inc(pocet[f(x)]) else pridaj( vedra[ f(x) ], x );
      end;
   end;
   kandidat:=-1; pkand:=0;
   for i:=0 to n-1 do begin { spracujeme vedro i }
      if pocet[i]>pkand then begin pkand:=pocet[i]; kandidat:=prve[i]; end;
      if vedra[i]<>nil then begin
         { ... prepiseme zvysny obsah vedra do pola ... }
         { ... pouzijeme na pole predchadzajuci program ... }
      end;
   end;
   writeln(kandidat);
end.
```

B-II-3 Kartári

Lackov balíček kariet zodpovedá dátovej štruktúre známejšej pod menom fronta. Môžeme si ho predstaviť ako takú dlhú rúru, do ktorej jedným koncom karty vkladáme a druhým koncom z nej *v tom istom poradí* vypadávajú.

Pomalé riešenie je pamätať si karty v poli a pri každom výbere ich o jedno posunúť. Takto totiž bude náš program tým pomalší, čím viac kariet bude mať Lacko v kôpke.

Lepšie riešenie je použiť na reprezentáciu Lackovej kôpky kariet spájaný zoznam. Takto vieme zobrať kartu zo začiatku aj pridať kartu na koniec priamo, bez toho, aby nás trápilo, koľko kariet je medzi nimi.

Listing programu:

```
program fronta1;
type pzaznam = ^tzaznam;
     tzaznam = record
        prev, next : pzaznam;
        hodnota : char;
     end;
var zaciatok, koniec : pzaznam;
    prikaz : longint;
    karta : string;

procedure pridaj(co : char);
var tmp : pzaznam;
begin
   { vytvorime a inicializujeme novy zaznam }
   getmem(tmp,sizeof(tzaznam));
   tmp^.hodnota := co; tmp^.next:=nil;

   { ak je fronta prazdna, vytvorime novu, inak ho pridame na koniec }
   if (zaciatok=nil) then begin
      tmp^.prev:=nil; zaciatok:=tmp;
   end else begin
      tmp^.prev:=koniec; koniec^.next:=tmp;
   end;
   koniec:=tmp;
end;

function vyber : char;
var tmp : pzaznam;
begin
   { vratime hodnotu z prveho zaznamu a vymazeme ho z pamate }
   vyber := zaciatok^.hodnota;
   tmp := zaciatok;
   zaciatok := zaciatok^.next;
   freemem(tmp);
end;

begin
   zaciatok := nil; koniec := nil;
   while true do begin
      read(prikaz);
      case prikaz of
         1 : begin readln(karta); pridaj(karta[2]); end;
         2 : writeln(vyber);
         0 : halt;
      end;
   end;
end.
```

Rovnako efektívne riešenie, ale jednoduchšie na naprogramovanie dostaneme nasledovne: budeme si Lackove karty pamätať ako súvislý úsek prvkov poľa. Keď príde nová karta, pridáme ju na koniec úseku (ten sa tým natiahne „doprava“), keď máme zahrať kartu, zoberieme ju zo začiatku (tým sa nám úsek skráti).

Ak ale zvolíme pole konečnej dĺžky, ako zabezpečiť, aby sme z neho časom nevybehli? Jednoducho: Vždy, keď takto prídeme úsekom na koniec poľa, začneme nové prvky pridávať znova od začiatku. A ešte jeden detail: Aby sa nám to celé nepomiešalo, zvolíme si pole väčšej dĺžky ako je celkový počet kariet.

Listing programu:

```
program fronta2;
var fronta : array[0..146] of char;
    zac, kon : longint; { zac=zaciatok useku, kon=prve volne miesto za nim }
    prikaz : longint;
    karta : string;

procedure pridaj(co : char);
begin
   fronta[kon] := co;
   kon := (kon+1) mod 147;
end;
function vyber : char;
begin
   vyber := fronta[zac];
   zac := (zac+1) mod 147;
end;

begin
   zac:=0; kon:=0;
   while true do begin
      read(prikaz);
      case prikaz of
         1 : begin readln(karta); pridaj(karta[2]); end;
         2 : writeln(vyber);
         0 : halt;
      end;
   end;
end.
```

Obe uvedené riešenia spracujú každú inštrukciu s konštantnou časovou zložitosťou.

B-II-4 Assembler II

Prvú úlohu vyriešime ľahko. Presunieme obsah R_j do R_7. Zmažeme obsah R_i. Na záver presunieme obsah R_7 naraz aj do R_i, aj do R_j. Musíme si dať pozor na situácie, kedy je v R_i alebo R_j nula.

Definujeme teraz pomocnú inštrukciu „**add** R_i, R_j“, ktorá k hodnote v R_i pridá hodnotu z R_j (a tú nechá nezmenenú). Toto spravíme rovnako ako pri inštrukcii **mov**, len vynecháme dva riadky, ktoré nulujú obsah R_i.

Pomocou inštrukcie **add** teraz definujeme inštrukciu **mul** zo zadania. Presunieme hodnotu R_i do R_6. Skopírujeme hodnotu R_j do R_5. Teraz hodnotu z R_6 (hodnota z R_5)-krát pripočítame k hodnote (pôvodne teda nule) v R_i. Nakoniec už len vynulujeme R_6.

```
///// mov R_i R_j /////
a: test R_j, R_j
   jz b
   dec R_j
   inc R_7
   jmp a
b: dec R_i
   jnz b
c: test R_7, R_7
   jz k
   dec R_7
   inc R_i
   inc R_j
   jmp c
k:
```

```
///// mul R_i R_j /////
a: test R_j, R_j
   jz b
   dec R_j
   inc R_6
   jmp a
b: mov R_j, R_5
c: test R_5, R_5
   jz d
   dec R_5
   add R_i, R_6
   jmp c
d: dec R_6
   jnz d
```

Všimnime si ešte, ako dlho trvá vykonanie nami definovaných inštrukcií. Na vykonanie **mov** potrebujeme inštrukcií lineárne veľa od súčtu hodnôt R_i a R_j. Pri **add** nemusíme nulovať R_i, takže počet inštrukcií je priamo úmerný hodnote v R_j. Pri inštrukcii **mul** bude počet krokov lineárne závisieť od výsledného súčinu. *(V špeciálnom prípade, kedy je jeden z činiteľov 0, nebude počet inštrukcii 0, ale bude úmerný obsahu druhého z násobených registrov. Toto nás ale pri našom využívaní tejto inštrukcie trápiť nebude.)*

S počítaním odmocniny to bude trochu zložitejšie. Optimálne riešenie je pomerne jednoduché, ale ukážeme si predtým niekoľko iných, pomalších riešení.

Začnime riešením, na ktoré netreba v podstate nič vymýšľať. Postupne budeme zväčšovať hodnotu v registri R_0. Zakaždým spočítame (pomocou inštrukcie **mul**) jej druhú mocninu a tú porovnáme s hodnotou v R_1. Akonáhle bude vypočítaná druhá mocnina väčšia ako obsah R_1, zmenšíme obsah R_0 o jedna a skončíme.

Jediná vec, ktorú ešte nevieme robiť, je „tú porovnáme s hodnotou v R_1“.

Definujme inštrukciu na porovnanie registrov „**jg** R_i, R_j, x“ (jump if greater), ktorá bude fungovať nasledovne: Porovnaj obsahy R_i a R_j. Oba registre vynuluj. Ak bol obsah R_i väčší, pokračuj skokom na návestie x, inak pokračuj ďalšou inštrukciou.

```
///// jg R_i R_j x /////
a:      test R_i, R_i
        jz zle
        dec R_i
        test R_j, R_j
        jz dobre
        dec R_j
        jmp a
dobre:  dec R_i
        jz x
        jmp dobre
zle:    dec R_j
        jnz zle
```

Aké efektívne je toto riešenie? Nech je správna odpoveď k. Potom náš algoritmus postupne vypočíta hodnoty $1^2, 2^2, \ldots, (k+1)^2$, a každú z nich porovná s k. Na výpočet hodnôt potrebujeme čas úmerný súčtu vypočítaných hodnôt, teda $O(k^3)$ krokov. Porovnanie dvoch hodnôt trvá čas úmerný väčšej oboch hodnôt. V našom prípade je to skoro vždy obsah registra R_1, a ten je približne rovný k^2. Porovnaní robíme k, preto pri všetky porovnaniach dokopy spravíme $O(k^3)$ krokov.

Ak označíme hodnotu v R_1 ako n, náš algoritmus potrebuje $O(n\sqrt{n})$ krokov.

Funkcia na porovnanie hodnôt sa dá v našom prípade napísať aj šikovnejšie, ak si trochu zmeníme, ako má fungovať. Spolu so zmenšovaním oboch hodnôt budeme zväčšovať hodnotu v novom pomocnom registri. Keď teraz vynulujeme niektorý z registrov, zapamätáme si (skokom do vhodnej časti programu), ktorý

register to bol, a spravíme opačný proces. Pomocný register budeme nulovať, pôvodné dva zároveň s tým zväčšovať. Skončíme v situácii, v ktorej sme začínali, iba navyše vieme, ktorý z registrov má väčší obsah. Takto upravené porovnávanie je lineárne od **menšieho** z oboch porovnávaných čísel. (Asymptotickú časovú zložitosť predchádzajúceho riešenia to nezlepší, ale zlepšenie to je.)

Prefíkanejšie riešenie je neskúšať postupne všetky hodnoty k, ale len niektoré. Použijeme niečo podobné binárnemu vyhľadávaniu: postupne skúsime ako k hodnoty 1, 2, 4, 8, 16, ..., až kým k^2 neprekročí obsah R_1. V tomto okamihu máme v R_0 hodnotu 2^x a vieme, že správne k je v rozsahu $2^{x-1} \leq k < 2^x$. A teraz už môžeme použiť sľubované binárne vyhľadávanie a na ďalších $x-1$ „otázok“ nájsť presnú hodnotu k.

(Príklad: Ak sme zistili, že $k = 8$ ešte nie je veľa a $k = 16$ už veľa je, budeme postupne skúšať $k = 12$, a napr. ďalej $k = 14$ a $k = 13$.)

Oproti predchádzajúcemu algoritmu sme na tom lepšie: zatiaľ čo ten potreboval vyskúšať k hodnôt, tomuto stačí $2\log_2 k$. Náš nový algoritmus teda spraví $O(k^2 \log k)$ krokov. Ak označíme hodnotu v R_1 ako n, tento algoritmus potrebuje $O(n \log n)$ krokov.

Stále to ale ide aj lepšie, a prekvapujúco jednoducho. Predstavme si, že v premennej y máme štvorec premennej x. Ako vypočítať štvorec premennej $x+1$? Platí $(x+1)^2 = x^2 + 2x + 1$. Takže stačí k y pripočítať dvojnásobok x a ešte 1.

V našom programe budeme robiť presne to isté, len namiesto pripočítavania k y budeme rovno odpočítavať od hodnoty v registri R_1. Bude teda platiť, že keď v R_0 bude hodnota x, v R_1 bude $n - x^2$. A jednoducho povedané, keď sa nám R_1 minie, skončíme.

Časová zložitosť tohoto riešenia je zjavne lineárna od obsahu R_1, teda $O(n)$. Program vyzerá nasledovne:

```
///// odmocnina /////
a:      sub R_1, R_0
        sub R_1, R_0
        jz koniec
        dec R_1
        inc R_0
        jmp a     // teraz plati: v R_0 je x, v R_1 je n-x^2
koniec:
```

(V programe používame inštrukciu „**sub** R_i, R_j“. Tá vyzerá presne rovnako ako „**add** R_i, R_j“, len namiesto inštrukcie „**inc** R_i“ je v nej „**dec** R_i“. Výsledok je teda taký, že od hodnoty v R_i odčíta hodnotu z R_j.)

Riešenia celoštátneho kola kategórie A

A-III-1 Pizza vracia úder

Efektívne riešenia tejto úlohy budú založené na metóde dynamického programovania. Zaujímať nás budú hodnoty $D(s,p)$ definované nasledovne: $D(s,p)$ je 1 (true, pravda), ak sa dá dosiahnuť výsledná suma s použitím prvých p položiek zo vstupu. Riešením zadanej úlohy je zjavne najmenšie nezáporné s také, že $D(s,N)$ je pravda.

Základné riešenie:

Nech S je súčet všetkých položiek a_i zo vstupu. Zjavne žiadny dosiahnuteľný výsledok nebude v absolútnej hodnote väčší ako S. Preto sa stačí obmedziť na hodnoty $D(s,p)$, kde $s \in \{-S,\ldots,S\}$.

Hodnoty $D(s,p)$ vieme počítať pomocou nasledujúcich vzťahov:

$$\begin{aligned} D(s,0) &\stackrel{def}{=} [s=0] \\ D(s,p) &\stackrel{def}{=} D(s-a_p,p-1) \ \vee \ D(s+a_p,p-1) \end{aligned}$$

Slovne: Pomocou 0 položiek vieme vyrobiť len sumu 0. Pomocou p položiek vieme sumu s vyrobiť práve vtedy, ak vieme pomocou $p-1$ položiek vyrobiť aspoň jednu zo súm $s-a_p$ a $s+a_p$.

Takéto riešenie má časovú zložitosť $O(SN)$. Keďže každé a_i je najviac K, vieme, že $S \leq KN$, a teda časová zložitosť tohoto algoritmu je $O(KN^2)$.

V nasledujúcich častiach budeme toto riešenie postupne zlepšovať.

Kompresia vstupných údajov:

Všimnime si, že výsledok nezáleží na poradí položiek na vstupe, iba na tom, koľkokrát sa ktoré číslo z intervalu 1 až K vyskytuje v Marcovej postupnosti. Označme n_i počet výskytov čísla i. Úlohu si teraz môžeme preformulovať nasledovne: hľadáme K čísel x_i (kde $0 \leq x_i \leq n_i$) takých, že pokiaľ x_i výskytov čísla i prehlásime za príjem a $n_i - x_i$ za výdaj, zisk bude nezáporný a najmenší možný.

Veľkosť výsledku:

Riešenie úlohy je menšie ako $2K$. Ak by totiž mal byť výsledný zisk $2K$ alebo ešte viac, môžeme zmeniť niektoré `+` na `-`, čím zmenšíme zisk a ešte neklesneme do záporu.

Ako na veľké počty položiek:

Predstavme si napríklad situáciu, kedy $K = 5$ a Marco má na zozname milión položiek s hodnotou 3. Intuícia hovorí, že nič nepokazí, ak zhruba polovicu z nich dá ako prijmy a druhú polovicu ako výdaje.

Ak by sme takéto niečo vedeli poriadne sformulovať a dokázať, dosť silne by to nášmu riešeniu pomohlo.

Ako by takéto tvrdenie mohlo vyzerať? Nejako takto: „Nech je nejaké n_i **dosť veľké**, potom existuje optimálne riešenie, kde nemajú všetky položky veľkosti i rovnaké znamienka.“

Ak takéto tvrdenie dokážeme, vieme ho nasledovne využiť: „Nech je nejaké n_i **dosť veľké**. Potom existuje optimálne riešenie, v ktorom je aspoň jeden výskyt i ako príjem a jeden ako výdaj. Tieto dva výskyty môžeme teda vynechať. To znamená, že ak v pôvodnej úlohe zmenšíme n_i o 2, optimálne riešenie sa tým nezmení.“

Zostáva prísť na to, čo máme dosadiť namiesto „**dosť veľké**“, aby to platilo.

Predstavme si, že už máme nejaké optimálne riešenie. Nech p je veľkosť položky, ktorá nás zaujíma. Jediné zaujímavé situácie sú keď $x_p = 0$ alebo $x_p = n_p$, teda keď majú všetky výskyty položky p rovnaké znamienko. Rozoberieme situáciu, keď sú všetky kladné, pre záporné to bude vyzerať podobne.

Rozdeľme položky na príjmy a výdaje. To, čo teraz chceme dosiahnuť, je nájsť niekoľko výdajov, ktoré budú mať dokopy súčet p, $2p$, ..., alebo $(n_p - 1)p$. Ak sa nám toto podarí, môžeme tieto výdaje zmeniť na príjmy a zodpovedajúci počet príjmov veľkosti p na výdaje. A tým vyhráme, lebo dosiahneme, že niektoré výskyty položky p budú príjmy a niektoré budú výdaje.

Nech P je súčet príjmov. Teraz využijeme, že vieme ohraničiť výsledok: Keďže výsledok je menší ako $2K$, musí byť súčet výdajov $V > P - 2K$. Keďže každý výdaj je najviac K, bude výdajov aspoň $\lceil V/K \rceil$.

Z toho dostávame, že ak $P \geq K(p+1)$, bude výdajov aspoň p.

Keď teraz máme aspoň p výdajov, vyberme si z nich presne p a označme ich $v_1, \ldots, v_p$. Všimneme si teraz tieto súčty: 0, v_1, $v_1 + v_2$, ..., $v_1 + \cdots + v_p$. Toto je $p + 1$ rôznych čísel. Podľa Dirichletovho princípu niektoré dve z nich musia dávať po delení p rovnaký zvyšok. Nech sú to súčty po v_i a po v_j.

Potom číslo $v_{i+1} + \cdots + v_j$ je násobkom p. Presnejšie, je to číslo z množiny $\{p, 2p, \ldots, Kp\}$.

Platí teda tvrdenie: „Nech pre nejaké p platí $n_p > 2K$. Potom optimálne riešenie je rovnaké ako keby n_p bolo o 2 menšie.“

Dôkaz: Zoberme ľubovoľné optimálne riešenie. Ako sme už povedali, jediné dva zlé prípady sú, ak v ňom berieme všetky položky p ako príjmy / všetky ako výdaje.

V prvom prípade ale vieme, že $P \geq n_p p \geq (2K+1)p = Kp + (Kp+p) > Kp + K$, a teda bude aspoň p výdajov. Potom ale vieme nájsť niekoľko výdajov, ktorých súčet je násobkom p a je menší ako $n_p p$. Tieto výdaje zmeníme na príjmy a niekoľko príjmov veľkosti p na výdaje.

Druhý prípad vyzerá analogicky. Ak by všetky p_i boli výdaje, príjmov musí byť aspoň toľko ako v prvom prípade výdajov. Preto tentokrát medzi príjmami vyberieme vhodnú množinu, ktorú vymeníme s výdajmi veľkosti p.

Prvý lepší algoritmus:

Pozrieme sa na hodnoty n_i a každú z nich upravíme, aby bola nanajvýš rovná $2K$. (Ak teda nejaké n_i bolo viac ako $2K$, po úprave bude buď $2K$, alebo $2K-1$, podľa jeho parity.)

Teraz zoberieme takto zmenšenú množinu príjmov a výdajov a pre ne použijeme pôvodný algoritmus s časovou zložitosťou $O(NS)$.

Teraz už ale vieme povedať, že (upravený) počet všetkých položiek je $N = O(K^2)$ a súčet všetkých položiek je $S = O(K^3)$, a teda náš algoritmus má časovú zložitosť $O(K^5)$. Presnejšie $O(N + K^5)$, lebo v $O(N)$ musíme načítať vstup.

Veľkosť priebežných výsledkov:

Zatiaľ sme sa nijako nezaoberali tým, či nám naozaj treba uvažovať všetky možné priebežné súčty, ktoré sa dajú dosiahnuť. Podobne ako pri obmedzení počtov položiek budeme chcieť ukázať, že (bez ohľadu na usporiadanie položiek) existuje optimálne riešenie, v ktorom žiaden priebežný zisk nie je (v absolútnej hodnote) príliš veľký.

Základná myšlienka bude podobná ako pri zmenšovaní počtov položiek:

Predstavme si, že máme optimálne riešenie, pri ktorom je nejaký medzisúčet M veľmi veľký. Ako by sa dalo takéto riešenie upraviť na iné, rovnako dobré, pri ktorom bude tento medzisúčet menší?

Keďže M je veľké, znamená to, že z položiek, ktoré dávajú tento medzisúčet, sme veľa prehlásili za príjmy. A naopak, keďže vieme, že výsledok je malý, bude medzi zvyšnými položkami veľa takých, ktoré sme prehlásili za výdaje.

Ak by sme teraz vybrali niekoľko zo spominaných príjmov a niekoľko zo spomínaných výdajov tak, aby dali dokopy súčet nula, vieme medzisúčet M zmenšiť tak, že zmeníme znamienko všetkým nájdeným príjmom a výdajom.

Ešte raz a poriadne. Dokážeme, že platí nasledovné tvrdenie: „Existuje optimálne riešenie, v ktorom je absolútna hodnota každého čiastočného súčtu menšia ako $2K^2 + 3K$.“

Aby sme toto dokázali, ukážeme, že ak máme väčší čiastočný súčet, vieme ho zmenšiť, a pritom nezväčšiť žiaden iný čiastočný súčet a nepokaziť optimálnosť riešenia. (Pre záporné čiastočné súčty dôkaz neuvádzame, vyzerá analogicky.)

Majme teda optimálne riešenie, v ktorom súčet prvých x položiek je $M \geq 2K^2 + 3K$. Teraz pôjdeme od x-tej položky k prvej a budeme vyberať položky, ktoré sú v našom optimálnom riešení príjmami. Prestaneme, keď súčet vybraných položiek prekročí K^2.

Ďalej pôjdeme od $(x + 1)$-ej položky k N-tej a budeme vyberať tie, ktoré sú v našom riešení výdaje. Prestaneme tesne pred tým, ako by súčet všetkých vybraných položiek klesol pod nulu.

Zjavne takto vyberieme aspoň K príjmov a aspoň K výdajov.

Všimnime si, že všetky čiastočné súčty, ktoré zodpovedajú úseku, z ktorého sme práve vybrali príjmy, sú väčšie alebo rovné $K^2 + 2K$. A teda ak by sme aj všetkých K vybratých položiek zmenili z príjmov na výdaje, budú všetky tieto čiastočné súčty naďalej kladné, a teda sa ich absolútne hodnoty zmenšia.

Teraz potrebujeme ukázať, že existuje nejaká neprázdna podmnožina nami vybraných príjmov a výdajov, ktorá má súčet 0.

To spravíme nasledovne: Usporiadajme vybrané príjmy a výdaje do postupnosti $p_1, p_2, \ldots$ tak, aby všetky čiastočné súčty $p_1 + \cdots + p_q$ boli z množiny $\{0, \ldots, 2K - 1\}$. (Rozmyslite si, že toto vieme vždy spraviť.) Keďže má naša postupnosť aspoň $2K$ členov, a teda aspoň $2K+1$ čiastočných súčtov, musia byť dva čiastočné súčty rovnaké, a teda má zodpovedajúci úsek postupnosti súčet nula.

Ak teraz zoberieme zodpovedajúce príjmy a výdaje v optimálnom riešení a zmeníme im znamienka, dostaneme nové optimálne riešenie, pričom sme zmenšili niektoré čiastočné súčty (vrátane toho vybraného veľkého) a ostatné čiastočné súčty zostali nezmenené.

Tento proces môžeme opakovať, až kým nedostaneme optimálne riešenie, kde sú všetky čiastočné súčty malé. Analogicky sa vieme zbaviť príliš záporných čiastočných súčtov.

Druhý lepší algoritmus:

Vieme už, že stačí hľadať také riešenia, v ktorých sú všetky čiastočné súčty medzi $-2K^2 - 3K$ a $2K^2 + 3K$. Keď pridáme toto ohraničenie do predchádzajúceho riešenia, dostávame algoritmus s časovou zložitosťou $O(N + K^4)$.

Šikovnejšie počítanie dosiahnuteľných medzisúčtov:

Položky s rovnakou hodnotou budeme spracúvať všetky naraz.

Teraz nás teda budú zaujímať nové hodnoty $E(s,p)$ definované nasledovne: $E(s,p)$ je 1 (true, pravda), ak sa dá dosiahnuť výsledná suma s použitím všetkých položiek, ktoré majú hodnoty 1 až p. Riešením zadanej úlohy je zjavne najmenšie nezáporné s také, že $E(s,K)$ je pravda.

Pre $p > 0$ platí: $E(s,p)$ je pravda práve vtedy, ak je pravda aspoň jedna z hodnôt $E(s - pn_p, p-1)$, $E(s - p(n_p - 2), p-1)$, $\ldots$, $E(s + pn_p, p-1)$.

Slovne: pomocou položiek s veľkosťami 1 až p vieme sumu s vyrobiť vtedy a len vtedy, ak vieme pomocou položiek menších ako p vyrobiť nejakú sumu s' takú, že $s - s'$ vieme vyrobiť pomocou položiek veľkosti p.

Ak by sme hodnoty $E(s,p)$ počítali priamo podľa tejto definície, dostali by sme rovnako efektívne riešenie ako v predchádzajúcom prípade.

Na to, aby sme dosiahli lepšiu časovú zložitosť, všimnime si nasledujúcu vec: Ak nás zaujíma, či platí $E(s+2p,p)$, budeme sa pozerať na skoro tie isté hodnoty $E(?,p-1)$ ako keď sme zisťovali, či platí $E(s,p)$.

Ako si túto prácu ušetriť? Označme
$M(s,p) = E(s - pn_p, p-1) + E(s - p(n_p - 2), p-1) + \cdots + E(s + pn_p, p-1)$.
Slovne, $M(s,p)$ je počet takých súm, ktoré sa dajú naskladať z položiek menších ako p, a z ktorých vieme položkami veľkosti p vyrobiť sumu s.

$M(s,p)$ je takmer to isté ako $E(s,p)$, len namiesto $\vee$ sme použili $+$. Zjavne teda $E(s,p)$ je pravda práve vtedy, keď $M(s,p)$ je kladné.

Čo sme tým získali? To, že vieme, že platí nasledujúci vzťah:
$M(s+2p,p) = M(s,p) - E(s - pn_p, p-1) + E(s + 2p + pn_p, p-1)$

Takže „hrubou silou“ stačí spočítať prvých $2p$ hodnôt $M(?,p)$, každú ďalšiu dostaneme v konštantnom čase.

Na výpočet jednej hodnoty hrubou silou potrebujeme čas $O(n_p)$, čo vieme zhora odhadnúť ako $O(K)$. Výpočet $2p$ hodnôt hrubou silou teda bude trvať $O(K^2)$. Ostatných hodnôt, ktoré budeme počítať, je $O(K^2)$.

Celkovo teda vieme pre konkrétne p spočítať všetky hodnoty $E(?,p)$ v čase $O(K^2)$. Tým dostávame algoritmus s časovou zložitosťou $O(N + K^3)$.

Listing programu:

```
program zisk;
const MAXK = 100;                          { maximální velikost příjmu či výdaje }
      MAXZ = 2 * MAXK * (MAXK + 2);              { maximální velikost prvku Z_t }
```

```
type mnozina = record       { true je nastaveno na pozicích všech prvku množiny }
       prvky : array[-MAXZ .. MAXZ] of boolean;
    end;
var m : array[1 .. 2] of mnozina;                        { množiny Z_t a Z_(t+1) }
    predchozi : integer;       { Z_t je m[predchozi], Z_(t+1) je m[3-predchozi] }
    k : integer;
    amaxz : integer;                                                   { 2k(k+2) }
    ni : array[1 .. MAXK] of integer;                                { čísla n_i }
    mz : array[-MAXZ .. MAXZ] of integer;                         { čísla m_z ... }
                     { ve skutečnosti by stačilo si pamatovat jen posledních 2t }

{ Inicializuje MN na prázdnou množinu }
procedure smaz (var mn : mnozina);
var i : integer;
begin for i := -amaxz to amaxz do mn.prvky[i] := false; end;

{ Vrátí true pokud X je prvkem množiny MN }
function je_prvek (var mn : mnozina; x : integer) : boolean;
begin je_prvek := (abs (x) <= amaxz) and mn.prvky[x]; end;

{ Z množiny Z_(T-1) (P) vytvoří množinu Z_T (MN) }
procedure dalsi_zisky (var p, mn : mnozina; t : integer);
var yt, i, z, max_yt : integer;
begin
  max_yt := t * ni[t];
  { spočteme čísla m_z pro z v rozmezí -amaxz ... -amaxz + 2t-1 }
  for z := -amaxz to -amaxz + 2 * t - 1 do begin
    i := 0;
    yt := -max_yt;
    while yt <= max_yt do begin
      if je_prvek (p, z + yt) then inc (i);
      inc (yt, 2 * t);
    end;
    mz[z] := i;
  end;
  { a nyní zbývající hodnoty m_z }
  for z := -amaxz + 2 * t to amaxz do begin
    i := mz[z - 2 * t];
    if je_prvek (p, z + max_yt) then inc (i);
    if je_prvek (p, z - 2 * t - max_yt) then dec (i);
    mz[z] := i;
  end;
  { do množiny mn dáme čísla z taková, že m_z není nula }
  for z := -amaxz to amaxz do if mz[z] <> 0 then mn.prvky[z] := true;
end;

{ Vrátí nejmenší nezáporné číslo v množině MN }
function minimum (var mn : mnozina) : integer;
var i : integer;
begin
```

```
  for i := 0 to amaxz do
    if mn.prvky[i] then begin minimum := i; break; end;
end;

{ Načte seznam příjmů či výdajů }
procedure nacti;
var i, n, a : integer;
begin
  readln (n, k);
  amaxz := 2 * k * (k + 2);
  for i := 1 to k do ni[i] := 0;
  for i := 1 to n do begin read (a);  inc (ni[a]);  end;
end;

{ Omezí čísla n_i na nanejvýš 2k+1 }
procedure omez_ni;
var i : integer;
begin
  for i := 1 to k do
    if ni[i] > 2 * k + 1 then begin
      if odd(ni[i]) then ni[i] := 2 * k + 1 else ni[i] := 2 * k;
    end;
end;

var t : integer;
begin
  nacti;
  omez_ni;
  smaz (m[1]);   smaz (m[2]);
  predchozi := 1;
  { Z_0 = (0) }
  m[predchozi].prvky[0] := true;
  for t := 1 to k do begin
    dalsi_zisky (m[predchozi], m[3 - predchozi], t);
    smaz (m[predchozi]);
    predchozi := 3 - predchozi;
  end;
  writeln (minimum (m[predchozi]));
end.
```

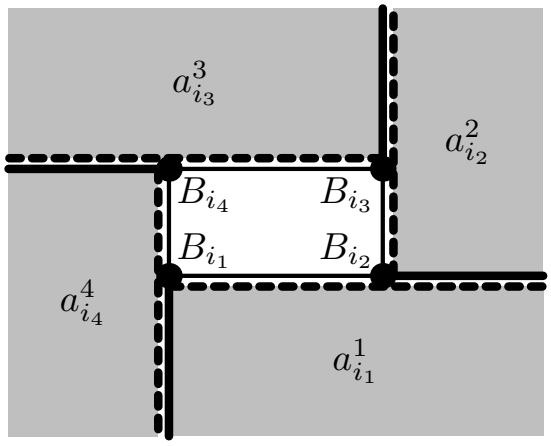
A-III-2 Obdĺžnik

Ukážeme si riešenie, ktorého časová zložitosť je $O(N^2)$ a jeho pamäťové nároky sú lineárne od počtu daných bodov.

Najprv vymyslíme, ako pre daný obdĺžnik $A_1A_2A_3A_4$ rýchlo určiť počet bodov, ktoré ležia vnútri tohto obdĺžnika. Pre každý bod B_i spočítame počet bodov $B_{i'}$, ktorých x-ová súradnica je väčšia alebo rovná x-ovej súradnici bodu

B_i a zároveň y-ová súradnica je ostro menšia než y-ová súradnica bodu B_i. Tento počet označíme a_i^1. Podobne, a_i^2 bude počet bodov s x-ovou súradnicou ostro väčšou a y-ovou súradnicou väčšou alebo rovnou, a_i^3 počet bodov s x-ovou súradnicou menšou alebo rovnou a y-ovou ostro väčšou a konečne a_i^4 bude počet bodov s x-ovou ostro menšou a y-ovou menšou alebo rovnou. Každé z čísel a_i^1, a_i^2, a_i^3 a a_i^4 je ľahké vypočítat v čase $O(N)$ (pre jeden bod B_i). Teda na spočítanie všetkých $4N$ čísel a_i^1, a_i^2, a_i^3 a a_i^4, $1 \leq i \leq n$, spotrebujeme čas $O(N^2)$.

Ľahko nahliadneme, že pokiaľ vrchol A_j obdĺžnika $A_1A_2A_3A_4$ je bod B_{i_j}, potom počet bodov ležiacich vnútri obdĺžnika $A_1A_2A_3A_4$ je $N - a_{i_1}^1 - a_{i_2}^2 - a_{i_3}^3 - a_{i_4}^4$ (viď obrázok). Pre jeden obdĺžnik $A_1A_2A_3A_4$ môžeme teda určiť počet bodov, ktoré ležia vnútri tohto obdĺžnika, v konštantnom čase.



(Poznámka: Existujú aj iné možné postupy, napr. pre vhodné body si spočítame počet bodov, ktoré ležia v ľavom hornom kvadrante od daného bodu, a z týchto informácií poskladáme v konštantnom čase počet bodov v ľubovoľnom obdĺžniku. Väčšina alternatívnych postupov ale potrebuje pamäť $O(N^2)$.)

Teraz si popíšeme, ako môžeme rýchlo najsť všetky obdĺžniky $A_1A_2A_3A_4$, ktorých hrany sú rovnoběžné s osami a ktorých vrcholy sú v niektorých zo zadaných bodov. Najprv si všetky body zotriedme vzostupne podľa x-ovej súradnice a tie body, ktoré majú zhodnú x-ovú súradnicu, zotriedime medzi sebou vzostupne podľa y-ovej súradnice. Pre každý bod B_i teraz môžeme vypísať tie body, ktoré majú rovnakú x-ovú súradnicu ako B_i a väčšiu y-ovú súradnicu, v čase lineárnom od ich počtu. Podobne si zotriedime všetky body podľa y-ovej súradnice a v prípade zhody podľa x-ovej, aby sme mohli ľahko nájsť body s rovnakou y-ovou a väčšou x-ovou súradnicou. Na zotriedenie bodov potrebujeme čas $O(N \log N)$ (použijeme napr. triediaci algoritmus heapsort alebo quicksort)

a na uloženie zotriedeného zoznamu bodov vrátane odkazov doňho potrebujeme pamäť lineárnu od N.

Teraz si popíšeme postup, ako nájsť všetky obdĺžniky $A_1A_2A_3A_4$, ktoré spĺňajú podmienky zo zadania úlohy. Zafixujeme jeden bod B_i a určíme všetky body B_j také, že bod B_i je vrchol A_1 a bod B_j vrchol A_3 nejakého obdĺžniku $A_1A_2A_3A_4$ (všimnite si, že vrcholy A_2 a A_4 sú bodmi B_i a B_j jednoznačne určené).

K nájdeniu všetkých takýchto bodov B_j pre bod B_i si vytvoríme pomocné pole C, ktorého všetky zložky $C[j]$ budú na začiatku rovné nule. Potom vezmeme všetky body $B_{i'}$, ktoré majú rovnakú x-ovú súradnicu ako B_i a zároveň väčšiu y-ovú súradnicu, a pre každý takýto bod $B_{i'}$ vezmeme všetky body $B_{i''}$, ktoré majú rovnakú y-ovú a väčšiu x-ovú súradnicu než $B_{i'}$. Body $B_{i'}$ zvládneme najsť v lineárnom čase od ich počtu a rovnako body $B_{i''}$ pre každý bod $B_{i'}$ vieme najsť v lineárnom čase od ich počtu. Pretože body $B_{i''}$ sú rôzne pre rôzne body $B_{i'}$, trvá nájdenie všetkých bodov $B_{i''}$ čas lineárny od N. Zložku $C[i'']$ poľa C zvýšime o jedna pre každý bod $B_{i''}$, ktorý sme takto našli. Zdôrazňujeme, že teraz je každá zložka poľa C rovná buď 0 alebo 1.

Potom pre bod B_i vezmeme všetky body $B_{i'}$ s rovnakou y-ovou a väčšou x-ovou súradnicou a pre také body $B_{i'}$ nájdeme všetky body $B_{i''}$ s rovnakou x-ovou súradnicou a väčšou y-ovou súradnicou. Za každý takýto bod zvýšime hodnotu $C[i'']$ o jedna. Zložky poľa C sú teraz rovné $0, 1$ alebo 2. Ďalej platí, že $C[j]$ je rovné 2 vtedy a len vtedy, keď pre body B_i a B_j existuje bod, ktorý má rovnakú x-ovú súradnicu ako B_i a rovnakú y-ovú súradnicu ako B_j, a zároveň existuje bod, ktorý má rovnakú x-ovú súradnicu ako B_j a rovnakú y-ovou súradnicu ako B_i. Potom ale takéto dva body tvoria vrcholy A_2 a A_4 obdĺžnika, ktorého vrchol A_1 je B_i a vrchol A_3 je B_j. Pre daný bod B_i môžeme teda v čase $O(N)$ nájsť všetky body B_j, ktoré tvoria protiľahlé vrcholy A_1 a A_3 nejakého obdĺžnika $A_1A_2A_3A_4$. Určiť počet vnútorných bodov takéhoto obdĺžnika vieme v konštantnom čase, ako sme si vysvetlili na začiatku riešenia.

Ako sme si práve popísali, dá sa pre daný bod B_i v čase $O(N)$ nájsť všetky obdĺžniky s ľavým dolným rohom B_i, a pre každý z nich spočítať, koľko bodov v ňom leží. Toto spravíme pre každé i.

Výsledný algoritmus má časovú zložitosť $O(N^2)$ a pamäťovú $O(N)$.

Listing programu:

```
program obdelniky;
const MAXN=1000; { maximální počet zadaných bodů }
var N,i: longint;
    sorted_by_x, sorted_by_y, index_by_x, index_by_y: array[1..MAXN] of longint;
```

```
    Bx, By, A1, A2, A3, A4, C: array[1..MAXN] of longint;
    nejlepsi_pocet: longint;

procedure spocitej_A;
var i,j: longint;
begin
  for i:=1 to N do begin
    A1[i]:=0; A2[i]:=0; A3[i]:=0; A4[i]:=0;
    for j:=1 to N do begin
      if (Bx[j]>=Bx[i]) and (By[j]<By[i]) then inc(A1[i]);
      if (Bx[j]>Bx[i]) and (By[j]>=By[i]) then inc(A2[i]);
      if (Bx[j]<=Bx[i]) and (By[j]>By[i]) then inc(A3[i]);
      if (Bx[j]<Bx[i]) and (By[j]<=By[i]) then inc(A4[i]);
    end;
  end;
end;

{ funkcie quick_?, ktore pouzitim quicksortu utriedia pole sorted_by_?
  podla ?-ovej suradnice, z priestorovych dovodov neuvadzame }

procedure setrid;
var i:longint;
begin
  for i:=1 to N do begin sorted_by_x[i]:=i; sorted_by_y[i]:=i; end;
  quick_x(1,N);  for i:=1 to N do index_by_x[sorted_by_x[i]]:=i;
  quick_y(1,N);  for i:=1 to N do index_by_y[sorted_by_y[i]]:=i;
end;

procedure zkus_vrchol(i: longint);
var i1, i2, j: longint;
    a2_index, a4_index: array[1..MAXN] of longint;
begin
  for j:=1 to N do C[j]:=0;
  i1:=index_by_x[i]+1;
  while Bx[sorted_by_x[i1]]=Bx[i] do begin
    i2:=index_by_y[sorted_by_x[i1]]+1;
    while By[sorted_by_y[i2]]=By[sorted_by_x[i1]] do begin
      inc(C[sorted_by_y[i2]]);
      inc(i2);
      a2_index[i2]:=sorted_by_x[i1];
    end;
    inc(i1)
  end;
  i1:=index_by_y[i]+1;
  while By[sorted_by_y[i1]]=By[i] do begin
    i2:=index_by_x[sorted_by_y[i1]]+1;
    while Bx[sorted_by_x[i2]]=Bx[sorted_by_y[i1]] do begin
      inc(C[sorted_by_x[i2]]);
      inc(i2);
      a4_index[i2]:=sorted_by_y[i1];
```

```
    end;
    inc(i1)
  end;
  for j:=1 to N do
    if (C[j]=2)
       and (N-A1[i]-A2[a2_index[j]]-A3[j]-A4[a4_index[j]] > nejlepsi_pocet)
    then begin
      nejlepsi_pocet := N - A1[i] - A2[a2_index[j]] - A3[j] - A4[a4_index[j]];
    end;
end;

begin
  readln(N); for i:=1 to N do readln(Bx[i],By[i]); end;
  spocitej_A;
  setrid;
  nejlepsi_pocet:=0;
  for i:=1 to N do zkus_vrchol(i);
  if nejlepsi_pocet=0 then writeln('Nemá řešení.') else writeln(nejlepsi_pocet);
end.
```

A-III-3 Grafomat a šamani

Na úvod si uvedomme, že stačí vedieť zadanú úlohu riešiť pre stromy. Ak totiž máme našu úlohu riešiť pre 3-graf, môžeme ho z vrcholu, ktorý je na začiatku označený, prehľadať do šírky. Podobne ako v riešení úlohy z domáceho kola si navyše v každom vrchole zapamätáme, odkiaľ sme doň prvýkrát prišli. Takto dostaneme zakorenený strom s koreňom vo vrchole, ktorý bol na začiatku označený.

Jednoduchšie ale pomalšie riešenie:

Chceme rozdeliť vrcholy stromu na dve rovnako veľké množiny. Keby sme túto úlohu riešili na klasickom počítači, mohli by sme vymyslieť napríklad nasledujúce riešenie: prehľadáme celý strom do hĺbky a vrcholy v poradí, v akom ich objavujeme, farbíme striedavo na červeno a na modro. V Pascale by to vyzeralo napríklad nejako takto:

```
function prejdi_strom ( v:vrchol ; f:farba ) : farba;
var i : integer;
begin
   if (f=cervena) f:=modra else f:=cervena;
   v.farba := f;
   for i:=1 to v.pocetSynov do   f := prejdi_strom( v.syn[i], f );
end;
```

Toto riešenie vieme naprogramovať aj v grafomate. Namiesto rekurzie si budeme medzi vrcholmi odovzdávať značku, ktorá bude reprezentovať miesto, kde sa práve prehľadávanie nachádza. Po každom takte bude v grafe práve jedna

značka. Vo vrchole, ktorý má značku, si navyše budeme pamätať naposledy použitú farbu a to, či sme do vrcholu práve prišli, alebo už chceme odísť z jeho podstromu.

Presnejšie to bude prebiehať takto:

1. Na začiatku dostane značku koreň, pamätáme si v ňom, že sme práve prišli a posledná použitá farba bola 0.
2. Ak vrchol má značku a práve sme doň prišli, v nasledujúcom takte zmení pamätanú farbu na opačnú a ofarbí sa ňou. Potom:
   - Ak nemá žiadnych synov, značku si nechá a zapamätá si, že chceme odísť preč.
   - Ak má nejakých synov, pošle značku prvému z nich.
3. Ak vrchol má značku a pamätáme si, že chceme z neho odísť:
   - Ak má mladšieho brata (teda vrchol, ktorý má rovnakého otca a u otca o jedno väčšie poradové číslo), pošle jemu značku a u brata si zapamätáme, že sme práve prišli.
   - Ak mladšieho brata nemá, odovzdá značku späť otcovi a zapamätáme si u otca, že budeme z neho chcieť odísť.
   - Ak už nemá ani otca, sme v koreni, celý graf je ofarbený a končíme.

Tieto pravidlá vieme priamočiaro prepísať do programu pre grafomat. Bude to fungovať tak trochu naopak. Napríklad namiesto toho, aby sme raz poslali značku mladšiemu bratovi, bude sa každý vrchol v každom takte pozerať na svojho staršieho brata, či ten nemá značku, ktorú chce poslať preč. Keď niekedy vrchol uvidí, že ju jeho starší brat má, nastaví si, že teraz je značka u neho.

Toto riešenie má časovú zložitosť úmernú počtu vrcholov v grafe.

Ide to však aj lepšie. Vzorové riešenie bude mať časovú zložitosť úmernú tomu, ako ďaleko je najvzdialenejší vrchol od toho, kde výpočet začíname – teda od hĺbky stromu.

Zložitejšie ale rýchlejšie riešenie:

Podobne ako v predchádzajúcom riešení začneme tým, že zadaný 3-graf prehľadáme do šírky, čím dostaneme zakorenený strom. Aby bolo naše riešenie rýchle, budeme chcieť jeho podstromy ofarbovať paralelne. Čo na to ale potrebujeme vedieť?

Ukážeme si najskôr na príklade približnú myšlienku riešenia, potom doplníme implementačné detaily.

Predstavme si, že máme napríklad vrchol v, ktorý má troch synov: a, b a c. Pozrime sa teraz na podstromy s koreňmi a, b, c. V niektorých z nich je párny

počet vrcholov. Tieto nás netrápia, lebo sa ich dá ofarbiť presne. Trápiť nás budú tie, kde je počet vrcholov nepárny. Keď ofarbíme taký podstrom „približne správne“, bude v ňom vrcholov jednej farby o jedno viac ako druhej.

Keby sme vo vrchole v vedeli, ktoré z podstromov s koreňmi a, b, c majú párnu a ktoré nepárnu veľkosť, ľahko by sme všetko vyriešili: do párnych podstromov oznámime „ofarbite sa rovnomerne“, do približne polovice nepárnych podstromov oznámime „ofarbite sa tak, aby bolo o jedno viac červených“ a do zvyšku „ofarbite sa tak, aby bolo o jedno viac modrých“.

Teraz už k tým sľúbeným detailom. Podrobnejšie bude náš algoritmus vyzerať nasledovne:

1. Prehľadáme graf zo začiatočného vrcholu, čím vznikne strom.
2. Budeme si posielať informácie od listov do koreňa a pre každý vrchol si spočítame, či má jeho podstrom párnu alebo nepárnu veľkosť.
3. Keď sa už koreň dozvie svoju paritu (tá určite vyjde párna), začneme po hladinách ofarbovať. V prvom takte tejto časti ofarbíme koreň farbou 0.
4. Ak už je nejaký vrchol v ofarbený, postupne sa ofarbia jeho synovia. Prvý syn dostane opačnú farbu ako v. Farba každého ďalšieho syna závisí od farby predchádzajúceho a parity veľkosti jeho podstromu.

   (Presnejšie, ak bol predchádzajúci podstrom párnej veľkosti, dostane ďalší syn rovnakú farbu, inak dostane opačnú.)

Uvedomte si, že ofarbenie, ktoré tento algoritmus vyrobí, bude úplne rovnaké ako to, ktoré vyrobil pomalší algoritmus. Iba sme na zrýchlenie použili to, že keď vieme paritu počtu vrcholov v podstrome, nemusíme čakať na to, ktorá farba v ňom bude použitá ako posledná – vieme si to vypočítať a rovno začať farbiť aj susedný podstrom.

Keďže pôvodný graf je 3-graf, nemá žiaden vrchol v strome viac ako troch synov. (Presnejšie, každý vrchol okrem koreňa má najviac dvoch synov.) Preto každú vrstvu stromu ofarbíme na najviac tri takty.

Listing programu:

```
var x: 0..1;                                  { 1=počáteční vrchol, 0=ostatní }
    y: 0..2 = 2;              { výstupní barva: 0=červená, 1=zelená, 2=neurčena }
    z: 0..4 = 0;            { hrana vedoucí k otci vrcholu, 0=neurčena, 4=kořen }
    q: 0..2 = 2;                { parita podstromu: 0=sudá, 1=lichá, 2=neurčena }
    i, k, l: 0..3 = 0;                                      { pomocné proměnné }

begin
   if z=0 then begin          { 1. fáze: prohledávání do šířky (stavění stromu) }
```

```
      if x=1 then z := 4                          { Kořen označíme speciálně }
      else for i:=1 to 3 do          { Když máme označeného souseda, máme otce }
         if (S[i].z > 0) or (S[i].x = 1) then z := i;
   end
   else if q=2 then begin                         { 2. fáze: počítání parity }
      l := 0;                                   { kolik synů má lichou paritu }
      k := 0;                                       { už můžeme paritu určit? }
      for i:=1 to 3 do
         if S[i].z = 0 then k := 1      { neprohledaný soused => určit nemůžeme }
         else if S[i].z = P[i] then                                { je to syn }
            if S[i].q = 2 then k := 1               { syn nemá paritu => ani my }
            else if S[i].q = 1 then l := l+1;                      { lichý syn }
      if k = 0 then q := 1 - l mod 2;    { toto funguje i když má vrchol 0 synů }
   end
   else if y=2 then begin                                   { 3. fáze: barvení }
      if (x=1) and (q<2) then y := 0      { dopočítáme paritu => obarvíme kořen }
      else if (z>0) and (S[z].y < 2) then begin         { otec už je obarven => }
         y := 1 - S[z].y;        { => dopočítáme svou barvu podle parity bratrů }
         k := P[z];                              { číslo hrany vedoucí z otce sem }
         for i := 1 to k-1 do          { za každého lichého staršího bratra ... }
            if S[z].S[i].q = 1 then y := 1 - y;             { ... barvu otočíme }
      end;
   end;
end.
```

A-III-4 Polícia zasahuje

Na začiatku si všimnime, že mapa kanalizácie predstavuje strom (čiže súvislý graf bez kružníc). Označme tento strom T. Môžeme si zvoliť ľubovoľný vrchol u a v ňom tento strom zakoreniť – odteraz bude pre nás u koreňom daného stromu T. Môžete si to predstaviť tak, že strom chytíme za vrchol u a zavesíme ho zaň na stenu.

V ďalšom texte budeme označovať T_v podstrom s koreňom v. To je tá časť stromu, ktorá by odpadla, keby sme vrchol v odstránili.

Vrchol v budeme nazývať *bezpečný*, ak sa žiaden mafián, ktorý býva v podstrome T_v, nevie dostať nestráženou cestou do vrcholu v.

Náš algoritmus bude postupovať smerom od listov stromu T k jeho koreňu u. Keď budeme spracúvať nejaký vrchol v, budeme pre neho chcieť spočítať dve veci:

- Aký je minimálny počet strážcov $S(v)$ potrebný na pokrytie podstromu T_v tak, aby sa žiadni dvaja mafiáni z neho nemohli stretnúť,

- a tiež, či vieme týmto počtom strážcov navyše dosiahnuť, aby bol vrchol v bezpečný.

Pre listy je situácia jednoduchá: V liste sa môže nachádzať najviac jeden mafián, a teda nie je potrebné ho oddeľovať. Počet potrebných strážcov je teda 0. Tento vrchol je bezpečný práve vtedy, ak v ňom nie je mafián.

Nech teraz náš algoritmus ide spracovať vrchol v. Označme si jeho priamych potomkov $v_1, \ldots, v_k$. Pre každý z podstromov $T_{v_1}, \ldots, T_{v_k}$ máme už spočítaný minimálny potrebný počet strážcov a aj to, či je pri ich použití vrchol v_i bezpečný.

Označme $S = S(v_1) + \cdots + S(v_k)$. Optimálne riešenie pre T_v je určite väčšie alebo rovné ako S (teda súčet optimálnych riešení pre jednotlivé podstromy), lebo v každom z týchto podstromov musíme od seba oddeliť všetkých mafiánov. My ale navyše musíme oddeliť od seba aj mafiánov v rôznych podstromoch, a možno ešte aj mafiána vo vrchole v. Ukážeme si, ako na to. Rozoberieme dva prípady.

Vo vrchole v je mafián. Ak majú všetky podstromy bezpečné optimálne riešenie, stačí nám na celý strom T_v použiť S strážcov. Lepšie to zjavne nejde. Výsledné rozmiestnenie strážcov určite nebude bezpečné, lebo mafián bývajúci vo v sa do v dostať vie.

V opačnom prípade musíme od seba oddeliť mafiána vo vrchole v a mafiánov v nebezpečných podstromoch. Musíme teda pridať aspoň jedného strážcu. Zjavne stačí pridať práve jedného strážcu, a to do vrcholu v. Takto teda dostaneme riešenie s $S + 1$ strážcami. Toto riešenie je určite bezpečné.

Vo vrchole v nie je mafián. Ak je najviac jeden podstrom, ktorý nemá bezpečné optimálne riešenie, stačí nám na celý strom T_v použiť S strážcov. Lepšie to zjavne nejde. Ak boli všetky podstromy bezpečné, bude bezpečné aj celkové rozmiestnenie, inak nie. (Ten istý mafián, ktorý to kazil v nebezpečnom podstrome, to kazí naďalej.)

V opačnom prípade musíme od seba oddeliť (aspoň dvoch) mafiánov v nebezpečných podstromoch. Musíme teda pridať aspoň jedného strážcu. Zjavne stačí pridať práve jedného strážcu, a to do vrcholu v. Takto teda dostaneme riešenie s $S + 1$ strážcami. Toto riešenie je určite bezpečné.

Keď takto nakoniec spracujeme vrchol, ktorý sme si zvolili ako koreň stromu, dostaneme optimálny počet strážcov. Ten vypíšeme na výstup a sme hotoví.

Spracovať jeden konkrétny vrchol vieme v čase priamo úmernom počtu jeho podstromov, čiže počtu hrán, ktoré z neho vychádzajú. Celkovo má teda tento algoritmus časovú zložitosť priamo úmernú súčtu stupňov vrcholov. Ale súčet stupňov vrcholov nie je nič iné ako dvojnásobok počtu hrán. (Každá hrana je zarátaná v dvoch stupňoch vrcholov.)

No a keďže zadaný graf je strom a má $N - 1$ hrán, je časová zložitosť tohto algoritmu $O(N)$.

Iné riešenie:

Táto úloha má aj iné lineárne riešenia, sú však náročnejšie na implementáciu ako vzorové riešenie. Uvedieme myšlienku jedného z nich.

Ak máme list stromu, v ktorom nebýva mafián, môžeme tento list a hranu, ktorá doň vedie, zahodiť, hodnotu optimálneho riešenia to nezmení. Podobne ak máme vrchol, z ktorého vedú dve hrany a v ktorom nebýva mafián, môžeme ho z grafu odstrániť a jeho dvoch susedov spojiť priamou hranou.

Ak už sa v nejakom okamihu žiadna z týchto operácií nedá spraviť, musí existovať vrchol, ktorého najviac jeden sused nie je list. (Dôkaz a návod na hľadanie: Ak by sme si strom zakorenili, stačí zobrať otca niektorého najhlbšieho listu.)

Rozmyslite si, že do tohto vrcholu potom musíme umiestniť strážnika. Tým sme sa ale zbavili tohto vrcholu a všetkých listov, ktoré s ním susedia. Vyhodíme ich z grafu a pokračujeme vyššie popísaným postupom ďalej, až kým už nie je čo riešiť.

Listing programu:

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

#define MAXN 100000

FILE *in, *out;          /* Vstupní a výstupní soubor */
int n, p;                /* Počet větvení a počet mafiánů */
int deg[MAXN];           /* Počet stok vedoucích z větvení */
int e[MAXN][2];          /* Seznam stok */
int edge_ptr[MAXN];      /* Index v neib, kde začínají sousedé každého větvení */
int neib[2*MAXN];        /* Seznamy sousedů větvení */
char podezrely[MAXN];    /* Je do daného větvení připojen dům mafiána? */
int hlidek;              /* Potřebný počet hlídek */

void nacti_vstup(void) {
  int i, a, b;
  int tmp_ptr[MAXN];
```

```
  /* Načteme stoky */
  fscanf(in, "%d %d", &n, &p);
  memset(deg, 0, n*sizeof(int));
  for (i = 0; i < n-1; i++) {
    fscanf(in, "%d %d", &a, &b);
    a--; b--;
    e[i][0] = a;
    e[i][1] = b;
    deg[a]++;
    deg[b]++;
  }

  /* Vytvoříme seznamy sousedů a nastavíme správně jejich počátky */
  edge_ptr[0] = 0;
  for (i = 1; i < n; i++)
    edge_ptr[i] = edge_ptr[i-1]+deg[i-1];
  memset(tmp_ptr, 0, n*sizeof(int));
  for (i = 0; i < n-1; i++) {
    neib[edge_ptr[e[i][0]]+tmp_ptr[e[i][0]]++] = e[i][1];
    neib[edge_ptr[e[i][1]]+tmp_ptr[e[i][1]]++] = e[i][0];
  }

  /* Načteme seznam mafiánů */
  for (i = 0; i < p; i++) {
    fscanf(in, "%d", &a);
    a--;
    podezrely[a] = 1;
  }
}

/* Spočte počet hlídek pro T_v, vrátí 1, pokud je T_v nehlídaná. */
int hledej(int v, int rodic) {
  int i, nehlidanych = 0;

  /* Projdeme všechny sousedy, ze kterých k nám vede stoka, a zjistíme situaci */
  for (i = 0; i < deg[v]; i++)
    if (neib[edge_ptr[v]+i] != rodic)
      nehlidanych += hledej(neib[edge_ptr[v]+i], v);
  /* Musíme větvení v hlídat? */
  /* To je třeba buď pokud máme alespoň 2 nehlídané sousedy (pak je třeba
   * izolovat mafiány z nich), nebo pokud ve 'v' sídlí mafián a máme alespoň
   * jednoho nehlídaného souseda. */
  if (nehlidanych > 1 || (podezrely[v] && nehlidanych > 0)) {
      hlidek++;
      return 0;
  }
  /* Jinak nemusíme hlídat, jen vrátíme, jestli je T_v nehlídaný */
  return nehlidanych || podezrely[v];
}
```

```
int main(void) {
  in = fopen("policie.in", "r");
  out = fopen("policie.out", "w");
  nacti_vstup();
  hledej(0,0);
  fprintf(out, "%d\n", hlidek);
  return 0;
}
```

A-III-5 Rybka Julka

Najjednoduchším správnym riešením by bolo postupne pre každý čas $t = 1, 2, 3, \ldots$ spočítať množinu všetkých políčok, kde sa Julka môže v danom čase nachádzať.

Ak M_t je množina tých políčok, kde mohla byť Julka v čase t, množinu M_{t+1} ľahko zostrojíme z množiny M_t. Stačí pridať tie políčka, ktoré susedia s nejakým políčkom z M_t a majú teplotu takú, že tam Julka smie v $(t+1)$-ej sekunde preplávať. A naopak, vyhodíme tie políčka, ktoré už sú pre Julku príliš horúce.

Časom buď narazíme na také T, že M_T bude obsahovať cieľové políčko, alebo zistíme, že aktuálna množina M_T je prázdna. V prvom prípade sme práve našli najrýchlejšiu cestu do cieľa, v druhom sa nám práve Julka uvarila.

Takéto riešenie vieme ľahko implementovať tak, aby malo časovú zložitosť $O(RST)$, kde R, S sú rozmery rybníka a T je čas, kedy prestaneme hľadať.

Myšlienka efektívnejšieho riešenia:

Lepšie riešenie bude založené na nasledujúcom pozorovaní: Pre každé políčko nám stačí vedieť, ako najrýchlejšie (a kadiaľ) sa naň vieme dostať. Totiž ak sa rozhodneme, že naša trasa povedie cez nejaké konkrétne políčko, nemá zmysel prísť tam inou ako najrýchlejšou cestou.

Pre každé políčko $[r, s]$ budeme teda chcieť spočítať hodnotu $c(r, s)$ – najmenší počet sekúnd, po ktorom vieme byť na tomto políčku. (Navyše si budeme pamätať aj smer, odkiaľ sem vedie jedna možná najrýchlejšia cesta.)

Na počítanie týchto hodnôt použijeme Dijkstrov algoritmus na hľadanie najkratšej cesty v grafe.

Myšlienka je nasledovná: Všetky možné políčka budeme mať rozdelené do dvoch množín: $\mathcal{H}$ budú hotové políčka, pre ktoré už vieme správnu hodnotu $c(r, s)$, a $\mathcal{S}$ budú ešte spracúvané políčka.

Pre každé políčko $[r, s]$ v $\mathcal{S}$ bude aktuálna hodnota $c(r, s)$ predstavovať najmenší čas, v akom sa vieme na políčko $[r, s]$ dostať, *ak ideme len cez políčka v* $\mathcal{H}$. (Prípadne $c(r, s) = \infty$, ak žiadna takáto cesta neexistuje).

Na začiatku nie je hotové žiadne políčko, teda $\mathcal{H} = \emptyset$. Zjavne $c(J_r, J_s) = 0$ a pre ostatné políčka je $c(r, s) = \infty$.

Zvyšok algoritmu bude prebiehať v kolách. V každom kole nájdeme v množine $\mathcal{S}$ jedno políčko, o ktorom vieme dokázať, že čas najkratšej cesty doň je už spočítaný správne. Toto políčko zakaždým presunieme do $\mathcal{H}$ a upravíme najlepšie časy pre políčka, ktoré zostali v $\mathcal{S}$. Akonáhle sa cieľové políčko dostane medzi hotové, môžeme skončiť.

Ako teda nájsť v množine $\mathcal{S}$ políčko, ktoré už môžeme presunúť medzi hotové? Jednoducho vyberieme z množiny $\mathcal{S}$ to políčko $[r, s]$, ktoré má najmenšiu hodnotu $c(r, s)$. (Ak by takých bolo viac, tak ľubovoľné z nich.) Tvrdíme, že hodnota $c(r, s)$ pre toto políčko je už správna.

Prečo? Všimnime si všetky možné spôsoby, ako sa dostať na políčko $[r, s]$. Už sme zobrali do úvahy všetky cesty, ktoré používajú len políčka z $\mathcal{H}$. Každá zo zvyšných ciest teda musí ísť cez aspoň jedno iné políčko $[r', s'] \in \mathcal{S}$. Lenže vieme, že už $c(r', s') \geq c(r, s)$, a teda žiadna takáto cesta nebude rýchlejšia ako $c(r, s)$.

Políčko $[r, s]$ teda prehlásime za hotové a presunieme ho z $\mathcal{S}$ do $\mathcal{H}$.

Teraz ešte zostáva upraviť hodnoty $c(r, s)$ pre políčka, ktoré zostali v $\mathcal{S}$. To je ale jednoduché: Jediné políčka, kde sa niečo mohlo zmeniť, sú susedia políčka $[r, s]$. Tam pribudla možnosť, že najrýchlejší spôsob, ako sa dostať na takéto políčko, je práve cez $[r, s]$.

Jednoducho teda prejdeme všetkých (nanajvýš štyroch) susedov políčka $[r, s]$. Pre každého suseda $p = [r_p, s_p]$: Ak p patrí do $\mathcal{H}$, nič sa nedeje. Ak patrí do $\mathcal{S}$, nová hodnota $c(r_p, s_p)$ bude minimum z doterajšej hodnoty a z hodnoty $c(r, s) + t$, kde t je čas potrebný na presun z $[r, s]$ na $[r_p, s_p]$.

(Čas t potrebný na presun závisí od $c(r, s)$ a od teplôt oboch políčok. Z týchto údajov ho vieme ľahko vypočítať v konštantnom čase.)

Implementácia efektívnejšieho riešenia:

Sú v princípe dva rôzne spôsoby, ako Dijkstrov algoritmus implementovať.

Ten jednoduchší a priamočiarejší vyzerá nasledovne: použijeme dve polia rozmerov rovnakých ako má rybník. V jednom si pre každé políčko budeme pamätať, či už je hotové, v druhom aktuálne hodnoty $c(r, s)$.

Nájsť nové políčko, ktoré treba presunúť do $\mathcal{H}$, vieme v čase $O(RS)$ tak, že prezrieme všetky políčka a nájdeme najlepšie z nich. Zvyšok kola, teda upraviť

hodnoty c pre susedov, už vieme v konštantnom čase.

Keďže v každom kole presunieme do $\mathcal{H}$ jedno políčko, kôl bude rádovo toľko ako políčok. Celková časová zložitosť takéhoto riešenia je teda $O(R^2S^2)$.

Je zjavné, že najpomalšou časťou predchádzajúcej implementácie je výber nového hotového políčka. Zbytočne znova a znova prechádzame všetky políčka. Nešlo by to lepšie?

Čo by to bola za rečnícka otázka, keby odpoveď nebola „áno“. Použijeme dátovú štruktúru nazývanú halda. (Podrobný popis haldy nebudeme uvádzať, v prípade záujmu ho nájdete napr. v Programátorskej liahni, rovnako ako aj podrobnejší popis tejto implementácie Dijkstrovho algoritmu.)

V halde si budeme pamätať políčka množiny $\mathcal{S}$, zoradené podľa doteraz zisteného času cesty do nich. Takto vieme nové hotové políčko vybrať v čase $O(\log(RS))$.

Keď teraz získame pre nejakého suseda nový, lepší čas, jednoducho vložíme do haldy pre tohto suseda nový záznam s týmto časom. (Netrápi nás, že ten starý tam niekde zostane, lebo ten nový vyberieme skôr.) Každú takúto úpravu vieme teda spraviť v čase $O(\log(RS))$.

Podarilo sa nám teda vylepšiť trvanie jedného kola z $O(RS)$ na $O(\log(RS))$, a tým časovú zložitosť celého algoritmu na $O(RS\log(RS))$.

Program uvedený v listingu namiesto nami uvedeného triku na upravovanie vzdialeností susedov používa inú metódu: pamätá si, kde v halde sa nachádza záznam pre ktoré políčko, a keď sa nejakémuj políčku zmenší vzdialenosť, „prebuble“ týmto políčkom v halde dohora.

Listing programu:

```
program rybka;

const   MAXM=1000;
        MAXN=1000;
        INFTY=3000000;

type    PPole=^TPole;
        TPole=record
                x,y:longint;
                halda:longint;
                t,c:longint;
                zpole:PPole;
        end;

var     m,n,x1,y1,x2,y2,t1,t2:longint;          { zadané parametry }
```

```
        rybnik:array[1..MAXM,1..MAXN] of TPole;  { pole rybníka }
        halda:array[1..MAXM*MAXN] of PPole;      { halda v poli }
        velhaldy:longint;

procedure prehod(pos1:longint; pos2:longint);    { přehoď v haldě dvě pole }
var tmp:PPole;
begin
        halda[pos1]^.halda:=pos2;
        halda[pos2]^.halda:=pos1;
        tmp:=halda[pos1];
        halda[pos1]:=halda[pos2];
        halda[pos2]:=tmp;
end;

procedure upravhaldu(var pole:TPole);    { sniž hodnotu c(x,y) pole v haldě }
var     hpos:longint;
begin
        hpos:=pole.halda;
        while (hpos>1) and (halda[hpos]^.c < halda[hpos div 2]^.c) do begin
                prehod(hpos,hpos div 2);
                hpos:=hpos div 2;
        end;
end;

function vyberzhaldy:PPole;         { vyber z haldy pole s nejnižším c(x,y) }
var     hpos,minpos:longint;
        hotovo:boolean;
begin
        if velhaldy=0 then
                vyberzhaldy:=nil
        else begin
                vyberzhaldy:=halda[1];
                halda[1]:=halda[velhaldy];
                halda[1]^.halda:=1;
                dec(velhaldy);
                hotovo:=false;
                hpos:=1;
                repeat
                        { buď už jsme na spodku haldy }
                        if (hpos*2>velhaldy) then hotovo:=true
                        { nebo máme jen jednoho potomka }
                        else if (hpos*2=velhaldy) then begin
                                hotovo:=true;
                                if (halda[hpos]^.c>halda[hpos*2]^.c) then
                                        prehod(hpos,hpos*2);
                        end
                        { nebo máme dva potomky }
                        else begin
                                minpos:=hpos;
                                if (halda[minpos]^.c>halda[hpos*2]^.c) then
```

```
                                        minpos:=hpos*2;
                                if (halda[minpos]^.c>halda[hpos*2+1]^.c) then
                                        minpos:=hpos*2+1;
                                if (minpos=hpos) then
                                        hotovo:=true
                                else begin
                                        prehod(hpos,minpos);
                                        hpos:=minpos;
                                end;
                        end;
                until hotovo;
        end;
end;

procedure vypis(var f:text; var pole:TPole);
begin
        if pole.zpole<>nil then begin
                vypis(f,pole.zpole^);
                if pole.c-pole.zpole^.c>1 then
                        write(f,pole.c-pole.zpole^.c-1,' ');
                if pole.x>pole.zpole^.x then write(f,'V ');
                if pole.x<pole.zpole^.x then write(f,'Z ');
                if pole.y>pole.zpole^.y then write(f,'J ');
                if pole.y<pole.zpole^.y then write(f,'S ');
        end;
end;

procedure vylepsipole(var ktere:TPole; var odkud:TPole);
var cek:longint;
begin
        { jak dlouho musím počkat, než se pole dost ohřeje? }
        cek:=t1-(odkud.c+ktere.t);
        if cek<0 then cek:=0;
        { jedná se o cílové pole? pokud ano, vůbec nečekej }
        if (ktere.x=x2) and (ktere.y=y2) then begin
                ktere.c:=odkud.c+1;
                ktere.zpole:=@odkud;
                upravhaldu(ktere);
        end
        { když počkám tolik, bude to ok na tomto i cílovém poli?
          je to lepší cesta? }
        else if (odkud.c+cek+odkud.t<=t2) and (odkud.c+cek+ktere.t+1<=t2)
             and (ktere.c>odkud.c+cek+1) then begin
                ktere.c:=odkud.c+cek+1;
                ktere.zpole:=@odkud;
                upravhaldu(ktere);
        end;
end;

var     min:PPole;
```

```
        hotovo:boolean;
        f1,f2:text;
        i,j:longint;
begin
        assign(f1,'rybka.in');    reset(f1);
        assign(f2,'rybka.out');   rewrite(f2);
        readln(f1,n,m,t1,t2);
        readln(f1,y1,x1,y2,x2);
        for j:=1 to n do
                for i:=1 to m do begin
                        read(f1,rybnik[i,j].t);
                        rybnik[i,j].x:=i;
                        rybnik[i,j].y:=j;
                        rybnik[i,j].c:=INFTY;
                        rybnik[i,j].zpole:=nil;
                        rybnik[i,j].halda:=velhaldy+1;
                        halda[velhaldy+1]:=@rybnik[i,j];
                        inc(velhaldy);
                end;
        rybnik[x1,y1].c:=0;
        upravhaldu(rybnik[x1,y1]);

        { hlavní smyčka výběru z haldy }
        hotovo:=false;
        repeat
                min:=vyberzhaldy;
                { buď už v haldě nic k přidání není }
                if (min=nil) or (min^.c=INFTY) then begin
                        writeln(f2,'Chudinka Julka!');
                        hotovo:=true;
                end
                { nebo vybírám cílové pole }
                else if (min^.x=x2) and (min^.y=y2) then begin
                        hotovo:=true;
                        vypis(f2,rybnik[x2,y2]);       { Ať žije Julka! }
                end
                { nebo je to nějaké obecné pole }
                else begin
                        if min^.x>1 then
                                vylepsipole(rybnik[min^.x-1,min^.y],min^);
                        if min^.y>1 then
                                vylepsipole(rybnik[min^.x,min^.y-1],min^);
                        if min^.x<M then
                                vylepsipole(rybnik[min^.x+1,min^.y],min^);
                        if min^.y<N then
                                vylepsipole(rybnik[min^.x,min^.y+1],min^);
                end;
        until hotovo;
        close(f1); close(f2);
end.
```

Výsledky krajského kola kategórie A

(Toto je spoločná výsledková listina po koordinácii bodovania, na základe poradia v tejto výsledkovej listine boli najlepší riešitelia pozvaní na celoštátne kolo.)

	Meno	Ročník, škola	1.	2.	3.	4.	Σ
1.	Vladimír Boža	3 Gym. Tatarku Poprad	10	4	7	9	30
	Jozef Jirásek	4 Gym. Zbrojničná Košice	10	3	7	10	30
3.	Ondrej Mikuláš	4 Gym. Haličská Lučenec	7	10	7	0	24
4.	Marcel Ďuriš	4 Gym. P. Horova Michalovce	7	3	3	9	22
	Ján Jerguš	4 Gym. Alejová Košice	4	1	7	10	22
	Tomáš Kočiský	3 Gym. Grösslingová BA	4	1	7	10	22
	Michal Szabados	4 ŠPMND BA	10	2		10	22
8.	Peter Herman	4 Gym. Jura Hronca BA	7	4	7	1	19
	Pavol Vaľkovič	Gym. Zlaté Moravce	10	2	7	0	19
10.	Milan Laslop	Gym. Nedožerského Prievidza	4	5	7	2	18
	Adam Okruhlica	4 Gym. Jura Hronca BA	2	4	2	10	18
12.	Samuel Brezáni	4 Gym. Rajec	8	2	7	0	17
	Peter Hlavatý	3 Gym. Jura Hronca BA	8	2	7		17
	Michal Sudolský	4 Gym. Tajovského B. Bystrica	10		7	0	17
15.	Dominika Fedáková	3 Gym. Stará Ľubovňa	6	4	6	0	16
	Peter Hrinčár	3 Gym. Tatarku Poprad	9	4	3	0	16
	Jakub Končok	3 Gym. Haličská Lučenec	6			10	16
	Peter Ondrúška	3 SPŠ Dubnica nad Váhom	5	3	6	2	16
19.	Michal Danilák	4 Gym. Hubeného BA	7	2	6	0	15
	Lenka Matejovičová	3 Gym. Grösslingová BA	9		4	2	15
21.	Timotej Betina	4 Gym. L. Stöckela Bardejov	7	0	5	2	14
	Juraj Danko	4 Gym. P. de Coubertina PN	6	1	7	0	14
23.	Samuel Hapák	3 Gym. Grösslingová BA	2		9	2	13
	Martin Višňovec	4 Gym. Lettricha Martin	8	1	2	2	13
25.	Balázs Kezes	4 Gym. Nové Zámky	2	2	7	1	12

	Meno	Ročník, škola	1.	2.	3.	4.	Σ
	Matúš Kotry	4 Gym. Párovská Nitra	2	4	6		12
	Tomáš Kovačovský	4 Gym. Jura Hronca BA	2	3	7		12
	Juraj Kušnier	3 Gym. Bánovce nad Bebravou	2	2	6	2	12
	Martin Sucha	3 Gym. Jura Hronca BA			2	10	12
30.	Kristína Čevorová	4 ŠPMND BA	2		7	2	11
	František Hajnovič	4 Gym. Jura Hronca BA	4		7		11
	Alena Košinárová	3 Gym. Grösslingová BA	2	2	7	0	11
	Pavol Otto	3 Gym. sv. Františka Žilina	2		7	2	11
	Martin Podolák	4 Gym. Grösslingová BA	2	2	7	0	11
	Ivan Srba	4 Gym. Liptovský Hrádok	7	2	2		11
	Tomáš Zábojník	4 Gym. Liptovský Hrádok	8		1	2	11
	Lukáš Zdechovan	4 Gym. Tajovského B. Bystrica	8	1	2		11
38.	Pavol Blaho	3 Gym. Jura Hronca BA	9		1		10
	Michal Kučiš	4 Gym. Čadca	1	3	4	2	10
	Kristián Nagy	3 Gym. Jura Hronca BA	2	4	2	2	10
	Peter Nikodem	3 Gym. Školská Spiš. Nová Ves	1		7	2	10
	Michal Petrucha	2 Gym. Metodova BA	2		6	2	10
43.	Daniel Bene	3 Gym. Haličská Lučenec	2		7	0	9
	Ivan Buštor	4 Gym. Jura Hronca BA	8		1		9
	Albert Herencsár	2 Gym. maď. Galanta	2		7		9
46.	Róbert Balent	Gym. Lettricha Martin	2		6		8
	Miroslav Brada	3 Gym. Varšavská Žilina - Vlčince	2	2	2	2	8
	Dušan Ďurech	Gym. 17. novembra Topoľčany	2	4	2		8
	Martin Hanes	3 Gym. P. Horova Michalovce	2	0	6	0	8
	Maroš Kasinec	3 Gym. P. Horova Michalovce	2	0	6	0	8
	Luboš Kuboň	2 Gym. Jura Hronca BA	2	4	2		8
	Andrej Pančík	4 Gym. Tajovského B. Bystrica	5	1	2	0	8
	Lucia Simanová	3 Gym. Grösslingová BA	2	2	2	2	8
	Emília Szabadosová	3 ŠPMND BA	7		1		8
	Tibor Szabó	4 Gym. Šuleka Komárno	2	4	2	0	8
	Danica Zajacová	4 Gym. Jura Hronca BA	2		4	2	8
57.	Jaroslav Böhmer	3 Gym. Školská Spiš. Nová Ves	2	1	4	0	7
	Michal Ferko	4 Gym. Jura Hronca BA	2	3	2	0	7

	Meno	Ročník, škola	1.	2.	3.	4.	Σ
	Marcel Švec	4 Gym. Jura Hronca BA	5		2		7
60.	Peter Korcsok	3 Gym. Mládežnícka Šahy	2	1	2	1	6
	Ladislav Rampášek	4 Gym. Jura Hronca BA	2	2	2		6
	Matúš Zamborský	Gym. Hollého Trnava	2	2	2		6
63.	Jaroslav Brdjar	4 Gym. Alejová Košice	3	0	1	1	5
	Peter Drábik	4 Gym. Čadca	2	1	2		5
	Ján Hreha	4 Gym. Liptovský Hrádok	1		2	2	5
	Marcel Kanta	Gym. Hubeného BA	2	2	1		5
	Ján Percsök	Gym. Daxnerova R. Sobota	1	1	2	1	5
	Adam Saleh	3 Gym. Jura Hronca BA	2		2	1	5
	Jakub Tomaga	4 Gym. P. Horova Michalovce	2	1	2	0	5
	András Varga	3 Gym. H. Selyeho Komárno	2	1	2	0	5
	Radoslav Zajonc	Gym. Športová Nové Mesto n.V.	2	2	1	0	5
	Martin Zruban	Gym. Hollého Trnava	2	1	2	0	5
73.	Róbert Barsa	4 Gym. Poštová Košice	2	0	2	0	4
	Katarína Gajdziková	4 Gym. Daxnera V.n. Topľou	2	0	2	0	4
	Miroslav Piter	Gym. Svidník	2	0	2	0	4
	Michal Pristáš	3 Gym. Daxnera V.n. Topľou	2	0	2	0	4
	Branislav Uljan	4 Gym. Poštová Košice	1	0	1	2	4
78.	Tomáš Benedikti	3 Gym. P. Horova Michalovce	1	0	2	0	3
	Jakub Bokšanský	4 SPŠ Spišská Nová Ves	1	0	2	0	3
	Filip Hanes	4 Gym. Tajovského B. Bystrica	1		2	0	3
	Peter Vida	Gym. Nedožerského Prievidza	3				3
82.	Erik Tilňák	4 Gym. Humenné	0	0	0	1	1
83.	Michal Klein	3 Gym. Školská Spiš. Nová Ves	0	0	0	0	0

Výsledky celoštátneho kola kategórie A

	Meno	Ročník, škola	1.	2.	3.	4.	5.	Σ
1.	Michal Danilák	4 G. Hubeného BA	7	9	10	15	13	54
2.	Vladimír Boža	3 G. Tatarku Poprad	6	9	7	15	9	46
3.	Samuel Hapák	3 G. Grösslingová BA	7	8	4	15	9	43
4.	Peter Herman	4 G. Jura Hronca BA	4	5	7	15	6	37
5.	Jozef Jirásek	4 G. Zbrojničná Košice	6	7	10	4	8	35
6.	Ondrej Mikuláš	4 G. Haličská Lučenec	6	6	4	15	0	31
7.	Peter Ondrúška	3 SPŠ Dubnica nad Váhom	6	8	10	1	4	29
8.	Ján Jerguš	4 G. Alejová Košice	3	1	7	0	11	22
9.	Michal Szabados	4 ŠPMND BA	4	9	7	1	0	21
10.	Peter Hlavatý	3 G. Jura Hronca BA	2	4	1	–	9	16
	Adam Okruhlica	4 G. Jura Hronca BA	7	8	1	–	0	16
12.	Tomáš Kočiský	3 G. Grösslingová BA	5	7	1	0	0	13
	Milan Laslop	G. Nedožerského Prievidza	1	6	2	3	1	13
	Balázs Kezes	4 G. Nové Zámky	6	3	1	0	3	13
15.	Peter Hrinčár	3 G. Tatarku Poprad	5	5	1	1	0	12
	Marcel Ďuriš	4 G. P. Horova Michalovce	4	6	1	1	0	12
17.	Matúš Kotry	4 G. Párovská Nitra	2	6	0	3	–	11
18.	Juraj Kušnier	3 G. Bánovce nad Bebravou	0	6	2	2	–	10
	Tomáš Kovačovský	4 G. Jura Hronca BA	1	6	0	3	0	10
	Michal Sudolský	4 G. Tajovského BB	0	6	2	2	0	10
21.	Juraj Danko	4 G. P. de Coubertina PN	0	6	2	0	0	8
	Kristína Čevorová	4 ŠPMND BA	0	6	2	0	–	8
	Alena Košinárová	3 G. Grösslingová BA	3	4	1	0	0	8
	Martin Sucha	3 G. Jura Hronca BA	0	6	0	1	1	8
25.	František Hajnovič	4 G. Jura Hronca BA	6	0	1	0	–	7
	Lenka Matejovičová	3 G. Grösslingová BA	2	3	2	0	0	7
27.	Dominika Fedáková	3 G. Stará Ľubovňa	0	5	1	0	0	6
	Jakub Končok	3 G. Haličská Lučenec	0	3	0	3	–	6
29.	Martin Višňovec	4 G. Lettricha Martin	0	3	1	0	–	4

Výsledky Česko-Poľsko-Slovenského prípravného stretnutia CPSPC 2007

	Meno (štát)	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	Σ
1.	Tomasz Kulczynski (pl)	100	0	27	100	–	66	100	–	100	0	100	100	693
2.	Jaroslaw Gomulka (pl)	100	30	–	80	–	5	40	20	40	0	100	60	475
3.	Piotr Niedzwiedz (pl)	30	–	–	60	–	1	100	10	100	0	50	100	451
4.	Vladimír Boža (sk)	40	20	36	10	30	38	40	0	40	0	100	60	414
5.	Maciej Klimek (pl)	–	50	–	60	–	16	40	–	40	0	100	100	406
6.	Jakub Kallas (pl)	100	–	–	100	–	34	–	–	–	–	100	35	369
7.	Miroslav Klimoš (cz)	0	40	0	0	95	4	40	0	30	0	40	70	319
8.	Jozef Jirásek (sk)	–	–	–	10	–	53	40	40	40	0	100	10	293
9.	Peter Ondrúška (sk)	–	–	–	–	–	53	0	0	40	0	100	75	268
10.	Michal Danilák (sk)	–	0	27	0	–	58	40	0	30	–	100	–	255
11.	Pavel Klavík (cz)	0	10	54	–	0	4	40	–	10	0	100	0	218
12.	Marcin Kurczych (pl)	–	0	9	–	–	–	–	100	0	–	100	–	209
13.	Lukáš Lánský (cz)	60	–	18	–	0	69	20	0	40	0	0	0	207
14.	Josef Pihera (cz)	70	0	0	10	0	22	20	20	0	0	50	–	192
15.	Alena Košinárová (sk)	–	30	0	0	–	23	30	–	20	–	50	–	153
16.	Roman Smrž (cz)	0	0	0	–	–	8	40	0	–	0	100	0	148
17.	Libor Plucnar (cz)	–	10	–	–	–	20	40	–	10	–	50	–	130
18.	Peter Hlavatý (sk)	–	–	0	–	–	34	20	0	–	–	–	–	54

Výsledky Medzinárodnej olympiády v informatike, IOI 2007

V roku 2007 sa Medzinárodná olympiáda v informatike (IOI) konala v dňoch 15. až 22. augusta v chorvátskom Záhrebe. Súťaže sa zúčastnilo 285 súťažiacich z takmer 80 krajín celého sveta.

Ako lídri zastupujúci našu krajinu pri hlasovaniach sa tejto IOI zúčastnili RNDr. Andrej Blaho a Juliana Lipková, obaja z FMFI UK v Bratislave. Ako člen medzinárodnej odbornej komisie (ISC) sa tejto IOI zúčastnil Mgr. Michal Forišek z FMFI UK v Bratislave.

Našu krajinu tento rok reprezentovala táto štvorica stredoškolákov: Vladimír Boža z Gymnázia D. Tatarku v Poprade, Michal Danilák z Gymnázia Hubeného v Bratislave, Peter Ondrúška z SPŠ v Dubnici nad Váhom a Jozef Jirásek z Gymnázia Zbrojničná v Košiciach. Výsledky našich súťažiacich uvádzame zhrnuté v tabuľke.

	Meno	Počet bodov
31.	Michal Danilák	366 – striebro
57.	Vladimír Boža	311 – striebro
78.	Peter Ondrúška	275 – bronz
	Jozef Jirásek	

Výsledky Stredoeurópskej olympiády v informatike, CEOI 2007

V roku 2007 sa Stredoeurópska olympiáda v informatike (CEOI) konala v dňoch 1. až 7. júla v neďalekom Brne, ČR. Súťaže sa zúčastnili tímy zo siedmich krajín: Českej republiky, Chorvátska, Maďarska, Nemecka, Poľska, Rumunska a Slovenska.

Ako lídri zastupujúci našu krajinu pri hlasovaniach sa tejto CEOI zúčastnili doc. RNDr. Gabriela Andrejková, CSc. z ÚI PF UPJŠ v Košiciach a Peter Perešíni z FMFI UK v Bratislave.

Našu krajinu tento rok reprezentovala tá istá štvorica stredoškolákov ako na IOI: Vladimír Boža z Gymnázia D. Tatarku v Poprade, Michal Danilák z Gymnázia Hubeného v Bratislave, Peter Ondrúška z SPŠ v Dubnici nad Váhom a Jozef Jirásek z Gymnázia Zbrojničná v Košiciach. Výsledky našich súťažiacich uvádzame zhrnuté v tabuľke.

	Meno	Počet bodov
12.	Vladimír Boža	338 – bronz
17.	Jozef Jirásek	246
20.	Peter Ondrúška	220
26.	Michal Danilák	177

RNDr. Michal Forišek, Bc. Jakub Kováč, Slovenská komisia OI
Dvadsiaty druhý ročník Olympiády v informatike
Vydal: Iuventa, Bratislava, 2008
119 strán, náklad 300 výtlačkov
Neprešlo jazykovou úpravou
ISBN 80-8072-053-3