# Dvadsiaty šiesty ročník Olympiády v informatike

**SLOVENSKÁ INFORMATICKÁ SPOLOČNOSŤ**

Odborným garantom súťaže Olympiáda v informatike je Slovenská informatická spoločnosť. Viac sa o nej dozviete na stránke `http://www.informatika.sk/`

RNDr. Michal Forišek PhD., Slovenská komisia OI
Dvadsiaty šiesty ročník Olympiády v informatike
ISBN 978-80-8072-117-6

# Obsah

# O priebehu 26. ročníka Olympiády v informatike

V školskom roku 2010/11 prebehol už dvadsiaty šiesty ročník Olympiády v informatike (OI).

Do kategórie B, určenej pre prvákov a druhákov klasických štvorročných stredných škôl, sa zapojilo 52 žiakov. Z nich sa 45 zúčastnilo krajského kola, ktorým v kategórii B súťaž končí.

Do ťažšej kategórie A, určenej pre všetkých stredoškolákov bez obmedzenia, sa zapojilo 94 žiakov. Najlepších 58 z nich postúpilo do krajského kola a odtiaľ 30 najlepších do celoštátneho kola OI. Traja z postupujúcich sa bohužiaľ celoštátneho kola zúčastniť nemohli, preto toto kolo malo nakoniec len 27 účastníkov. Celoštátne kolo prebehlo v Drienici, okres Sabinov (teoretická časť) a v Prešove (praktická časť).

Najlepší riešitelia kategórie A boli následne pozvaní na týždňové výberové sústredenie, kde sa určila reprezentácia Slovenska na medzinárodných súťažiach – Stredoeurópskej olympiáde v informatike (CEOI) a Medzinárodnej olympiáde v informatike (IOI).

Na celoslovenskej úrovni sa počas celého ročníka o plynulý chod OI starala Slovenská komisia OI v nasledujúcom zložení:

- doc. RNDr. Gabriela Andrejková, CSc.,
  predsedkyňa, ÚI PF UPJŠ, Košice
- RNDr. Michal Forišek, PhD., podpredseda, KI FMFI UK, Bratislava
- RNDr. Andrej Blaho, PhD., FMFI UK, Bratislava
- Mgr. Ján Katrenič, PF UPJŠ, Košice
- Mgr. Juliana Šišková, FMFI UK, Bratislava
- PaedDr. Ivan Brodenec, KI FPV UMB, krajský predseda pre BB
- RNDr. Eva Hanulová, Gym. J. Hronca, krajská predsedkyňa pre BA
- RNDr. Rastislav Krivoš-Belluš, PhD.,
  Ústav informatiky PF UPJŠ, krajský predseda pre KE
- prof. Ing. Veronika Stoffová, CSc.,
  KI PF UJS, krajská predsedkyňa pre NR
- Mgr. Mária Majherová, PhD.,
  Katedra matematiky FHPV PU, krajská predsedkyňa pre PO

- Ing. Andrea Julény, KI FMech TnUAD, krajská predsedkyňa pre TN
- doc. RNDr. Mária Lucká, CSc.,
  KMI PdF TU, krajská predsedkyňa pre TT
- RNDr. Peter Varša, PhD., KI FRI ŽU, krajský predseda pre ZA

Zástupcom IUVENTY zabezpečujúcim organizačnú stránku súťaže bol Ing. Tomáš Lučenič, v priebehu ročníka časť jeho povinností prevzal PhDr. Peter Barát.

Špeciálne poďakovanie patrí celému kolektívu organizátorov Korešpondenčného seminára z programovania (KSP). To, že naša malá krajina má stále šancu vychovať súťažiacich na svetovej úrovni, je z veľkej časti aj ich zásluhou. Mnohí z organizátorov KSP taktiež prispeli priamo k chodu olympiády – či už spisovaním zadaní a vzorových riešení, koordináciou krajského kola, alebo organizáciou odbornej aj technickej stránky celoštátneho kola. Bez tejto komunity nadaných mladých ľudí by bola organizácia kvalitných programátorských súťaží na Slovensku omnoho náročnejšia, ak nie nemožná.

Michal Forišek, podpredseda SK OI

# Zadania domáceho kola kategórie A

## A-I-1 Indiana a poklad

Po dlhej a nebezpečnej ceste plnej dobrodružstiev sa pred Indianom konečne rozprestrel pohľad na starodávne pohrebisko aztéckych kráľov. Ležal tam hrob pri hrobe pozdĺž dlhej cesty. Všetky boli navlas rovnaké, líšili sa len výškou náhrobných kameňov. Pri prvom kameni bol nasledujúci nápis:

*Ak múdry si, hneď pochopíš, kde poklad môj má svoju skrýš.*
*Nepomôže však nič po tom čo zastaneš nad zlým hrobom!*

Výklad týchto veršov je jednoduchý: ak otvoríte hrob s pokladom, tak tento poklad získate; v opačnom prípade nezískate nič a pravdepodobne vás niečo rozmliaždi. „To nie je príliš lákavá perspektíva," pomyslel si Indiana. V tej chvíli si však spomenul na starý nápis, ktorý kedysi videl v aztéckom múzeu. A zrazu bolo Indianovi jasné, kde sa poklad nachádza.

*Múdry je ten, kto volí z prostredných, ak nakoniec najväčší vyberie.*

**Súťažná úloha:**

Nech $K$ je nepárne prirodzené číslo. *Medián* postupnosti dĺžky $K$ je jej prostredný prvok podľa veľkosti. Teda keby sme danú postupnosť utriedili, medián by bol ten prvok, ktorý by skončil na pozícii číslo $(K+1)/2$.

Daná je postupnosť $N$ celých kladných čísel $v_1, v_2, \ldots, v_N$ a nepárne celé číslo $K$. Vašou úlohou je nájsť tú súvislú $K$-prvkovú podpostupnosť tejto postupnosti, ktorej medián je najväčší, a vypísať veľkosť tohto mediánu.

Napríklad pre postupnosť výšok hrobov $(4, 5, 1, 3, 2, 4)$ a $K = 3$ máme na výber štyri súvislé podpostupnosti: $(4, 5, 1)$, $(5, 1, 3)$, $(1, 3, 2)$ a $(3, 2, 4)$. Ich mediány sú 4, 3, 2 a 3. Hľadáme najväčší z nich, teda číslo 4.

**Formát vstupu:**

Prvý riadok vstupu obsahuje dve celé čísla $N$ a $K$ – počet hrobov a dĺžku uvažovaných podpostupností. Platí $1 \leq K \leq N \leq 1\,000\,000$ a tiež $K \leq 10\,000$. Každý z nasledujúcich $N$ riadkov obsahuje výšku jedného hrobu (v poradí $v_1, v_2, \ldots, v_N$). Výšky sú prirodzené čísla menšie ako $1\,000\,000\,000$.

Vo vstupoch, za ktoré budú dokopy 4 body, bude $K < 100$.

**Formát výstupu:**

Vypíšte jeden riadok a v ňom jedno číslo, ktoré je rovné maximu všetkých mediánov súvislých úsekov dĺžky $K$ v postupnosti čísel zadaných na vstupe.

**Príklad:**

| Vstup | Výstup |
|---|---|
| 6 3<br>4<br>5<br>1<br>3<br>2<br>4 | 4 |

## A-I-2 Ešte raz poklad

Potom, čo Indiana našiel najväčší z prostredných hrobov, zistil, že nápis nebol úplne presný. Namiesto pokladu bol totiž pod kameňom len vchod do ďalšej chodby. Tá bola vydláždená štvorcovými dlaždicami a na stene chodby bol vyrytý tento nápis:

*Návštevník, čo nie je hlúpy,*
*práve raz na každú stúpi,*
*nie však ak je označená,*
*lebo lebka smrť znamená!*

Chodba mala na šírku presne 3 dlaždice a dlhá bola kam až oko dovidelo. Na niektorých dlaždiciach boli namaľované lebky. (Pár z týchto dlaždíc zdobili aj skutočné lebky predchádzajúcich objaviteľov tajnej chodby.)

Indiana veľmi rýchlo nápis pochopil – musí na každú dlaždicu, okrem tých zakázaných, šliapnuť práve raz, lebo len tak sa živý dostane k pokladu. V tom momente ale začal ľutovať, že má tak veľké nohy. Nech sa snažil, ako chcel, nikdy sa mu nepodarilo stúpiť len na jednu dlaždicu, ale vždy stúpil na dve susedné. To mu jeho úlohu samozrejme značne skomplikovalo.

**Súťažná úloha:**

Daná je chodba dĺžky $N$, tvorí ju $3 \times N$ dlaždíc. Z týchto dlaždíc je $K$ zakázaných, na tie Indiana nesmie stúpiť. Vašou úlohou je určiť, koľkými spôsobmi

je možné pokryť túto chodbu stopami veľkosti $1 \times 2$ dlaždice tak, aby každá nezakázaná dlaždica bola pokrytá práve raz a žiadna zakázaná dlaždica nebola pokrytá. Keďže výsledné číslo môže byť veľmi veľké, stačí spočítať zvyšok, ktorý dáva po delení zadaným číslom $L$.

**Formát vstupu:**

V prvom riadku sú zadané prirodzené čísla $N$, $K$ a $L$. Môžete predpokladať, že platí $1 \leq N \leq 2\,500\,000$, $0 \leq K \leq \min(3N-1, 1\,000\,000)$ a $1 \leq L \leq 1\,000\,000$.

Nasleduje $K$ riadkov, v $i$-tom z nich je dvojica čísel $r_i$, $s_i$: súradnice $i$-tej zakázanej dlaždice. Pre každé $i$ platí $1 \leq r_i \leq 3$ a $1 \leq s_i \leq N$ a navyše platí aj $s_1 \leq s_2 \leq \cdots \leq s_K$.

V dvoch sadách testovacích vstupov bude platiť $N \leq 20$.
V ďalších dvoch sadách testovacích vstupov bude platiť, že počet dláždení neprekročí 500 000.
V ďalších dvoch sadách testovacích vstupov bude platiť $K = 0$.

**Formát výstupu:**

Nech $D$ je počet spôsobov ako chodbu pokryť. Vypíšte jediný riadok obsahujúci hodnotu $D$ mod $L$ (t. j. zvyšok, ktorý dáva $D$ po delení $L$). Všimnite si, že pre niektoré vstupy nemusí existovať žiadne riešenie, v takom prípade vypíšte nulu.

**Príklad:**

**Vstup**

```
5 3 13
3 1
1 2
2 4
```

**Výstup**

```
3
```

*Pre tento vstup existujú presne 3 riešenia.*

## A-I-3 Rieka

Za siedmimi horami sa nachádza rozsiahly prales, ktorým preteká dlhá rieka. Napriek svojej dĺžke nemá žiadne prítoky a jej tok sa nikde nerozvetvuje. V tomto pralese rastie veľa vzácnych druhov stromov, preto niet divu, že pri rieke ležia drevorubačské tábory. Vždy, keď drevorubači zotnú dosť stromov, postavia z nich plť a pošlú ju dole riekou.

Okrem drevorubačských táborov sa pri rieke nachádzajú aj píly. Zamestnanci každej píly číhajú pri rieke a keď zbadajú, že sa po nej plaví plť, vylovia ju a všetko drevo spotrebujú. Občas je píla zastavená kvôli údržbe, počas ktorej ignoruje plte a necháva ich plávať ďalej po rieke.

Zamestnancom píl prekáža, že nevedia, kedy očakávať dodávku dreva a tak zbytočne márnia čas pozorovaním rieky. Pomôžte im a napíšte program, ktorý im bude posielať upozornenia na blížiace sa zásielky dreva.

**Súťažná úloha:**

Rieka má dĺžku $N$ kilometrov. Pozície bodov na rieke budú označené ich vzdialenosťou od prameňa. V každom z bodov $1, \ldots, N$ sa nachádza buď drevorubačský tábor alebo píla. Umiestnenie píl na rieke je zadané na začiatku behu programu. Môžete predpokladať, že počet píl $P$ je rádovo rovnaký ako dĺžka rieky $N$.

Váš program bude dostávať udalosti nasledujúcich typov: „píla v bode $b$ začína údržbu“, „píla v bode $b$ je opäť v prevádzke“ a „z tábora v bode $b$ bola vyslaná plť“. Pre každú vyslanú plť váš program povie, v ktorej píle bude spracovaná. Môžete predpokladať, že rieka tečie naozaj rýchlo – medzi vyslaním plte a jej prijatím nestihnú v žiadnej píle spustiť ani ukončiť údržbu.

**Poznámka:** Existuje riešenie, ktoré na spracovanie každej udalosti potrebuje rádovo menej ako $\log N$ krokov.

**Formát vstupu:**

Prvý riadok vstupu obsahuje tri prirodzené čísla $N, P$ a $M$ udávajúce dĺžku rieky, počet píl na nej a počet udalostí, ktoré nastanú ($1 \leq N \leq 100\,000, 1 \leq P \leq N$ a $1 \leq M \leq 1\,000\,000$). Druhý riadok obsahuje čísla $n_1, \ldots, n_P$ udávajúce, na ktorých kilometroch ležia píly. Môžete predpokladať, že $1 \leq n_1 \leq \ldots \leq n_P \leq N$.

Zvyšných $M$ riadkov popisuje udalosti v chronologickom poradí. Popis udalosti tvorí jedno z písmen `Z`, `K` a `V` (Začiatok údržby, Koniec údržby, Vyslaná plť), nasledované kilometrom, na ktorom táto udalosť nastala.

**Formát výstupu:**

Pre každú vyslanú plť vypíšte jeden riadok a v ňom jedno celé číslo: vzdialenosť píly, ktorá spracuje túto plť, od prameňa rieky. Ak plť na rieke nezachytí žiadna píla, vypíšte 0.

**Príklad:**

**Vstup**

```
6 3 9
2 4 6
V 1
V 3
V 5
Z 2
V 1
K 2
V 1
Z 6
V 5
```

**Výstup**

```
2
4
6
4
2
0
```

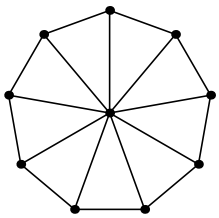
*V poradí štvrtú plť zachytila až píla na 4. km, lebo na píle na 2. km vtedy prebiehala údržba.*
*Posledná plť plávala len okolo poslednej píly a tá ju kvôli prebiehajúcej údržbe nezachytila.*

## A-I-4 Grafový počítač

V tomto ročníku olympiády sa budeme v teoretickej úlohe stretávať so špeciálnym grafovým počítačom. V študijnom texte uvedenom za zadaním tejto úlohy je popísané, ako tento počítač funguje.

**Súťažná úloha:**

a) (5 bodov) Ruské kolo rádu $n$ je graf, ktorý vznikne tak, že zoberieme kružnicu tvorenú $n$ vrcholmi a pridáme jeden nový vrchol („stred kolesa“), ktorý bude spojený so všetkými $n$ vrcholmi kružnice. Ruské kolo rádu $n$ má teda $n + 1$ vrcholov a $2n$ hrán.



Napíšte funkciu pre grafový počítač, ktorá pre zadané $n$ zostrojí ruské kolo rádu $n$. Na obrázku vpravo vidíte príklad pre $n = 9$.

b) (5 bodov) Majme graf opisujúci cestnú sieť: vrcholy sú mestá, hrany cesty ohodnotené nezápornými vzdialenosťami v kilometroch. Cesty sú prepojené len v mestách, všetky ostatné križovatky sú mimoúrovňové. Zaujíma nás, akú najmenšiu vzdialenosť musíme prejsť, aby sme sa dostali z mesta $x$ do mesta $y$. Inými slovami, máme v danom grafe nájsť cestu medzi $x$ a $y$, pre ktorú bude celkový súčet váh použitých hrán najmenší možný.

Napíšte funkciu pre grafový počítač, ktorá dostane na vstupe graf cestnej siete a identifikátory dvoch rôznych vrcholov $x$, $y$ a vráti vzdialenosť medzi týmito vrcholmi.

## Tabuľka inštrukcii

| **príkaz** | **účinok príkazu** |
|---|---|
| `AddV(G,z)` | Pridá nový vrchol so značkou `z` a najväčším id. |
| `DelV(G,id)` | Zmaže vrchol s identifikátorom `id`. |
| `AddE(G,x,y,w)` | Pridá hranu medzi `x` a `y` s váhou `w`. |
| `DelE(G,x,y)` | Zmaže hranu medzi `x` a `y`. |
| `TestE(G,x,y)` | Zistí, či sú `x` a `y` spojené hranou. |
| `GetV(G,id)` | Vráti značku daného vrcholu. |
| `SetV(G,id,z)` | Nastaví značku daného vrcholu na danú hodnotu. |
| `SetAllV(G,z)` | Nastaví každému vrcholu v `G` značku na `z`. |
| `ReplaceV(G,zold,znew)` | Každému vrcholu v `G`, ktorý mal doteraz značku `zold`, dá značku `znew`. |
| `GetE(G,x,y)` | Vráti váhu danej hrany. |
| `SetE(G,x,y,w)` | Nastaví váhu danej hrany na danú hodnotu (aj vyrobí takú hranu, ak treba). |
| `SetAllE(G,w)` | Nastaví každej hrane v `G` váhu na `w`. |
| `ReplaceE(G,wold,wnew)` | Každej hrane v `G`, ktorá mala doteraz váhu `wold`, dá váhu `wnew`. |
| `CountV(G)` | Vráti počet vrcholov v grafe. |
| `SumV(G)` | Vráti súčet značiek všetkých vrcholov, pričom `undef` sa počíta ako 0. |
| `CountE(G)` | Vráti počet hrán v grafe. |
| `SumE(G)` | Vráti súčet váh všetkých hrán, pričom `undef` sa počíta ako 0. |
| `Iso(G,H,veq,eeq)` | Zistí, či sú grafy `G` a `H` izomorfné pri danom porovnávaní vrcholov a hrán. |
| `Find(G,H,veq,eeq)` | Nájde jeden najľahší podgraf grafu `G`, ktorý je izomorfný s `H` pri danom porovnávaní vrcholov a hrán. Vráti `EmptyG` ak taký podgraf neexistuje. |

| **príkaz** | **účinok príkazu** |
|---|---|
| `Common(G,H,veq,eeq)` | Nájde jeden spoločný podgraf grafov `G` a `H` s najviac vrcholmi (a ak sa nevie rozhodnúť, tak aj s najviac hranami). |
| `Join(G,H,veq,eeq)` | Spojí grafy `G` a `H` do jedného, pričom ich akoby zlepí za `Common(G,H,veq,eeq)`. |

| **konštanta** | **hodnota / význam** |
|---|---|
| `EmptyG` | Prázdny graf (s 0 vrcholmi a 0 hranami). |
| `undef` | Špeciálna hodnota používaná ako značka vrcholu alebo váha hrany. |
| `any` | Ľubovoľné dva vrcholy/hrany sa rovnajú (na ohodnotenie sa nehľadí). |
| `value` | Zodpovedajúce si vrcholy/hrany musia mať rovnaké ohodnotenie. Hodnotu `undef` považujeme za „žolíka“, ktorý sa rovná ľubovoľnej hodnote. |
| `value_strict` | Vrcholy/hrany musia mať presne rovnaké ohodnotenie, `undef` sa rovná len `undef`-u. |
| `value_defined` | Vrcholy/hrany musia mať rovnaké ohodnotenie, `undef` sa nerovná ničomu, ani `undef`-u. |
| `id` | Vrcholy musia mať rovnaké id. (Pri hranách sa toto nedá použiť.) |
| `none` | Žiadne dva vrcholy/hrany nie sú identické. |

## Študijný text

Bežné počítače, ktoré máme všetci doma, všetko počítajú v číslach v dvojkovej sústave. Mnohým ľuďom to vyhovuje. Nie však železničným inžinierom v Tasmánii. Tí si jedného dňa uvedomili, že väčšina problémov, ktoré oni potrebujú riešiť, sa týka grafov. Preto si pre vlastnú potrebu navrhli *grafový počítač*, ktorý namiesto čísel pracuje priamo s grafmi. Vie s nimi robiť všetky bežné operácie, a to dokonca v konštantnom čase.

Formálne, *graf* je usporiadaná dvojica $(V, E)$, kde $V$ je konečná množina objektov, ktoré voláme *vrcholy*, a $E$ je konečná množina obsahujúca niektoré neusporiadané dvojice vrcholov. Prvky $E$ voláme *hrany*. Neformálne si graf môžeme predstaviť ako železničnú sieť. Vrcholy sú jednotlivé zastávky, hrany sú

priame úseky železničnej trate medzi dvojicami zastávok.

Grafy niekedy môžu byť *ohodnotené*: každému vrcholu, resp. každej hrane môžeme priradiť nejaké číslo. Ich význam môže byť v rôznych situáciách rôzny: raz to môže byť dĺžka koľajníc, inokedy cena cestovného lístka, a ešte inokedy môžeme pomocou čísel reprezentovať vlastnosti objektov: napríklad stanice, kde stoja rýchliky, môžu mať priradené číslo 1 a ostatné stanice číslo 2. Upresnime ešte, že naše grafy nebudú nikdy obsahovať násobné hrany ani slučky. Teda medzi každými dvomi vrcholmi bude viesť nanajvýš jedna hrana a žiadna hrana nebude začínať a končiť v tom istom vrchole.

**Reprezentácia grafu:**

Každý graf uložený v grafovom počítači má vrcholy očíslované prirodzenými číslami od 1 do ich počtu. Týmto číslam budeme hovoriť *identifikátory* vrcholov (skrátene: *id*).

Hranám identifikátory netreba, každá hrana je totiž jednoznačne určená pomocou id vrcholov, ktoré spája.

Každý vrchol má priradené svoje ohodnotenie, ktoré budeme volať *značka vrcholu*. Každá hrana má priradené svoje ohodnotenie, ktoré budeme volať *váha hrany*. Každé ohodnotenie je buď nezáporné celé číslo alebo špeciálna hodnota `undef` (nedefinované).

Programy budeme písať v bežnom programovacom jazyku, ktorý len rozšírime o niekoľko nových typov premenných a niekoľko nových funkcií. V tomto študijnom texte použijeme Pascal, ale pokojne môžete písať riešenia aj v iných jazykoch, do ktorých si nové súčasti doplníte analogicky.

**Nové typy premenných a konštánt:**

- Typ `Graph` slúži na uloženie grafu. Do premennej typu `Graph` teda môžeme uložiť jeden graf, je jedno ako veľký. Premenné typu `Graph` vieme priradzovať a vieme testovať rovnosť obsahu dvoch takýchto premenných. Obe tieto operácie počítač vykoná v konštantnom čase.
- Konštanta `EmptyG` typu `Graph` obsahuje prázdny graf, teda graf s 0 vrcholmi (a teda nutne aj 0 hranami).
- Typ `Value` slúži na uloženie jedného ohodnotenia. Premenná typu `Value` teda môže nadobudnúť ako hodnotu ľubovoľné nezáporné celé číslo alebo špeciálnu hodnotu `undef`. Až na `undef` fungujú operácie s týmto typom rovnako ako operácie s bežnými celočíselnými typmi premenných.

**Operácie meniace štruktúru grafu:**

- Funkcia `AddV(G,z)` pridá do grafu `G` nový vrchol a priradí mu značku `z`. Do nového vrcholu nevedú žiadne hrany a jeho id je rovné novému počtu vrcholov. Návratová hodnota funkcie `AddV(G,z)` je toto id.
- Procedúra `DelV(G,id)` zmaže z grafu `G` vrchol s identifikátorom `id` a všetky hrany, ktoré do tohto vrcholu viedli. Následne poposúva id vrcholov tak, aby nevznikla diera v ich číslovaní. Teda z vrcholu s číslom `id`+1 sa stane `id`, z `id`+2 sa stane `id`+1, atď.
- Procedúra `AddE(G,x,y,w)` vytvorí v grafe `G` hranu medzi vrcholmi s id `x` a `y` a priradí jej ohodnotenie `w`.
- Procedúra `DelE(G,x,y)` odstráni z grafu `G` hranu medzi vrcholmi s id `x` a `y`.
- Funkcia `TestE(G,x,y)` vráti `true` ak sú vrcholy s id `x` a `y` v grafe `G` spojené hranou a `false` ak nie sú.

Všetky tieto funkcie a procedúry bežia v konštantnom čase. Je nutné ich volať so správnymi parametrami, inak nastane chyba a program bude ukončený. Napr. procedúru `DelE` je povolené volať len s id vrcholov, ktoré skutočne sú v danom grafe spojené hranou.

**Operácie meniace značky vrcholov a váhy hrán:**

- Funkcia `GetV(G,id)` vráti značku zadaného vrcholu.
- Procedúra `SetV(G,id,z)` nastaví značku zadaného vrcholu na zadanú hodnotu.
- Procedúra `SetAllV(G,z)` nastaví každému vrcholu v `G` značku na `z`.
- Procedúra `ReplaceV(G,zold,znew)` každému vrcholu v `G`, ktorý mal doteraz značku `zold`, zmení značku na `znew`.
- Funkcia `GetE(G,x,y)` a procedúry `SetE(G,x,y,w)`, `SetAllE(G,w)` a tiež `ReplaceE(G,wold,wnew)` sú definované rovnako ako funkcie pracujúce s vrcholmi. Hrana je popísaná id vrcholov `x` a `y`, ktoré spája.

  Ak pri volaní `SetE(G,x,y,w)` hrana medzi `x` a `y` neexistovala, tak je najskôr do grafu pridaná.

**Štatistické funkcie:**

- Funkcia `CountV(G)` vráti počet vrcholov v grafe.

- Funkcia `SumV(G)` vráti súčet značiek všetkých vrcholov, pričom `undef` sa počíta ako 0. Pokiaľ graf nemá žiadne vrcholy, funkcia vráti 0.
- Analogicky definujeme funkcie `CountE(G)` a `SumE(G)` pre hrany a ich váhy.

**Globálne operácie:**

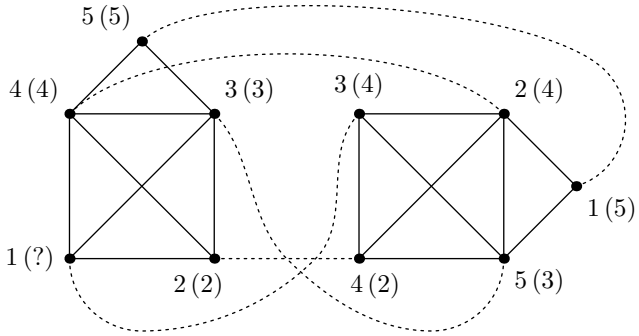
- Funkcia `Iso(G,H,veq,eeq)` zistí, či sú grafy `G` a `H` *izomorfné* – t. j., či možno jednému z grafov prečíslovať vrcholy tak, aby sa zhodoval s druhým grafom. Dva grafy sú zhodné, pokiaľ majú rovnaké množiny vrcholov aj hrán; navyše sa im musia zhodovať značky vrcholov a váhy hrán podľa toho, ako určujú parametre `veq` (pre vrcholy) a `eeq` (pre hrany). Tieto parametre môžu nadobúdať nasledujúce hodnoty:

  `any`: Ľubovoľné dva vrcholy/hrany sa rovnajú (na ohodnotenie sa nehľadí).
  `value`: Zodpovedajúce si vrcholy/hrany musia mať rovnaké ohodnotenie. Hodnotu `undef` ale považujeme za „žolíka“, ktorý sa rovná ľubovoľnej hodnote.
  `value_strict`: Vrcholy/hrany musia mať presne rovnaké ohodnotenie, `undef` sa rovná len `undef`-u.
  `value_defined`: Vrcholy/hrany musia mať rovnaké ohodnotenie, `undef` sa nerovná ničomu, ani `undef`-u. Teda akonáhle má nejaký vrchol/hrana ohodnotenie `undef`, nemôžeme ho priradiť žiadnemu vrcholu/hrane v druhom grafe.
  `id`: Vrcholy musia mať rovnaké id (toto možno aplikovať len na vrcholy, lebo hrany nemajú id). Inými slovami, zakazujeme prečíslovávať vrcholy, ale na ich ohodnotenie nehľadíme.
  `none`: Žiadne dva vrcholy/hrany nie sú identické. (I keď to vyzerá neužitočne, túto možnosť použijeme v ďalších funkciách.)

  Izomorfizmus je znázornený na nasledujúcom obrázku. Plné čiary predstavujú hrany dvoch 5-vrcholových grafov. Čísla pred zátvorkami sú id vrcholov, v zátvorkách sú ich značky (otáznik označuje hodnotu `undef`). Všetky hrany majú váhu `undef`.

  Volaním `Iso(G,H,any,any)` zistíme, či vieme zmeniť id vrcholov grafu `G` tak, aby sme dostali v `G` presne tú istú množinu vrcholov ako v `H`. Takýchto prečíslovaní môže vo všeobecnosti byť aj viac. V príklade na obrázku existujú dve možnosti, pre jednu z nich je priradenie vrcholov zobrazené čiarkovanými čiarami.

Ak navyše požadujeme, aby sa zhodovali aj značky vrcholov, resp. váhy hrán, dosiahneme to zmenením `veq`, resp. `eeq`, na inú hodnotu ako `any`. Grafy z nášho obrázku budú izomorfné, ak nastavíme `veq` na `value` alebo `any` a zároveň `eeq` na `any`, `value` alebo `value_strict`. (Čiarkované čiary predstavujú priradenie vrcholov, ktoré vo všetkých týchto prípadoch funguje.)



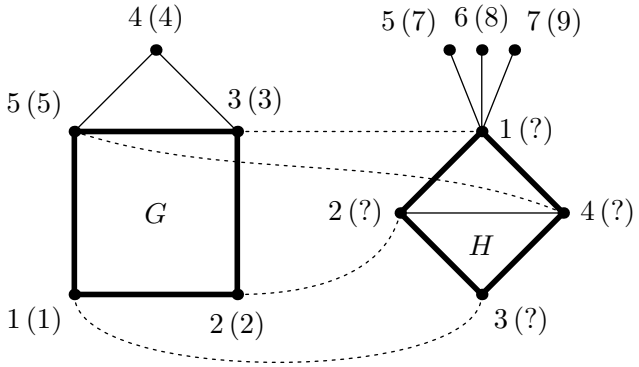

Pri ostatných kombináciách `veq` a `eeq` tieto dva grafy izomorfné nie sú.

- Funkcia `Find(G,H,veq,eeq)` nájde podgraf grafu `G`, ktorý je izomorfný s grafom `H`. (Podgraf je taký graf, ktorý možno získať z `G` odstránením niektorých vrcholov a hrán.) Návratovou hodnotou funkcie bude tento podgraf, pričom vrcholy budú očíslované podľa grafu `H` a ohodnotenie vrcholov a hrán bude pochádzať z grafu `G`. Pokiaľ hľadaný podgraf neexistuje, funkcia vráti `EmptyG`. Parametre `veq` a `eeq` určujú rovnako ako pri funkcii `Iso`, ako sa správa izomorfizmus.

  Pokiaľ existuje viacej izomorfných podgrafov, funkcia `Find` nájde najľahší z nich (t. j. ten, ktorý má najmenší súčet váh hrán, ako by ho spočítala funkcia `SumE`). Pokiaľ aj tak existuje viacej možností, `Find` vráti ľubovoľnú z nich.

- Funkcia `Common(G,H,veq,eeq)` nájde najväčší spoločný podgraf grafov `G` a `H`. Presnejšie, nájde graf, ktorý je izomorfný (podľa `veq` a `eeq`) s niektorým podgrafom `G` i s niektorým podgrafom `H`. Zo všetkých možných riešení si navyše vyberie také, ktoré má najväčší možný počet vrcholov, a z takých potom to s najväčším počtom hrán. Pokiaľ aj tých je viacej, vyberie si ľubovoľné z nich.

  Výsledný graf bude mať id vrcholov v rovnakom poradí, ako ho mal zodpovedajúci podgraf v `G`, len budú prečíslované od 1 po ich počet, aby v číslovaní neboli diery. Ohodnotenie vrcholov a hrán bude taktiež zdedené z grafu `G`.



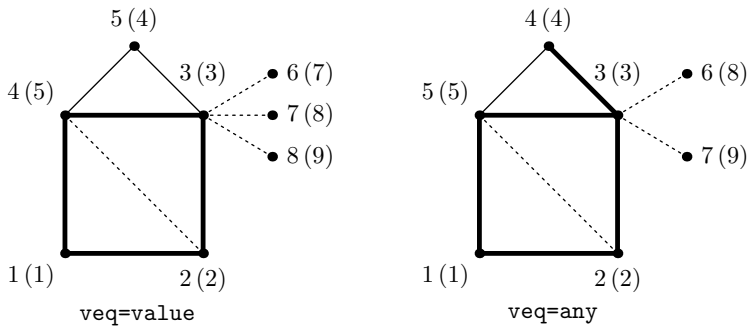

Na obrázku sú opäť plnými čiarami nakreslené dva grafy. Pri `veq=value` a `eeq=any` je ich najväčším spoločným podgrafom „štvorec“ obtiahnutý tučne. Čiarkované hrany ukazujú jedno možné priradenie vrcholov.

Ak zmeníme `veq` na `any`, pribudne do spoločného podgrafu ešte jedna nová hrana a jej koncový vrchol: v ľavom grafe to môže byť jedna z hrán 3-4 a 3-5, v pravom jedna z hrán 1-5, 1-6 a 1-7.

Najväčší spoločný podgraf nemusí byť určený jednoznačne. Napríklad vo vyššie popísanej situácii ním nemusí byť len zvýraznený štvorec. Rovnako veľký je aj podgraf, ktorý je v ľavom obrázku určený vrcholmi 3, 4, 5, 1 a v pravom 4, 1, 2, 3.

- Funkcia `Join(G,H,veq,eeq)` zlúči grafy `G` a `H`. Môžete si to predstaviť tak, že ich „zlepí za ich najväčší spoločný podgraf“. Urobí to tak, že najprv nájde najväčší spoločný podgraf (tak ako vo funkcii `Common`), potom k nemu doplní zostávajúce vrcholy grafu `G` a nakoniec vrcholy grafu `H` (id vrcholov výsledného grafu teda budú v tomto poradí). Hrany, váhy a značky pritom zdedí z oboch grafov, pričom pokiaľ sa nejaký vrchol alebo hrana vyskytujú v oboch grafoch, použije sa ohodnotenie z grafu `G`.

  Ak by sme pre grafy z predchádzajúceho obrázku použili `Join`, mohli by sme dostať napr. nasledujúci výstup:




Tučnými čiarami je vyznačená spoločná časť (všimnite si rozdiely v id

vrcholov), tenké neprerušované hrany pochádzajú z grafu `G`, čiarkované hrany sú z grafu `H`.

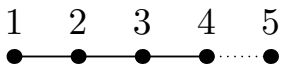
(Keďže aj pri `veq=value`, aj pri `veq=any` existovalo viac možností, ako vybrať najväčší spoločný podgraf, existuje aj viacero možností, ako bude vyzerať výstup funkcie `Join` pre tieto dva grafy. Výstup na obrázku vľavo zodpovedá priradeniu z obrázku k funkcii `Common`, výstup na obrázku vpravo zodpovedá tomu istému priradeniu a navyše vrchol 4 vľavo je priradený vrcholu 5 vpravo.)

Všetky operácie predpokladajú, že dostanú korektný vstup – nie je teda napríklad povolené volať ich s id neexistujúceho vrcholu alebo upravovať grafovú premennú, do ktorej ste ešte nepriradili, a podobne. Všetky grafové operácie trvajú konštantný čas.

**Príklad 1: Tvorba cesty:**

Ukážeme, ako vytvoriť cestu dĺžky $n$. To je graf s $n+1$ vrcholmi a $n$ hranami, v ktorom je každý vrchol spojený hranou s nasledujúcim. Určite by sme cestu mohli vytvárať postupne, napríklad takto:

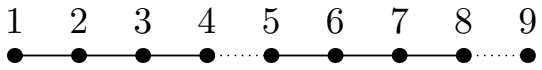
```
function cesta(n: Integer): Graph;
var i, posledny, novy: Integer;
    G: Graph;
begin
   G := EmptyG;
   posledny := AddV(G, 0);
   for i := 1 to n do begin
      novy := AddV(G, 0);
      AddE(G, posledny, novy, undef);
      posledny := novy;
   end;
   cesta := G;
end;
```




Začíname s jediným vrcholom (má id 1) a potom $n$-krát pridáme nový vrchol a hranu doň. (Vrcholom dávame značky 0, hranám nedefinované váhy, čo sa bude hodiť neskôr.) Celý postup teda trvá lineárne dlho a vytvorí cestu začínajúcu vo vrchole s id 1 a končiacu vo vrchole s id $n + 1$.

Ten istý graf však vieme zostrojiť aj rýchlejšie. Predstavme si na chvíľku, že už máme v `G` zostrojenú časť cesty, povedzme tvorenú $k$ vrcholmi. Pomocou `Join(G,G,none,none)` vytvoríme nový graf, ktorý obsahuje dve kópie tejto

cesty: jednu s id $1, \ldots, k$, druhú s id $k+1, \ldots, 2k$. Do toho stačí následne pridať hranu z $k$ do $k+1$ a máme cestu dĺžky $2k$. Tento prístup využijeme v nasledujúcom (rekurzívnom) riešení úlohy:



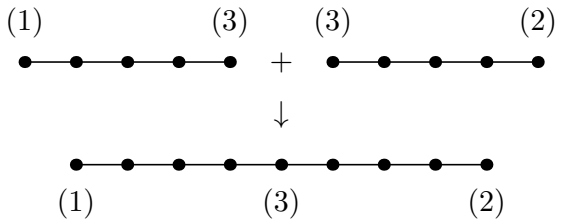

```
function cesta(n: Integer): Graph;
var vysledok: Graph;
    polovica: Integer;
begin
   if n = 0 then begin { cesta dlzky 0 je len jeden vrchol }
      vysledok := EmptyG;
      AddV(vysledok, 0);
   end else begin
      { rekurzivne vytvorime cestu polovicnej dlzky }
      polovica := (n-1) div 2;
      vysledok := cesta(polovica);
      { vyrobime 2 kopie a spojime ich }
      vysledok := Join(vysledok, vysledok, none, none);
      AddE(vysledok, polovica+1, polovica+2, undef);
      { ked polovica nevysla celociselna, pridame este hranu }
      if n mod 2 = 0 then begin
         AddV(vysledok, 0);
         AddE(vysledok, n, n+1, undef);
      end;
  end;
  cesta := vysledok;
end;
```

Pri každom rekurzívnom volaní sa $n$ zmenší aspoň na polovicu, časová zložitosť tohto riešenia je teda $O(\log n)$.

Ukážeme ešte jedno riešenie, tentoraz založené na spájaní ciest za vrchol. Budeme vytvárať cesty, ktorých začiatočný vrchol bude mať značku 1, koncový vrchol značku 2 a všetky ostatné vrcholy značku `undef`. Keď chceme dve cesty spojiť do jednej, preznačíme koncový vrchol prvej a začiatočný vrchol druhej na 3 a zavoláme `Join` s `veq=value_defined`. Tým spôsobíme, že sa vrcholy označené trojkou stotožnia a vznikne cesta dvojnásobnej dĺžky. (Všimnite si, že keby sme namiesto `value_defined` použili `value` alebo `value_strict`, stotožnili by sa aj vnútorné vrcholy ciest, čo nechceme.) Potom ešte odstránime pomocnú značku 3 a prepíšeme ju na `undef`. Program tentokrát pre jednoduchosť napíšeme len pre $n = 2^k$:

```
function cesta(n: Integer): Graph;
var G, T1, T2: Graph;
begin
   if n = 1 then begin
      G := EmptyG;
      AddV(G, 1);
```

```
      AddV(G, 2);
      AddE(G, 1, 2, undef);
  end else begin
      T1 := cesta(n div 2);
      T2 := T1;
      ReplaceV(T1, 2, 3);
      ReplaceV(T2, 1, 3);
      G := Join(t1, t2, value_defined, any);
      ReplaceV(G, 3, undef);
  end;
  cesta := G;
end;
```




Časová zložitosť tohto riešenia je opäť logaritmická od $n$.

**Príklad 2: Obchodný cestujúci:**

Všetci poznáme prešibaných obchodníkov. Predávajú ktovie čo a najradšej by boli, keby ich po predaji už kupujúci nikdy nenašiel. Predstavme si takého obchodníka. Teraz sa nachádza v meste (t. j. vo vrchole) číslo 1. Chce prejsť celú krajinu (graf) po cestách (hranách) tak, aby navštívil každé mesto práve raz a potom sa vrátil domov. Navyše pri tom chce najazdiť čo najmenej. Inými slovami, celková váha použitých hrán by mala byť najmenšia možná.

Na obvyklom počítači pre tento problém nepoznáme riešenie v polynomiálnom čase. Pokiaľ ale máme k dispozícii grafový počítač, pôjde to veľmi efektívne.

Stačí totiž vyrobiť cyklus z $n$ hrán a potom funkciou `Find` nájsť jeho najľahší výskyt v grafe predstavujúcom mapu krajiny. Cyklus vytvoríme tak, že podľa predchádzajúceho príkladu vytvoríme cestu s $n$ vrcholmi a $n-1$ hranami a potom spojíme hranou jej prvý vrchol s posledným. To vieme spraviť v logaritmickom čase. Následné volanie funkcie `Find` beží v konštantnom čase, a teda nám časovú zložitosť nepokazí.

```
function cestujuci(mapa: Graph): Graph;
var trasa: Graph;
begin
   trasa := cesta(CountV(mapa)-1);
   AddE(trasa, 1, CountV(mapa), undef);
   cestujuci := Find(mapa, trasa, any, any);
end;
```

# Zadania domáceho kola kategórie B

## B-I-1 Hľadanie mín

Hru „Hľadanie mín“ asi nemusíme nijak špeciálne predstavovať. Na niektorých políčkach obdĺžnikového hracieho plánu sa nachádzajú míny. Úlohou hráča je postupne odkryť všetky políčka, ktoré neobsahujú mínu. Aby hráč nemusel len náhodne tipovať, hra mu pomáha: vždy, keď odkryje prázdne políčko, dozvie sa, s koľkými mínami toto políčko (stranou alebo rohom) susedí.

**Súťažná úloha:**

Napíšte program, ktorý načíta rozmery hracieho plánu, počet mín a ich rozmiestnenie a vypočíta, ako by vyzerala úspešne ukončená hra na danom pláne.

**Formát vstupu:**

V prvom riadku každého vstupného súboru sú dve celé čísla: počet riadkov $R$ a počet stĺpcov $S$ hracieho plánu. Riadky sú očíslované zhora dole od 1 po $R$, stĺpce zľava doprava od 1 po $S$.

V druhom riadku je jedno celé číslo $N$ udávajúce počet mín. Nasleduje $N$ riadkov. V $i$-tom z nich sú dve celé čísla $r_i$ a $s_i$, udávajúce riadok a stĺpec jedného z políčok, ktoré obsahujú mínu.

**Formát výstupu:**

Váš program by mal vyrobiť textový súbor, v ktorom bude $R$ riadkov a v každom z nich $S$ znakov. Každý znak predstavuje jedno políčko hracieho plánu. Ak je na danom políčku mína, vypíšte znak ‘*’ (hviezdička). Ak dané prázdne políčko susedí s 1 až 8 mínami, vypíšte príslušnú cifru. Ak dané prázdne políčko nesusedí so žiadnou mínou, vypíšte znak ‘.’ (bodka).

**Príklad:**

**Vstup**

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

**Výstup**

```
......1**
.....1232
.....1*1.
.....111.
..111....
..1*11221
223221**1
**3*11221
3*311....
```

*Tento príklad vstupu a výstupu zodpovedá obrázku na vrchu zadania.*

## B-I-2 Starosta

Móricka už omrzelo, že sú v jeho rodnej dedine chodníky plné odpadkov, obecný policajt je nezvestný, psy starej Košáričky terorizujú celú ulicu a navyše stále prší. Rozhodol sa, že bude kandidovať na starostu a všetko to zmení k lepšiemu.

Lenže byť na dedine starostom, to nie je len tak. Na to je najlepšie poznať všetkých dedinčanov. A keď náhodou starosta nejakého človeka Č nepozná, tak nutne musí poznať aspoň niekoho, kto človeka Č pozná. (Inými slovami, starosta musí mať s človekom Č spoločného známeho.)

Móricko si teraz nie je istý, či už môže za starostu kandidovať. Napíšte program, ktorý mu to zistí.

**Súťažná úloha:**

Daný je zoznam všetkých dvojíc dedinčanov, ktorí sa navzájom poznajú. Zistite, či už Móricko môže byť starostom. Ak nie, nájdite všetkých ľudí, ktorých nepozná ani Móricko, ani nik, koho Móricko pozná.

**Formát vstupu:**

V prvom riadku vstupného súboru sú dve celé čísla $D$ a $Z$ ($1 \leq D \leq 1\,000\,000$, $0 \leq Z \leq 5\,000\,000$). $D$ je počet dedinčanov, $Z$ je počet známostí. Dedinčanov si očíslujeme od 1 po $D$, pričom číslo 1 dostane Móricko.

Každý zo zvyšných $Z$ riadkov obsahuje dve čísla od 1 po $D$ – čísla dvoch dedinčanov, ktorí sa navzájom poznajú.

**Formát výstupu:**

Ak Móricko môže byť starostom, vypíšte jeden riadok a v ňom reťazec „`starosta`“. Ak nie, vypíšte čísla všetkých dedinčanov, kvôli ktorým Móricko byť starostom nemôže. Čísla vypíšte v ľubovoľnom poradí, každé práve raz.

**Príklady:**

**Vstup**

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

**Výstup**

```
starosta
```

*Móricko pozná dedinčanov 3 a 4. Dedinčana 2 síce nepozná, ale pozná dedinčana 4 a ten pozná dedinčana 2. Takže je všetko v poriadku a Móricko môže byť starostom.*

**Vstup**

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

**Výstup**

```
6
4
```

*Móricko nemôže byť starostom. Dedinčanov 4 a 6 treba vypísať na výstup, keďže Móricko nepozná ani ich, ani žiadneho ich známeho.*

## B-I-3 Poriadok na polici

V knižnici Neviditeľnej univerzity je polica a na tej polici je šesť obrovských a strašne ťažkých zväzkov magickej encyklopédie. Tieto zväzky žijú vlastným životom a každú noc sa škodoradostne preusporiadajú.

Chudák knihovník ich potom musí každé ráno preusporiadať do správneho poradia. Keďže sú však strašne ťažké, jediné, čo zvláda, je vymeniť dva susedné zväzky. A navyše po každej takej výmene musí 10 minút oddychovať.

Včera napríklad boli zväzky ráno v poradí 1, 4, 2, 3, 5, 6. Keď ich chcel knihovník usporiadať, najskôr vymenil zväzky 4 a 2 (čím dostal poradie 1, 2, 4,

3, 5, 6), potom si oddýchol a potom vymenil zväzky 4 a 3 (čím dostal správne poradie 1, 2, 3, 4, 5, 6).

**Súťažná úloha:**

a) (3 body) Dnes sú zväzky v poradí 4, 2, 1, 6, 5, 3. Poraďte knihovníkovi, ako ich najrýchlejšie usporiada.

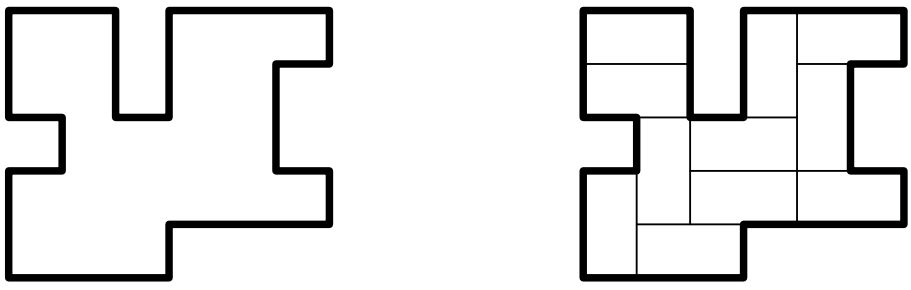
b) (3 body) Zdôvodnite, prečo je vaše riešenie podúlohy a) optimálne. Inými slovami, zdôvodnite, že neexistuje postup, ktorý by zväzky usporiadal pomocou menej výmen ako ten váš.

c) (4 body) Budúci mesiac vyjde nové vydanie magickej encyklopédie. To už nebude mať 6 zväzkov, ale $N$.

Dokážte, že bez ohľadu na to, ako sa v noci zväzky prehádžu, knihovníkovi bude určite stačiť spraviť nanajvýš $N(N-1)/2$ výmen na to, aby ich usporiadal.

## B-I-4 Dláždenie

Korina je architektka. Poslednou dobou sa pustila do navrhovania moderných bytov. Keďže si kúpila štvorčekový zošit, rozhodla sa, že každá miestnosť, ktorú navrhne, bude mať pôdorys tvorený niekoľkými jednotkovými štvorcami. Podlahu v miestnosti sa Korina rozhodla vydláždiť parketami rozmerov presne $2 \times 1$ štvorček.

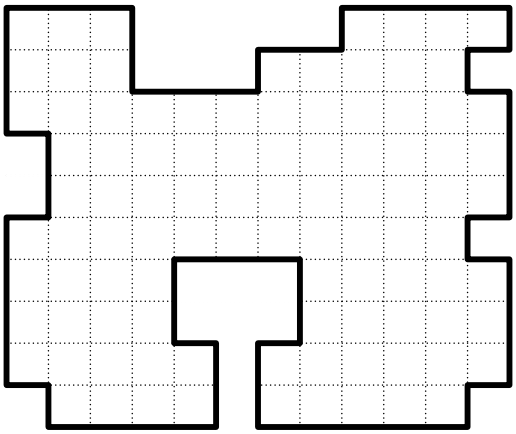


Na obrázku vľavo vidíte príklad jednej takejto miestnosti, na obrázku vpravo je jedno možné vydláždenie.

**Súťažná úloha:**

a) (2 body) Korina rýchlo zistila, že ak navrhne miestnosť tvorenú nepárnym počtom štvorčekov, vydláždiť sa jej ju nepodarí. Ukážte jej, že niekedy ani párny počet štvorčekov nepomôže. Nájdite miestnosť tvorenú presne 10 štvorčekmi, ktorá sa nebude dať vydláždiť.

b) (3 body) Na nasledujúcom obrázku je ďalšia z miestností, ktoré Korina navrhla. Nech sa snaží, ako chce, nedarí sa jej ju vydláždiť. Pomôžte jej, alebo dokážte, že sa táto miestnosť vydláždiť nedá.



c) (2 body) Pri niektorých miestnostiach má Korina na výber viacero spôsobov, ako ju vydláždiť. Navrhnite takú miestnosť, kde bude mať Korina na výber presne 3 možné dláždenia.

d) (3 body) Navrhnite takú miestnosť, kde bude mať Korina na výber presne 16 možných dláždení.

Všetky vami navrhnuté miestnosti musia byť súvislé (t. j. „držať pokope“) a nesmú vo svojom vnútri obsahovať diery.

# Zadania krajského kola kategórie A

## A-II-1 Vlak

Na nákladnej stanici v istom nemenovanom meste stojí utajený vlak. Teda, presnejšie povedané, iba vozne. A v nich prísne strážený náklad – v jednom gumové medvedíky, v inom fialové kravy, v ďalších zase iné dobroty. Pretože vlak obsahuje spomínané tajné materiály, riaditeľ stanice by ho chcel čím skôr vypraviť, aby sa po jednom anonymnom tipe náhodou celá stanica nezmenila na bojisko proti organizovanému detskému zločinu. (Predstavte si vagón plný čokolády, stovky detí snažiacich sa ho vyžrať a železničnú políciu márne sa snažiacu zabrániť tejto katastrofe.)

Lenže je tu istý háčik – podľa nového nariadenia musí výpravca poslať do konečnej stanice vlaku rozpis, v akom poradí majú prísť vozne. Na prvý pohľad sa zdá, že je to triviálna otázka, ale lokomotíva môže byť počas medzizastávky v niektorej stanici preradená na druhý koniec vlaku. A nikto nevie (z dôvodov utajenia), ktorou trasou ten vlak vlastne pôjde, nie to ešte koľkokrát bude lokomotíva preradená. Preto sa na stanici rozhodli, že z vlaku vyradia niekoľko vagónov tak, aby zostávajúce vagóny vyzerali rovnako, nech je už lokomotíva na ktoromkoľvek konci. A samozrejme, chcú vyradiť čo najmenej vagónov.

**Súťažná úloha:** Na vstupe je postupnosť písmen – typov vagónov. Výstupom vášho programu majú byť pozície vagónov (písmen), ktoré treba vyradiť z vlaku, aby bol výsledný vlak (postupnosť písmen) rovnaký pri čítaní spredu aj odzadu (teda aby výsledný vlak bol palindrómom).

Pokiaľ by existovalo viacero možností, nájdite a vypíšte tú, pri ktorej treba odstrániť najmenší možný počet vagónov. Pokiaľ by takýchto možností stále bolo viac, vypíšte ľubovoľnú z nich.

**Formát vstupu:** Na vstupe sú dva riadky. V prvom je jediné číslo $n$ ($1 \leq n \leq 10\,000$) udávajúce počet vagónov vlaku. Druhý riadok obsahuje postupnosť $n$ znakov, ktoré reprezentujú jednotlivé typy vagónov.

**Formát výstupu:** Na výstup vypíšte dva riadky. V prvom riadku vypíšte jediné číslo $k$ ($0 \leq k \leq n - 1$) udávajúce najmenší počet vagónov, ktoré je potrebné z vlaku vyradiť. V druhom riadku vypíšte $k$ rôznych čísel oddelených medzerou: čísla vagónov, ktoré treba vyradiť. Vagóny sú očíslované od 1 po $n$ v poradí, v akom sú na vstupe.

**Príklad:**

| Vstup | Výstup |
|---|---|
| 6<br>ABCDBA | 1<br>3 |

*Odstránením tretieho vagóna vznikne vlak s vagónmi ABDBA.*
*Iným optimálnym riešením je odstrániť vagón D.*

| Vstup | Výstup |
|---|---|
| 7<br>ABECEDA | 2<br>2 6 |

*Odstránením druhého a šiesteho vagóna vzniká AECEA.*

| Vstup | Výstup |
|---|---|
| 7<br>ABECADA | 4<br>3 4 6 7 |

*Výstupu zodpovedá vlak ABA. Iných, tiež optimálnych riešení je v tomto prípade mnoho.*

## A-II-2 Jabloňový sad

V jednom malom kráľovstve vyrástol strom so zlatými jablkami. Kráľ prikázal záhradníkom, nech takých jabloní vypestujú celý sad. Keby však vysadili zlaté jadierka len tak, vyrástli by z nich obyčajné plánky. Našťastie alchymisti vedia vypočítať, kam zasadiť nasledujúce jadierko, aby aj z neho vyrástol strom so zlatými jablkami.

Aby zlaté jablká nik neukradol, musel kráľ čoskoro začať so stavbou oplotenia. Najlacnejšie riešenie, ktoré vymyslel, vyzeralo nasledovne: Teraz nechá postaviť najkratší možný plot okolo všetkých $n$ už rastúcich stromov. V budúcnosti len bude tento plot podľa potreby rozširovať vždy, keď bude treba nový strom zasadiť mimo oplotenej oblasti. Navyše pri rozširovaní sa vždy dá časť plota rozobrať a použiť znova na inom mieste, takže bude treba dokúpiť len toľko materiálu, o koľko bude nový plot dlhší od starého.

**Súťažná úloha:**

Na začiatku dostanete kartézske súradnice $n$ bodov (jabloní) v rovine: $[x_1, y_1]$,

$\ldots, [x_n, y_n]$. Musíte zistiť dĺžku najkratšieho plota takého, že všetky jablone ležia na obvode alebo vo vnútri oplotenej oblasti.

Potom budete postupne dostávať súradnice ďalších $m$ bodov: miesta, kde budú zasadené nasledujúce jablone. Vždy, keď načítate súradnice ďalšieho bodu, musíte vypočítať a vypísať, o koľko práve narástla dĺžka najkratšieho plota. Vami vypočítaný údaj zodpovedá tomu, koľko pletiva je potrebné dokúpiť. (Uvedomte si, že nový plot nikdy nebude kratší od starého. Ak nový bod leží na oplotení alebo v jeho vnútri, výstupom je nula.)

Očakávajte, že čísla $n$ a $m$ budú veľké. Efektívne riešenie by teda malo pri spracúvaní nového bodu šikovne využiť skôr vypočítané informácie. (Nejaké body však samozrejme získate aj za korektné riešenie, ktoré zakaždým odznova správne zostrojí celý plot.)

**Dôležité:** Ak v riešení tejto úlohy použijete nejakú všeobecne známu dátovú štruktúru (napr. haldu alebo binárny vyhľadávací strom), stačí uviesť popis jej vlastností, ktoré v riešení využívate. Nie je potrebné rozpisovať jej implementáciu.

**Formát vstupu:**

V prvom riadku vstupu bude číslo $n$, ktoré udáva začiatočný počet jabloní. Nasleduje $n$ riadkov, v $i$-tom z nich sú dve čísla $x_i$ a $y_i$ oddelené medzerou: súradnice jedného z už zasadených stromov.

Ďalší riadok obsahuje číslo $m$. Nasleduje $m$ riadkov, v $i$-tom z nich sú dve čísla $x_{n+i}$ a $y_{n+i}$ oddelené medzerou: súradnice stromu, ktorý bude v budúcnosti vysadený ako $i$-ty v poradí.

**Formát výstupu:**

Na výstup vypíšte $m + 1$ riadkov. V prvom riadku bude súčasná dĺžka oplotenia, t.j. obvod najmenšieho obrazca, ktorý obsahuje všetky body $[x_1, y_1]$ až $[x_n, y_n]$. Nasleduje $m$ riadkov, v $i$-tom z nich má byť uvedené, o koľko sa obvod obrazca zväčší po pridaní bodu $[x_{n+i}, y_{n+i}]$ k ostatným.

**Príklad:**

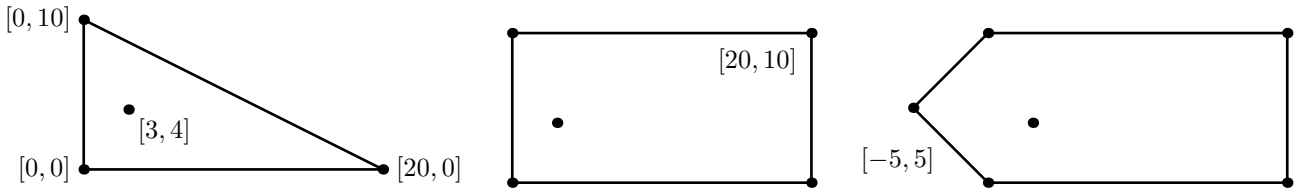
**Vstup**

```
4
0 0
0 10
20 0
3 4
2
20 10
-5 5
```

**Výstup**

```
52.36068
7.63932
4.14214
```

*Na obrázku vľavo je začiatočný plot, jeho dĺžka je* $10+20+\sqrt{10^2+20^2}$. *Po pridaní piateho bodu dostaneme obrázok v strede, jeho obvod je 60, teda narástol o 7.64 oproti predchádzajúcemu. Po pridaní šiesteho bodu dostaneme obrázok napravo.*




## A-II-3 Mažoretky

Na planéte Diks majú v hlavnom meste už vyše sto rokov slávny súbor mažoretiek. Každé ráno slávia príchod nového dňa sprievodom po uliciach mesta, pri ktorom kráčajú v rade jedna za druhou a predvádzajú rôzne prvky. Občania mesta sledujú vystúpenie a hneď sa im radostnejšie vstáva do školy a do práce.

Mažoretiek v súbore je presne $n$. Aby sa občania mesta nenudili, každý deň idú v sprievode v inom poradí. Zhodou okolností má rok na planéte Diks presne $1 \cdot 2 \cdot \ldots \cdot n = n!$ dní, takže za jeden rok použijú práve raz každé možné poradie.

Aby sa im to nepoplietlo, poradie si určujú systematicky, a to nasledovne. Každá mažoretka má svoje jednoznačné číslo od 1 po $n$. Ich poradie v konkrétny deň teda vieme popísať ako postupnosť čísel. Ľubovoľné dve poradia $(a_1, \ldots, a_n)$ a $(b_1, \ldots, b_n)$ môžeme teraz porovnať nasledovne: Nech $i$ je najmenší index, pre ktorý $a_i \neq b_i$. Potom to poradie, ktoré má na $i$-tej pozícii mažoretku s menším číslom, bude použité v skorší deň v roku ako to druhé.

Napríklad pre $n = 4$ bude poradie $(3, 1, 2, 4)$ použité skôr ako $(3, 4, 1, 2)$, lebo na druhej pozícii má prvé z nich mažoretku s menším číslom.

Počas jedného roka by pre $n = 3$ mažoretky postupne použili nasledovných 6 usporiadaní:
$(1, 2, 3)$, $(1, 3, 2)$, $(2, 1, 3)$, $(2, 3, 1)$, $(3, 1, 2)$ a nakoniec $(3, 2, 1)$.

**Súťažná úloha:**

Na vstupe je číslo $n \geq 2$ a jedno konkrétne poradie mažoretiek.

Napíšte program, ktorý:

a) (3 body) vypíše poradie mažoretiek v nasledujúci deň,

b) (7 bodov) vypíše poradie mažoretiek presne o pol roka.

V oboch podúlohách nezabudnite na to, že dotyčný deň môže byť v nasledujúcom kalendárnom roku.

**Príklad pre podúlohu a):**

| Vstup | Výstup |
|---|---|
| 5<br>1 4 2 5 3 | 1 4 3 2 5 |

**Príklady pre podúlohu b):**

| Vstup | Výstup |
|---|---|
| 3<br>1 3 2 | 3 1 2 |

*Rok na tejto planéte má* $3! = 6$ *dní, hľadáme teda poradie o* $6/2 = 3$ *dni. V zadaní je uvedený zoznam všetkých poradí pre* $n = 3$*, z neho vidíme, že o 3 dni po poradí* $(1, 3, 2)$ *bude použité poradie* $(3, 1, 2)$.

| Vstup | Výstup |
|---|---|
| 3<br>3 1 2 | 1 3 2 |

*O ďalšie 3 dni bude opäť použité poradie* $(1, 3, 2)$.

## A-II-4 Grafový počítač a kompletné grafy

V tomto ročníku OI-A sa v poslednej úlohe stretávame so špeciálnym grafovým počítačom. Jeho popis (rovnaký ako v domácom kole) nájdete v študijnom texte za zadaním tejto úlohy.

**Súťažná úloha:**

a) (5 bodov) Napíšte funkciu pre grafový počítač, ktorá dostane ako parameter číslo $n$ a na výstupe vráti premennú obsahujúcu kompletný graf $K_n$. To je graf s $n$ vrcholmi, z ktorých každé dva sú spojené hranou.

   (Najviac 4 body môžete získať za riešenie, ktoré túto úlohu rieši za predpokladu, že $n$ je mocninou dvoch.)

b) (5 bodov) Hovoríme, že graf obsahuje *kliku* veľkosti $k$, ak v ňom existuje $k$-tica vrcholov, ktoré sú spojené každý s každým. *Klikovosť* grafu je počet vrcholov v najväčšej klike, ktorú obsahuje.

   Napíšte funkciu pre grafový počítač, ktorá dostane ako parameter premennú obsahujúcu graf $G$ a na výstupe vráti celé číslo udávajúce jeho klikovosť.

Tabuľku s povolenými inštrukciami nájdete na strane 10, študijný text začína na strane 11.

# Zadania krajského kola kategórie B

## B-II-1 Hľadanie mín II

Známa hra „Hľadanie mín“ sa hrá na obdĺžnikovom hracom pláne, ktorý obsahuje niekoľko mín. Na začiatku hry sú všetky políčka zakryté. Úlohou hráča je odhaliť pozície mín. Hracia plocha obsahuje 3 druhy políčok:

- *mínové políčko*: Obsahuje mínu.
- *políčko s číslom*: Neobsahuje mínu, ale susedí (stranou alebo rohom) s aspoň jednou mínou. Číslo v takomto políčku určuje počet susediacich mínových políčok.
- *prázdne políčko*: Neobsahuje mínu ani nesusedí so žiadnou mínou.

V každom ťahu si hráč zvolí jedno zakryté políčko. Toto políčko sa mu odkryje. Ak obsahuje mínu, hráč prehral. Ak obsahuje číslo, hráč sa ho dozvie a ťah končí. Po kliknutí hráča na políčko $p$, ktoré je *prázdne*, sa okrem políčka $p$ odkryje aj viacero iných políčok. Odkryté políčka vieme nájsť systematickým aplikovaním nasledujúceho pravidla:

„*Ak existuje neodkryté políčko, ktoré stranou alebo rohom susedí s nejakým* ***prázdnym*** *odkrytým políčkom, odkry ho.*“

(Hra teda len urobí za hráča niekoľko ťahov, ktoré sú v danej situácii jasné – odkryje len políčka, na ktorých očividne žiadna mína byť nemôže.)

**Súťažná úloha:**

Uvažujme prvý ťah hry, v ktorom si hráč zvolil ľavé horné políčko hracieho plánu. Napíšte program, ktorý načíta rozmery hracieho plánu, počet mín a ich rozmiestnenie a vypočíta, ako bude vyzerať hrací plán po uvedenom ťahu.

**Formát vstupu:**

V prvom riadku vstupu sú dve celé čísla: počet riadkov $R$ a počet stĺpcov $S$ hracieho plánu. Riadky sú očíslované zhora dole od 1 po $R$, stĺpce zľava doprava od 1 po $S$. V druhom riadku je jedno celé číslo $N$, udávajúce počet mín. Nasleduje $N$ riadkov. V $i$-tom z nich sú dve celé čísla $r_i$ a $s_i$, udávajúce riadok a stĺpec jedného z políčok, ktoré obsahujú mínu. Môžete predpokladať, že políčko v prvom riadku a prvom stĺpci neobsahuje mínu.

**Formát výstupu:**

Výstup by mal obsahovať hrací plán po prvom ťahu do ľavého horného rohu. Vypíšte $R$ riadkov a v každom z nich $S$ znakov. Každý znak predstavuje jedno políčko hracieho plánu. Ak políčko nie je odkryté, vypíšte znak '`?`' (otáznik). Ak políčko je odkryté a susedí s 1 až 8 mínami, vypíšte príslušnú cifru. Ak políčko je odkryté, no nesusedí so žiadnou z mín (je *prázdne*) vypíšte znak '`.`' (bodka).

**Príklad:**

**Vstup**

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

**Výstup**

```
.1?1.....
.1?1..111
.111.12??
.....1???
1211.12??
???1..12?
?311...1?
?1..1111?
?1..1????
```

**Obrázok**

*Výstup zodpovedá nasledovnému hraciemu plánu:*

## B-II-2 Zakopaný pes

Strýko Jano je nadšeným zástancom všelijakých konšpiračných teórií. Nedávno sa z dôveryhodného zdroja dozvedel, že svetová tieňová vláda uverejňuje v dennej tlači skryté správy. Na ich prečítanie stačí vybrať z textu niektoré písmená. Skúsený konšpiračný teoretik tak dokáže vhodným výberom písmen nájsť v jedinom článku dôkazy o existencii mimozemšťanov a fingovanom pristátí na Mesiaci. Na takejto vysokej úrovni strýko ešte nie je, preto najprv trénuje hľadaním niečoho jednoduchšieho. V každom článku, ktorý dostane do ruky, hľadá a počíta skryté výskyty slova PES.

**Súťažná úloha:**

Napíšte program, ktorý načíta riadok textu a vypíše počet spôsobov, ako možno v tomto texte prečítať slovo PES. Snažte sa, aby váš program bol čo najefektívnejší, lebo niektoré novinové články sú veľmi dlhé.

**Formát vstupu a výstupu:**

V jedinom riadku vstupu je daný reťazec písmen veľkej anglickej abecedy. Výstup by mal pozostávať z jediného riadka obsahujúceho jedno číslo – počet spôsobov, ako možno zo vstupného reťazca odstránením niektorých písmen dostať slovo PES.

**Príklad:**

| Vstup | Výstup |
|---|---|
| APREDSATUJEPES | 5 |

*Takto sa dá vo vstupnom reťazci prečítať PES:*

```
-P-E-S--------
-P-E---------S
-P--------E--S
-P----------ES
-----------PES
```

## B-II-3 Poriadok na polici II

Istotne si pamätáte príbeh o Neviditeľnej univerzite a jej knihovníkovi. Ten má v polici $n$ zväzkov magickej encyklopédie, očíslovaných číslami 1 až $n$. Zlé víly tieto zväzky každú noc preusporiadajú a knihovník ich potom musí každé ráno zoradiť do správneho poradia – do postupnosti $1, 2, \ldots, n$. No keďže sú zväzky veľmi ťažké, jediné čo zvláda je vymeniť dva **susedné** zväzky. Po dlhých rokoch práce v knižnici mu dnes každá takáto výmena trvá presne 1 minútu (a verte, že rýchlejšie to už skutočne nejde).

Napríklad včera ráno boli zväzky v poradí $1, 4, 2, 3, 5, 6$. Keď ich chcel knihovník usporiadať, najskôr vymenil zväzky 4 a 2 (čím dostal poradie $1, 2, 4, 3, 5, 6$) a následne vymenil zväzky 4 a 3 (čím dostal správne poradie $1, 2, 3, 4, 5, 6$).

Z vašich riešení domáceho kola sa knihovník naučil postup (algoritmus), ktorým zaručene každú postupnosť kníh vie usporiadať, a navyše pri tom spraví najmenší možný počet výmen. Algoritmus je založený na jednoduchej myšlienke: Ak niekde vidíš vedľa seba dve knihy také, že ľavá má väčšie číslo ako pravá, vymeň ich. A ak takú dvojicu už nevidíš, môžeš skončiť.

Do knižnice dnes zavítala Optimalizačná Inšpekcia (OI), aby prešetrila, či sa knihovník pri preusporadúvaní kníh neulieva. Dnes mu usporiadanie $n$-zväzkovej

encyklopédie trvalo $k$ minút. Teraz by rád inšpektorom OI zdôvodnil, že pracoval optimálne, no akosi si už nevie spomenúť, v akom poradí to tie zväzky ráno boli. Pomôžte prosím knihovníkovi nájsť aspoň jedno poradie zväzkov také, že keď ho optimálnym spôsobom usporiadame, tak spravíme presne $k$ výmen.

**Súťažná úloha:**

a) (2 body) Zistite, či existuje postupnosť 6 zväzkov, ktorej usporiadanie optimálnym postupom vyžaduje presne 17 výmen. Ak áno, jednu takú postupnosť nájdite.

b) (2 body) Zistite, či existuje postupnosť 10 zväzkov, ktorej usporiadanie optimálnym postupom vyžaduje presne 22 výmen. Ak áno, jednu takú postupnosť nájdite.

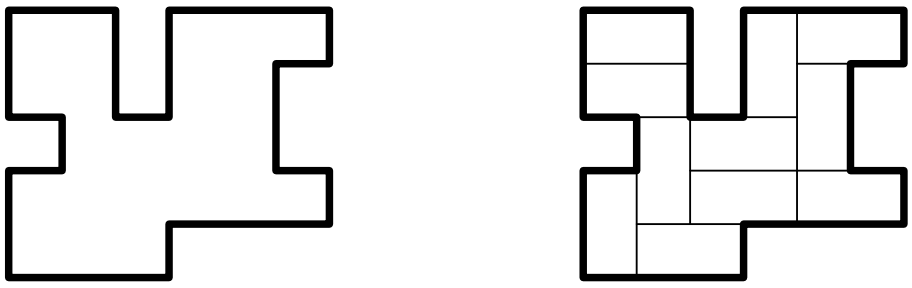
c) (6 bodov) Napíšte program, ktorý načíta $n$ a $k$ a nájde jednu konkrétnu postupnosť dĺžky $n$, na usporiadanie ktorej treba presne $k$ výmen. Ak taká postupnosť neexistuje, váš program by to mal zistiť a podať o tom vhodnú správu. Snažte sa, aby váš program bol čo najefektívnejší.

**Príklad:**

Pre $n = 4$ a $k = 3$ je jednou z takýchto postupností $4, 1, 2, 3$. (Na jej usporiadanie treba presne 3 výmeny. Inými slovami, existuje spôsob, ako zoradiť túto postupnosť pomocou troch výmen, a zároveň žiaden postup, ktorý použije menej ako tri výmeny, ju nezoradí.)

## B-II-4 Dláždenie pre pokročilých

Ako isto viete, túto sezónu je v móde používať v miestnostiach parkety rozmerov $2 \times 1$ štvorček. Nimi sa budeme aj dnes snažiť pokrývať podlahy rôznych miestností.



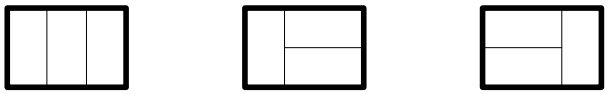
Vľavo: príklad miestnosti. Vpravo: jedno jej možné vydláždenie.

Štvorčeky, ktoré tvoria konkrétnu miestnosť, si môžeme vyfarbiť ako šachovnicu. Miestnosť, ktorá obsahuje rovnako veľa čiernych a bielych štvorčekov, budeme volať *vyvážená miestnosť*.

Z domáceho kola vieme, že ak miestnosť nie je vyvážená, tak sa nedá vydláždiť. (To preto, že každá parketa pokryje jedno čierne a jedno biele políčko.)

**Súťažná úloha:**

a) (4 body) V domácom kole sme zistili, že obdĺžnik $2 \times 3$ vieme vydláždiť presne tromi rôznymi spôsobmi:



Pre čo najviac $n$ z množiny $\{4, 5, 6, \ldots, 15\}$ zistite, koľkými spôsobmi sa dá vydláždiť obdĺžnik $2 \times n$.

b) (3 body) Navrhnite takú miestnosť, kde budeme mať na výber presne 150 možných dláždení. Zdôvodnite, že vami navrhnutá miestnosť ich má presne toľko.

c) (3 body) Architektka Korina by chcela navrhovať len také miestnosti, ktoré sa určite dajú vydláždiť. Používa preto dve pravidlá:
– Každá miestnosť, ktorú Korina navrhne, je vyvážená.
– Každý štvorček tvoriaci jej miestnosť susedí stranami s aspoň dvoma inými štvorčekmi.

Je zaručené, že sa každá Korinina miestnosť bude dať vydláždiť? Ak áno, slovne popíšte postup, ktorý pre každú takúto miestnosť nájde nejaké vydláždenie. Ak nie, nájdite konkrétnu miestnosť, ktorá spĺňa obe podmienky a predsa sa nedá vydláždiť.

# Zadania celoštátneho kola kategórie A

## A-III-1 Romantické básničky

Hermi má rád dievčatá, to nie je žiadna novinka. Pre mnohých z vás ani nie je novinkou, že jeho otec je Hviezdoslav. A keďže genetika melie pomaly ale isto, má aj on básnické črevo. A iste je vám jasné, na čo ho využíva – na romantické básničky.

Neviem, či ste už niekedy písali milostnú básničku, ale vedzte, nie je to jednoduché. Keby bol človek zviazaný iba myšlienkami, ešte by to nejak šlo. No básničky sa predsa musia rýmovať! A hľadať také rýmy nebýva jednoduché. Teraz sa ale blíži sústredenie s orchestrom a Hermi musí napísať každej spoluhráčke básničku. A to by sa z toho zjašil, keby musel všetky tie rýmy vymýšľať. Našťastie to môže prepašovať do zadaní olympiády a všetko budete musieť odmakať vy. S programom na hľadanie rýmov už potom Hermi básničky stvorí raz-dva.

**Súťažná úloha:**

Na vstupe je zoznam všetkých $n$ pekných slov, ktoré Hermi pozná. Tento zoznam budeme volať *lexikón*. Ďalej nasleduje $m$ otázok – slov, ku ktorým treba v lexikóne nájsť čo najlepší rým.

Pre jednoduchosť budeme predpokladať, že rým je tým lepší, čím viac spoločných písmen majú dve slová na konci. Odborne povedané, dĺžka spoločného sufixu má byť čo najväčšia. Napríklad slová „`tvari`“ a „`podari`“ majú najdlhší spoločný sufix „`ari`“ dĺžky 3, zatiaľčo „`tvari`“ a „`frajeri`“ majú len spoločný sufix dĺžky 2. Preto ku slovu „`tvari`“ je slovo „`podari`“ lepším rýmom ako slovo „`frajeri`“.

Hermi je navyše veľmi metodický. Ak je v lexikóne na výber viacero slov, ktoré sa s jeho slovom rýmujú rovnako dobre, chce vždy použiť to, ktoré je z nich prvé v abecede. A toto slovo musíte vy nájsť. (Odborne hovoríme, že hľadáme lexikograficky najmenšie slovo. Napr. „`had`“ < „`hadica`“ < „`pes`“.)

Keďže otázok môže byť veľa, snažte sa optimalizovať časovú zložitosť na zodpovedanie jednej otázky – a to aj za cenu pomalšieho predspracovania lexikónu.

**Poznámka:** Ak úlohu neviete efektívne vyriešiť, uveďte aspoň riešenie, ktoré spomedzi najviac sa rýmujúcich slov vypíše ľubovoľné (teda nie nutne to, ktoré je prvé v abecede).

**Formát vstupu:**

V prvom riadku je číslo $n$. Nasleduje lexikón: $n$ navzájom rôznych reťazcov zložených z malých písmen anglickej abecedy, každý v samostatnom riadku.

V nasledujúcom riadku je číslo $m$. Nasledujú otázky: $m$ reťazcov zložených z malých písmen anglickej abecedy, každý v samostatnom riadku.

Môžete predpokladať, že súčet dĺžok všetkých slov v slovníku nepresiahne $10^6$ znakov a žiadne slovo na vstupe nebude dlhšie ako $10^4$ znakov.

**Formát výstupu:**

Pre každú z Hermiho otázok vypíšte na samostatný riadok slovo z lexikónu, ktoré s ním má najdlhší spoločný sufix. Ak je viac možností, musíte vypísať tú z nich, ktorá je prvá v abecednom poradí.

**Príklad:**

**Vstup**

```
5
prasa
pomysel
ziari
tvari
zabronel
3
krasa
zmari
bager
```

**Výstup**

```
prasa
tvari
pomysel
```

*Pri druhom rýme mal Hermi na výber dve slová, ale* `tvari` *je v abecede skôr ako* `ziari`*.*

*Pri tretej otázke si všimnite, že žiadne zo slov sa s otázkou vôbec nerýmuje. V takomto prípade je samozrejme správnym výstupom to z nich, ktoré je úplne prvé v abecede.*

## A-III-2 Ministerstvo

Ministerstvo pre minimalizáciu byrokracie zamestnáva takmer milión úradníkov. Úradníci sú usporiadaní v typickej hierarchickej štruktúre, kde každý úradník má práve jedného priameho nadriadeného. Výnimkou je minister, ktorý nadriadeného nemá. Navyše každý úradník vykonáva len jedinú činnosť – niektorí vyškrtávajú kolónky sprava doľava, iní zľava doprava, iní dávajú okrúhle pečiatky, ... Výnimkou je opäť minister, ktorý ako jediný môže a vie robiť všetko.

Keď chce bežný človek na úrade niečo vybaviť, najprv navštívi nejakého úradníka, ktorý má za úlohu styk s verejnosťou. Ten už ale nevie robiť nič iné, preto ho pošle za svojím nadriadeným. Ak ten robí to, čo klient potrebuje, tak mu pomôže, ináč ho pošle za svojím nadriadeným. A toto sa opakuje, kým daného klienta neobslúžia. (V najhoršom prípade sa klient dostane k samotnému ministrovi a ten ho už zaručene obslúži.)

Po niekoľkých analýzach chodu ministerstva sa zistilo, že niektorí úradníci fakt nič nerobia a ani nikdy robiť nebudú. (Napríklad nikdy nič nerobia úradníci, ktorí nemajú žiadneho podriadeného pre styk s verejnosťou. Iným príkladom sú takí, ktorých všetci priami podriadení robia to isté, čo oni.) No a v duchu hesla „kto nič nerobí, nič nepokazí“, treba nájsť a odmeniť **všetkých** takýchto úradníkov.

**Súťažná úloha:**

Úradníci sú očíslovaní číslami $1, 2, \ldots, n$, pričom číslom 1 je označený minister. Pre každého úradníka $i$ okrem ministra vieme číslo $p_i$ jeho priameho nadriadeného, pričom platí $p_i < i$. Činnosti sú očíslované číslami $1, 2, \ldots, m$, pričom styk s verejnosťou má číslo 1. Pre každého úradníka $i$ okrem ministra vieme číslo $c_i$ činnosti, ktorú vykonáva.

Vypíšte čísla všetkých úradníkov, ktorí zaručene nič nerobia a ani nikdy nebudú robiť.

Formálne, nech úradník $u$ môže robiť činnosť $c$. Kedy **môže** tento úradník niečo robiť? Jeden prípad je, ak $c = 1$. Druhý prípad je, ak $c > 1$ a existuje postupnosť úradníkov $u_1, u_2, \ldots, u_t = u$, taká, že platí:

– úradník $u_1$ vykonáva činnosť 1 (styk s verejnosťou),
– pre každé $i < t$ je úradník $u_{i+1}$ priamym nadriadeným úradníka $u_i$
– pre žiadne $i < t$ úradník $u_i$ nevykonáva činnosť $c$.

Ak pre nejakého úradníka nenastáva ani prvý, ani druhý prípad, tak máme istotu, že tento úradník nikdy nič robiť nebude. (Všimnite si, že minister vykonáva všetky činnosti vrátane styku s verejnosťou, preto sa mu môže stať, že niečo robí.)

Pri odhade časovej zložitosti (a analýze, ktoré z možných riešení je ako dobré) predpokladajte, že počet činností $m$ je rádovo menší ako počet úradníkov $n$.

**Formát vstupu:**

V prvom riadku sú čísla $n$ a $m$. V druhom riadku je pre každého úradníka (od 2 do $n$) číslo jeho priameho nadriadeného. V treťom riadku je pre každého úradníka (od 2 do $n$) číslo jeho činnosti.

**Formát výstupu:**
Vypíšte v ľubovoľnom poradí čísla úradníkov, čo nikdy nič nerobia.

**Príklad:**

**Vstup**

```
10 3
1 2 2 2 1 6 6 4 5
2 3 2 2 2 2 1 1 1
```

**Výstup**

```
2 3 7
```

## A-III-3 Grafový počítač pomáha krtkovi

V tomto ročníku OI-A sa v poslednej úlohe stretávame so špeciálnym grafovým počítačom. Jeho popis (rovnaký ako v domácom kole) nájdete v študijnom texte za zadaním tejto úlohy.

**Súťažná úloha:**

a) (3 body) Napíšte funkciu pre grafový počítač, ktorá načíta zo vstupu $n$ prirodzených čísel a na výstup ich vypíše usporiadané od najmenšieho po najväčšie. Môžete predpokladať (a využiť) to, že zadané čísla sa zmestia do dátového typu `Value`.

Riešenie s časovou zložitosťou $\Theta(n \log n)$ môže získať nanajvýš jeden bod, pomalšie riešenia nedostanú žiadne body.

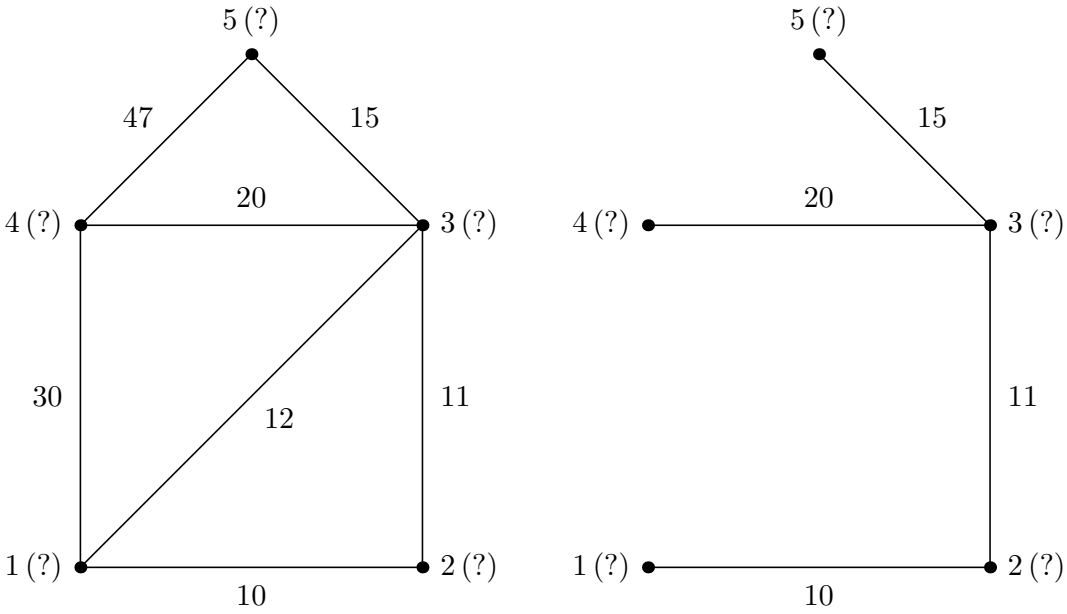
**Príklad.** Pre $n = 5$ a čísla 42, 47, 3, 1, 2 by mala vaša funkcia vypísať na výstup 1, 2, 3, 42, 47.

b) (7 bodov) Malému lenivému krtkovi pri nedávnych dažďoch podmáčalo celú noru. Nora sa kedysi skladala z $n$ brlôžkov a nejakých **navzájom rôzne dlhých** chodbičiek medzi nimi. Po chodbičkách sa dalo (nie nutne priamo) prejsť medzi hociktorými dvoma brlôžkami. Teraz sú ale všetky chodbičky samé blato. Krtko by potreboval **niektoré** chodbičky spevniť, a to tak, aby sa potom cez spevnené chodbičky vedel dostať do všetkých brlôžkov. A keďže je náš krtko lenivý, hľadá takú množinu chodbičiek, ktorá má **najmenšiu celkovú dĺžku**.

Napíšte funkciu pre grafový počítač, ktorá dostane na vstupe graf predstavujúci pôvodnú sieť brlôžkov a chodbičiek a vráti na výstupe jeho podgraf

tvorený len tými chodbičkami, ktoré má krtko spevniť. Vo vstupnom grafe má každá hrana váhu predstavujúcu dĺžku príslušnej chodbičky. Váhy sú kladné a navzájom rôzne.

Aj v tejto podúlohe bude pri hodnotení kladený dôraz na to, či je vaše riešenie efektívnejšie ako riešenia, ktoré sa dajú naprogramovať na klasickom počítači. Nezabudnite uviesť dôkaz správnosti vášho algoritmu.




**Príklad.** Na obrázku vľavo je príklad vstupného grafu. Čísla pri vrcholoch sú ich id, každý vrchol má na začiatku značku `undef`. Čísla pri hranách sú ich váhy. Vpravo je zodpovedajúci výstupný graf. Vo výstupnom grafe môžu byť značky vrcholov a váhy hrán ľubovoľné, zaujíma nás len množina hrán, ktoré obsahuje. Celková dĺžka chodbičiek, ktoré treba spevniť, je 56.

Tabuľku s povolenými inštrukciami nájdete na strane 10, študijný text začína na strane 11.

## A-III-4 Asfaltistan

V Asfaltistane nedávno objavili jedno veľké územie zatiaľ nedotknuté asfaltom. Rozhodli sa to napraviť tak, že krížom cez neho postavia diaľnicu. Minister financií si ale kladie podmienky: diaľnica musí byť naozaj zjazdná automobilmi a zároveň čo najlacnejšia, lebo je kríza a nikto nemá peniaze na rozhadzovanie.

Projektanti si celé územie rozdelili na políčka (štvorce so stranou dlhou 1 km) a pre každé políčko si spočítali jeho priemernú nadmorskú výšku. V našej úlohe tieto nadmorské výšky udávame v takých jednotkách, aby platilo: diaľnica vedúca medzi dvoma susednými políčkami je zjazdná vtedy a len vtedy, ak sa ich nadmorské výšky líšia **nanajvýš o 1**.

Navyše môžu stavať mosty a tunely – mostom sa dajú prepojiť dve políčka v rovnakom stĺpci alebo riadku, ak je ich nadmorská výška zhodná a nadmorská výška **všetkých** políčok medzi nimi je **ostro** menšia. Tunelom sa tiež dajú prepojiť dve políčka s rovnakou nadmorskou výškou v rovnakom stĺpci alebo riadku, ale v tomto prípade nadmorská výška **všetkých** políčok medzi nimi musí byť **ostro** väčšia.

Je povolené, aby na jednom políčku zároveň končil jeden most či tunel a zároveň tam začínal druhý.

**Súťažná úloha:**

Územie má rozmery $r \times s$ km, jeho mapa má teda $r$ riadkov po $s$ stĺpcov. Vašou úlohou je nájsť najlacnejšiu diaľnicu medzi jej ľavým horným a pravým dolným rohom, teda políčkami $(0, 0)$ a $(r-1, s-1)$. Na týchto dvoch políčkach musí viesť diaľnica po povrchu, teda nesmie byť niekde uprostred mosta alebo tunelu.

**Formát vstupu:**

Na vstupe dostanete v prvom riadku celé čísla $r$, $s$, $c$, $m$ a $t$.

Čísla $c$, $m$ a $t$ udávajú cenu za postavenie jedného políčka obyčajnej diaľnice, mosta a tunela. Celkovú cenu cesty zistíme tak, že za každé vnútorné políčko mosta (všetky okrem začiatku a konca) zarátame $m$, za každé vnútorné políčko tunela zarátame $t$ a za každé iné políčko, ktorým cesta vedie, zarátame $c$.

V každom z ďalších $r$ riadkov dostanete $s$ čísel $h_{i,j}$ oddelených medzerou. Tieto čísla predstavujú nadmorské výšky jednotlivých políčok.

Môžete predpokladať, že platia nasledujúce obmedzenia:
$1 \leq r, s \leq 1\,000$, $1 \leq c, m, t \leq 1\,000\,000$ a $0 \leq h_{i,j} \leq 1\,000\,000$.

Niektoré zo vstupov sú **malé**: platí v nich $r, s \leq 50$. Ak váš program správne vyrieši všetky malé vstupy, dostane aspoň 5 bodov.

Niektoré zo vstupov sú **ľahšie**: platí v nich, že existuje najlacnejšia cesta, ktorá neobsahuje žiaden most ani tunel. Ak váš program správne vyrieši všetky ľahšie vstupy, dostane aspoň 5 bodov.

(Niektoré vstupy môžu byť súčasne malé aj ľahšie.)

**Formát výstupu:**

V prvom riadku výstupu vypíšte najmenšiu sumu, za ktorú sa dá postaviť celá diaľnica. V nasledujúcich riadkoch vypíšte jednu možnú trasu, ktorá sa dá za dané peniaze postaviť. V každom riadku vypíšte dvojicu celých čísel $r_i$, $s_i$: riadok a stĺpec políčka, ktorým cesta prechádza. Vynechajte políčka vo vnútri mostov a tunelov.

Ak žiadne riešenie neexistuje, vypíšte namiesto toho jediný riadok výstupu a v ňom slovo „`NEEXISTUJE`“. V prípade, že existuje viacero optimálnych riešení, vypíšte ľubovoľné jedno z nich.

Všimnite si, že celková cena cesty môže byť veľmi **veľká**. Na uloženie celkovej ceny cesty vám nemusí stačiť 32-bitová celočíselná premenná.

Tiež si všimnite, že je teoreticky možné, že sa na optimálnej ceste budú dva mosty alebo tunely križovať. Toto je povolené, naši šikovní inžinieri to zvládnu vyriešiť tak, aby sa autá nezrážali.

**Príklady:**

**Vstup**

```
2 7 1 20 20
5 1 1 5 5 5 5
5 5 5 5 9 9 5
```

*Stavať mosty a tunely je drahé. Najlacnejšia cesta ide po políčkach s výškou 5, obíde jamu aj kopec.*

výstup

```
10
0 0
1 0
1 1
1 2
1 3
0 3
0 4
0 5
0 6
1 6
```

**Vstup**

```
2 7 1 1 20
5 1 1 5 5 5 5
5 5 5 5 9 9 5
```

*Stavať mosty je teraz rovnako lacné ako stavať obyčajnú cestu. Je teda lacnejšie postaviť most z (0,0) na (0,3) ponad jamu ako ju obchádzať.*

výstup

```
8
0 0
0 3
0 4
0 5
0 6
1 6
```

**Vstup**

```
2 7 1 20 1
5 1 1 5 5 5 5
5 5 5 5 9 9 5
```

výstup

```
8
0 0
1 0
1 1
1 2
1 3
1 6
```

*V tomto príklade je zase lacné stavať tunely, preto optimálna cesta vedie tunelom z (1,3) na (1,6).*

**Vstup**

```
3 3 1 1 1
42 32 32
32 22 12
22 12 2
```

**Výstup**

```
NEEXISTUJE
```

*Nevieme sa ani len pohnúť z políčka (0,0).*

## A-III-5 Romantické básničky II

Po tom, ako včera Hermi (aj vďaka vášmu programu!) poobletoval všetky huslistky z orchestra, je naňho jeho priateľka Milka poriadne nasrdená (a teraz raď, Hviezdoslav).

Aby si ju udobril, potrebuje Hermi zložiť najkrajšiu báseň všetkých čias. Preto sa rozhodol, že nenechá nič na náhodu a zloží básničku s pôsobivou štruktúrou – cyklickým združeným rýmom. Báseň je písaná cyklickým združeným rýmom, ak majú rýmy schému *ab bc cd de ... yz za*; t.j. každá sloha má dva verše, pričom sa vždy ten druhý rýmuje s prvým veršom nasledujúcej slohy; navyše druhý verš poslednej slohy sa rýmuje s prvým veršom prvej slohy.

Hermi má skutočne bohatú fantáziu (a tiež váš včerajší program; a tiež sa bojí, že ho Milka zbije), takže už stihol vymyslieť celkový obsah básne aj jej jednotlivé slohy. Pre niektoré slohy už dokonca vymyslel niekoľko rôznych variantov, ktoré síce obsahujú iné dvojice rýmov, ale vyjadrujú v podstate tú istú myšlenku. Stále sa mu však akosi nedarí zo slôh poskladať celú básničku. Ten cyklický rým mu robí problémy. Hermi by potreboval s výberom jednotlivých variantov do výslednej básne pomôcť.

Aby ste si nemuseli lámať hlavu nad tým, ktoré verše se dostatočne rýmujú a ktoré nie, očísloval Hermi všetky možné rýmy prirodzenými číslami. Každý variant slohy potom popísal dvojicou $(x, y)$ obsahujúcou čísla rýmov v oboch veršoch. Za slohou $(x, y)$ se teda môže vyskytovať len sloha tvaru $(y, z)$ pre nejaké $z$.

**Súťažná úloha:**

Daný je počet slôh $n$, pre každú slohu počet variantov $s_i$, ktoré Hermi vymyslel, a pre každý z nich popis rýmov, ktoré ho tvoria. Zistite, či sa dá výberom vhodných variantov zostrojiť báseň, ktorá bude mať cyklický združený rým. Ak áno, jednu takúto báseň zostrojte.

**Formát vstupu:**

Prvý riadok vstupu obsahuje prirodzené číslo $n$, ktoré udáva počet slôh básničky. Nasleduje $n$ skupín riadkov. Skupina $i$ začína riadkom obsahujúcim číslo $s_i$ udávajúce počet variantov pre $i$-tu slohu. Nasleduje $s_i$ riadkov, každý s dvoma číslami $a_{i,j}$ a $b_{i,j}$, popisujúcimi prvý a druhý verš jednotlivých variantov. Môžete predpokladať, že $1 \leq n \leq 1\,000$, $1 \leq s_i \leq 1\,000$ a $1 \leq a_{i,j}, b_{i,j} \leq 10^9$.

Niektoré zo vstupov sú **malé**: platí v nich $n \leq 10$ a $s_i \leq 10$. Ak váš program správne vyrieši všetky malé vstupy, dostane aspoň 5 bodov.

**Formát výstupu:**

Program vypíše na výstup buď jeden riadok s reťazcom „`NEEXISTUJE`“, ak sa básnička z daných variantov nedá zostaviť, alebo $n$ riadkov, pričom $i$-ty riadok bude obsahovať číslo variantu $v_i$ ($1 \leq v_i \leq s_i$), ktorý bol vybratý do $i$-tej slohy. Ak existuje viacero riešení, vypíšte ľubovoľné jedno z nich.

**Príklady:**

| Vstup | Výstup |
|---|---|
| 2<br>1<br>2 1<br>1<br>1 3 | NEEXISTUJE |

**Vstup**

```
5
1
1 1
2
1 5
1 6
2
5 2
6 3
1
3 8
3
10 2
8 1
4 2
```

**Výstup**

```
1
2
2
1
2
```

*Jednotlivé slohy básničky budú mať rýmy:*
*11, 16, 63, 38, 81.*

# Riešenia domáceho kola kategórie A

## A-I-1 Indiana a poklad

Našu úlohu si môžeme jemne zovšeobecniť: našim cieľom bude vymyslieť, ako v pamäti reprezentovať množinu prvkov tak, aby sme vedeli prvky do nej vkladať, vyberať ich z nej a zisťovať aktuálnu hodnotu mediánu.

Ako na to? Keby sme túto úlohu mali riešiť ručne, pravdepodobne by sme skúsili prvky nejak rozhádzať: malé pohádžeme niekam naľavo, veľké napravo a v strede budeme mať aktuálny medián.

Takýto prístup vieme použiť aj v programe. Formálne, budeme mať aktuálnu množinu prvkov rozdelenú na dve časti, ktoré budeme volať „malé prvky“ a „veľké prvky“. Rozdelené budú tak, aby bolo malých rovnako ako veľkých, resp. o jeden viac, ak je dokopy prvkov nepárny počet. A navyše musí samozrejme platiť, že žiaden veľký prvok nesmie byť menší ako niektorý z malých. Teda napr. prvky $(1, 4, 77, 4, 9, 2, 3, 13, 4)$ rozdelíme na $(1, 4, 4, 2, 3)$ a $(77, 9, 13, 4)$. Takéto rozdelenie má užitočnú vlastnosť: vždy platí, že medián všetkých prvkov je najväčší spomedzi všetkých malých prvkov.

Pozrime sa na to, čo sa zmení, ak nejaký nový prvok chceme pridať. Porovnáme ho s aktuálnym mediánom a podľa toho ho pridáme buď medzi malé, alebo medzi veľké prvky. Tým sme ale ešte neskončili. Môže sa totiž stať, že tých prvkov, medzi ktoré sme ho pridali, je teraz priveľa. Aby sme to napravili, môže byť potrebné buď presunúť najväčší malý prvok medzi veľké, alebo najmenší veľký medzi malé.

Odobratie prvku vyzerá takmer rovnako. Podľa porovnania s mediánom zistíme, v ktorej časti sa nachádza, odstránime ho z nej a následne ak je to potrebné presunieme jeden prvok z jednej časti do druhej, aby si opäť zodpovedali ich veľkosti.

Aby sme všetky tieto operácie vedeli robiť efektívne, použijeme na uloženie malých a veľkých prvkov dva vyvažované binárne stromy. (V nižšie uvedenej implementácii je použitý `multiset`, ktorý na rozdiel od `set`u môže obsahovať viacero prvkov s rovnakou hodnotou.) Tie nám umožnia každú potrebnú operáciu spraviť v čase logaritmickom od počtu prvkov.

Ako túto dátovú štruktúru použiť na riešenie pôvodnej úlohy? Veľmi jednoducho: najskôr do nej vložíme prvých $K$ prvkov poľa, no a následne striedavo

vkladáme nový prvok a vyberáme najstarší, až kým neprejdeme celé pole. Takto teda postupne pre každý $K$-prvkový úsek nájdeme jeho medián. Časová zložitosť tohto riešenia je $O(N \log K)$. Naša implementácia má pamäťovú zložitosť $O(N)$, ale je zjavné, ako by sa dala zlepšiť na $O(K)$, keby to bolo potrebné.

### Listing programu:

```
#include <iostream>
#include <vector>
#include <set>
using namespace std;

multiset<int> male, velke;

void fix() {
    multiset<int>::iterator tmp;
    while (velke.size() > male.size()) {
        tmp=velke.begin(); male.insert(*tmp); velke.erase(tmp);
    }
    while (velke.size()+1 < male.size()) {
        tmp=--male.end(); velke.insert(*tmp); male.erase(tmp);
    }
}

int main() {
    int N, K;
    cin >> N >> K;
    vector<int> A(N);
    for (int n=0; n<N; ++n) cin >> A[n];

    for (int n=0; n<K; ++n) velke.insert(A[n]);
    fix();
    int best = *male.rbegin();
    for (int n=0; n<N-K; ++n) {
        if (A[n+K] <= *male.rbegin()) male.insert(A[n+K]);
        else velke.insert(A[n+K]);
        if (A[n] <= *male.rbegin()) male.erase(male.find(A[n]));
        else velke.erase(velke.find(A[n]));
        fix();
        best = max( best, *male.rbegin() );
    }
    cout << best << endl;
}
```

### Haldy namiesto stromov:

Namiesto vyvažovaných stromov môžeme použiť dve haldy – na uloženie malých prvkov haldu na vrchu s maximom, na uloženie veľkých haldu na vrchu s minimom. Jediný problém potom predstavuje odstraňovanie prvkov. To môžeme riešiť dvoma spôsobmi: Jedna možnosť je, že ku každému indexu do vstupného poľa si budeme udržiavať ukazovateľ na miesto, kde sa v jednej z háld dotyčný prvok práve nachádza. Pomocou tohto ukazovateľa potom ľahko dotyčný prvok z haldy vo vhodnej chvíli odstránime.

Druhá možnosť je zapamätať si len boolovskú premennú hovoriacu, že tento prvok už „nežije“. Neodstránime ho hneď, ale až keď sa nám náhodou niekedy ocitne na vrchu haldy. Pri takejto implementácii navyše potrebujeme ku každej halde jedno počítadlo, ktoré nám bude hovoriť, koľko „živých“ prvkov halda momentálne obsahuje.

**Vlastný vyvažovaný strom:**

Ak by sa nám chcelo implementovať si vlastný vyvažovaný binárny vyhľadávací strom, dostávame tretie možné riešenie. Potom si totiž stačí v každom vrchole stromu pamätať navyše informáciu, koľko prvkov sa nachádza v jeho podstrome. Pomocou tejto informácie vieme pre ľubovoľné $x$ v logaritmickom čase nájsť v strome $x$-tý najmenší prvok.

Nemuseli by sme sa teda vôbec obťažovať s rozdeľovaním prvkov na malé a veľké. Jednoducho by sme všetkých $K$ prvkov nahádzali do nášho stromu a potom sa pozreli na $\lceil K/2 \rceil$-ty najmenší prvok.

Kompromis medzi týmto prístupom a prvým vzorovým riešením sa dá implementovať v C++ pomocou STL. Všetky prvky uložíme do jednej usporiadanej množiny, pričom si budeme udržiavať iterátor ukazujúci na aktuálny medián. Podľa toho, či vkladáme/vyberáme prvok menší/väčší ako aktuálny medián vieme v každom kroku výpočtu povedať, kam iterátor posunúť.

**Listing programu:**

```
#include <algorithm>
#include <iostream>
#include <vector>
#include <set>
using namespace std;

typedef pair<int,int> zaznam;

class median_set {
    set<zaznam> data;
    set<zaznam>::iterator current;
    unsigned less_than_current;
    void fix();
  public:
    median_set();
    void insert(zaznam z);
    void erase(zaznam z);
    int median();
};

median_set::median_set() { less_than_current=0; current=data.end(); }
int median_set::median() { return current->first; }

void median_set::fix() {
  while (less_than_current < (data.size()-1)/2) ++less_than_current, ++current;
  while (less_than_current > (data.size()-1)/2) --less_than_current, --current;
}
```

```
void median_set::insert(zaznam z) {
  if (data.empty() || z < *current) ++less_than_current;
  data.insert(z);
  fix();
}

void median_set::erase(zaznam z) {
  if (z < *current) --less_than_current;
  if (z == *current) ++current;
  data.erase(z);
  fix();
}

int main() {
  int N, K;
  cin >> N >> K;
  vector<int> A(N);
  for (int n=0; n<N; ++n) cin >> A[n];

  median_set MS;
  for (int n=0; n<K; ++n) MS.insert(zaznam(A[n],n));
  int best = MS.median();
  for (int n=K; n<N; ++n) {
    MS.insert( zaznam(A[n],n) );
    MS.erase( zaznam(A[n-K],n-K) );
    best = max( best, MS.median() );
  }
  cout << best << endl;
}
```

**Intervalový strom:**

V poradí štvrté riešenie s približne rovnakou časovou zložitosťou využíva jednoduchú dátovú štruktúru: intervalový strom. Na začiatku si prvky poľa usporiadame a prečíslujeme na 1 až (nanajvýš) $N$. Následne robíme to isté ako v predchádzajúcom riešení, ale na ukladanie prvkov a hľadanie $x$-tého najmenšieho použijeme namiesto „plnohodnotného“ vyvažovaného stromu náš intervalový strom. Na záver už len získanú hodnotu mediánu prevedieme na pôvodnú hodnotu spred prečíslovania.

**Úplne iné riešenie: vyhľadávanie odpovede:**

Na zadanú úlohu sa môžeme pozrieť aj z úplne inej strany. Našou úlohou je nájsť najväčšie $M$ také, že niekde v poli sa nachádza úsek $K$ prvkov s mediánom aspoň $M$. Častokrát keď nevieme, ako nejaký objekt *nájsť*, môžeme sa najskôr zamyslieť nad potenciálne ľahším problémom: ako *overiť*, či nejaký objekt existuje. V našom prípade si teda položme otázku: Ako vieme pre dané $M$ overiť, či v našom poli existuje nejaký $K$-prvkový úsek s mediánom aspoň $M$?

Pri zodpovedaní tejto otázky nás vôbec nezaujímajú konkrétne hodnoty v poli. Jediné, čo je podstatné, je, ktoré z nich sú menšie ako $M$ (tie sú zlé) a

ktoré sú aspoň rovné $M$ (tie sú dobré). Aby sme mali medián aspoň $M$, musíme nájsť úsek, v ktorom je aspoň $\lceil K/2 \rceil$ dobrých čísel.

Ak si zlé čísla nahradíme nulami a dobré jednotkami, dostávame jednoduchú úlohu: existuje v novom poli úsek dĺžky $K$ so súčtom aspoň $\lceil K/2 \rceil$? Túto otázku zodpovieme hravo v lineárnom čase.

Ako nám však toto pomôže pri riešení pôvodnej úlohy? Jednoducho: ak vieme pre ľubovoľné $M$ *overiť*, či existuje úsek s mediánom aspoň $M$, môžeme na nájdenie optimálnej hodnoty $M$ použiť *binárne vyhľadávanie*.

Takéto riešenie má časovú zložitosť $O(N \log X)$, kde $X$ je rozdiel medzi najväčšou a najmenšou hodnotou na vstupe. Spomedzi všetkých programov uvedených v tomto vzorovom riešení je tento najjednoduchší.

**Listing programu:**

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    int N, K;
    cin >> N >> K;
    vector<int> A(N);
    for (int n=0; n<N; ++n) cin >> A[n];
    int lo = *min_element(A.begin(), A.end());
    int hi = 1 + *max_element(A.begin(), A.end());
    while (hi - lo > 1) {
        int med = (hi+lo)/2, sum = 0, good = 0;
        for (int n=0; n<K; ++n)
            sum += (A[n] >= med), good |= (sum >= (K+1)/2);
        for (int n=K; n<N; ++n)
            sum += (A[n] >= med) - (A[n-K] >= med), good |= (sum >= (K+1)/2);
        if (good) lo=med; else hi=med;
    }
    cout << lo << endl;
}
```

## A-I-2 Ešte raz poklad

Úlohu si môžeme preformulovať nasledovne: koľkými spôsobmi sa dá daná chodba vydláždiť dlaždicami rozmerov $2 \times 1$?

Ako už býva pri podobných kombinatorických úlohách zvykom, nemôžeme si dovoliť generovať všetky možné dláždenia, lebo ich počet môže rásť až exponenciálne v závislosti od dĺžky chodby.

Vzorové riešenie bude používať metódu dynamického programovania.

Existuje veľa spôsobov, ako dostať riešenie s časovou zložitosťou lineárnou od dĺžky chodby, teda $O(N)$. My popíšeme riešenie, ktoré síce nie je najrýchlejšie, ale je dostatočne ľahké na implementáciu.

**Približný nápad:**

Riešiteľovi, ktorý už má aké-také skúsenosti s dynamickým programovaním, by malo byť jasné, ako *približne* bude riešenie vyzerať. Začneme tým, že spočítame, koľkými spôsobmi sa dá vydláždiť prvý stĺpec. Potom prvé dva. Potom prvé tri. A tak ďalej. Keď už budeme poznať počet dláždení pre prvých $k$ stĺpcov, použijeme túto informáciu pri výpočte počtu dláždení prvých $k+1$ stĺpcov.

Teda, tak nejak by sme si to predstavovali. Nebude to ale také jednoduché.

V čom je problém? Informácia o počte spôsobov, ako vydláždiť prvých $k$ stĺpcov, nám pri výpočte toho istého pre $k+1$ stĺpcov nestačí. My totiž môžeme v stĺpci číslo $k+1$ položiť nejaké dlaždice vodorovne. A tie budú potom zasahovať do predchádzajúceho stĺpca. Takže to, čo nám ostane po vydláždení stĺpca $k+1$, nie je prvých $k$ stĺpcov chodby. Prvých $k-1$ je nedotknutých, ale v $k$-tom môžu už oproti pôvodnému stavu byť niektoré políčka pokryté.

**Opravený nápad:**

Podproblémy, ktoré budeme riešiť, budú nasledujúceho tvaru: „Koľkými spôsobmi sa dá vydláždiť prvých $k$ stĺpcov chodby, ak z posledného stĺpca vyškrtnem tieto konkrétne políčka?“

Každý takýto podproblém vieme vyriešiť v konštantnom čase, ak už poznáme riešenie pre podproblémy s o 1 menším $k$. Jednoducho vyskúšame všetky možnosti, ako môžu byť uložené dlaždice $2 \times 1$, ktoré zasahujú do posledného stĺpca. To, čo nám zostane nevydláždené, zakaždým zodpovedá nejakému podproblému s o 1 menším $k$.

**Dláždenia posledného stĺpca:**

V ľubovoľnom podprobléme existujú nanajvýš tri spôsoby, ako môže byť vydláždený posledný stĺpec:

1. prvé dva riadky sú pokryté jednou „zvislou“ dlaždicou
2. posledné dva riadky sú pokryté jednou „zvislou“ dlaždicou
3. nepoužijeme žiadnu „zvislú“ dlaždicu

V každej možnosti sú navyše všetky políčka posledného stĺpca, ktoré neobsahujú prekážku ani zvislú dlaždicu, pokryté vodorovnými dlaždicami. V niektorých situáciách sa môže stať, že niektoré z týchto troch spôsobov nebudú

prípustné. Celkový počet dláždení získame tak, že sčítame počty dláždení zodpovedajúce prípustným spôsobom pokrytia posledného stĺpca.

**Bitové masky:**

Na jednoduchšiu prácu s podproblémami používame v našom programe bitové masky. To, ktoré políčka v poslednom stĺpci ponechať a ktoré zakázať, vieme popísať 3 bitmi – pre každý riadok jeden bit, 0 znamená zakázať, 1 ponechať. A na tieto 3 bity sa vieme pozerať ako na číslo od 0 do 7.

Máme teda $8N$ podproblémov, každý z nich je určený počtom stĺpcov $n$ a bitovou maskou $mask$ z rozsahu od 0 do 7, ktorá nám v poslednom stĺpci nejaké políčka (navyše oproti vstupu) zakáže.

Implementáciu si tiež zjednodušíme nasledovne: Pri načítaní vstupu si vytvoríme mapu celej chodby, v ktorej 0 bude voľné políčko a 1 prekážka. Vždy, keď spracúvame konkrétny podproblém, tak si do mapy číslom 2 zaznačíme políčka zakázané aktuálnou maskou. A keď daný podproblém doriešime, 2-ky po sebe zase zmažeme.

Posledné zjednodušenie, ktoré sme spravili, je nasledovné pri prvých dvoch spôsoboch dláždenia sa neodvoláme na podproblém s menej stĺpcami. Len položíme zvislú dlaždicu, čím dostaneme podproblém rovnakej veľkosti, ale s menšou bitovou maskou. Ten sme už predtým spracovali, takže všetko funguje ako má.

**Listing programu:**

```
#include <iostream>
using namespace std;

int N, K, MOD;
char A[3][10000000];
int sol[10000000][8];

int main() {
    cin >> N >> K >> MOD;
    for (int k=0; k<K; ++k) { int x,y; cin >> x >> y; A[x-1][y-1]=1; }
    sol[0][7] = 1;
    for (int c=0; c<N; ++c) for (int mask=0; mask<8; ++mask) {
        sol[c+1][mask] = 0;

        // pridame do mapy policka zakazane maskou
        for (int r=0; r<3; ++r) if (!A[r][c]) if (!(mask & 1<<r)) A[r][c]=2;

        // prve dve moznosti dlazdenia: je pouzita zvisla dlazdica
        if (!A[0][c] && !A[1][c]) sol[c+1][mask] += sol[c+1][mask & 4];
        if (!A[1][c] && !A[2][c]) sol[c+1][mask] += sol[c+1][mask & 1];

        // posledna moznost: vsetky volne policka pokryte vodorovnymi dlazdicami
        int lmask = 7;
        bool ok = true;
        for (int r=0; r<3; ++r)
            if (!A[r][c]) { lmask ^= 1<<r; if (c==0 || A[r][c-1]) ok=false; }
        if (ok) sol[c+1][mask] += sol[c][lmask];
```

```
        // upravime vysledok a upraceme po sebe
        sol[c+1][mask] %= MOD;
        for (int r=0; r<3; ++r) if (A[r][c]==2) A[r][c]=0;
    }
    cout << sol[N][7] << endl;
}
```

**Efektívnejšie riešenie:**

Pri rovnakom obmedzení na počet zakázaných dlaždíc je úloha riešiteľná ešte pre omnoho väčšie dĺžky chodby. Ak si vyfarbíme stĺpce, ktoré obsahujú aspoň jednu prekážku, dostaneme medzi nimi prázdne obdĺžniky. A pre prázdny obdĺžnik sa počet dláždení ráta ľahko.

Presnejšie by takéto riešenie vyzeralo napr. tak, že budeme mať nový typ podproblémov: obdĺžnik, ktorý je určený svojou dĺžkou a dvomi bitovými maskami popisujúcimi jeho okraje. Pre každú dĺžku, ktorá je mocninou dvoch, a každé dve bitové masky spočítame počet dláždení takéhoto obdĺžnika. Z tejto informácie potom vieme počet dláždení ľubovoľnej voľnej oblasti zistiť v čase logaritmickom od jej dĺžky.

Toto riešenie samozrejme pri obmedzeniach v zadaní nebolo potrebné implementovať.

## A-I-3 Rieka

Na vyriešenie tejto úlohy potrebujeme dátovú štruktúru, ktorá okrem základných operácií (vloženie a výber prvku) podporuje aj operáciu *nasledovník* – pre ľubovoľnú hodnotu $x$ potrebujeme vedieť v našej dátovej štruktúre efektívne nájsť najmenší prvok väčší ako $x$.

Toto vzorové riešenie začneme popisom niekoľkých pomalých riešení. Pokračovať budeme popisom dátových štruktúr, ktoré budú všetky operácie vedieť realizovať v čase logaritmickom od počtu prvkov. Potom ukážeme, ako sa dá dosiahnuť lepšia časová zložitosť v prípade, že vieme, že všetky prvky patria do malého rozsahu (v našom prípade ide o celé čísla od 1 po $N$).

**Dve pomalé riešenia:**

Prvé triviálne riešenie: použijeme pole, kde si pre každý prvok budeme pamätať, či sa v uloženej množine nachádza alebo nie. Vkladať a vyberať prvky tak vieme v konštantnom čase. Pomalá je však operácia nasledovníka – v najhoršom prípade prejdeme celé pole, kým ho nájdeme. Hľadanie nasledovníka má teda lineárnu časovú zložitosť.

„Opačné“ triviálne riešenie: použijeme navyše aj pole, kde si pre každú hodnotu budeme pamätať jej aktuálneho nasledovníka. Pomocou neho budeme vedieť nasledovníka, pochopiteľne, hľadať v konštantnom čase. Pokazí nám to však operácie vkladania a výberu, tie teraz budú v najhoršom prípade potrebovať až lineárny čas. Totiž keď napr. vložíme nový prvok $x$, tak toto $x$ sa stane nasledovníkom pre všetky hodnoty od $y$ po $x-1$ (kde $y$ je najväčší z prvkov menších ako $x$).

**Kombinácia dvoch pomalých riešení do lepšieho:**

Aby sme využili výhody oboch triviálnych riešení, rozdelíme si pole na $\sqrt{N}$ úsekov dĺžky $\sqrt{N}$. Pre každý úsek si budeme pamätať, koľko prvkov obsahuje.

Vkladanie a výber budú naďalej v konštantnom čase – okrem zaznačenia si nového prvku do poľa zvýšime/znížime počítadlo pre úsek, do ktorého patrí.

Hľadanie nasledovníka hodnoty $x$ vieme spraviť v čase $O(\sqrt{N})$: Najskôr v čase $O(\sqrt{N})$ zistíme, či má $x$ nasledovníka vo svojom vlastnom úseku. Ak nie, v čase $O(\sqrt{N})$ nájdeme nasledujúci neprázdny úsek a taktiež v čase $O(\sqrt{N})$ v ňom nájdeme najmenší prvok.

**Vyvažovaný strom:**

Základnou v praxi používanou dátovou štruktúrou podporujúcou hľadanie nasledovníka je vyvažovaný binárny vyhľadávací strom. Hľadanie nasledovníka vyzerá takmer rovnako ako hľadanie prvku s danou hodnotou.

V C++ máme vyvažovaný binárny strom k dispozícii v dátovej štruktúre `set`. Ten má metódu `upper_bound(x)`, ktorá vráti iterátor na najmenší prvok väčší ako `x`. Rovnako ako vkladanie a výber, aj táto metóda má časovú zložitosť logaritmickú od počtu prvkov. Pomocou nej dostávame program na pár riadkov.

**Listing programu:**

```
#include <iostream>
#include <set>
using namespace std;
int main() {
    int N, P, M; cin >> N >> P >> M;
    set<int> pily;
    while (P--) { int p; cin >> p; pily.insert(p); }
    while (M--) {
        char cmd; int loc; cin >> cmd >> loc;
        if (cmd=='Z') pily.erase(loc);
        if (cmd=='K') pily.insert(loc);
        if (cmd=='V') {
            set<int>::iterator it = pily.upper_bound(loc);
            cout << ( (it == pily.end()) ? 0 : *it ) << endl;
        }
    }
}
```

**Intervalový strom:**

Rovnakú časovú zložitosť vieme dosiahnuť aj použitím iných dátových štruktúr. Ľahko sa implementuje napríklad intervalový strom. Na túto úlohu stačí použiť jeho najjednoduchšiu verziu.

Intervalový strom je úplný binárny strom, ktorého hĺbku $H$ zvolíme tak, aby mal aspoň $N$ listov. Každý vrchol tohto stromu bude zodpovedať nejakému intervalu prvkov. Každý list predstavuje jeden prvok, list číslo $x$ teda zodpovedá intervalu dĺžky 1. Vnútorný vrchol stromu zodpovedá zjednoteniu intervalov, ktorým zodpovedajú jeho deti.

V každom vrchole intervalového stromu, vrátane listov, si pamätáme, koľko prvkov z uloženej množiny leží v jeho intervale.

Vkladanie prvku je jednoduché: začneme v príslušnom liste a po ceste do koreňa zvyšujeme všetky pamätané hodnoty o 1. Výber prvku funguje presne opačne. No a hľadanie nasledovníka hodnoty $x$ sa napríklad dá implementovať tak, že začneme v liste zodpovedajúcom hodnote $x$. Z neho ideme dohora, teda smerom ku koreňu, až kým sa nedostaneme do situácie, že sme práve prišli do vrcholu z jeho ľavého syna a zároveň v podstrome pod pravým synom sa nachádza aspoň jeden prvok. Prejdeme do dotyčného pravého syna a od tejto chvíle pôjdeme v strome dodola až kým nenájdeme najmenší prvok v jeho podstrome.

(Vyššie popísaný algoritmus je aj implementovaný v nižšie uvedenom zdrojovom kóde. Nie je to však jediný možný prístup. Iné, rovnako dobré riešenie: najskôr zistíme počet $p$ prvkov menších alebo rovných $x$. Následne začínajúc v koreni nájdeme prvok s poradovým číslom $p + 1$.)

Všetky potrebné operácie vieme spraviť v čase $O(\log N)$. Na pamätanie si intervalového stromu nám stačí obyčajné pole, strom doň uložíme rovnako ako ukladáme haldu.

**Listing programu:**

```
#include <iostream>
using namespace std;

int T[1<<21], H=20; // intervalovy strom
void update(int x, int zmena) { x += 1<<H; while (x) T[x]+=zmena, x/=2; }

int successor(int x) {
  x += 1<<H;
  while (x%2==1 || T[x+1]==0) { x/=2; if (x==0) return 0; }
  ++x;
  while (x<(1<<H)) if (T[2*x]) x=2*x; else x=2*x+1;
  return x-(1<<H);
}

int main() {
  int N, P, M; cin >> N >> P >> M;
  while (P--) { int p; cin >> p; update(p,1); }
```

```
  while (M--) {
    char cmd; int loc;
    cin >> cmd >> loc;
    switch (cmd) {
      case 'Z': update(loc,-1); break;
      case 'K': update(loc,1); break;
      case 'V': cout << successor(loc) << endl; break;
    }
  }
}
```

**Na ceste ku van Emde Boasovmu stromu:**

Vzorovým riešením tejto úlohy je dátová štruktúra, ktorú vymyslel Holanďan Peter van Emde Boas v roku 1977. Ideovo to bude „kríženec“ medzi intervalovým stromom a riešením v $O(\sqrt{N})$, ktoré sme si predviedli vyššie.

Začnime tým, že sa poriadne pozrieme na naše riešenie s časovou zložitosťou $O(\sqrt{N})$. Zopakujme, že vkladanie a výber prvku v ňom boli ľahké. To jediné zaujímavé bolo hľadanie nasledovníka, ktoré fungovalo nasledovne:

Hľadanie nasledovníka hodnoty $x$ vieme spraviť v čase $O(\sqrt{N})$: Najskôr v čase $O(\sqrt{N})$ zistíme, či má $x$ nasledovníka vo svojom vlastnom úseku. Ak nie, v čase $O(\sqrt{N})$ nájdeme nasledujúci neprázdny úsek a taktiež v čase $O(\sqrt{N})$ v ňom nájdeme najmenší prvok.

Čo že sa to tu vlastne udialo? Zobrali sme pôvodnú úlohu „nájsť nasledovníka v poli dĺžky $N$“ a previedli sme ju na tri menšie úlohy. Prvá aj tretia z nich sú priamo „nájsť nasledovníka v poli dĺžky $\sqrt{N}$“. A tá druhá je to isté, len prvky, medzi ktorými hľadáme nasledovníka, sú tentokrát jednotlivé úseky.

Čo keby sme teda skúsili na riešenie týchto podúloh rekurzívne použiť ten istý prístup? Každý úsek dĺžky $Z = \sqrt{N}$ by sme rozdelili na $\sqrt{Z}$ podúsekov dĺžky $\sqrt{Z}$, a tak ďalej, až kým sa nedostaneme k jednoprvkovým úsekom.

Dobrá správa je, že strom, ktorý takto dostaneme, bude mať málo vrstiev. Ukážme si to najskôr na príklade: Nech $N = 2^{32}$, teda čísla, ktoré budeme spracúvať, zodpovedajú tomu, čo môžeme uložiť do štandardnej 32-bitovej celočíselnej premennej.

Celý úsek dĺžky $2^{32}$ rozdelíme na úseky dĺžky $2^{16}$. Každý z nich budeme mať rozdelený na úseky dĺžky $2^8$, tie na úseky dĺžky $2^4$, tie na úseky dĺžky $2^2$, a tie na úseky dĺžky $2^1$. Celý strom má teda len 5 vrstiev. Pre porovnanie, intervalový strom by ich mal 32.

Vo všeobecnosti platí, že ak $N = 2^{2^K}$, tak náš strom bude mať $K$ vrstiev. Inými slovami, počet vrstiev $K = \log_2 \log_2 N$.

Zlá správa je, že rekurzívny algoritmus, ktorý sme vymysleli, je ešte príliš pomalý. V najhoršom prípade na vyriešenie problému veľkosti $N$ potrebuje tri

rekurzívne volania na problém veľkosti $\sqrt{N}$. Časová zložitosť takéhoto algoritmu teda spĺňa rekurenciu $T(N) = 3T(\sqrt{N}) + O(1)$, z čoho vieme odvodiť, že $T(n) = O((\log N)^{1.585})$. Detaily tohto výpočtu vynecháme, keďže nie sú potrebné pre naše vzorové riešenie. Povšimnite si ale, že sme dostali riešenie, ktoré je horšie ako to s vyváženým stromom.

**Vylepšená verzia:**

Ak chceme rýchlejšie riešenie, potrebujeme sa zbaviť niektorých rekurzívnych volaní. Zmena, ktorou to dosiahneme, je jednoduchá: v každom vrchole stromu, ktorý nie je prázdny, si budeme pamätať najväčší aj najmenší prvok, ktoré sú v ňom uložené. Pomocou tejto informácie navyše teraz vieme lepšie hľadať nasledovníka hodnoty $x$:

1. Porovnáme $x$ s najväčším prvkom uloženým v jeho úseku.
2. Ak nenastala rovnosť, vieme, že nasledovník sa nachádza v tom istom úseku ako $x$. Preto ho nájdeme jediným rekurzívnym volaním na príslušný úsek.
3. Ak nastala rovnosť, vieme, že nasledovník sa nachádza v nasledujúcom neprázdnom úseku. Rekurzívnym volaním nájdeme najbližší neprázdny úsek.
4. Vrátime minimum v práve nájdenom úseku.

Na každej úrovni stromu tento algoritmus urobí jedno rekurzívne volanie a konštantne veľa iných operácií. Časovú zložitosť tohto algoritmu je teda priamo úmerná počtu vrstiev v strome, teda $O(\log \log N)$.

Potrebujeme si ešte rozmyslieť presnú realizáciu vkladania a vymazávania prvkov. Obe sa dajú implementovať v časovej zložitosti $O(\log \log N)$.

**Listing programu:**

```
#include <iostream>
#include <vector>
using namespace std;

class vanEmdeBoasTree {
  public:
    vector<vanEmdeBoasTree*> sub; // O(sqrt(N)) podstromov
    vanEmdeBoasTree* summary;     // jeden podstrom s informaciami o blokoch
    int bits;      // pocet bitov zodpovedajuci tomuto stromu
    int smallest;  // min cislo v strome (ulozene len tu, -1 ak je prazdny)
    int largest;   // max cislo v strome (ulozene aj v podstrome, ak >smallest)

    vanEmdeBoasTree(int bits);
    void insert(int x);
    void erase(int x);
    int successor(int x);
```

```
};

// konstruktor len inicializuje vsetko na prazdne
vanEmdeBoasTree::vanEmdeBoasTree(int bits) : bits(bits) {
  summary = NULL;
  sub.resize(1 << (bits/2), NULL);
  smallest = largest = -1;
}

// vkladanie prvku
void vanEmdeBoasTree::insert(int x) {
  // ak este nemame ziaden, len ho dame do smallest
  if (smallest == -1) { smallest = largest = x; return; }

  // pozrieme ci ho nevymenit so smallest a ci neupravit largest
  if (x < smallest) swap(smallest,x);
  if (x > largest) largest = x;
  if (bits==1) return;

  int nbits=bits/2;
  int hix = x >> nbits, lox = x ^ (hix << nbits);

  if (!sub[hix]) sub[hix] = new vanEmdeBoasTree(bits/2);
  if (sub[hix]->smallest >= 0) {
    // uz v tom podstrome nieco je
    sub[hix]->insert(lox); // toto je v O(log log U)
  } else {
    // este v tom podstrome nic nie je, treba aj summary updatovat
    sub[hix]->insert(lox); // toto je v konstantnom case
    if (!summary) summary = new vanEmdeBoasTree(bits/2);
    summary->insert(hix); // toto je v O(log log U)
  }
}

// vyber prvku
void vanEmdeBoasTree::erase(int x) {
  // ak uz sme v liste, ten vyriesime v konstantnom case
  if (bits==1) {
    if (smallest==x && smallest==largest) { smallest=largest=-1; return; }
    if (smallest==x && smallest<largest) { smallest=largest; return; }
    largest=smallest; return;
  }

  // ak prave odstranujeme minimum, treba ho nahradit nasledujucim prvkom
  if (x==smallest) {
    // trivialny pripad -- nic ine nezostalo
    int tmp = summary ? summary->smallest : -1;
    if (tmp==-1) { smallest=largest=-1; return; }
    // vseobecny pripad -- najdeme nasledujuci prvok, dame ho do smallest
    // a zmenime x nan, nech ho nasledne zmazeme z podstromu
    int s = sub[tmp]->smallest;
    x = smallest = (tmp << (bits/2)) + s;
  }

  int nbits=bits/2;
  int hix = x >> nbits, lox = x ^ (hix << nbits);

  // odstranime x z prislusneho podstromu, ak treba upravime aj summary
  sub[hix]->erase(lox);
  if (sub[hix]->smallest == -1) summary->erase(hix);

  // a este sa nam mohol zmenit largest, preto ho prepocitame
```

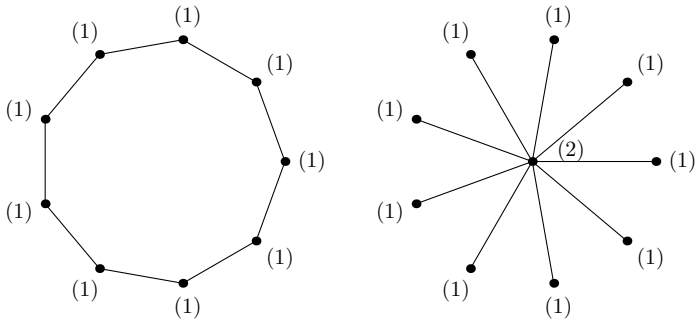
```
  if (summary->largest == -1) largest=smallest;
  else largest = ((summary->largest) << nbits) + sub[summary->largest]->largest;
}

// hladanie nasledovnika
int vanEmdeBoasTree::successor(int x) {
  // osetrime okrajove pripady
  if (x < smallest) return smallest;
  if (x >= largest) return 0; // nenasiel
  if (bits==1 && x<largest) return largest;

  int nbits=bits/2;
  int hix = x >> nbits, lox = x ^ (hix << nbits);

  // ak este mame v tom istom useku ako x nieco vacsie od x, staci hladat tam
  if (sub[hix] && lox < sub[hix]->largest)
    return sub[hix]->successor(lox) + (hix << nbits);

  // a ak nie, najdeme nasledujuci neprazdny usek a vratime jeho minimum
  int suc = summary->successor(hix);
  return (suc << nbits) + sub[suc]->smallest;
}

int main() {
  vanEmdeBoasTree *vEB = new vanEmdeBoasTree(16);
  int N, P, M; cin >> N >> P >> M;
  while (P--) { int p; cin >> p; vEB->insert(p); }
  while (M--) {
    char cmd; int loc;
    cin >> cmd >> loc;
    switch (cmd) {
      case 'Z': vEB->erase(loc); break;
      case 'K': vEB->insert(loc); break;
      case 'V': cout << vEB->successor(loc) << endl; break;
    }
  }
}
```

## A-I-4 Grafový počítač

**Konštrukcia ruského kola z kružnice a hviezdy:**

Podľa študijného textu už vieme v logaritmickom čase zostrojiť cestu s $n-1$ hranami a $n$ vrcholmi a z nej kružnicu s $n$ vrcholmi.

Teraz ešte potrebujeme doplniť stredný vrchol a hrany z neho. To spravíme v dvoch krokoch: najskôr (podobným spôsobom ako pri ceste) zostrojíme chýbajúcu časť grafu, tzv. hviezdu, a následne vhodným volaním `Join` spojíme kružnicu a hviezdu do ruského kola.




Kružnica a hviezda so značkami vrcholov pripravenými na `Join`.

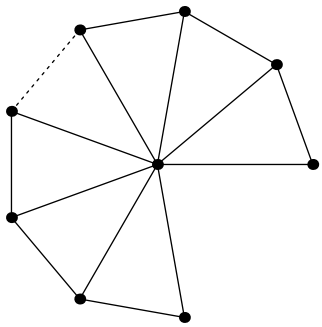
Aj kružnicu, aj hviezdu vieme zostrojiť v čase $O(\log n)$, ich spojenie prebehne v konštantnom čase, celková časová zložitosť je teda logaritmická od $n$.

## Listing programu:

```
function cesta(n: longint): Graph;
{ zostroji cestu s n hranami, implementacia je v studijnom texte }

function hviezda(n: longint): Graph;
{ zostroji hviezdu s n vrcholmi na obvode + jednym v strede }
var vysledok : Graph;
begin
   if n=1 then begin
      vysledok := EmptyG;
      { vyrobime stredny vrchol, ma znacku 1 kvoli spajaniu }
      AddV(vysledok,1);
      { vyrobime jediny vrchol na obvode a hranu don }
      AddV(vysledok,undef);
      AddE(vysledok,1,2,undef);
   end else begin
      { zostrojime hviezdu polovicnej velkosti }
      vysledok := hviezda(n div 2);
      { spojime dve jej kopie za stredny vrchol }
      vysledok := Join(vysledok, vysledok, value_defined, none);
      { pre neparne n este pridame jeden vrchol na obvod }
      if n mod 2 = 1 then begin
         AddV(vysledok,undef);
         AddE(vysledok,1,CountV(vysledok),undef);
      end;
   end;
   hviezda := vysledok;
end;

function ruske_kolo(n : longint) : Graph;
var G, H : Graph;
begin
   { zostrojime kruznicu a hviezdu }
   G := cesta(n-1); AddE(G,1,n,undef); H := hviezda(n);
   { preznacime ich aby sa dali spojit }
   SetAllV(G,1); SetAllV(H,1); SetV(H,1,2);
   { spojime ich }
   ruske_kolo := Join(G,H,value,none);
end;
```

**Konštrukcia ruského kola z reťaze trojuholníkov:**

Keď z ruského kola odstránime jednu z hrán na obvode, môžeme si všimnúť, že nám zostala reťaz na seba nadväzujúcich trojuholníkov. Tú vieme priamo zostrojiť podobne ako sme v študijnom texte zostrojovali cestu: ak chceme reťaz, ktorá má na obvode $2k$ vrcholov, zostrojíme reťaz, ktorá ich má $k$, dve jej kópie spojíme dokopy príkazom `Join` (jedine stredový vrchol bude spoločný) a pridáme jednu hranu na obvode.



Spojenie dvoch reťazí do jednej s dvakrát toľko vrcholmi na obvode kruhu.

Takto v logaritmickom čase zostrojíme ruské kolo bez jednej hrany a následne pridáme chýbajúcu hranu a sme hotoví.

## Listing programu:

```
function retaz(n: longint): Graph;
{ zostroji retaz trojuholnikov s 1 vrcholom v strede a n na obvode }
var vysledok : Graph;
begin
   if n=1 then begin
      vysledok := EmptyG;
      { vyrobime stredny vrchol, ma znacku 1 kvoli spajaniu }
      AddV(vysledok,1);
      { vyrobime jediny vrchol na obvode a hranu don }
      AddV(vysledok,undef);
      AddE(vysledok,1,2,undef);
   end else begin
      { vyrobime retaz s polovicnym poctom vrcholov na obvode }
      vysledok := retaz(n div 2);
      { spojime dve jej kopie }
      vysledok := Join(vysledok, vysledok, value_defined, none);
      AddE(vysledok, (n div 2)+1, (n div 2)+2, undef);
      { pre neparne n pridame este jeden trojuholnik }
      if n mod 2 = 1 then begin
         AddV(vysledok,undef);
         AddE(vysledok,1,n+1,undef);
         AddE(vysledok,n,n+1,undef);
      end;
   end;
   retaz := vysledok;
end;

function ruske_kolo(n : longint) : Graph;
```

```
var vysledok : Graph;
begin
   vysledok := retaz(n);
   AddE(vysledok, 2, n+1, undef);
   ruske_kolo := vysledok;
end;
```

**Hľadanie najkratšej cesty v lineárnom čase:**

Základný trik potrebný na riešenie úlohy bol ukázaný v príklade 2 v študijnom texte: využijeme to, že ak má funkcia `Find` na výber viac možností, nájde tú z nich s najmenším súčtom váh hrán. Ak teda budeme pomocou `Find` v zadanom grafe $G$ vyhľadávať nejakú cestu $C$, nájdeme najlacnejšiu zo všetkých ciest, ktoré sú s $C$ izomorfné. Do hotového riešenia nám už chýbajú len dva technické detaily.

1. Potrebujeme zaistiť, aby s $C$ boli izomorfné len cesty vedúce z $x$ do $y$.
2. Dve cesty sú izomorfné len vtedy, ak majú rovnaký počet hrán. My však nevieme, koľko hrán má najkratšia cesta z $x$ do $y$.

Prvý problém vyriešime nasledovne: V grafe $G$ priradíme vrcholu $x$ značku 1, vrcholu $y$ značku 2. V ceste $C$ priradíme jednému koncovému vrcholu značku 1, druhému 2. Všetky ostatné vrcholy budú mať značku `undef`. Pri testovaní izomorfizmu použijeme `veq=value_strict` (1 pasuje na 1, 2 na 2 a `undef` len na `undef`) a napr. `eeq=any`.

Druhý problém je najľahšie vyriešiť tak, že budeme ako $C$ postupne skúšať cestu tvorenú 1, 2, 3, ..., až `CountV(G)`-1 hranami a zo všetkých nájdených možností vyberieme tú najlepšiu.

Ak má teda graf $G$ práve $N$ vrcholov, potrebujeme vyskúšať $N-1$ možných dĺžok cesty $C$. Každú z nich vieme vyskúšať v konštantnom čase, takéto riešenie má teda časovú zložitosť lineárnu od počtu vrcholov grafu.

**Listing programu:**

```
function najkratsia_cesta(G : Graph; x,y : longint) : longint;
var C, match : Graph;
    i, best : longint;
begin
   { oznacime vrcholy G a vyrobime cestu C s 0 hranami }
   SetAllV(G,undef); SetV(G,x,1); SetV(G,y,2);
   C := EmptyG;
   AddV(C,1);
   best := -1;

   { postupne pre kazde i najdeme najkratsiu cestu s i hranami }
   for i:=1 to CountV(G)-1 do begin
      { predlzime cestu C }
      AddV(C,2);
```

```
      if i>1 then SetV(C,i,undef);
      AddE(C,i,i+1,undef);
      { najdeme cestu C v grafe G }
      match := Find(G,C,value_strict,any);
      { skontrolujeme, ci ju naozaj nasiel }
      if CountV(match) <> i+1 then continue;
      { ak ano, zistime, ci je lepsia ako doteraz najlepsia }
      if (best=-1) or (SumE(match)<best) then best := SumE(match);
   end;
   najkratsia_cesta := best;
end;
```

**Hľadanie najkratšej cesty v logaritmickom čase:**

V predchádzajúcom riešení nás najviac zdržalo to, že sme nevedeli správnu dĺžku cesty $C$. Aby sme sa tohto problému zbavili, pomôžeme si trikom: upravíme daný graf $G$ tak, aby nám potom stačilo cestu hľadať raz. Takáto úprava môže vyzerať napr. nasledovne:

Z vrcholu $y$ povedie navyše nová cesta, tvorená vrcholmi $z_1, \ldots, z_N$ (kde $N$ je počet vrcholov $G$). Všetky hrany tejto cesty budú mať váhu 0. Takto upravený graf nazveme $G'$.

Všimnime si teraz dve skutočnosti: Za prvé, ak máme ľubovoľnú cestu z $x$ do niektorého $z_i$, táto cesta musí prechádzať cez $y$. Navyše dĺžka celej tejto cesty je rovná dĺžke jej časti vedúcej z $x$ do $y$, lebo na úseku z $y$ do $z_i$ majú všetky hrany váhu 0.

A za druhé, všimnime si ľubovoľnú cestu z $x$ do $y$. Označme jej počet hrán $k$. (Platí $k \leq N - 1$.) Ak na jej koniec pridáme ešte $N - k$ hrán vedúcich z $y$ cez vrcholy $z_1, \ldots, z_{N-k}$, dostaneme cestu s rovnakou dĺžkou tvorenú presne $N$ hranami.

Čo z týchto dvoch pozorovaní vyplýva? Namiesto pôvodnej úlohy „nájdi v grafe $G$ najkratšiu cestu, ktorá začína v $x$ a končí v $y$“ môžeme riešiť novú úlohu: „nájdi v grafe $G'$ cestu tvorenú presne $N$ hranami, ktorá začína v $x$ a končí v niektorom $z_i$“. Riešenie oboch úloh je vždy rovnaké.

No a tu už máme omnoho jednoduchšiu situáciu: poznáme totiž dĺžku cesty. Celý algoritmus si teda môžeme zhrnúť nasledovne:

1. Zostrojíme cestu dĺžky $N$.
2. Jednu kópiu tejto cesty vhodne označíme a pripojíme k vrcholu $y$ v $G$.
3. Následne druhú kópiu tejto cesty vhodne označíme a v $G'$ nájdeme jej najlacnejší výskyt.

Prvý krok vieme spraviť v čase logaritmickom od počtu vrcholov pomocou jedného z algoritmov v študijnom texte. Zvyšné dva kroky vieme spraviť v konštantnom čase. Celková časová zložitosť je teda logaritmická od počtu vrcholov.

### Listing programu:

```
function cesta(n: longint): Graph;
{ zostroji cestu s n hranami, implementacia je v studijnom texte }

function najkratsia_cesta(G : Graph; x,y : longint) : longint;
var C, match : Graph;
    N : longint;
begin
   { vyrobime cestu tvorenu N novymi vrcholmi }
   N := CountV(G);
   C := cesta(N);
   SetAllV(C,2);
   SetAllE(C,0);
   { nastavime G znacky a pridame tu cestu don }
   SetAllV(G,0);
   SetV(G,x,1);
   G := Join(G,C,none,none);
   AddE(G,y,N+1,0);
   { najdeme najlacnejsiu N-hranovu cestu z x do noveho vrcholu }
   SetAllV(C,undef);
   SetV(C,1,1);
   SetV(C,N+1,2);
   match := Find(G,C,value,any);
   najkratsia_cesta := SumE(match);
end;
```

# Riešenia domáceho kola kategórie B

## B-I-1 Hľadanie mín

Na reprezentáciu hracieho plánu použijeme dvojrozmerné pole. Pri načítaní vstupu si do neho zaznačíme polohu mín. Pri výpise výstupu potom pre každé políčko, ktoré neobsahuje mínu, prejdeme všetkých jeho susedov a zistíme počet tých, ktorí mínu obsahujú.

Šikovný trik, ako riešiť okraje hracieho plánu: namiesto toho, aby sme mali pole, ktorého riadky a stĺpce budú číslované od 1 po $R$, resp. od 1 po $S$, budeme mať pole v každom smere o políčko väčšie. Riadky teda budú mať čísla od 0 po $R+1$ a stĺpce od 0 po $S+1$. Nové políčka samozrejme žiadne míny neobsahujú.

### Listing programu:

```
const MAX_ROZMER = 5000;

var miny : array[0..MAX_ROZMER+1, 0..MAX_ROZMER+1] of boolean;
    R, S, N, i, j, k, l, mr, ms, pocet : longint;

begin
   { nacitame rozmery }
   readln(R,S);

   { inicializujeme hraci plan: nikde nie je mina }
   for i:=0 to R+1 do for j:=0 to S+1 do miny[r,s]:=false;

   { nacitame jednotlive miny a zaznacime ich do pola }
   readln(N);
   for i:=1 to N do begin
      readln(mr,ms);
      miny[mr,ms] := true;
   end;

   { prechadzame cez policka a vyrabame vystup }
   for i:=1 to R do begin
      for j:=1 to S do begin
         if miny[i,j] then write('*') else begin
            pocet := 0;
            { prezrieme vsetkych susedov policka (i,j) }
            for k:=i-1 to i+1 do for l:=j-1 to j+1 do
               if miny[k,l] then inc(pocet);
            if pocet=0 then write('.') else write(pocet);
         end;
      end;
      writeln;
   end;
end.
```

Iné možné riešenie je zostrojovať výstup rovno pri načítaní vstupu. Na začiatku každé políčko inicializujeme na nulu. Vždy, keď načítame súradnice míny,

zvýšime o 1 hodnoty na všetkých 8 susedných políčkach. Prítomnosť míny si môžeme zaznačiť napr. tak, že k hodnote políčka, kde mína leží, pripočítame 10.

## B-I-2 Starosta

Najjednoduchšie riešenie tejto úlohy je načítať celý vstup do poľa a toto pole spracovať v dvoch prechodoch. V prvom prechode nájdeme všetkých Mórickovych známych. Na ich zapamätanie použijeme pole boolovských premenných, v ktorom si o každom dedinčanovi budeme pamätať, či sa s Mórickom pozná.

V druhom prechode vyplníme druhé pole boolovských premenných, v ktorom budeme mať o každom dedinčanovi uložené, či pozná nejakého Mórickovho známeho.

### Listing programu:

```
const MAX_D = 1000000; MAX_Z = 5000000;

var vstup : array[1..MAX_Z, 1..2] of longint;
    znami1, znami2 : array[1..MAX_D] of boolean;
    D, Z, i, j : longint;

begin
   { nacitame vstup }
   read(D,Z);
   for i:=1 to Z do read(vstup[i,1],vstup[i,2]);

   { prejdeme vstup a zistime vsetkych Morickovych znamych }
   for i:=1 to D do znami1[i] := false;
   for i:=1 to Z do begin
      if vstup[i,1]=1 then znami1[ vstup[i,2] ] := true;
      if vstup[i,2]=1 then znami1[ vstup[i,1] ] := true;
   end;

   { znova prejdeme vstup a zistime vsetkych ich znamych }
   for i:=1 to D do znami2[i] := false;
   for i:=1 to Z do begin
      if znami1[ vstup[i,1] ] then znami2[ vstup[i,2] ] := true;
      if znami1[ vstup[i,2] ] then znami2[ vstup[i,1] ] := true;
   end;

   { vypiseme vsetkych, ktori nie su znami, v premennej j je ich pocet }
   j := 0;
   for i:=2 to D do
      if (not znami1[i]) and (not znami2[i]) then begin
         writeln(i);
         inc(j);
      end;
   if j=0 then writeln('starosta');
end.
```

**Alternatívne riešenia:**

Predchádzajúce riešenie je bezkonkurenčne najjednoduchšie. Ukážeme si však aj iné možné prístupy, ktoré obsahujú viacero užitočných techník.

Základná myšlienka týchto riešení je jednoduchá: najskôr v cykle prejdeme všetkých Mórickovych známych a potom pomocou dvoch vnorených cyklov prejdeme pre každého známeho všetkých jeho známych. O každom z nich si zaznačíme, že už sme ho videli. Na záver už len prejdeme zoznam dedinčanov a vypíšeme všetkých, ktorých sme ani raz nevideli.

Na to, aby bolo naše riešenie dostatočne rýchle, si potrebujeme informácie o dedinčanoch pamätať v nejakej vhodnejšej podobe – prechádzať celý zoznam známostí toľkokrát, koľko známych má Móricko, už je predsa len priveľa zbytočnej práce.

**Matica susednosti:**

Pre vstupy, kde je počet dedinčanov $D$ nanajvýš pár tisíc, si môžeme dovoliť použiť maticu susednosti – teda dvojrozmerné pole, kde si pre každú dvojicu dedinčanov zaznačíme, či sa poznajú.

**Listing programu:**

```
const MAX_D = 5000;
var pozna : array[1..MAX_D, 1..MAX_D] of boolean;
    OK : array[1..MAX_D] of boolean;
    D, Z, i, j, k : longint;

begin
   { nacitame vstup a vyplnime si maticu susednosti }
   read(D,Z);
   for i:=1 to D do for j:=1 to D do pozna[i,j] := false;
   for i:=1 to Z do begin
      read(j,k);
      pozna[j,k] := true;
      pozna[k,j] := true;
   end;

   { najdeme vsetkych Morickovych znamych a zaznacime ich do pola OK }
   OK[1] := true;
   for i:=2 to D do OK[i] := pozna[1,i];

   { pre kazdeho Morickovho znameho najdeme jeho znamych a zaznacime aj tych }
   for i:=2 to D do
      if pozna[1,i] then
         for j:=2 to D do
            if pozna[i,j] then
               OK[j] := true;

   { vypiseme vsetkych, ktori nie su OK, v premennej k je ich pocet }
   k := 0;
   for i:=2 to D do if not OK[i] then begin writeln(i); inc(k); end;
   if k=0 then writeln('starosta');
end.
```

**Zoznamy známych:**

Ešte lepšie riešenie je vytvoriť si zoznamy známych: Pre každého dedinčana $x$ budeme mať zoznam, v ktorom budú uložení všetci dedinčania, ktorých $x$ pozná.

Oproti predchádzajúcemu riešeniu má toto hneď dve výhody. Za prvé, spotreba pamäte je omnoho nižšia – pamätáme si len údaje, ktoré sú na vstupe, neplýtvame pamäťou na dvojice, ktoré sa nepoznajú.

(Aby sme boli presnejší, riešenie používajúce maticu susednosti potrebuje pamäť priamo úmernú hodnote $D^2$, teda počtu všetkých dvojíc, zatiaľ čo pri tomto riešení bude množstvo pamäte lineárne od $Z$, teda počtu dvojíc, ktoré sa poznajú.)

Za druhé, zlepší nám to aj časovú zložitosť – aj pre Móricka, aj pre každého jeho známeho budeme vedieť rovno vymenovať všetkých jeho známych, nebudeme musieť zbytočne kontrolovať všetkých dedinčanov.

Jeden možný spôsob, ako zoznamy známych uložiť, je použiť dátovú štruktúru *spájaný zoznam*.

**Listing programu:**

```
const MAX_D = 1000000;
type pznamy = ^tznamy;
     tznamy = record
        kto : longint;
        dalsi : pznamy;
     end;

var znami : array[1..MAX_D] of pznamy;
    p, q : pznamy;
    OK : array[1..MAX_D] of boolean;
    D, Z, i, j, k : longint;

begin
   { nacitame vstup a vyplnime si zoznamy znamych }
   read(D,Z);
   for i:=1 to D do znami[i] := nil;
   for i:=1 to Z do begin
      read(j,k);
      p := new(pznamy);  p^.kto := k;  p^.dalsi := znami[j];  znami[j] := p;
      p := new(pznamy);  p^.kto := j;  p^.dalsi := znami[k];  znami[k] := p;
   end;

   { najdeme vsetkych Morickovych znamych a zaznacime ich do pola OK }
   OK[1] := true;
   for i:=2 to D do OK[i] := false;
   p := znami[1];
   while p <> nil do begin
      OK[ p^.kto ] := true;
      p := p^.dalsi;
   end;

   { pre kazdeho znameho prejdeme vsetkych jeho znamych a zaznacime aj tych }
   p := znami[1];
```

```
   while p <> nil do begin
      q := znami[ p^.kto ];
      while q <> nil do begin
         OK[ q^.kto ] := true;
         q := q^.dalsi;
      end;
      p := p^.dalsi;
   end;

   { vypiseme vsetkych, ktori nie su OK, v premennej k je ich pocet }
   k := 0;
   for i:=2 to D do if not OK[i] then begin writeln(i); inc(k); end;
   if k=0 then writeln('starosta');
end.
```

### Zoznamy známych v obyčajnom poli:

V predchádzajúcom riešení sme použili spájané zoznamy, implementované pomocou ukazovateľov (pointrov). Vystačíme si však aj bez nich – rovnako efektívne riešenie sa dá naprogramovať aj s obyčajnými poľami. Moderné programovacie jazyky ponúkajú polia, ktorých veľkosť vieme nastaviť počas behu programu. (Napr. vo FreePascale/Delphi môžeme na nastavenie dĺžky poľa použiť `SetLength`.)

Takéto riešenie bude fungovať v dvoch fázach. V prvej načítame celý vstup do pomocného poľa a pre každého dedinčana spočítame jeho *stupeň* – teda počet ľudí, s ktorými sa pozná. Potom pre každého dedinčana vyrobíme pole príslušnej dĺžky a následne všetky tieto polia vyplníme potrebnými údajmi.

(V C++ máme k dispozícii dátovú štruktúru `vector`, pomocou ktorej je riešenie ešte pohodlnejšie – nepotrebujeme poznať veľkosť poľa vopred, stačí postupne na jeho koniec pridávať nové a nové prvky.)

### Listing programu:

```
const MAX_D = 1000000; MAX_Z = 5000000;

var vstup : array[1..MAX_Z, 1..2] of longint;
    stupne : array[1..MAX_D] of longint;
    znami : array[1..MAX_D] of array of longint;
    OK : array[1..MAX_D] of boolean;
    D, Z, i, j : longint;

begin
   { nacitame vstup a spocitame stupne }
   read(D,Z);
   for i:=1 to D do stupne[i] := 0;
   for i:=1 to Z do begin
      read(vstup[i,1],vstup[i,2]);
      inc( stupne[ vstup[i,1] ] );
      inc( stupne[ vstup[i,2] ] );
   end;

   { nastavime velkosti poli a vyplnime ich }
   for i:=1 to D do setlength( znami[i], stupne[i] );
```

```
   for i:=1 to D do stupne[i]:=0;
   for i:=1 to Z do begin
      inc( stupne[ vstup[i,1] ] );
      znami[ vstup[i,1] ][ stupne[vstup[i,1]] ] := vstup[i,2];
      inc( stupne[ vstup[i,2] ] );
      znami[ vstup[i,2] ][ stupne[vstup[i,2]] ] := vstup[i,1];
   end;

   { najdeme vsetkych Morickovych znamych a zaznacime ich do pola OK }
   OK[1] := true;
   for i:=2 to D do OK[i] := false;
   for i:=1 to stupne[1] do OK[ znami[1,i] ] := true;

   { pre kazdeho znameho prejdeme vsetkych jeho znamych a zaznacime aj tych }
   for i:=1 to stupne[1] do
      for j:=1 to stupne[znami[1,i]] do
         OK[ znami[znami[1,i],j] ] := true;

   { vypiseme vsetkych, ktori nie su OK, v premennej k je ich pocet }
   k := 0;
   for i:=2 to D do if not OK[i] then begin writeln(i); inc(k); end;
   if k=0 then writeln('starosta');
end.
```

## B-I-3 Poriadok na polici

**Optimálne usporiadanie šestice zo zadania:**

Na polici ležali zväzky v poradí 4, 2, 1, 6, 5, 3. Mali sme ich usporiadať použitím najmenšieho možného počtu výmen susedných dvojíc a dokázať, že naše riešenie je najlepšie možné.

Jeden optimálny postup vyzerá nasledovne:

- Knihu 1 vymeníme s knihou 2 a následne s knihou 4.
  Dostávame poradie 1, 4, 2, 6, 5, 3.
- Knihu 2 vymeníme s knihou 4. Dostávame 1, 2, 4, 6, 5, 3.
- Knihu 3 vymeníme s knihou 5, knihou 6 a nakoniec aj s knihou 4.
  Dostávame 1, 2, 3, 4, 6, 5.
- Vymeníme knihy 5 a 6 a sme hotoví.

Dokopy sme spravili sedem výmen.

**Prečo to na menej výmen nejde?:**

Všimnime si napríklad knihy 1 a 2. Na začiatku je kniha 1 napravo od knihy 2, na konci však už musí byť od nej naľavo. No a jediný spôsob, ako sa tam dostane, je ten, že ich niekedy počas riešenia vymeníme.

To isté platí napríklad aj pre knihy 3 a 6. V ľubovoľnom riešení musíme niekedy tieto dve knihy vymeniť, aby sme dostali knihu 3 naľavo od knihy 6.

Vyzbrojení týmto pozorovaním si už ľahko všimneme, že vo vyššie uvedenom riešení to platilo pre všetkých 7 výmen, ktoré sme spravili. Vždy sme menili dvojicu kníh takú, že kniha, ktorú sme presúvali sprava doľava, mala od tej druhej menšie číslo. Hocijaké iné riešenie musí obsahovať každú z týchto 7 výmen, preto lepšie riešenie neexistuje.

Poznámka: V odbornej literatúre sa dvojica kníh s vlastnosťou „kniha s väčším číslom je naľavo od knihy s menším číslom“ volá *inverzia*. Počet inverzii je zaujímavou mierou toho, nakoľko je postupnosť čísel neutriedená.

**Usporiadanie veľa kníh:**

Budúci mesiac vyjde nové vydanie magickej encyklopédie. To už nebude mať 6 zväzkov, ale $N$. Našou úlohou je dokázať, že knihovníkovi bude určite stačiť spraviť nanajvýš $N(N-1)/2$ výmen na to, aby ich usporiadal.

Toto je jednoduché – stačí, ak bude knihovník postupovať rovnako, ako my pri riešení prvej podúlohy. Najskôr postupnými výmenami presunie na prvú pozíciu knihu číslo 1, potom za ňu knihu číslo 2, a tak ďalej.

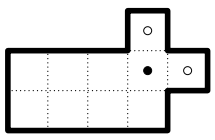
Nech by bola na začiatku kniha 1 hocikde, určite nám stačí nanajvýš $N-1$ výmen na to, aby sme ju dostali na začiatok. Pre knihu 2 nám už bude stačiť nanajvýš $N-2$ výmen – prvá pozícia je totiž už obsadená knihou 1. A tak ďalej. Celkový počet výmen teda vieme zhora ohraničiť hodnotou

$$(N-1)+(N-2)+\cdots+2+1=N(N-1)/2.$$

## B-I-4 Dláždenie

**Podúloha a):**

Mali sme zostrojiť miestnosť tvorenú presne 10 štvorčekmi, ktorá sa nebude dať vydláždiť. Jedna takáto miestnosť je na obrázku vpravo.

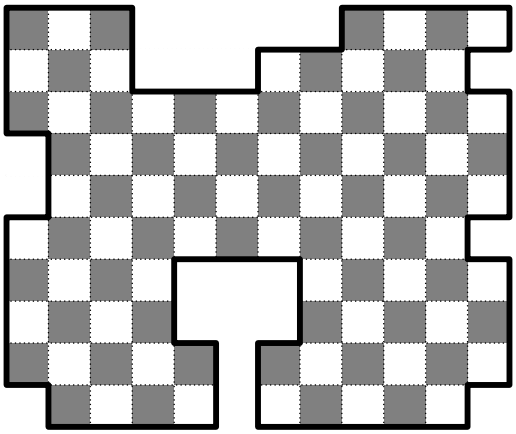


Ľahko dokážeme, že sa naozaj nedá vydláždiť. Všimnime si jedno z políčok označených prázdnym krúžkom. Jediný spôsob, ako ho pokryť dlaždicou, pokryje aj políčko označené plným krúžkom. Potom už ale nemáme ako pokryť druhé políčko označené prázdnym krúžkom.

**Podúloha b):**

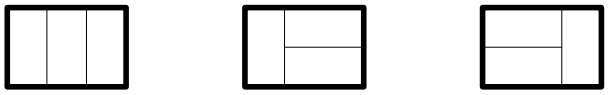
Útvar v zadaní sa vydláždiť nedá. Aby sme to dokázali, stačí si ho ofarbiť šachovnicovým vzorom. Zjavne každá dlaždica vždy pokryje jedno čierne a jedno biele políčko. Náš útvar však obsahuje 48 čiernych a 50 bielych políčok. To

znamená, že nech budeme prikladať dlaždice ako len chceme, vždy nám ostanú aspoň dve biele políčka nepokryté.



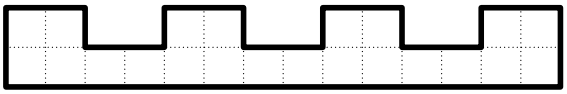
**Podúloha c):**

V tejto podúlohe sme mali nájsť miestnosť, ktorá bude mať presne 3 možné dláždenia. Najjednoduchšou takouto miestnosťou je obdĺžnik rozmerov $2 \times 3$. Na nasledujúcom obrázku sú všetky jeho možné dláždenia.



**Podúloha d):**

Existuje veľa miestností, ktoré majú 16 možných dláždení. V našom riešení začneme tým, že si všimneme, že štvorec $2 \times 2$ má dve možné dláždenia. Teraz použijeme jednoduchý trik: Ak máme dve miestnosti, z ktorých prvá má $A$ možných dláždení a druhá $B$ a vhodne ich spojíme do jednej väčšej miestnosti, dostaneme miestnosť s $AB$ rôznymi dláždeniami.



Miestnosť na obrázku vznikla takýmto spojením štyroch štvorcov. Je zjavné, že má práve 16 dláždení: v každom zo štvorcov, ktoré obsahuje, máme na výber z dvoch možností, „spájacie“ dlaždice medzi štvorcami sú určené jednoznačne.

# Riešenia krajského kola kategórie A

## A-II-1 Vlak

Priamočiarym riešením je napríklad vyskúšať všetky možnosti, ako sa dajú odobrať vagóny, a pre každú z nich vyskúšať, či vznikne palindróm. Takýchto možností je toľko, koľko je podmnožín $n$-prvkovej množiny. A tých je až $2^n$, čo je už pre $n = 50$ neúnosne veľa.

Skúsme teda úlohu vyriešiť po krokoch. Reprezentujme si vlak ako postupnosť písmen $a_1a_2a_3 \ldots a_{n-1}a_n$. Pre podpostupnosť $a_ia_{i+1} \ldots a_{j-1}a_j$ označme $V[i, j]$ najmenší počet vagónov, ktoré z nej treba vyradiť, aby bola symetrická. Špeciálne teda riešenie pre celý pôvodný vlak (to, ktoré nás zaujíma) bude označené $V[1, n]$.

Čo vieme o týchto hodnotách povedať? Ak $i = j$, tak zjavne $V[i, j] = 0$, lebo vlak s jedným vagónom je vždy symetrický. Nech teda $i < j$. Ako vyzerá optimálne riešenie pre vlak $a_ia_{i+1} \ldots a_{j-1}a_j$?

V prípade, že $a_i = a_j$, prvý ani posledný vagón evidentne v optimálnom riešení vyradiť netreba.[1] V takomto prípade nám ostáva vyriešiť našu úlohu pre vagóny $a_{i+1} \ldots a_{j-1}$. A teda platí $V[i, j] = V[i + 1, j - 1]$.

V prípade, že $a_i \neq a_j$, musíme aspoň jeden z týchto dvoch vagónov vyhodiť (inak by hneď za lokomotívou bol po otočení iný typ vagónu). Ak by sme vyhodili vagón $a_i$, ostane nám vlak tvorený vagónmi $a_{i+1} \ldots a_{j-1}a_j$. Pre tento vlak potrebujeme vyhodiť ďalších $V[i + 1, j]$ vagónov. A naopak, ak by sme vyhodili vagón $a_j$, ostane nám $a_ia_{i+1} \ldots a_{j-1}$ a pre ten treba ešte $V[i, j - 1]$ zmien. Z týchto dvoch možností, ktoré máme na výber, si samozrejme vyberieme tú lepšiu pre nás. Preto v tomto prípade platí $V[i, j] = 1 + \min(V[i + 1, j], V[i, j - 1])$.

V oboch prípadoch sme teda našli vzťah, ktorý optimálne riešenie $V[i, j]$ spočíta v konštantnom čase z optimálnych riešení pre iné, kratšie vlaky. A toto nám už stačí na napísanie algoritmu s časovou zložitosťou $\Theta(n^2)$: Budeme postupne spracúvať všetky podreťazce zadaného vlaku, začínajúc od najkratších a končiac celým vlakom. A pre každý úsek si spočítame a zapamätáme optimálny počet vyhodení vagónu.

[1] Dôkaz: Od najlepšieho riešenia, v ktorom oba vyradíme, je lepšie to isté riešenie, v ktorom ale oba ponecháme. Ak by sme v optimálnom riešení vyradili len jeden z nich, BUNV nech to je $a_j$, dostaneme rovnako dobré riešenie, ak namiesto $a_j$ vyradíme ten vagón, ktorý zostal posledný nevyradený.

Pamäťová zložitosť takéhoto riešenia je tiež $\Theta(n^2)$. Musíme si totiž zapamätať všetky hodnoty $V[i,j]$, aby sme na konci vedeli jedno optimálne riešenie zostrojiť. (Existuje aj riešenie s časovou zložitosťou $\Theta(n^2)$ a pamäťovou $\Theta(n)$, ale je zbytočne komplikované, preto ho nebudeme uvádzať.)

## Listing programu:

```
#include <cstdio>
#include <algorithm>

#define MAX 10010

int N;
char vstup[MAX];
unsigned short V[MAX][MAX]; // hodnoty V[i,j] z popisu riesenia, zmestia sa do
unsigned short

int main() {
    scanf("%d %s", &N, vstup);

    // podvlaky dlzky 0 a 1 su optimalne
    for (int i = 0; i < N; i++) V[i][i-1] = V[i][i] = 0;

    // pre dlzky 2..N si spocitame optimalnu dlzku riesenia
    for (int dlzka = 2; dlzka <= N; dlzka++) {
        for (int zaciatok = 0; zaciatok+dlzka <= N; zaciatok++) {
            int koniec = zaciatok + dlzka - 1;
            if (vstup[zaciatok] == vstup[koniec]) {
                V[zaciatok][koniec] = V[zaciatok+1][koniec-1];
            } else {
                V[zaciatok][koniec] =
                    1 + std::min( V[zaciatok+1][koniec], V[zaciatok][koniec-1] );
            }
        }
    }

    // teraz ideme zrekonstruovat jedno optimalne riesenie
    bool vyhodit[MAX];
    for (int i = 0; i < N; i++) vyhodit[i] = false;
    int zaciatok = 0, koniec = N-1;
    while (koniec > zaciatok) {
        if (vstup[zaciatok] == vstup[koniec]) {
            // prvy aj posledny vagon ostavaju
            zaciatok++;
            koniec--;
        } else {
            // odstranime bud zaciatok alebo koniec, podla toho co je vyhodnejsie
            if (V[zaciatok][koniec] == 1+V[zaciatok+1][koniec]) {
                vyhodit[zaciatok] = true;
                zaciatok++;
            } else {
                vyhodit[koniec] = true;
                koniec--;
            }
        }
    }

    // a ideme vypisat najdene riesenie
    printf("%d\n", V[0][N-1]);
    bool medzera = false;
```

```
    for (int i = 0; i < N; i++) {
        if (vyhodit[i]) {
            if (medzera) printf(" ");
            medzera = true;
            printf("%d", i + 1); // v programe indexujeme od 0, v zadani od 1
        }
    }
    printf("\n");
}
```

**Dodatok: riešenie s lepšou pamäťovou zložitosťou:**

Existuje aj riešenie úlohy Vlak v čase $O(n^2)$ a pamäti $O(n)$. V nasledujúcom texte si ho ukážeme.

Pripomeňme si, že $V[i,j]$ je najmenší počet vagónov, ktoré treba vyradiť z $a_i a_{i+1} \ldots a_{j-1} a_j$, aby sme dostali palindróm. Platí, že ak $a_i = a_j$, tak $V[i,j] = V[i+1, j-1]$, inak $V[i,j] = 1 + \min(V[i+1,j], V[i,j-1])$. Všimnite si, že v prvom prípade sa odkazujeme na postupnosť o 2 znaky kratšiu, v druhom prípade na dve postupnosti, ktoré sú obe od pôvodnej o 1 znak kratšie.

Keď budeme teda hodnoty $V[i,j]$ počítať najskôr pre všetky postupnosti dĺžky 1, potom všetky postupnosti dĺžky 2, dĺžky 3, atď., tak nám v každom okamihu stačí pamäť $O(n)$: keď spracúvame postupnosti dĺžky $k$, stačí si pamätať optimálne riešenia pre postupnosti dĺžok $k-1$ a $k-2$.

Optimálny počet vagónov, ktoré treba odstrániť, vieme teda v lineárnej pamäti spočítať ľahko. Problém je v samotnom zostrojení riešenia.

**Zložitejšia úloha**

Namiesto toho, aby sme ukázali, ako tento problém riešiť, si našu úlohu ešte trochu zovšeobecníme: dovolíme, aby náš vlak obsahoval **nanajvýš jeden** špeciálny vozeň „`*`“, ktorý nesmieme odobrať. Táto úloha sa zjavne rieši rovnako ako pôvodná – len v prípade, že máme špeciálny vozeň, neskúšame možnosti, ktoré ho odoberú.

**Rozoberáme vlak**

Predstavme si, že postupne rozoberáme vlak, ktorý sme dostali na vstupe, a to spôsobom, ktorý zodpovedá (jednému možnému) optimálnemu riešeniu. Pre vlak `ABCXDFABECEDAFDCBYZZA` by tento proces vyzeral takto:

– zober do riešenia `A` zo začiatku aj z konca
– zahoď `Z` z konca
– zahoď `Z` z konca
– zahoď `Y` z konca
– zober do riešenia `B` zo začiatku aj z konca
– ...

Po pár takýchto krokoch sa z reťazca dĺžky $n$ dostaneme k reťazcu polovičnej dĺžky, presnejšie $\lfloor n/2 \rfloor$ alebo $\lfloor n/2 \rfloor + 1$. V našom príklade by to mohol byť reťazec `DFABECEDAFD`.

No a presne tento reťazec (presnejšie: indexy, kde v pôvodnom reťazci začína a končí) vieme nájsť počas toho, ako počítame hodnoty $V[i, j]$: Kým spracúvame úseky dĺžky menšej ako $\lfloor n/2 \rfloor$, len počítame hodnoty $V[i, j]$ ako v predchádzajúcom riešení, nerobíme nič navyše. Akonáhle začneme spracúvať dlhšie úseky, budeme si pre každý z nich pamätať nie len optimálnu hodnotu $V[i, j]$, ale aj to, ktorý reťazec dĺžky $\lfloor n/2 \rfloor$ alebo $\lfloor n/2 \rfloor + 1$ by sme dostali počas optimálneho riešenia úseku $a_i a_{i+1} \ldots a_{j-1} a_j$.

V čase $O(n^2)$ a pamäti $O(n)$ teda vieme okrem hodnoty $V[1, n]$ získať trochu informácie navyše: vieme v pôvodnom reťazci nájsť „strednú časť“ s vyššie uvedeným významom.

**A pasca sklapne**

A teraz sa ukáže, načo bola dobrá zložitejšia úloha. Nech sme pre pôvodný reťazec $w = \alpha\beta\gamma$ zistili, že počas výroby optimálneho riešenia sa „na pol ceste“ stretneme s reťazcom $\beta$. To ale znamená, že úlohu „optimálne zostroj palindróm z $w$“ vieme rozbiť na dve menšie, nezávislé podúlohy: „optimálne zostroj palindróm z $\beta$“ a „optimálne zostroj palindróm z $\alpha * \gamma$“. Obe tieto úlohy vieme vyriešiť rekurzívnym volaním pôvodného algoritmu.

Napríklad uvažujme vlak z vyššie uvedeného príkladu. Najdlhší palindróm, ktorý sa v ňom nachádza, vieme poskladať z najdlhších palindrómov, ktoré sa nachádzajú v reťazcoch `DFABECEDAFD` a `ABCX*CBYZZA`.

**Analýza časovej zložitosti**

Označme $T(n)$ čas výpočtu tohto algoritmu na reťazci dĺžky $n$. Z popisu algoritmu dostávame, že $T(n) = \Theta(n^2) + 2T(n/2)$. Dá sa dokázať, že potom $T(n) = \Theta(n^2)$, teda asymptotická časová zložitosť je rovnaká ako pri pôvodnom riešení. Inými slovami, prácu navyše, ktorú toto nové riešenie spraví, môžeme zanedbať pri porovnaní s pôvodným dynamickým programovaním. Intuitívne zdôvodnenie:

- Pre pôvodný reťazec spravíme rádovo $n^2$ krokov výpočtu.
- Pre každý z dvoch polovičných reťazcov spravíme rádovo $(n/2)^2 = n^2/4$ krokov výpočtu, teda dokopy rádovo $n^2/2$.
- Z každého z nich vzniknú dva reťazce štvrtinovej dĺžky. Pre všetky štyri takéto reťazce dokopy spravíme rádovo $4 \cdot (n/4)^2 = n^2/4$ práce.

– A tak ďalej. Celkový počet krokov je teda rádovo $n^2+n^2/2+n^2/4+n^2/8+\cdots = 2n^2$.

Formálne sa dá napríklad matematickou indukciou dokázať, že z $T(x) = 1$ pre $x \leq 1$ a $T(n) \leq cn^2 + 2T(n/2)$ vyplýva $T(n) \leq 2cn^2$.

**Poznámka na záver**

Podobný trik sa dá aplikovať aj v iných úlohách. Vyskúšajte si napríklad v čase $O(n^2)$ a pamäti $O(n)$ zostrojiť jednu najdlhšiu postupnosť, ktorá sa dá vybrať aj z postupnosti $a_1, \ldots, a_n$, aj z postupnosti $b_1, \ldots, b_n$.

**Dodatok: nesprávne riešenie a ako ho opraviť:**

Veľmi často sa pri tejto úlohe vyskytuje nasledujúce **nesprávne** riešenie: najdlhší palindróm v postupnosti $a_1a_2\ldots a_n$ nájdem ako najdlhšiu spoločnú podpostupnosť postupností $A = a_1a_2\ldots a_n$ a $B = a_n\ldots a_2a_1$.

Tento prístup nie je uplny zlý (podobá sa predsa na naše vzorové riešenie), ale má niekoľko problémov. **Platí** síce, že najdlhší palindróm v postupnosti $A$ má rovnakú dĺžku ako najdlhšia spoločná podpostupnosť $A$ a $B$. Ale ani omylom **neplatí**, že každá ich spoločná podpostupnosť je aj palindrómom. Napríklad pre vstup `ACBBAC` je hľadaný palindróm buď `ABBA` alebo `CBBC`, ale nájdená podpostupnosť môže byť aj `ABBC` alebo `CBBA`.

Pozrime sa najskôr na samotný algoritmus na hľadanie najdlhšej spoločnej podpostupnosti postupností $a_1a_2\ldots a_n$ a $b_1b_2\ldots b_n$. Podobne ako vo vzorovom riešení použijeme dynamické programovanie: $D[i,j]$ bude dĺžka najdlhšej spoločnej podpostupnosti pre postupnosti $a_1\ldots a_i$ a $b_1\ldots b_j$. Pre tieto hodnoty platí okrajová podmienka $D[0,j] = D[i,0] = 0$ a všeobecný vzťah nasledovný: ak $a_i = b_j$, tak $D[i,j] = 1+D[i-1,j-1]$, inak $D[i,j] = \max(D[i-1,j], D[i,j-1])$.

Predstavme si teda, že sme tieto hodnoty $D[i,j]$ spočítali. My ale v skutočnosti nehľadáme najdlhšiu spoločnú podpostupnosť. Predstavme si, že hľadáme palindróm párnej dĺžky. Potom určite existuje nejaké $k$ také, že z $a_1\ldots a_k$ vyberieme jeho prvú polovicu a z $a_{k+1}\ldots a_n$ druhú. Inými slovami, polovica hľadaného palindrómu je spoločnou podpostupnosťou postupností $a_1\ldots a_k$ a $a_n\ldots a_{k+1}$. A jej dĺžku predsa poznáme: je to $D[k, n-k]$. Stačí teda vyskúšať všetky možné $k$ a vybrať si najlepšiu možnosť. Palindrómy nepárnej dĺžky vyriešime analogicky – zoberieme doň $a_k$ a zvyšok nájdeme ako najdlhšiu spoločnú podpostupnosť $a_1\ldots a_{k-1}$ a $a_n\ldots a_{k+1}$.

Iné, trikové riešenie: Predstavme si, že sme našli najdlhšiu spoločnú podpostupnosť $P$ zadaného reťazca a jeho reverzu. Táto síce nemusí byť palindrómom,

vieme z nej však vždy jeden vyhovujúci palindróm rovnakej dĺžky zostrojiť. Určite totiž zafunguje aspoň jeden z postupov „zober prvú polovicu $P$ a jej reverz“ a „zober reverz druhej polovice $P$ a druhú polovicu $P$“. Rozmyslite si, prečo je to tak.

## A-II-2 Jabloňový sad

Hovoríme, že množina bodov v rovine je *konvexná*, ak pre každé dva jej body $A$ a $B$ platí, že obsahuje celú úsečku $AB$. Preložené do ľudskej reči, konvexný útvar drží pokope a nemá na obvode žiadne preliačenia dovnútra. Pre ľubovoľnú množinu $M$ je jednoznačne definovaný jej *konvexný obal*: najmenšia konvexná množina $\overline{M}$, ktorá obsahuje celú množinu $M$.

Ak je $M$ tvorená len konečne veľa bodmi, intuitívne si jej konvexný obal môžeme predstaviť nasledovne: body si predstavíme ako klince na doske, okolo všetkých natiahneme gumičku a pustíme ju. Gumička sa bude skracovať, až kým nenarazí na niektoré z klincov, a medzi nimi zostane napnutá. (Pozrite si ešte raz obrázky v zadaní, ak si to neviete predstaviť.)

Asi nikoho teraz neprekvapí, že hľadaný plot je vždy práve obvodom konvexného obalu aktuálnej množiny bodov. Zopakujme si teda niektoré tvrdenia, ktoré pre konvexný obal množiny bodov zjavne platia:

- Konvexný obal je vždy mnohouholník, jeho vrcholy sú niektoré z daných bodov.
- Body najviac vľavo a najviac vpravo (najmenšia a najväčšia súradnica $x$) ležia na jeho obvode.
- Pre každý vrchol konvexného obalu platí, že vnútorný uhol pri ňom je menší ako $180°$.

Nájsť konvexný obal pre zadané množiny bodov v rovine je pomerne známy problém, pre ktorý existujú algoritmy pracujúce v čase $O(n \log n)$. Jeden takýto algoritmus popíšeme.

Konvexný obal môžeme rozdeliť na horný a dolný polobal. Začiatkom oboch polobalov bude najľavejší bod (ak je takých viac, najspodnejší z nich), koncom je najpravejší bod (ak je takých viac, najvrchnejší z nich). V našom algoritme samostatne nájdeme horný a samostatne dolný konvexný polobal.

Na nájdenie horného polobalu si najskôr usporiadame body podľa súradníc (najskôr podľa $x$ a v prípade rovnosti podľa $y$). Potom si body postupne pridávame do polobalu. Vždy, keď pridáme bod do polobalu, skontrolujeme, či nám

tým pri predchádzajúcom bode v polobale nevznikol vnútorný uhol väčší alebo rovný 180°. Ak áno, tak ten predchádzajúci bod práve prestal byť na hornom polobale, preto ho vyhodíme. Toto budeme opakovať, kým v obale neostanú len dva body, alebo kým pri predchádzajúcom bode v polobale nedostaneme vnútorný uhol menší ako 180°. Dolný polobal hľadáme analogicky, len prechádzame body v opačnom poradí.

Ak by stačilo nájsť konvexný obal, tu by sme mohli prestať. Naša úloha však nekončí. Teraz, keď máme hotový konvexný obal, potrebujeme doň vedieť rýchlo pridať ešte jeden bod.

Pozrime sa, ako sa zmení horný polobal. Ak pridaný bod leží pod ním alebo na ňom, polobal zostane nezmenený. Predpokladajme teda, že nový bod $A = [x_A, y_A]$ leží nad horným polobalom. Body na polobale vždy idú za sebou podľa $x$-ových súradníc (v prípade rovnosti podľa $y$-ových). Je teda jasne určené miesto, na ktoré bod $A$ patrí. Pridajme ho teda naň. Teraz mohli nastať dva prípady: buď naďalej všetky vrcholy zvierajú vnútorný uhol menší ako 180° a všetko je v poriadku, alebo sa táto vlastnosť na niektorých miestach porušila. Ale jediné (nanajvýš) dva body, u ktorých sa to mohlo stať, sú práve susedia čerstvo pridaného bodu $A$. Tieto body sa teda ocitli vo vnútri nového konvexného obalu, z nášho polobalu ich teda vyhodíme. Toto budeme opakovať až kým také body v polobale nebudú existovať (pripomíname, že sa stačí stále pozerať na aktuálne susedné body bodu $A$). Keď už pravý aj ľavý sused bodu $A$ majú správne vnútorné uhly, tak máme správny nový horný polobal. Dolný polobal upravíme analogicky.

Pozrime sa teraz ako tento algoritmus implementovať. Potrebujeme vedieť rýchlo nájsť ľavého a pravého suseda v postupnosti a potrebujeme vedieť rýchlo pridávať a odoberať nové body. Pole alebo zoznam nie sú vhodné: v poli vkladanie/odoberanie prvkov niekde v strede poľa trvá lineárny čas, v zozname zase hľadanie prvku trvá lineárny čas. Dobrou dátovou štruktúrou pre nás je vyvažovaný binárny vyhľadávací strom (napríklad AVL strom, červeno-čierny strom, ...). Takéto stromy majú časovú zložitosť každej operácie, ktorú budeme potrebovať, v najhoršom prípade $O(\log x)$, kde $x$ je aktuálny počet prvkov v strome.

Aby sme nemuseli písať dva krát podobný kód, jeden pre horný polobal a druhý pre dolný polobal, tak použijeme nasledovný trik: pri práci s dolným polobalom všetky súradnice vynásobíme $-1$ a použijeme kód pre horný polobal. Všimnite si, že takto vyberieme nielen správne body, ale aj každá zvislá

hraná bude v práve jednom polobale. Navyše ani nemusíme samostatne zostrojovať začiatočný konvexný obal. Jednoducho začneme s prázdnymi polobalmi a postupne pridáme všetkých $n + m$ bodov.

Ako efektívny je náš algoritmus? Pozrime sa, čo sa stane, keď pridáme nový bod. V polobale môže byť najviac $n + m$ bodov. Nájdenie miesta, kam nový bod patrí, trvá vo vyvažovanom strome $O(\log(n + m))$, toľko isto trvá aj jeho pridanie. Potom niekoľkokrát (nech je to $k$) vyhodíme z polobalu jedného z jeho susedov. Každú kontrolu a vyhodenie vieme opäť spraviť v čase $O(\log(n+m))$, celé vyhadzovanie teda trvá $O(k \log(n + m))$. Jedno konkrétne pridanie bodu môže byť teda vcelku pomalé – čím viac bodov z obalu zmizne, tým dlhšie nám bude trvať jeho prepočítanie. Teraz si ale treba uvedomiť, že dokopy tento algoritmus musí byť efektívny. Presnejšie, pozrime sa na jeho celkovú časovú zložitosť. Každý bod najviac raz pridáme do daného polobalu a najviac raz ho odtiaľ vyhodíme. Aj pridanie, aj vyradenie spotrebuje čas $O(\log(n+m))$, preto celková časová zložitosť nášho algoritmu je $O((n + m) \log(n + m))$.

Pamäťová zložitosť algoritmu je $O(n + m)$, lebo pre každý polobal si udržujeme jeden vyhľadávací strom, ktorý v najhoršom prípade obsahuje všetky prvky.

Vzorové riešenie je naprogramované v jazyku C++. Aby sme si ušetrili implementáciu vyváženého vyhľadávacieho stromu, použili sme *set* z STL. Premenná typu *set::iterator* je inteligentný ukazovateľ na prvok v strome. Operátory *++* a *--* na iterátore ho posunú na nasledujúci resp. predchádzajúci prvok. Metóda *upper_bound(B)* nájde v strome prvý prvok väčší od $B$, metóda *begin()* ukazuje na prvý prvok v strome a *end()* ukazuje o jedno za posledný prvok v strome. (Všetky intervaly v STL sú polouzavreté – vľavo uzavreté, vpravo otvorené.) Metóda *erase(a,b)* zoberie polouzavretý interval určený dvomi iterátormi a vymaže ho zo stromu.

### Listing programu:

```
#include <iostream>
#include <set>
#include <cmath>
using namespace std;

struct Bod {
    double x, y;
    Bod (double xx, double yy) { x = xx; y = yy; }
    void flip() { x = -x; y = -y; }
    bool operator < (const Bod &a) const {
        if (a.x == this->x) return this->y < a.y; else return this->x < a.x;
    }
};

double vektorovy_sucin (Bod a, Bod b, Bod c) {
```

```
    return (b.x-a.x)*(c.y-b.y) - (b.y-a.y)*(c.x-b.x);
}
double sqr (double a) { return a * a; }
double vzdialenost (Bod a, Bod b) { return hypot (a.x - b.x, a.y - b.y); }

set<Bod> horny_obal, dolny_obal;

double pridaj_do_obalu (set<Bod> &obal, Bod &bod) {
    if (obal.count (bod) > 0) return 0;
    set<Bod>::iterator stred, pred, pred2, za, za2 ;
    pred = stred = obal.upper_bound (bod);
    if (pred != obal.begin() ) pred--;
    if (stred != obal.begin() && stred != obal.end()
        && vektorovy_sucin(*pred,bod,*stred) <= 0) return 0;
    double zmena = 0;
    if (stred != obal.end() ) zmena -= vzdialenost (*pred, *stred);
    //zaciatok
    pred2 = pred;
    if (pred2 != obal.begin() ) pred2--;
    while (pred2 != pred && vektorovy_sucin (*pred2, *pred, bod) <= 0) {
        zmena -= vzdialenost (*pred, *pred2);
        pred = pred2;
        if (pred2 != obal.begin() ) pred2--;
    }
    //koniec
    za = za2 = stred;
    if (za2 != obal.end() ) za2++;
    while (za2 != obal.end() && vektorovy_sucin (bod, *za, *za2) <= 0) {
        zmena -= vzdialenost (*za, *za2);
        za = za2;
        if (za2 != obal.end() ) za2++;
    }
    if (pred != stred) zmena += vzdialenost(*pred,bod);
    if (za != obal.end()) zmena += vzdialenost(bod, *za);
    if (pred != stred) pred++;
    obal.erase (pred, za);
    obal.insert (bod);
    return zmena;
}

double update (double x, double y) {
    Bod b (x, y);
    double zmena = 0;
    zmena += pridaj_do_obalu (horny_obal, b);
    b.flip();
    zmena += pridaj_do_obalu (dolny_obal, b);
    return zmena;
}

int main() {
    int N, M;
    double x, y, obvod=0;
    cin >> N;
    for (int i = 0; i < N; i++) { cin >> x >> y; obvod += update (x, y); }
    cout << obvod << endl;
    cin >> M;
    for (int i = 0; i < M; i++) { cin >> x >> y; cout << update (x, y) << endl; }
}
```

## A-II-3 Mažoretky

Na úvod trocha terminológie: Poradia mažoretiek zodpovedajú *permutáciám* čísel od 1 po $n$. Poradie definované v zadaní sa zvykne nazývať ich *lexikografickým usporiadaním*. Pre jednoduchosť budeme namiesto „permutácia $\pi$ je v lexikografickom usporiadaní pred permutáciou $\rho$“ hovoriť „$\pi$ je menšia ako $\rho$“.

**Nasledujúca permutácia:**

Našim cieľom v prvej podúlohe je z danej permutácie $a = (a_1, \ldots, a_n)$ vyrobiť najbližšiu väčšiu. Jedinou výnimkou je posledná, najväčšia permutácia $(n, n-1, \ldots, 2, 1)$, po ktorej opäť nasleduje prvá permutácia $(1, 2, \ldots, n-1, n)$. Túto výnimku môžeme v programe ošetriť samostatne. V ďalšom texte teda predpokladáme, že zadaná permutácia $a$ nie je najväčšia.

Pozrime sa na ľubovoľnú permutáciu $b$, ktorá je väčšia ako $a$. Čo o nej vieme povedať? Presne to, čo nám hovorí definícia ich poradia: že na prvom mieste, na ktorom sa $b$ a $a$ líšia, musí byť v $b$ väčšia hodnota ako v $a$.

Spomedzi všetkých permutácií, ktoré sú väčšie ako $a$, hľadáme tú najmenšiu. Ktorá to bude? Majme dve permutácie $b$ a $c$, ktoré sú obe väčšie ako $a$, pričom $b$ sa od $a$ začne líšiť neskôr ako $c$. Potom je zjavné, že $b$ je menšia ako $c$ – totiž na prvom mieste, kde sa $b$ a $c$ líšia, má $b$ pôvodnú hodnotu z $a$, zatiaľ čo $c$ tam má inú, väčšiu hodnotu. Ak teda chceme vyrobiť najmenšiu permutáciu väčšiu ako $a$, musíme sa v prvom rade snažiť nechať na začiatku čo najviac hodnôt nedotknutých.

V rôznych situáciách však bude počet hodnôt, ktoré vieme nechať nedotknuté, rôzny. Všimnime si napríklad permutáciu: $(7, 3, 1, 6, 9, 8, 5, 4, 2)$. Existuje iná permutácia tvaru $(7, 3, 1, 6, \ldots)$, ktorá je od tejto väčšia? Neexistuje – lebo zvyšné prvky $(9, 8, 5, 4, 2)$ sú už usporiadané klesajúco, a teda žiadnu väčšiu permutáciu ich prehádzaním vyrobiť nevieme.

V tomto prípade teda hľadáme permutáciu tvaru $(7, 3, 1, x, \ldots)$, kde $x > 6$. Samozrejme, čím menšie $x$ použijeme, tým menšiu permutáciu dostaneme. V našom prípade máme na výber $x = 8$ alebo $x = 9$, použijeme teda $x = 8$.

Vieme už teda, že hľadáme permutáciu tvaru $(7, 3, 1, 8, \ldots)$. No a teraz si už len stačí uvedomiť, že všetky takéto permutácie sú zaručene väčšie ako tá, ktorú sme dostali na vstupe. Hľadáme teda najmenšiu z nich – tú, kde sú všetky ďalšie čísla uvedené v rastúcom poradí. Po permutácii $(7, 3, 1, 6, 9, 8, 5, 4, 2)$ teda nasleduje permutácia $(7, 3, 1, 8, 2, 4, 5, 6, 9)$.

Celý algoritmus výroby nasledujúcej permutácie teda môžeme sformulovať nasledovne: Nech $A$ je pole, ktoré na začiatku obsahuje vstupnú permutáciu.

1. Idúc od konca nájdi najväčšie $x$ také, že $A[x] < A[x+1]$.
   (Pozícia $x$ je prvá pozícia, ktorá sa bude meniť. Väčšie $x$ nevyhovujú, lebo za touto pozíciou už je naša permutácia usporiadaná klesajúco. Ak také $x$ neexistuje, v poli $A$ je posledná permutácia, zmeníme ju na prvú a skončíme.)
2. Idúc od konca nájdi najväčšie $y$ také, že $A[y] > A[x]$.
   (Keďže $y$ bude najväčšie možné, $A[y]$ bude najmenšie možné – spomedzi hodnôt, ktoré máme na výber, je to teda najmenšia hodnota, ktorá je väčšia ako $A[x]$. Takéto $y$ určite existuje, lebo určite vyhovuje $y = x + 1$.)
3. Vymeň $A[x]$ a $A[y]$.
   (V tejto chvíli sme na pozíciu $x$ dostali hodnotu, ktorá tam patrí. A navyše vieme, že v časti, ktorú meníme, sme vymenili dve po sebe nasledujúce hodnoty. Úsek poľa $A$ od pozície $x+1$ do konca je teda naďalej usporiadaný klesajúco.)
4. Obráť úsek poľa $A$ od pozície $x + 1$ po koniec.
   (Z klesajúceho úseku, teda najväčšieho možného, urobíme rastúci, teda najmenší možný.)

Príklad: pre vyššie uvedenú permutáciu $(7, 3, 1, 6, 9, 8, 5, 4, 2)$ najskôr nájdeme $x = 4$ (pozícia hodnoty 6) a $y = 6$ (pozícia hodnoty 8). Potom vymeníme dotyčné dve hodnoty, čím dostaneme $(7, 3, 1, 8, 9, 6, 5, 4, 2)$. A na záver otočíme naopak klesajúci úsek $(9, 6, 5, 4, 2)$, čím dostaneme správnu nasledujúcu permutáciu.

Časová zložitosť tohto algoritmu je zjavne $O(n)$. Vieme ju však odhadnúť ešte o čosi presnejšie: keď priamo meníme pole obsahujúce permutáciu tak, ako sme to opísali v našom algoritme, je celkový počet krokov priamo úmerný počtu pozícií, ktoré sa zmenia – a teda optimálny.

Poznámka na záver: Tento algoritmus je v C++ implementovaný vo funkcii `next_permutation`. Navyše sa oplatí vedieť, že tento algoritmus bez zmeny funguje pre ľubovoľné hodnoty vo vstupnom poli, vrátane situácie, keď sú niektoré z hodnôt navzájom rovné.

### Listing programu:

```
void dalsia_permutacia(vector<int> &A) {
    int x = A.size()-2;
    while (x>=0 && A[x]>=A[x+1]) --x;
    if (x >= 0) { // nerobime posledna -> prva
        int y = A.size()-1;
```

```
        while (A[y] <= A[x]) --y;
        swap( A[x], A[y] );
    }
    reverse( A.begin()+x+1, A.end() );
}
```

**Protiľahlá permutácia:**

Máme z daného poradia mažoretiek vyrobiť poradie, v ktorom budú o $n!/2$ dní. Táto úloha sa bude ináč riešiť pre párne a ináč pre nepárne $n$. Pre párne $n$ to bude jednoduchšie, začneme teda týmto prípadom.

Pre každé $x$ od 1 po $n$ zjavne platí, že permutácií, ktoré začínajú číslom $x$, je presne $(n-1)!$. Ak teda v nejaký deň vidíme prvú z týchto permutácií, vieme, že prvú permutáciu začínajúcu nasledujúcim číslom uvidíme presne o $(n-1)!$ dní.

Pozrime sa teda na našu úlohu: k danej permutácii $(a_1, \ldots, a_n)$ máme zostrojiť tú, ktorú uvidíme o $n!/2$ dní. Túto hodnotu môžeme zapísať ako $(n/2)\cdot(n-1)!$, pričom pre párne $n$ je číslo $n/2$ celé.

Už teda vieme povedať, ako vyzerá prvé číslo hľadanej permutácie: je to jednoducho $b_1 = a_1 + n/2$. (Samozrejme počítané cyklicky, teda po $n$ opäť nasleduje 1.) A vlastne vieme aj to, ako bude vyzerať zvyšok hľadanej permutácie. Totiž ak by napríklad $a$ bola prvá z permutácií začínajúcich číslom $a_1$, tak hľadáme prvú z permutácií začínajúcich $b_1$, ak druhá tak druhú, a tak ďalej.

Na pozíciách 2 až $n$ máme teda niekoľkú permutáciu čísel $\{1, 2, \ldots, a_1 - 1, a_1 + 1, \ldots, n\}$. Túto teraz potrebujeme prerobiť na toľkú istú permutáciu čísel $\{1, 2, \ldots, b_1 - 1, b_1 + 1, \ldots, n\}$. To však vieme ľahko urobiť: stačí pre každé $k$ prepísať $k$-ty najmenší prvok prvej množiny na $k$-ty najmenší prvok druhej množiny. To znamená, že čísla od 1 po $\min(a_1, b_1) - 1$ aj čísla od $\max(a_1, b_1) + 1$ po $n$ zostanú nedotknuté a tým medzi nimi zmeníme hodnoty o 1.

Príklad: Nech je vstupná permutácia $(2, 7, 4, 1, 6, 8, 3, 5)$. Potom vieme, že výstupná permutácia je tvaru $(6, \ldots)$. Zvyšok dostaneme tak, že v pôvodnej postupnosti $(7, 4, 1, 6, 8, 3, 5)$ premenujeme čísla 1345678 na čísla 1234578. Výsledkom je teda permutácia $(6, 7, 3, 1, 5, 8, 2, 4)$.

Pozrime sa teraz na nepárne $n$. Môžeme zopakovať pozorovanie, že $n!/2 = (n/2) \cdot (n-1)!$. Teraz je ale $n$ nepárne, a teda $n/2$ nie je celé. Označme si $c = \lfloor n/2 \rfloor$. Potom posun o $n!/2$ dní môžeme zložiť z dvoch posunov: najskôr o $c(n-1)!$ dní a potom o $(n-1)!/2$ dní.

Posun o $c(n-1)!$ dní vyriešime rovnako ako pre párne $n$: zmeníme prvý prvok z $a_1$ na $b_1 = a_1 + c$ (opäť počítané cyklicky) a zvyšok permutácie upravíme, aby používal namiesto všetkých prvkov okrem $a_1$ všetky prvky okrem $b_1$.

Ostáva nám doriešiť posun o $(n-1)!/2$ dní. No a keďže tento posun v princípe ovplyvňuje posledných $n-1$ pozícii a $n-1$ je párne číslo, môžeme použiť ten istý postup, ktorým sme vyššie vyriešili vstupy s párnym $n$. Je tu ale jedna vec, na ktorú si musíme dať pozor: ak je hodnota $a_2$ veľká, nastane niekedy počas týchto $(n-1)!/2$ dní ďalšia zmena na prvom mieste permutácie. Napr. ak by sme z permutácie $(3,4,1,5,2)$ spravili $(n-1)!/2 = 12$ krokov, tak dostaneme permutáciu $(4,1,2,5,3)$, ktorá už nezačína hodnotou 3.

Najjednoduchšie riešenie tohto problému je nasledovné: ak je $a_2$ také veľké, že by nastalo toto pretečenie, tak namiesto $c(n-1)!$ dní sa v prvej fáze posunieme až o $(c+1)(n-1)!$ dní dopredu. Následne sa potrebujeme posunúť o $(n-1)!/2$ dní *späť*. Pri tomto posune máme istotu, že sa počas neho prvá hodnota nezmení. A zároveň vieme, že keď sa dívame len na pozície 2 až $n$, tak posun o $(n-1)!/2$ dopredu a dozadu vedie na tú istú postupnosť, takže môžeme spokojne použiť vyššie popísaný postup pre párne $n$.

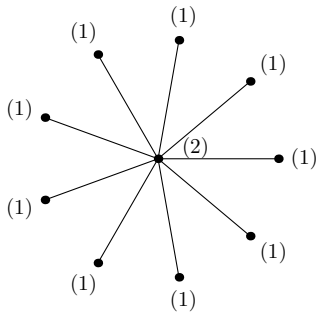
Naša implementácia má časovú zložitosť lineárnu od počtu prvkov permutácie, teda $O(n)$.

### Listing programu:

```
inline int cyklicky(int x, int N) { if (x>N) return x-N; return x; }

void premenuj(vector<int> &A, int stare, int nove) {
    for (int n=1; n<int(A.size()); ++n) {
        if (stare < nove && A[n]>stare && A[n]<=nove) --A[n];
        if (stare > nove && A[n]>=nove && A[n]<stare) ++A[n];
    }
}

void opacna_permutacia(vector<int> &A) {
    int N = A.size();
    int old=A[0];
    A[0] = cyklicky(A[0]+N/2,N);
    premenuj(A,old,A[0]);
    if (N % 2 == 0) return;

    int velke = ((N+1)/2) + (A[0]<=N/2);
    if (A[1] >= velke) {
        old = A[0];
        A[0] = cyklicky(A[0]+1,N);
        premenuj(A,old,A[0]);
    }

    premenuj(A,A[0],N);
    vector<int> B(A.begin()+1,A.end());
    opacna_permutacia(B);
    for (int n=1; n<N; ++n) A[n]=B[n-1];
    premenuj(A,N,A[0]);
}
```

## A-II-4 Grafový počítač a kompletné grafy

**Výroba kompletného grafu v lineárnom čase:**




Na klasickom počítači na výrobu kompletného grafu $K_n$ potrebujeme spraviť $\Theta(n^2)$ krokov, keďže tento graf má až $\binom{n}{2}$ hrán.

Prvé lepšie riešenie na grafovom počítači je postupne vyrobiť a pozliepať dokopy hviezdy s 1 až $n$ vrcholmi. Predstavme si totiž, že máme $K_n$, v ktorom majú všetky vrcholy značku 1, a hviezdu, v ktorej je $n$ vrcholov so značkou 1 a jeden vrchol so značkou 2. Vhodným volaním `Join` vieme tieto dva grafy spojiť tak, aby sme získali graf $K_{n+1}$. Následne jeho novému vrcholu zmeníme značku na 1, do hviezdy prídáme nový vrchol a novú hranu a môžeme celý proces začať odznova.

Takéto riešenie má časovú zložitosť $\Theta(n)$, teda lineárnu od počtu vrcholov.

**Výroba kompletného grafu pre mocniny dvoch:**

Pre jednoduchosť zatiaľ predpokladajme, že $n$ je mocninou dvoch. Ukážeme, ako celý graf $K_n$ zostrojiť v čase $\Theta(\log n)$.

Ak chceme dosiahnuť takúto časovú zložitosť, musíme v princípe vedieť pomocou konštantne veľa krokov zdvojnásobiť veľkosť zostrojeného kompletného grafu.

Zdvojnásobiť počet vrcholov je ľahké. Keď máme kompletný graf $K_n$, môžeme použiť `Join` na dve jeho kópie, pričom nastavíme `veq=none`. Tým dostaneme graf s $2n$ vrcholmi. Do kompletného grafu mu však ešte chýba celkom veľa hrán – presne sú to hrany $(x, y)$, kde $x \leq n < y$.

Samozrejme, po jednej tieto hrany pridávať nemôžeme, to by trvalo pridlho. Potrebujeme ich vedieť pridať veľa naraz. My však už máme graf, ktorý obsahuje veľa hrán: samotný $K_n$. Graf $K_{2n}$ teda „nazliepame“ z niekoľkých kópií grafu $K_n$.

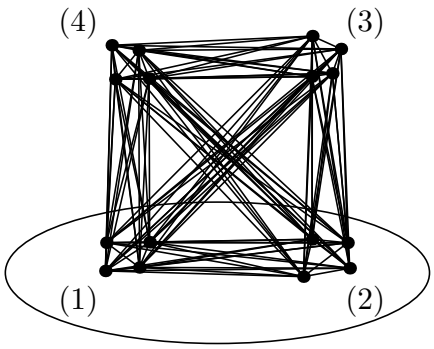
Teraz prichádza jediný „trik“ v celom riešení. Aby sme vedeli šikovne nazliepať menšie grafy na väčší, vhodne použijeme značky vrcholov.

Majme graf $K_n$, pričom $n$ je párne a platí, že v $K_n$ má polovica vrcholov značku 1 a polovica značku 2. Vyrobíme si 5 ďalších kópií tohto grafu, ktoré budú namiesto značiek (1,2) používať značky (1,3), (1,4), (2,3), (2,4) a (3,4).

Čo sa stane, keď týchto šesť grafov postupne pospájam volaniami `Join`, pričom `veq=value`? Zjavne dostaneme kompletný graf $K_{2n}$, v ktorom bude mať po $n/2$ vrcholov každú zo značiek 1, 2, 3 a 4. No a v takomto grafe už len stačí

dvoma príkazmi zmeniť všetky značky z 3 na 1 a zo 4 na 2, čím dostaneme správne označený graf $K_{2n}$.

Na obrázku je príklad grafu $K_{16}$, zostrojeného týmto postupom zo šiestich kópií grafu $K_8$. Zakrúžkovaná časť grafu zodpovedá pôvodnému grafu $K_8$, z ktorého sme konštrukciu začínali.




### Listing programu:

```
function kompletny_moc2(n: Integer): Graph;
{ funguje len pre n rovne kladnej mocnine dvoch
}
var stary, novy : Graph;
begin
    if n = 2 then begin
        novy := EmptyG;
        AddV(novy,1);
        AddV(novy,2);
        AddE(novy,1,2,undef);
        kompletny_moc2 := novy;
    end else begin
        stary := kompletny_moc2(n div 2);
        novy := stary;                              {zoberieme graf s labelmi (1,2) }
        ReplaceV(stary,2,3); novy := Join(novy,stary,IM_value,IM_any); {a (1,3) }
        ReplaceV(stary,3,4); novy := Join(novy,stary,IM_value,IM_any); {a (1,4) }
        ReplaceV(stary,1,2); novy := Join(novy,stary,IM_value,IM_any); {a (2,4) }
        ReplaceV(stary,4,3); novy := Join(novy,stary,IM_value,IM_any); {a (2,3) }
        ReplaceV(stary,2,4); novy := Join(novy,stary,IM_value,IM_any); {a (3,4) }
        ReplaceV(novy,3,1); ReplaceV(novy,4,2); {a upravime znacky z 1234 na 12 }
        kompletny_moc2 := novy;
    end;
end;
```

### Výroba kompletného grafu pre všeobecný počet vrcholov:

Najjednoduchšie riešenie pozostáva z dvoch krokov: keď máme vyrobiť $K_n$, najskôr vyrobíme $K_m$, kde $m$ je prvá mocnina dvoch väčšia alebo rovná $n$. Následne z tohto grafu zahodíme $m - n$ vrcholov, spolu s hranami, ktoré do nich vedú.

Ako ale rýchlo zahodiť veľa vrcholov? Pomôžeme si jednoduchým trikom. Najskôr zostrojíme graf s $m - n$ izolovanými vrcholmi, to vieme spraviť ľahko. Všetkým vrcholom $K_m$ dáme značku 1, všetkým vrcholom nášho nového grafu dáme značku 2. Vhodným zavolaním `Join` teraz vieme vyrobiť graf $K_m$, v ktorom $m-n$ vrcholov bude mať značku 2 a ostatných $n$ značku 1. No a vhodným zavolaním `Common` na pôvodné $K_m$ a označené $K_m$ dostaneme želaný graf $K_n$.

Akú má tento algoritmus časovú zložitosť? Stačí si všimnúť, že nutne $m \leq 2n$. Preto má zostrojenie kompletného grafu $K_m$ časovú zložitosť $\Theta(\log n)$. Ana-

logicky v nanajvýš logaritmickom čase zostrojíme aj prázdny graf s $m - n$ vrcholmi; zvyšok algoritmu už beží v konštantnom čase. Celková časová zložitosť je teda $\Theta(\log n)$.

### Listing programu:

```
{ vyrobi graf s n vrcholmi (kazdy so znackou 2) a 0 hranami }
function prazdny(n : Integer) : Graph;
var tmp : Graph;
begin
    if n=0 then prazdny:=EmptyG else begin
        tmp := prazdny(n div 2);
        tmp := Join(tmp,tmp,IM_none,IM_none);
        if n mod 2 = 1 then AddV(tmp,2);
        prazdny := tmp;
    end;
end;

{ vyrobi kompletny graf s n vrcholmi }
function kompletny(n : Integer) : Graph;
var m : Integer;
    g, h : Graph;
begin
    m := 2; while m<n do m := 2*m;
    g := kompletny_moc2(m);
    SetAllV(g,1);
    h := prazdny(m-n);
    h := Join(h,g,IM_any,IM_any); { znacky sa beru prednostne z h }
    kompletny := Common(g,h,IM_value,IM_any);
end;
```

### Výroba kompletného grafu trochu ináč:

Bez nejakého zahadzovania (alebo naopak pridávania) vrcholov sa pri konštrukcii všeobecného $K_n$ nezaobídeme – totiž jediné, čo vieme robiť, je, že z vhodne označeného $K_{2x}$ vieme vyrobiť $K_{4x}$. Grafy, ktorých počet vrcholov nie je deliteľný 4, teda pomocou nášho postupu priamočiaro vyrobiť nevieme.

Namiesto toho, aby sme zahadzovali prebytočné vrcholy všetky naraz na konci, ich však môžeme zahadzovať aj počas algoritmu. Jeden možný spôsob, ako to robiť, vyzerá nasledovne:

Nech máme vyrobiť graf $K_n$. Pre $n \leq 3$ ho vyrobíme priamo. Inak nájdeme najmenšie $m$ také, že $4m \geq n$. Rekurzívnym volaním vyrobíme graf $K_{2m}$. Z jeho šiestich kópií vyrobíme graf $K_{4m}$ a z toho následne nanajvýš tri vrcholy zahodíme, čím dostaneme želaný graf $K_n$.

Všimnite si, že pri rekurzívnom volaní klesol počet vrcholov zostrojovaného grafu približne na polovicu. Poriadnejším sformulovaním tohto pozorovania sa dá dokázať, že aj tento algoritmus má časovú zložitosť $\Theta(\log n)$.

**Efektívne hľadanie najväčšej kliky:**

Otestovať, či graf $G$ obsahuje kliku veľkosti $x$, vieme ľahko: Zostrojíme kompletný graf $K_x$ a zavoláme `Find`, nech nám nájde jeden jeho výskyt v $G$. Podľa toho, či `Find` naozaj nájde výskyt alebo vráti `EmptyG`, vieme, či sa v $G$ klika danej veľkosti nachádza alebo nie.

O grafe $G$, ktorý má $n \geq 1$ vrcholov, vieme, že jeho klikovosť je aspoň 1 (lebo má vrchol) a tiež je menej ako $n + 1$ (toľko vrcholov nemá). Správnu hodnotu môžeme na tomto intervale nájsť binárnym vyhľadávaním: budeme postupne generovať kompletné grafy rôznych veľkostí a testovať, či sa v $G$ nachádzajú.

Pri binárnom vyhľadávaní potrebujeme spraviť $\log_2 n$ testov. Tieto testy však nie sú zadarmo: pri každom z nich treba zostrojiť nejaký kompletný graf.

Ak by zadaný graf $G$ bol hustý (napr. $G = K_n$), tak každý z grafov, ktoré zostrojíme počas binárneho vyhľadávania, bude mať aspoň $n/2$ vrcholov. Preto každá iterácia binárneho vyhľadávania bude mať časovú zložitosť $\Theta(\log n)$, a teda celková časová zložitosť tohto algoritmu bude $\Theta(\log^2 n)$.

**Ešte efektívnejšie hľadanie najväčšej kliky:**

Aby sme predchádzajúci algoritmus zrýchlili, musíme sa zbaviť jeho zbytočne pomalej časti: graf, ktorý v $G$ hľadáme, zostrojujeme vždy úplne odznova.

Ak chceme dosiahnuť optimálnu časovú zložitosť binárneho vyhľadávania, potrebujeme na každú otázku spotrebovať len konštantný čas – a teda nový graf, ktorý chceme vyhľadávať, musíme vedieť v konštantnom čase vyrobiť.

Naše riešenie bude fungovať v dvoch fázach. V prvej fáze budeme postupne zostrojovať grafy $K_1$, $K_2$, $K_4$, $K_8$, atď., až kým nenájdeme prvý z nich, ktorý sa v danom grafe $G$ nenachádza. V tomto okamihu vieme, že sa klikovosť grafu $G$ nachádza v intervale $\langle 2^a, 2^{a+1}\rangle$, a tiež máme zostrojené kompletné grafy veľkostí $2^0$ až $2^{a+1}$. Keďže každý graf vieme z predchádzajúceho zostrojiť v konštantnom čase a platí $2^a \leq n$, táto fáza trvá $O(\log n)$.

Od tohto okamihu budeme v intervale, ktorý sme práve našli, binárne vyhľadávať. Všimnite si, že po každej iterácii bude dĺžka intervalu, v ktorom hľadáme, rovná mocnine dvojky. Totiž majme interval $\langle x, x + 2^y)$ pre $y > 0$. V jeho strede leží hodnota $x + 2^{y-1}$ a keď sa opýtame na ňu, dostaneme tak či onak interval presne polovičnej dĺžky. Navyše si všimnite, že stále bude platiť $x \geq 2^y$, lebo na začiatku to platí a počas hľadania sa $y$ bude zmenšovať, zatiaľ čo $x$ nie.

Počas binárneho vyhľadávania si budeme v jednej premennej udržiavať kompletný graf $K_x$, ktorého počet vrcholov bude rovný spodnej hranici intervalu, v ktorom práve hľadáme. Každá iterácia binárneho vyhľadávania bude vyzerať nasledovne: Hľadáme v intervale $\langle x, x + 2^y)$ pre nejaké $y > 0$. Nech $z = 2^{y-1}$.

Potom hodnota uprostred intervalu, v ktorom hľadáme, je práve $x + z$. Zoberieme graf $K_x$ a graf $K_z$ (ten máme ešte z prvej fázy, keďže $z$ je mocnina dvoch) a vyrobíme z nich graf $K_{x+z}$. Na ten sa opýtame. Ak ho graf $G$ obsahuje, ďalej pokračujeme s intervalom $\langle x + z, x + 2z)$, ak nie, tak s intervalom $\langle x, x + z)$.

Jediné, čo potrebujeme ukázať, je, ako z grafov $K_x$ a $K_z$ vyrobiť v konštantnom čase graf $K_{x+z}$. Konštrukcia bude podobná ako keď sme z viacerých kópií $K_n$ vyrábali $K_{2n}$. Tentokrát využijeme, že platí $x \geq 2^y$. Graf $K_x$ má teda aspoň dvakrát toľko vrcholov ako graf $K_z$. Konštrukciu grafu $K_{x+z}$ rozpíšeme po myšlienkových krokoch.

- Na začiatku sú v premenných `K_x` a `K_z` grafy $K_x$ a $K_z$, všetky ich vrcholy majú značku 1.
- Vyrobíme graf $C_{12}$, čo je $K_x$, v ktorom má $z$ vrcholov značku 2 a ostatné značku 1.
  `K_z_2 := SetAllV(K_z,2); C_12 := Join(K_z_2,K_x,any,any);`
- Vyrobíme graf $D$ s $x + z$ vrcholmi, z ktorých $z$ má značku 2, $z$ má značku 3 a ostatné značku 1. Do kompletného grafu budú grafu $D$ chýbať hrany medzi vrcholmi so značkami 2 a 3.
  `C_13 := ReplaceV(C_12,2,3); D := Join(C_12,C_13,value,any);`
- Preznačíme vrcholy grafu $C_{12}$, aby sme mali istotu, že vrcholy, ktoré majú id 1 až $z$, majú značku 1.
  `tmp := Join(K_z,C_{12},value,any);`
- Vyrobíme z práve preznačeného grafu graf $C_{123}$, v ktorom vrcholy s id 1 až $z$ preznačíme na značku 3. V tomto grafe máme teda po $z$ vrcholoch so značkami 2 a 3, ostatné vrcholy (ak tam ešte nejaké sú) majú značku 1.
  `K_z_3 := SetAllV(K_z,3); C_123 := Join(K_z_3,tmp,id,any);`
- Spojením grafov $D$ a $C_{123}$ dostávame hľadaný graf $K_{x+z}$.
  `vysledok := Join(D,C_123,value,any);`

Takto dostávame riešenie, ktorého celková časová zložitosť je $O(\log n)$. Poznámka na záver: aj pamäťová zložitosť tohto riešenia je $O(\log n)$, presnejšie, potrebujeme si súčasne pamätať $O(\log n)$ grafov. Existuje aj riešenie s časovou zložitosťou $O(\log n)$, ktoré si v každom okamihu pamätá len konštantne veľa grafov. Jedna možnosť je namiesto postupne idúcich mocnín dvojky použiť Fibonacciho čísla. Najskôr nájdeme také $x$, že hľadané číslo leží v intervale $\langle F_x, F_{x+1})$ a od tejto chvíle si stačí pamätať tri kompletné grafy: keď máme interval $\langle a, a + F_y)$, tak si pamätáme kompletné grafy veľkosti $a$, $F_y$ a $F_{y-1}$.

# Riešenia krajského kola kategórie B

## B-II-1 Hľadanie mín II

Na reprezentáciu hracieho plánu použijeme dvojrozmerné pole. Pri načítaní vstupu si do neho zaznačíme polohu mín a následne vypočítame hodnoty ostatných políčok, t.j. pre každé políčko neobsahujúce mínu spočítame počet jeho susediacich mínových políčok. (Toto bola presne vaša úloha v domácom kole.)

Algoritmus pre túto fázu má časovú zložitosť $O(RS)$, teda čas výpočtu rastie približne priamo úmerne s plochou hracieho plánu – hodnotou $RS$. To preto, že pre každé políčko spravíme len konštantné množstvo operácií: najskôr si najviac raz zaznačíme, že je na ňom mína, a ak nie je, tak prezrieme jeho najviac 8 susedov, aby sme zistili, s koľkými mínami susedí.

**Odkrývanie políčok – pomalšie riešenie:**

V druhej časti riešenia potrebujeme zistiť, ktoré políčka budú odkryté. Priamočiary prístup môže vyzerať napríklad nasledovne: Odkryjeme políčko v ľavom hornom rohu. Odteraz bude výpočet prebiehať v kolách. Každé kolo vyzerá nasledovne: Prejdeme zaradom všetky políčka hracieho plánu a pre každé z nich si položíme otázku, či ho môžeme odkryť. (Podľa definície v zadaní: políčko môžeme odkryť, ak ešte nie je odkryté a zároveň susedí s odkrytým prázdnym políčkom.) Skončíme, akonáhle sa nám stane, že v niektorom kole vôbec nič nové neodkryjeme – inými slovami, keď už na celom pláne nenájdeme žiadne políčko, ktoré môžeme odkryť.

Akú má tento postup časovú zložitosť? V každom kole je počet krokov výpočtu priamo úmerný hodnote $RS$, teda ploche hracieho plánu. A koľko najviac môže byť kôl? Jeden možný horný odhad hovorí: najviac $RS$, teda toľko ako políčok na pláne. Totiž v každom kole okrem posledného aspoň jedno nové políčko odkryjeme, a na začiatku už je jedno políčko odkryté.

O celkovej časovej zložitosti teda vieme, že v najhoršom prípade je priamo úmerná hodnote $(RS)^2 = R^2S^2$. Inými slovami, časová zložitosť je $O(R^2S^2)$.

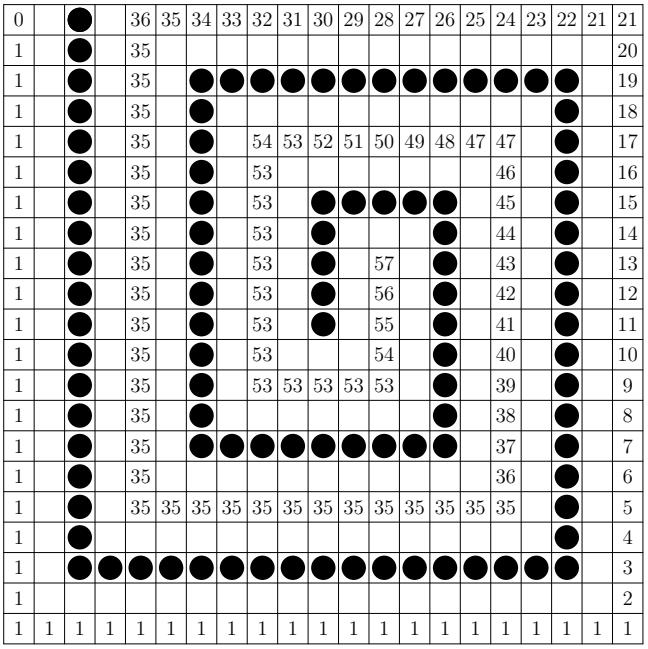
**Odbočka pre zaujímavosť: aké pomalé je to naozaj?:**

Tu si ale zvedavé povahy položia otázku: „Tak moment, toto bol predsa len horný odhad! Je vôbec tesný? Je naozaj tento algoritmus niekedy až taký pomalý?“

Ukážeme si, že naozaj áno: nájdeme konkrétny vstup, ktorý tento algoritmus donúti spraviť veľa kôl. Budeme pre jednoduchosť predpokladať, že každé kolo vyzerá rovnako:

```
for riadok:=1 to R do for stlpec:=1 to S do skus_odkryt_policko(riadok,stlpec);
```

Sľubovaným vstupom bude jednoduchá špirála, ako na obrázku vpravo. Míny sú znázornené plnými krúžkami. Prázdne políčka, t.j. políčka, ktoré nesusedia s mínou, sú práve tie políčka, ktoré na obrázku obsahujú čísla. Číslo na políčku je poradové číslo kola, v ktorom je toto prázdne políčko odkryté. (Okrem prázdnych políčok budú odkryté aj všetky políčka, ktoré s nimi susedia, na tých nám ale momentálne nezáleží.)




Vidíme, že pre $R = S = 21$ náš algoritmus posledné prázdne políčko odkryje až v kole číslo 57. A takúto špirálu vieme zostrojiť pre každé $R = S = 4k + 1$. Koľko kôl na spracovanie takejto špirály potrebujeme? Hrubý odhad hovorí, že približne $RS/8$. Prečo? Prázdne políčka tvoria približne štvrtinu všetkých políčok, a z nich polovicu (tie, ktoré ležia na úsekoch vedúcich dohora a doľava) budeme odkrývať rýchlosťou políčko za kolo.

Pre zaujímavosť, toto pozorovanie sa dá aj exaktne dokázať. Vieme spočítať (alebo uhádnuť a dokázať matematickou indukciou), že pre špirálu nakreslenú na pláne rozmerov $(4k+1)\times(4k+1)$ odkryjeme posledné prázdne políčko presne v kole číslo $2k^2 + k + 2$. Pre takýto vstup je teda potrebný počet kôl približne osminou plochy – takže skutočne potrebný počet kôl rastie priamo úmerne s veľkosťou plochy hracieho plánu.

**Odkrývanie políčok – vzorové riešenie:**

Prečo bolo predchádzajúce riešenie neefektívne? Preto, že sme pri hľadaní políčok, ktoré budeme odkrývať, vždy zbytočne prezerali celý hrací plán. Prečo je to zbytočné? Preto, že my predsa vieme, kde tie ďalšie políčka hľadať: hneď vedľa tých, ktoré sme odkryli doteraz!

Aby sme každé políčko odkryli najviac raz, použijeme pomocné dvojrozmerné pole, v ktorom si budeme značiť, či dané políčko už je alebo ešte nie je

odkryté. Na začiatku priradíme každému políčku v tomto poli hodnotu `false`, reprezentujúcu ešte neodkryté políčko.

Následne implementujeme jednoduchú rekurzívnu funkciu `odkry`, ktorá odkryje jedno dané políčko. Táto funkcia bude fungovať nasledovne: Ak odkrývané políčko nie je prázdne, nič sa neudeje. Ak však je prázdne, postupne prezrieme jeho susedov. Zakaždým, keď stretneme nejakého ešte neodkrytého, môžeme ho odkryť. To urobíme rekurzívnym volaním našej funkcie, aby sme zabezpečili, že sa odkryje aj všetko za ním. Tým splníme definíciu zadania, podľa ktorého ak sa odkryje prázdne políčko, majú sa odkryť aj všetky jeho susedné políčka. A keďže máme naše pomocné pole, tak platí, že pre každé políčko hracej plochy sa naša funkcia zavolá nanajvýš raz. Vďaka tomu je časová zložitosť nášho riešenia lineárna od veľkosti hracej plochy, teda $O(RS)$.

### Listing programu:

```
const MINA = 47;
      MAX_ROZMER = 100;
var   a : array [0..MAX_ROZMER+1, 0..MAX_ROZMER+1] of longint;
      odkryte : array [0..MAX_ROZMER+1,0..MAX_ROZMER+1] of boolean;
      { "a": policka hracieho planu, "odkryte": ci je dane policko uz odkryte}
      i, j, k, l, S, R, N, x, y : longint;

procedure odkry(i,j:longint);
var k,l:longint;
begin
    odkryte[i,j]:=true;
    if a[i,j]=0 then { prazdne policko: postupne odkryjeme jeho susedov }
        for k:=-1 to +1 do for l:=-1 to +1 do
            if not odkryte[i+k,j+l] then odkry(i+k,j+l);
end;

begin
    { do pola "a" vytvorime hraci plan }
    read(S,R,N);
    for i:=0 to R+1 do for j:=0 to S+1 do a[i,j]:=0;
    for i:=1 to N do begin readln(x,y); a[x,y]:=MINA; end;

    { vyratame hodnoty ciselnych policok }
    for i:=1 to R do for j:=1 to S do
        if a[i,j]<>MINA then { prezrieme susedov policka i,j }
            for k:=-1 to +1 do for l:=-1 to +1 do
                if a[i+k,j+l]=MINA then inc(a[i,j]);

    { odkryjeme cely okraj, ktory sluzi ako zarazka }
    for i:=0 to R+1 do for j:=0 to S+1 do
        odkryte[i,j] := (i=0) or (j=0) or (i=R+1) or (j=S+1);

    { odkryjeme policko [1,1] a vsetko, co z toho vyplyva }
    odkry(1,1);

    for i:=1 to R do begin
        for j:=1 to S do
            if not odkryte[i,j] then write('?')
            else if a[i,j]=0 then write('.')
            else write(a[i,j]);
```

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

**Alternatívne vzorové riešenie:**

Vyššie popísané riešenie je vlastne aplikáciou všeobecnejšieho grafového algoritmu, tzv. prehľadávania do hĺbky. Úlohu je možné riešiť aj nerekurzívne, napríklad pomocou príbuzného algoritmu: prehľadávania do šírky.

Rovnako ako v predchádzajúcom riešení by sme mali pomocné pole hovoriace, ktoré políčka sú už odkryté. Tentokrát by sme ale implementovali v ďalšom poli dátovú štruktúru fronta. V tej by sme si pamätali zoznam políčok, ktoré je ešte potrebné spracovať. (Teda také, ktoré sme už odkryli, ale ešte sme neprezreli ich susedov.)

Myšlienka algoritmu je nasledujúca. V prvom kroku odkryjeme políčko $(1, 1)$ a zároveň ho aj vložíme do fronty. Potom systematicky v každom kroku vyberieme jedno políčko z fronty. Ak je práve vybrané políčko prázdne, tak každé políčko, ktoré s ním susedí a ešte nie je odkryté, odkryjeme a pridáme do fronty. Keďže každé políčko hracej plochy sa takto dostane do fronty najviac raz, časová zložitosť takéhoto riešenia je, rovnako ako v predchádzajúcom prípade, $O(RS)$.

## B-II-2 Zakopaný pes

Označme si dĺžku zadaného reťazca $n$. Najjednoduchším riešením je pre každú trojicu písmen overiť, či je z nich prvé v poradí P, druhé E a tretie S. Tento prístup má však časovú zložitosť $O(n^3)$.

Takéto riešenie vieme mierne zlepšiť: pre každé z písmen P, E a S si spravíme zoznam jeho výskytov. Potom budeme prechádzať len cez tieto výskyty. Takéto riešenie má časovú zložitosť $O(pes)$, kde $p$, $e$ a $s$ sú počty výskytov P, E, a S. Toto zlepšenie nám však príliš nepomôže – v najhoršom prípade takéto riešenie ešte stále spraví počet krokov priamo úmerný tretej mocnine dĺžky vstupného reťazca. Totiž existujú vstupné reťazce dĺžky $n$, ktoré naozaj obsahujú rádovo $n^3$ výskytov slova PES. Napríklad taký reťazec dĺžky $n = 3k$, tvorený $k$ znakmi P, $k$ znakmi E a nakoniec $k$ znakmi S. Z toho vyplýva, že ak chceme dosiahnuť lepšiu časovú zložitosť ako rádovo $n^3$, nesmieme výskyty slova PES rátať po jednom.

Jeden možný trik: Pre každé písmeno E v zadanom reťazci nájdeme počet takých výskytov PES, ktoré obsahujú toto E. Zjavne takto dokopy zarátame každý výskyt slova PES práve raz.

Pre konkrétne E každý výskyt slova PES dostaneme tak, že vezmeme nejaké písmeno P naľavo od nášho E a nejaké písmeno S napravo od nášho E. A tieto P a S vieme voliť nezávisle na sebe. Stačí nám teda len zistiť, koľko máme na výber P a koľko S, a tieto dve hodnoty vynásobiť. Takto dostávame algoritmus s časovou zložitosťou $O(n^2)$.

**Listing programu:**

```
var A: AnsiString;
    i, j, pocet_p, pocet_s, n, res: longint;

begin
    ReadLn(A);
    n := Length(A);
    res := 0;
    for i := 1 to n do if A[i] = 'E' then begin
        pocet_p := 0;  for j := 1 to i-1 do if A[j] = 'P' then inc(pocet_p);
        pocet_s := 0;  for j := i+1 to n do if A[j] = 'S' then inc(pocet_s);
        res := res + pocet_p * pocet_s;
    end;
    WriteLn(res);
end.
```

Časovú zložitosť vieme zlepšiť až na $O(n)$. Stačí si pre každú pozíciu predpočítať dve čísla: počet P od nej doľava a počet S od nej doprava. Presnejšie, v poli $P$ si budeme na $i$-tej pozícii pamätať počet písmen P od začiatku reťazca po $i$-te písmeno. Hodnotu $P[i]$ ľahko zistíme v konštantnom čase z $P[i-1]$. Podobne si v poli $S$ budeme pamätať počet S od $i$-teho písmena po koniec reťazca. Hodnoty v poli $S$ spočítame v opačnom smere, od konca reťazca ku začiatku. Teda hodnotu $S[i]$ spočítame z $i$-teho znaku reťazca a z hodnoty $S[i+1]$.

**Listing programu:**

```
var A: AnsiString;
    n, i, res: longint;
    P, S: array of longint;

begin
    ReadLn(A);
    n := Length(A);
    SetLength(P,n+1); SetLength(S,n+1);
    P[0] := 0;
    for i:= 1 to n do     if A[i]='P' then P[i] := P[i-1]+1 else P[i] := P[i-1];
    S[n + 1] := 0;
    for i:= n downto 1 do if A[i]='S' then S[i] := S[i+1]+1 else S[i] := S[i+1];
    res := 0;
    for i:= 1 to n do     if A[i]='E' then res := res + P[i] * S[i];
    WriteLn(res);
end.
```

**Alternatívne vzorové riešenie:** Budeme postupne prechádzať vstupný reťazec zľava doprava a v každom kroku si pamätať, koľko výskytov slov P, PE a PES bolo v zatiaľ spracovanom prefixe.

So slovami P je to jednoduché: ich počet sa zmení (zväčší o 1) iba v prípade, ak pridaným písmenom je práve P. Nové slovo PE nám môže pribudnúť len vtedy, keď načítame písmeno E. Počet slov PE sa potom zväčší o počet slov P v zatiaľ spracovanom prefixe reťazca. S novými výskytmi slova PES je to rovnaké: v prípade, že pridaným písmenom je S, ich počet sa zväčší o počet doteraz nájdených slov PE.

Nakoniec po spracovaní celého reťazca vypíšeme vypočítaný počet výskytov slova PES. Časová zložitosť tohto riešenia je $O(n)$. A navyše si pri ňom vystačíme s konštantnou pamäťou – vstupný reťazec stačí čítať po znakoch a rovno ich spracúvať.

**Listing programu:**

```
var c : char;
    P, PE, PES: longint;

begin
    P := 0; PE := 0; PES := 0;
    while not SeekEOF do begin
        read(c);
        if c='P' then P := P+1;
        if c='E' then PE := PE+P;
        if c='S' then PES := PES+PE;
    end;
    WriteLn(PES);
end.
```

## B-II-3 Poriadok na polici II

Najprv popíšeme spôsob, ktorým budeme pre konkrétne poradie kníh argumentovať, že optimálne riešenie vyžaduje práve $k$ výmen.

Podobne ako v riešení domáceho kola použijeme pojem *inverzia* pre označenie stavu kedy kniha s väčším číslom je (ľubovoľne ďaleko) naľavo od knihy s menším číslom. Napríklad v postupnosti $2, 4, 1, 6, 3, 5$ nájdeme 5 inverzií, konkrétne sú to dvojice 2–1, 4–1, 4–3, 6–3 a 6–5. Každá inverzia má tú vlastnosť, že v ľubovoľnom riešení musíme niekedy jej prvky vzájomne vymeniť, kedže skôr alebo neskôr sa menšie číslo musí dostať naľavo od väčšieho čísla. Preto každé preusporiadanie postupnosti do správneho poradia musí použiť minimálne toľko výmen, koľko inverzií sa v danej postupnosti nachádza.

Takéto riešenie vždy aj existuje. Presnejšie, každé riešenie, ktoré sa drží postupu opísaného v zadaní, je naozaj optimálne. Totiž ak postupnosť nie je usporiadaná, tak obsahuje aspoň jednu dvojicu **po sebe idúcich** kníh, ktoré

tvoria inverziu. (Rozmyslite si, prečo.) Tým, že hocijakú takúto dvojicu vymeníme, znížime celkový počet inverzií o jednu. Každý algoritmus, ktorý takéto dvojice hľadá a vymieňa, teda vyrobí optimálne riešenie.

Tým sme dokázali, že ľubovoľné optimálne riešenie potrebuje na usporiadanie presne toľko výmen, koľko má postupnosť inverzií. Našou úlohou je teda k danému $n$ a $k$ nájsť postupnosť $n$ čísel, ktorá má práve $k$ inverzií.

**Prvé dve podúlohy:**

Pre $n = 6$ a $k = 17$ žiadna vyhovujúca postupnosť neexistuje. To vyplýva z výsledku, ktorý sme si dokázali v domácom kole: každú postupnosť $n$ čísel sa dá usporiadať na nanajvýš $n(n-1)/2$ výmen, teda pre $n = 6$ vždy stačí nanajvýš 15 výmen. (Iný pohľad: každá 6-prvková postupnosť má nanajvýš $5 + 4 + 3 + 2 + 1 = 15$ inverzií.)

Príkladom 10-prvkovej postupnosti, ktorá obsahuje 22 inverzií, je postupnosť $10, 9, 6, 1, 2, 3, 4, 5, 7, 8$.

Obsahuje nasledovné inverzie: 10–1, 10–2, ..., až 10–9 (9 inverzií), 9–1, 9–2, ..., až 9–8 (ďalších 8 inverzií), 6–1, 6–2, 6–3, 6–4 a 6–5. Samozrejme, existuje aj mnoho ďalších postupností. Postup, ako sme našli túto, popíšeme vo zvyšku vzorového riešenia.

**Jednoduchšie ale pomalšie riešenie tretej podúlohy:**

Na vytvorenie $n$-prvkovej postupnosti, ktorá by obsahovala práve $k$ inverzií, môžeme napríklad využiť postup presne opačný od toho, ktorý používame pri jej usporadúvaní. Začneme s usporiadanou postupnosťou, ktorá má 0 inverzií. Následne v každom kroku nájdeme a vymeníme dva susedné prvky tak, aby vytvorili inverziu. (Teda nájdeme dvojicu „menší väčší“ a zmeníme ju na „väčší menší“). Tým zvýšime počet inverzií v tejto postupnosti o 1. Toto zjavne môžeme robiť až kým nedostaneme obrátene usporiadanú postupnosť. A tá už obsahuje maximálny počet inverzií: $1 + 2 + \ldots + (n-1)$.

Kostra programu pre takéto riešenie by mohla vyzerať nasledovne:

```
if K > (N*(N-1) div 2) then begin writeln('postupnost neexistuje'); halt; end;
for i:=1 to N do a[i]:=i;
for j:=1 to K do begin
    for i:=1 to N-1 do
        if a[i] < a[i+1] then begin
            swap(a[i],a[i+1]);   { vymenime hodnoty a[i] a a[i+1] }
            break;
        end;
end;
```

Časová zložitosť tohto riešenia je $O(kn)$, pretože $k$-krát vyhľadávame dvojicu susedných prvkov, ktoré vymeníme. No a pri každom hľadaní v najhoršom prípade prejdeme celú postupnosť – teda každé hľadanie trvá nanajvýš rádovo $n$ krokov. Tento postup je najpomalší vtedy, keď je $k$ najväčšie, čiže keď $k = n(n-1)/2$. Pre takto zvolené $k$ je počet krokov tohto algoritmu kubickou funkciou čísla $n$, teda $\Theta(n^3)$.

**Lepšie riešenie tretej podúlohy:**

Na predchádzajúcom riešení bolo neefektívne to, že sme dvojicu na výmenu hľadali vždy od začiatku. To vôbec nie je potrebné – stačí nám ľubovoľná taká dvojica. Program teda zrýchlime tak, že budeme po postupnosti dokola prechádzať od začiatku až úplne po koniec a **zakaždým**, keď stretneme vhodnú dvojicu, tak ju vymeníme.

V programe by to vyzeralo takto:

```
if K > (N*(N-1) div 2) then begin writeln('postupnost neexistuje'); halt; end;
for i:=1 to N do a[i]:=i;
while K > 0 do begin
    for i:=1 to N-1 do
        if a[i] < a[i+1] then begin
            swap(a[i],a[i+1]);
            dec(K);
            if K=0 then break; { len ak uz koncime }
        end;
end;
```

Ako veľmi sme tým zrýchlili naše riešenie? Pozrime sa, čo sa napríklad stane pre $n = 6$ a najväčšie možné $k$.

- Začneme s postupnosťou $1, 2, 3, 4, 5, 6$.
- Počas prvej iterácie while-cyklu náš program postupne vymení dvojice 1–2 (teraz máme postupnosť 2,1,3,4,5,6), potom 1–3, ..., až 1–6 a dostane postupnosť $2, 3, 4, 5, 6, 1$.
- Počas druhej iterácie while-cyklu náš program postupne vymení dvojice 2–3, 2–4, ..., až 2–6 a dostane postupnosť $3, 4, 5, 6, 2, 1$.
- A už je zjavné, ako to bude pokračovať ďalej: po ďalších troch (teda dokopy piatich) iteráciách dostaneme postupnosť $6, 5, 4, 3, 2, 1$ a skončíme.

Zovšeobecnením tejto úvahy ľahko dokážeme, že pre konkrétne $n$ sa nikdy nevykoná viac ako $n - 1$ iterácií while-cyklu, bez ohľadu na to, aké veľké je $k$.

Časová zložitosť tohto algoritmu je teda v najhoršom prípade kvadratická od $n$, čiže $O(n^2)$.

Poznámka: Za povšimnutie stojí, že to, čo sme práve naprogramovali, je vlastne obyčajný triediaci algoritmus BubbleSort, postupne preusporadúvajúci prvky do poradia od najväčšieho po najmenší.

**Vzorové riešenie tretej podúlohy:**

Existuje viacero riešení, ktoré vedia hľadané poradie čísel vyrobiť v optimálnom čase – lineárnom od $n$. Ukážeme si dve podobné. Prvé z nich dostaneme ďalším zrýchlením predchádzajúceho riešenia.

Môžeme si totiž všimnúť, čo presnejšie sa pri predchádzajúcom algoritme udeje. V prvej iterácii while-cyklu presunieme jednotku zo začiatku na koniec. Na to potrebujeme $n-1$ výmen, a teda vyrobíme $n-1$ inverzií. V druhej iterácii presunieme dvojku zo začiatku až skoro na koniec – tesne pred jednotku. Na to potrebujeme $n-2$ výmen, a teda vyrobíme $n-2$ inverzií. A tak ďalej. Prvých niekoľko prvkov posunieme najďalej, ako sa dá, a potom (v poslednej ešte vykonanej iterácii while-cyklu) jeden prvok posunieme toľkokrát, koľko inverzií nám ešte chýba.

Uvažujme napríklad $n = 6$ a $k = 13$. Pre tieto hodnoty si priebeh vyššie popísaného algoritmu môžeme zhrnúť nasledovne:

1. Presuň jednotku najďalej ako to ide, teda až na koniec (5 inverzií).
2. Presuň dvojku najďalej ako ide (ďalšie 4 inverzie).
3. Presuň trojku najďalej ako ide (ďalšie 3 inverzie, to už je dokopy 12).
4. Následne ešte vymeň štvorku s päťkou.

A teraz si už len stačí všimnúť, že prvé tri kroky predchádzajúceho riešenia vôbec netreba simulovať. Je predsa jasné, že ako bude vyzerať pole po tom, ako presunieme na koniec čísla 1, 2 a 3: na začiatku budú v rastúcom poradí čísla, ktoré sme nepresúvali, a za nimi v klesajúcom poradí tie, ktoré sme presúvali. V našom prípade teda $4, 5, 6, 3, 2, 1$. A takéto poradie vieme rovno vyrobiť.

A teda namiesto toho, aby sme simulovali každú jednu výmenu prvkov, si naše riešenie spočíta, koľko prvkov treba posunúť až na koniec, potom vyrobí taký obsah poľa, ktorý by vznikol ich posunutím, a následne ešte dorobí posledných pár inverzií.

**Listing programu:**

```
procedure swap(var a,b : longint);
var c : longint;
begin c:=a; a:=b; b:=c; end;

var   a : array[1..1000] of longint;
      i,N,K,mam,presunul : longint;
```

```
begin
    if K > (N*(N-1) div 2) then
        writeln('Taka postupnost neexistuje')
    else begin
        { spocitame, kolko prvkov treba presunut az na koniec }
        mam := 0;                    { kolko inverzii uz mam }
        presunul := 0;         { kolko prvkov som uz presunul }
        while true do begin
            { dalsi prvok by vyrobil (N-presunul-1) inverzii }
            if (mam + N - presunul - 1) > K then break;
            mam := mam + N - presunul - 1;
            inc(presunul);
        end;

        { vyrobime pole po danom pocte presunov }
        for i:=1 to N-presunul do a[i] := i+presunul;
        for i:=1 to presunul do a[i+N-presunul] := presunul+1-i;

        { a uz len dorobime chybajuci pocet inverzii a vypiseme }
        for i:=1 to K-mam do swap(a[i],a[i+1]);
        for i:=1 to N do writeln(a[i]);
    end;
end.
```

Počas každej fázy (počítanie, výroba poľa, záverečné úpravy a výpis) je počet krokov tohto algoritmu priamo úmerný veľkosti poľa. Celková časová zložitosť je teda $O(n)$.

**Stručnejšie vzorové riešenie tretej podúlohy:**

Na záver si ukážeme, ako jedno hľadané poradie ľahko zostrojiť po jednotlivých číslach zľava doprava. Keď už rozumieme tomu, ako vznikajú inverzie, bude to veľmi jednoduché.

Pozrime sa na hodnoty $n$ a $k$. Ak $k \leq n-1$, vieme úlohu vyriešiť jednoducho: na prvé miesto dáme číslo $k+1$. To nám vyrobí presne $k$ inverzií (ono samo s každým z čísel od 1 po $k$). Žiadne ďalšie inverzie už teda nechceme, preto ostatné čísla vypíšeme v rastúcom poradí a skončili sme.

A čo v opačnom prípade? Nech $k \geq n$, teda ešte treba viac inverzií ako vieme vyrobiť len prvým číslom. V takomto prípade určite nič nepokazíme, keď na začiatok umiestnime číslo $n$. To nám vyrobí najviac inverzií, presne $n-1$. A čo nám zostalo? Čísla 1 až $n-1$, z ktorých treba vyrobiť postupnosť s $k-(n-1)$ inverziami. A to je presne tá istá úloha ako na začiatku, len už máme menej čísel a menej inverzií. Zopakujeme teda odznova ten istý postup.

**Listing programu:**

```
var i,N,K : longint;
begin
    readln(N,K);
    if K > (N*(N-1) div 2) then
        begin writeln('Taka postupnost neexistuje'); halt; end;
    while K > N-1 do begin writeln(N); dec(K,N-1); dec(N); end;
    writeln(K+1);
    for i:=1 to K do writeln(i);
    for i:=K+2 to N do writeln(i);
end.
```

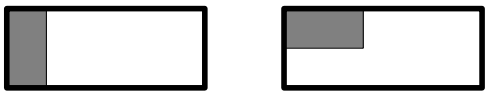
Časová zložitosť tohto programu je, rovnako ako v predchádzajúcom riešení, lineárna od $n$. Mimochodom, všimli ste si, že výsledné poradie čísel vyrobíme presne „opačne“ ako v predchádzajúcom riešení? V tomto riešení najskôr akoby presunieme $n$ z konca na začiatok, potom $n-1$ z konca za $n$, a tak ďalej.

## B-II-4 Dláždenie pre pokročilých

**Dláždenia obdĺžnikov:**

Označme si počet vydláždení obdĺžnika $2 \times n$ ako $D_n$. Už vieme, že $D_1 = 1$, $D_2 = 2$ a $D_3 = 3$. Čo vieme povedať o $D_n$ pre väčšie $n$?

Predstavme si, že ideme vyrobiť jedno konkrétne dláždenie obdĺžnika $2 \times n$. Ako prvé sa musíme rozhodnúť, či štvorček v ľavom hornom rohu pokryjeme zvislou alebo vodorovnou parketou:



Ak sme si vybrali zvislú parketu, zostal nám ešte nevydláždený (na obrázku biely) obdĺžnik rozmerov $2 \times (n-1)$. A pre ten máme na výber presne $D_{n-1}$ rôznych dláždení.

Ak sme si vybrali vodorovnú parketu, pozrime sa teraz na štvorček v ľavom dolnom rohu. Tam už nemáme na výber: musíme ho tiež pokryť vodorovnou parketou. A akonáhle to spravíme, bude nevydláždená časť tvoriť obdĺžnik rozmerov $2 \times (n-2)$. V tomto prípade máme teda presne $D_{n-2}$ možností, ako dokončiť naše dláždenie.

Tým sme práve dokázali, že pre každé $n \geq 2$ platí vzťah: $D_n = D_{n-1} + D_{n-2}$.

Pomocou neho môžeme ľahko spočítať požadované počty dláždení. Odpovede sú v nasledujúcej tabuľke:

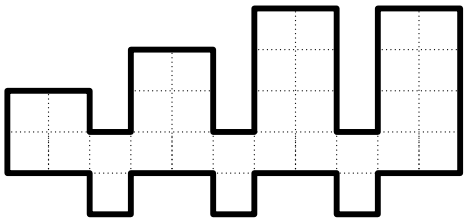
| $n$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $D_n$ | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233 | 377 | 610 | 987 |

Poznámka na záver: postupnosť, ktorú sme práve vypočítali, je známa pod menom Fibonacciho postupnosť.

**Miestnosť so 150 dláždeniami:**

Číslo 150 si môžeme zapísať ako $2 \times 3 \times 5 \times 5$. Najjednoduchší spôsob, ako vyrobiť miestnosť so 150 dláždeniami, je vhodne spojiť štyri menšie miestnosti, ktoré budú mať 2, 3, 5 a 5 možných vydláždení. A takéto miestnosti už poznáme: sú to napríklad obdĺžniky $2 \times 2$, $2 \times 3$ a $2 \times 4$.

Jedno možné riešenie vyzerá nasledovne:

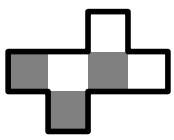


Štvorčeky v spodnom riadku zjavne musíme pokryť zvislými parketami. A tým, že tieto parkety položíme, sa nám miestnosť rozpadne na niekoľko nezávislých častí, ktoré môžeme dláždiť každú zvlášť. Preto celkový počet dláždení dostaneme vynásobením počtov dláždení pre jednotlivé časti.

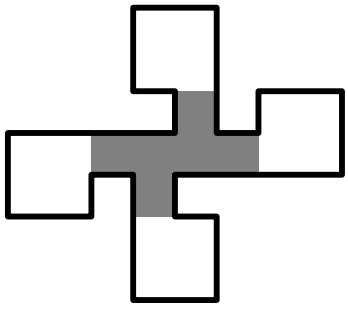
**Nie každá miestnosť sa dá vydláždiť:**

V prvom rade zabudnime na druhú podmienku zo zadania a zamyslime sa nad vyváženými miestnosťami. Stačí samotná vyváženosť na to, aby sa miestnosť dala vydláždiť?

Ľahko nahliadneme, že nestačí. Existuje dokonca protipríklad tvorený iba šiestimi štvorčekmi:



Nás ale zaujímajú len také miestnosti, v ktorých nemáme štvorčeky s len jedným susedom. Skúsme teda náš protipríklad „vylepšiť“ – prirobíme každému z krajných štvorčekov susedov:



Je zjavné, že táto miestnosť je vyvážená a každý jej štvorček má aspoň dvoch susedov. Ale je tiež zjavné, že táto miestnosť nemá žiadne platné vydláždenie. (Nemôžeme položiť parketu tak, aby pokryla jedno políčko v bielom štvorci a jedno mimo neho, lebo by nám ostali tri biele políčka a tie nemáme ako vydláždiť. Musíme teda každý z pridaných štvorcov vydláždiť samostatne. Tým nám ale zostane miestnosť z predchádzajúceho obrázka a tá sa vydláždiť nedá.)

# Riešenia celoštátneho kola kategórie A

## A-III-1 Romantické básničky

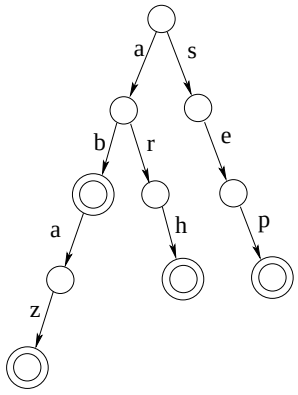
Vyriešime najskôr jednoduchšiu úlohu: ku zadanému slovu nájsť v lexikóne **ľubovoľné** z tých, ktoré sa s ním najviac rýmujú.

Jedno možné efektívne riešenie je použiť dátovú štruktúru nazývanú písmenkový strom (po anglicky *trie*). Písmenkový strom je zakorenený strom, v ktorom každý vrchol má najviac 26 synov, a hrany do synov sú označené rôznymi písmenkami (od `a` po `z`). Každá cesta z koreňa nadol zodpovedá slovu, ktoré si „prečítame“ na hranách, po ktorých ideme.

Písmenkový strom sa dá použiť na uloženie množiny slov. Jednoducho vytvoríme všetky vrcholy, ktoré treba na to, aby sme si mohli na ceste z koreňa dole „prečítať“ každé z našich slov, a následne označíme tie vrcholy, kde niektoré z uložených slov končí. (Napríklad si do toho vrcholu napíšeme poradové číslo slova, ktoré sa tam končí.)

Ako pomocou písmenkového stromu vyriešiť našu zjednodušenú úlohu? Úplne jednoducho. Na začiatku zoberieme všetky slová v lexikóne a vložíme ich do písmenkového stromu – lenže nie normálne, ale odzadu, teda začínajúc ich posledným písmenom.

Príklad takéhoto stromu si môžete pozrieť na obrázku 1. Dvojitým krúžkom sú znázornené vrcholy, kde niektoré zo slov končí. Z každého vrcholu dodola vedie v skutočnosti až 26 hrán – tie, ktoré nevedú nikam (NULL pointre), sme pre prehľadnosť nekreslili.




Obr. 1: Trie, do ktorého sme odzadu vložili slová `ba`, `hra`, `pes` a `zaba`.

Keď nám teraz príde ľubovoľná otázka, môžeme aj tú čítať odzadu a zároveň putovať dodola po písmenkovom strome. Napríklad majme lexikón z obrázku 1 a predstavme si, že nám prišla otázka `kobra`. Čítajúc písmená `a` a `r` sa dostaneme do jedného z vrcholov v našom písmenkovom strome. Odtiaľ však už ďalej dodola hrana na písmeno `b` nevedie, vieme teda, že lepší rým ako dve písmená nevyrobíme.

Ako teraz nájsť jedno z najlepšie sa rýmujúcich slov? Na to sa stačí z vrcholu, kde práve sme, ľubovoľnou cestou pohnúť dodola, a to až kým nenájdeme vrchol, kde nejaké slovo končí (respektíve začína). Toto sa nám skôr či neskôr musí stať – najneskôr vtedy, keď prídeme do listu, keďže každý list nutne zodpovedá nejakému slovu.

V našom príklade by sme mali jedinú možnosť: pohnúť sa dodola na písmeno `h`, čím sme úspešne zistili, že najlepším rýmom ku slovu `kobra` je slovo `hra`.

V zadaní súťažnej úlohy ale bola ešte jedna požiadavka: ak je viac možných odpovedí, musíme nájsť tú lexikograficky najmenšiu (čítané samozrejme odpredu). Vyššie popísané riešenie teda musíme upraviť tak, aby vedelo splniť aj túto požiadavku.

V prvom rade si rozmyslite, že samotný písmenkový strom, ktorý sme si zostrojili, nám nestačí – nevieme z neho lokálne zistiť, ktoré slovo si vybrať, ak ich máme na výber viac. Napríklad si predstavte, že máme v lexikóne slová `zaba` a `huba` a hľadáme najlepší rým k slovu `hudba`. Pri hľadaní nájdeme vrchol zodpovedajúci ich spoločnému suffixu `ba`. Z neho dodola ale vedie viac možností – hrana na písmeno `a` aj hrana na písmeno `u`. A správna odpoveď `huba` sa v tomto prípade skrýva pod písmenom `u` – to ale nemáme ako v danej chvíli zistiť, lebo prvé písmená slov `zaba` a `huba` sú ukryté niekde hlbšie v strome.

Keďže primárnym kritériom hodnotenia riešenia je rýchlosť odpovedania na otázky, potrebujeme si hľadané odpovede nejak predpočítať. Presnejšie, v každom vrchole nášho písmenkového stromu si budeme pamätať odpoveď – číslo slova z lexikónu, ktoré máme vypísať na výstup, ak sme pri hľadaní rýmu skončili v danom vrchole.

Ako tieto hodnoty spočítať efektívne? Pomohlo by nám, keby slová v lexikóne boli usporiadané podľa abecedy. Potom totiž stačí v tomto poradí vkladať ich reverzy do písmenkového stromu a v každom vrchole si zapamätať číslo slova, pri spracúvaní ktorého sme dotyčný vrchol vytvorili.

No a ako najefektívnejšie usporiadať lexikón? Jednoducho: využijeme na to opäť písmenkový strom. (Spravíme si inú premennú, ale použijeme tú istú im-

plementáciu písmenkového stromu.) Tentokrát doň najskôr klasicky (odpredu) vložíme všetky slová z lexikónu, a následne tento strom prehľadáme do hĺbky. Synov každého vrcholu budeme spracúvať v abecednom poradí. Poradie, v ktorom stretneme vrcholy zodpovedajúce koncom slov, takto bude zjavne zodpovedať ich abecednému poradiu.

Na usporiadanie lexikónu pomocou písmenkového stromu potrebujeme čas priamo úmerný súčtu dĺžok jeho slov. Taktiež vytvorenie písmenkového stromu reverzov slov z lexikónu vieme potom spraviť v rádovo rovnakom čase. Následne spracovanie každej otázky prebehne priamočiaro: načítame otázku, v čase najviac lineárnom od jej dĺžky nájdeme vrchol zodpovedajúci najlepšiemu rýmu, z toho vrcholu v konštantnom čase vieme číslo slova, ktoré je správnou odpoveďou, a to len vypíšeme.

Aj fáza predspracovania, aj spracovanie jednej otázky, má teda optimálnu časovú zložitosť. (Lepšie to nejde, lebo musíme aspoň čítať vstup a vypisovať výstup.)

### Listing programu:

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

const int MAX_V = 1000047;
int n,m;
vector <string> lexikon;

// Vrati reverz stringu
string reversed(string s) {
        reverse(s.begin(),s.end());
        return s;
}

struct node {
        int sons[26], id; // id = Ktoremu slovu patrim?
        node();
};

node::node() {
        id=-1;
        for (int i=0;i<26;++i) sons[i]=-1;
}

struct trie {
        node a[MAX_V]; // Miesto pointrov pouzijeme pole
        int root,num;
        trie();
        void insert(const string &s, int n, bool vsetky);
        void lexicographic(int pos);
        int find(const string &s);
```

```
};

trie sorter, suffix; // Jeden strom na triedenie odpredu, druhy na reverzy

trie::trie() {
        root=0;
        num=1;
}

// Vlozi do trie retazec s
// ak vsetky==false, vklada klasicky: posledny vrchol oznaci n-kom
// ak vsetky==true, oznaci n-kom vsetky novovytvorene vrcholy
void trie::insert(const string &s, int n, bool vsetky=false) {
        int node=root;
        if (vsetky && a[node].id==-1) a[node].id=n;
        for (unsigned i=0;i<s.size();++i) {
                if (a[node].sons[s[i] - 'a']==-1) a[node].sons[s[i] - 'a'] =
num++;
                node=a[node].sons[s[i] - 'a'];
                if (vsetky && a[node].id==-1) a[node].id=n;
        }
        if (!vsetky) a[node].id=n;
}

// Prechadza strom v lexikografickom poradi (prehladava ho do hlbky)
// Vzdy ked najde slovo, vlozi jeho reverz do stromu reverzov
void trie::lexicographic(int pos=0) {
        if (a[pos].id!=-1) { // Ak vo vrchole konci slovo
                suffix.insert( reversed(lexikon[a[pos].id]), a[pos].id, true );
        }
        for (int i=0;i<26;i++) { // Rekurzivne prehladaj
                if (a[pos].sons[i]!=-1)
                lexicographic(a[pos].sons[i]);
        }
}

// Najde najdlhsi prefix a vrati ID slova, ktoremu patri
int trie::find(const string &s) {
        int node=root;
        for (unsigned i=0;i<s.size();++i) {
                int next = a[node].sons[ s[i]-'a' ];
                if (next==-1) break; else node=next;
        }
        return a[node].id;
}

int main() {
        string s;
        cin >> n;
        for (int i=0;i<n;++i) {
                cin >> s;
                lexikon.push_back(s);
                sorter.insert( lexikon[i], i );
        }
        sorter.lexicographic();
        cin >> m;
        for (int i=0;i<m;++i) {
                cin >> s;
                cout << lexikon[ suffix.find(reversed(s)) ] << endl;
        }
}
```

## A-III-2 Úrad

Štruktúra úradnikov tvorí strom, v ktorom sú úradníci vrcholy. Úlohy, ktoré vykonávajú úradnici budeme niekedy označovať ako farby vrcholov.

**Pomalšie riešenia:**

Najjednoduchšie riešenie je pustiť z každého vrcholu prehľadávanie smerom nadol, ktoré hľadá vrcholy farby 1. Pokiaľ narazíme na vrchol rovnakej farby, tak prehľadávanie zastavíme. Pokiaľ narazíme na vrchol farby 1, tak vieme, že daný úradník niečo robí. Jedno prehľadávanie nám trvá $O(n)$ času, takže máme celkový čas $O(n^2)$.

Tento horný odhad časovej zložitosti je síce správny, ale nie tesný. Všimnime si ľubovoľnú konkrétnu farbu $f$. Z každého vrcholu tejto farby raz spustíme prehľadávanie. To sa ale zastaví, keď nájde pod sebou iné vrcholy tej istej farby, pod nimi už teda neprehľadáva. Keď teda uvažujeme dokopy všetky prehľadávania z vrcholov farby $f$, vieme, že prezrú dokopy každú hranu stromu najviac raz. Preto majú všetky tieto prehľadávania spolu časovú zložitosť $O(n)$. A keďže máme $m$ farieb, je celková časová zložitosť $O(mn)$.

Iné riešenie s rovnakou časovou zložitosťou: Budeme priamo riešiť každú farbu jedným prechodom. Pre každú farbu $f \in \{2, 3, \ldots, m\}$ vykonáme nasledovný proces: Zmeníme strom na orientovaný graf, ktorého hrany smerujú smerom nahor. Do grafu pridáme nový „štartovný“ vrchol $X$, z neho budú smerovať všetky hrany do vrcholov s farbou 1. Zároveň pri spracúvaní farby $f$ neuvažujeme hrany vedúce dohora z vrcholov farby $f$. Teraz už len spustíme z vrcholu $X$ prehľadávanie. Ak narazíme na vrchol farby $f$, tak vieme, že tento úradník niečo robí. Takto nájdeme všetkých úradnikov farby $f$, čo niečo robia. Jedno prehľadávanie opäť trvá čas $O(n)$. Celkový čas máme teda $O(mn)$.

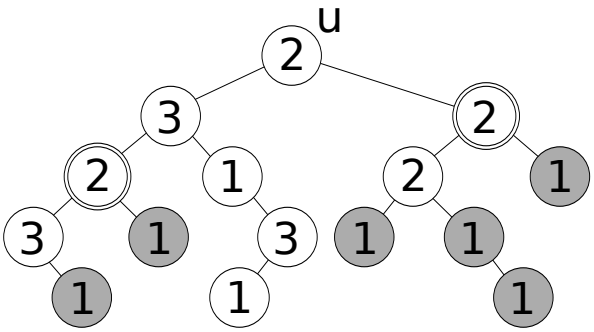
**Vzorové riešenie:**

Medzi všetkými (aj nie priamymi) podriadenými úradníka $u$ je niekoľko podriadených pre styk s verejnosťou. Ich počet označíme $P(u)$. Všetky hodnoty $P(u)$ vieme ľahko spočítať v čase $O(n)$ pomocou jedného prehľadávania do hĺbky. Stačí si uvedomiť, že $P(u)$ vieme spočítať ako súčet všetkých $P(x)$, kde $x$ je priamym podriadeným $u$, a ešte $+1$ ak samotný $u$ robí styk s verejnosťou.

Predstavme si teraz, že z úradníka $u$ sme spustili smerom nadol prehľadávanie z prvého vyššie popísaného pomalšieho riešenia – zisťujeme teda, odkiaľ

všadiaľ ku nám môžu prísť klienti. Zamestnancov pre styk s verejnosťou, ktorých toto prehľadávanie nájde, budeme volať *nepokrytými* a ich počet označíme $N(u)$. Zjavne nám stačí pre každé $u$ spočítať $N(u)$: totiž $u$ nič nerobí práve vtedy, keď je táto hodnota nulová.

Okrem zamestnancov pre styk s verejnosťou toto prehľadávanie nájde aj niekoľko iných zamestnancov, podriadených úradníka $u$, ktorí majú rovnakú farbu ako on. Týchto budeme volať jeho *poskokmi*. Poskok úradníka $u$ je teda každý úradník $v$, ktorý má rovnakú farbu ako $u$ a zároveň platí, že $u$ je jeho najbližší nadriadený s touto farbou.




Obr. 2: Ukážka podstromu pre nejakého úradníka $u$.

Na obrázku 2 je príklad časti hierarchie. Čísla vo vrcholoch sú farby, teda typy úloh, ktoré jednotliví úradníci robia. Úradník $u$ je úplne hore (v koreni stromu). Dvojitým krúžkom sú vyznačení jeho poskokovia. Sivou farbou sú vyznačení tí úradníci pre styk s verejnosťou, ktorých poskokovia pokryli.

Hodnotu $N(u)$ teraz vieme spočítať tak, že od hodnoty $P(u)$ odpočítame tých úradníkov, ktorých pokryjú poskokovia vrcholu $u$ – teda odpočítame všetky $P(v)$, kde $v$ je poskokom $u$. (Všimnite si, napr. pomocou obrázku 2, že toto môžeme spraviť, lebo množiny pracovníkov pokrytých rôznymi poskokmi úradníka $u$ sú disjunktné. Formálny dôkaz by vyzeral takto: Uvažujme ľubovoľného úradníka $x$ spomedzi $P(u)$ úradníkov pre styk s verejnosťou, ktorí sú podriadenými $u$. Na ceste medzi ním a $u$ leží najviac jeden poskok $u$ – viac ako jeden tam byť nemôže, lebo potom by ten z nich, ktorý je ďalej od $u$, nebol poskokom.)

Potrebujeme teda efektívne nájsť pre každého úradníka všetkých jeho poskokov. Všimnime si, že každý úradník iný ako minister je poskokom práve jedného iného. Preto nám pre každý vrchol stačí nájsť toho jeho nadriadeného, komu je poskokom. Ak to zvládneme v lineárnom čase pre celý strom, máme lineárne riešenie úlohy.

Najbližšieho nadriadeného rovnakej farby vieme pre všetky vrcholy nájsť takto: Najprv si naicializujeme $m$ zásobníkov, jeden pre každú farbu. Teraz začneme strom prehľadávať do hĺbky. Keď prídeme do vrcholu $v$ s farbou $f$, tak sa pozrieme na vrch zásobníku pre farbu $f$. Vrchol, ktorý je tam umiestnený, je ten, ktorému je aktuálny vrchol $v$ poskokom. Toto si zapamätáme. Následne vložíme do zásobníku pre farbu $f$ vrchol $v$ a pokračujeme prehľadávaním jeho podstromu. A keď toto prehľadávanie skončí a z vrchola $v$ odchádzame, tak ho zase vyberieme z vrchu príslušného zásobníku.

Celkovo teda vykonáme nasledovné:

1. Vypočítame prehľadávaním do hĺbky hodnoty $P(u)$ – počet podriadených pre styk s verejnosťou.
2. Vypočítame prehľadávaním do hĺbky pomocou $m$ zásobníkov pre každého úradníka jeho poskokov.
3. Pre každého úradníka $u$ vypočítame jeho hodnotu $N(u)$.
4. Úradníkov s $N(u) = 0$ vypíšeme na výstup.

Každý z týchto krokov nám zaberie $O(n)$ času a spotrebujeme $O(n)$ pamäte. V programe prvé dva kroky robíme súčasne počas jedného prehľadávania. Treba ešte dodať, že ministra je potrebné ošetriť samostatne – ten nič nerobí len vtedy, ak by nič nerobil v každej možnej farbe.

### Listing programu:

```
#include <cstdio>
#include <vector>
#include <stack>
using namespace std;

struct Vrchol {
    vector<int> next;
    int color, seen, boss;
};

int N, M;
Vrchol v[100000];
stack<int> color_stack[100000];

// dfs(x) prehlada podstrom pod vrcholom x,
//   vrati pocet uplne vsetkych 1 v tomto podstrome (vratane pripadne vrcholu x)
//   a tiez ku kazdemu vrcholu najde najblizsieho predka rovnakej farby
int dfs(int x) {
    v[x].seen = (v[x].color==1 ? 1 : 0); // 1 ak my sme farby 1, 0 inak

    // Ak sme v koreni, vlozime ho do vsetkych zasobnikov, inak len do spravneho
    //   a tiez si zapamatame najblizsieho predka rovnakej farby ako mame my
    if (x == 0) {
        for (int i = 0; i <= M; i++) color_stack[i].push(0);
    } else {
        v[x].boss = color_stack[ v[x].color ].top();
```

```
        color_stack[ v[x].color ].push(x);
    }

    // Prehladame strom pod sebou
    for (unsigned i = 0; i < v[x].next.size(); i++)
        v[x].seen += dfs(v[x].next[i]);

    // Upraceme spravny zasobnik a vratime spocitany pocet jedniciek
    color_stack[v[x].color].pop();
    return v[x].seen;
}

int main() {
    scanf("%d%d", &N, &M);
    for (int i = 1; i < N; i++) {
        v[i].seen = 0;
        int p;
        scanf("%d", &p);
        v[p-1].next.push_back(i);
    }
    v[0].color = 0;
    for (int i = 1; i < N; i++) scanf("%d", &v[i].color);
    dfs(0);

    // v premennej seen v kazdom vrchole mame teraz pocet vsetkych jednotiek
    // pod nim, odratame tie pokryte vrcholmi, ktorym je on boss
    for (int i=1; i<N; i++) v[ v[i].boss ].seen -= v[i].seen;

    // vypiseme vsetkych, ktori nemaju pod sebou nepokryte jednotky
    for (int i=1; i<N; i++) if (v[i].seen==0) printf("%d\n",i+1);
}
```

## A-III-3 Grafový počítač pomáha krtkovi

**Triedenie:**

Našim cieľom je nájsť algoritmus, ktorý pomocou grafového počítača usporiada čísla efektívnejšie ako to vieme bez neho. Určite nebude existovať riešenie v lepšom ako lineárnom čase, keďže zadaných $n$ čísel musíme aspoň načítať a vypísať. Ak teda nájdeme riešenie v čase $\Theta(n)$, budeme mať istotu, že je optimálne.

Ako na to? Pohľad do tabuľky operácií, ktoré vieme na grafovom počítači robiť, nám ponúka jedinú z nich, ktorá prihliada na váhy hrán: `Find`. Ak má `Find` viac možností, ktorý podgraf vybrať, máme zaručené, že vždy vyberie ten s najmenšou váhou. Na tejto operácii postavíme naše riešenie.

Začneme s prázdnym grafom. Postupne v ňom vyrobíme $n$ hrán, ktorých váhy budú triedené čísla. A potom dokola $n$-krát pomocou `Find` nájdeme najľahšiu z hrán, jej váhu vypíšeme na výstup a dotyčnú hranu z grafu odstránime. Takéto riešenie bude zjavne správne a bude mať lineárnu časovú zložitosť.

Zopár detailov k implementácii: Graf, ktorý v prvej fáze vyrobíme, bude mať tvar hviezdy – $i$-temu číslu na vstupe bude zodpovedať hrana medzi vrcholmi s id 1 a $i+1$. Každý vrchol navyše dostane značku rovnú svojmu id. Toto potrebujeme na to, aby sme v druhej fáze vedeli, ktorú hranu vlastne vymazať.

V druhej fáze budeme pomocou `Find` hľadať najlacnejší podgraf tvorený jednou hranou. Tej prvý vrchol musí mať značku 1, druhý môže byť ľubovoľný. Keď tento podgraf nájdeme, cena jeho hrany je číslo, ktoré vypíšeme na výstup, a značka druhého vrcholu nám hovorí, ktorú hranu zmazať v pôvodnom grafe.

**Listing programu:**

```
procedure triedenie;
var n, i, x : Integer;
    G, H, hrana : Graph;
begin
   read(n);
   G := EmptyG; AddV(G,1);
   { nacitavame cisla a pridavame hrany }
   for i:=1 to n do begin read(x); AddV(G,i+1); AddE(G,1,i+1,x); end;

   { dokola hladame a vypisujeme najlacnejsiu hranu }
   hrana := EmptyG; AddV(hrana,1); AddV(hrana,undef); AddE(hrana,1,2,undef);
   for i:=1 to n do begin
      H := Find(G,hrana,IM_value,IM_any);
      writeln( GetE(H,1,2) );
      DelE(G,1,GetV(H,2));
   end;
end;
```

**Krtkove brlôžky:**

Našou úlohou je nájsť súvislý podgraf s najmenším súčtom dĺžok hrán. Keďže sú dĺžky hrán kladné, hľadaný podgraf určite nebude obsahovať žiaden cyklus. Totiž keby nejaký cyklus obsahoval, môžeme jeho ľubovoľnú hranu zahodiť. Tým dostaneme podgraf, ktorý bude lacnejší (lebo sme zahodili hranu) a zároveň naďalej súvislý (lebo zvyšok cyklu naďalej spája vrcholy, ktoré spájala zahodená hrana).

Z toho plynie záver, že hľadaný najlacnejší podgraf je vždy strom obsahujúci všetkých $n$ vrcholov. Takýto strom sa nazýva *kostra grafu*. Má vždy presne $n-1$ hrán. (Predstavme si krtka ako postupne čistí chodbičky. Na začiatku má $n$ izolovaných brlôžkov – teda $n$ komponentov súvislosti. Vždy, keď očistí chodbičku, spojí tým dva komponenty do jedného. Takže po očistení $n-1$ chodbičiek už bude celá nora tvoriť jeden komponent súvislosti a krtko môže ísť oddychovať.)

Náš cieľ teda môžeme sformulovať nasledovne: v danom grafe potrebujeme nájsť jeho *najlacnejšiu (najľahšiu) kostru*. Algoritmus, ktorý bude vzorovým riešením, si najskôr popíšeme všeobecne, až potom ukážeme, ako ho efektívne

implementovať na grafovom počítači. (Pôjde o algoritmus, ktorý sa dá efektívne, hoci s horšou časovou zložitosťou, implementovať aj na počítači klasickom.)

**Lema (pomocné tvrdenie):** Nech $G$ je súvislý ohodnotený graf, ktorého množina vrcholov je $V$ a ktorého hrany majú navzájom rôzne ceny. Rozdeľme vrcholy $G$ do dvoch disjunktných množín $A$ a $B$. Potom najlacnejšia hrana medzi $A$ a $B$ musí byť súčasťou najlacnejšej kostry.

**Dôkaz:** Sporom. Najlacnejšiu hranu, ktorej jeden koniec je v množine $A$ a druhý v množine $B$, nazvime $h$. Majme teda najlacnejšiu kostru, ktorá hranu $h$ neobsahuje. Pridajme hranu $h$ do nej. Tým nám určite vznikol cyklus: tvorí ho pridaná hrana $h$ a cesta po pôvodnej kostre medzi koncovými vrcholmi hrany $h$. Jeden koniec tejto cesty leží v množine $A$, druhý v množine $B$, preto musí na tejto ceste existovať aspoň jedna hrana $h'$, ktorá vedie medzi množinami $A$ a $B$.

No a čo sa stane, keď teraz z nášho podgrafu túto hranu $h'$ zahodíme? Výsledný graf bude naďalej súvislý, lebo sme zahodili hranu z cyklu. Bude to teda opäť nejaká kostra. Lenže hrana $h$ bola najlacnejšia zo všetkých hrán vedúcich medzi $A$ do $B$. Špeciálne teda je $h$ lacnejšia aj ako $h'$. Ale to znamená, že nová kostra je lacnejšia ako tá pôvodná, a to je spor s predpokladom, že pôvodná kostra bola najlacnejšia.

Práve dokázaná lema nám umožní zostrojiť najlacnejšiu kostru postupne – v každom kroku k nej pridáme jednu novú hranu. Tento algoritmus ako prvý objavil český matematik Vojtěch Jarník, nezávisle na ňom neskôr Robert Prim, po nich sa nazýva Jarníkov-Primov algoritmus. Vyzerá nasledovne: Vrcholy grafu budeme mať rozdelené do dvoch skupín: už spojené (množina $A$) a ešte nespojené (množina $B$). Na začiatku je v množine $A$ jeden ľubovoľný vrchol, všetky ostatné sú v množine $B$. V každom kroku teraz nájdeme najlacnejšiu hranu medzi $A$ a $B$. Tú pridáme do zostrojovanej kostry a jej doteraz nepripojený koniec presunieme z $B$ do $A$. Takto zjavne platí, že po každom kroku máme množinu $A$ celú spojenú hranami, ktoré sme dovtedy vybrali. A vďaka dokázanej leme vieme, že každá z hrán, ktoré sme vybrali, musí v najlacnejšej kostre ležať. Keď teda po $n - 1$ krokoch tento proces skončí, všetky vrcholy sú v množine $A$ a vybrané hrany nutne tvoria najlacnejšiu kostru.

Všimnite si, že Jarníkov-Primov algoritmus zodpovedá „pažravému“ rozhodovaniu sa krtka. Ten začne v brlôžku, v ktorom sa prebudil, a v každom kroku si vyberie na prečistenie najkratšiu z chodbičiek, ku ktorým sa vie dostať (a

ktoré nie je zbytočné čistiť). Tým postupne rozširuje sieť brlôžkov, do ktorých sa už dá dostať.

Priamočiara implementácia tohto algoritmu na klasickom počítači by mala časovú zložitosť $\Theta(n^3)$. Totiž $(n-1)$-krát hľadáme novú hranu do kostry. Na jej nájdenie vždy potrebujeme prezrieť všetky hrany medzi aktuálnymi množinami $A$ a $B$, a tých môže byť až rádovo $n^2$.

Časová zložitosť sa dá zlepšiť na $\Theta(n^2)$ tak, že si budeme pre každý vrchol v $B$ pamätať najkratšiu hranu, ktorá z neho vedie do nejakého vrcholu v $A$. Vždy, keď presúvame nejaký vrchol $v$ z $B$ do $A$, tak si tieto hodnoty v čase $O(n)$ prepočítame.

Pre riedke grafy existuje aj implementácia s časovou zložitosťou $\Theta(m \log n)$, kde $m$ je počet hrán grafu. (Úprava vyzerá podobne ako u Dijkstrovho algoritmu: vrcholy množiny $B$ máme uložené vo vhodnej prioritnej fronte tak, aby sme vedeli efektívnejšie nájsť ten, ktorý je momentálne k množine $A$ najbližšie.)

Ako Jarníkov-Primov algoritmus efektívne implementovať na grafovom počítači? Naše riešenie bude mať časovú zložitosť $\Theta(n)$. Použijeme postup veľmi podobný riešeniu prvej podúlohy: novú hranu, ktorú treba pridať do kostry, nájdeme vhodným volaním `Find` v konštantnom čase.

Potrebujeme ešte vyriešiť jeden technický detail: Keď nájdeme novú hranu kostry, potrebujeme vedieť zistiť nie len jej cenu, ale aj čísla vrcholov, ktoré spája. Na to potrebujeme použiť značky. Lenže zároveň potrebujeme nejak vedieť špecifikovať množinu hrán, spomedzi ktorých najlacnejšiu hľadáme.

Existuje viacero možností, ako toto dosiahnuť, my si ukážeme jednu z nich: pridáme do grafu dva nové vrcholy $a$ a $b$, pričom po každom kole algoritmu bude platiť, že vrchol $a$ je spojený s vrcholmi v množine $A$ a vrchol $b$ s vrcholmi v množine $B$. Všetky tieto nové hrany budú mať váhu 0. Potom najlacnejšiu hranu z $A$ do $B$ nájdeme tak, že budeme hľadať najlacnejšiu cestu tvaru $a-x-y-b$. Keď ju nájdeme, pozrieme sa na cenu hrany $x-y$ a na značky týchto dvoch vrcholov. Takúto hranu pridáme do výstupu. (Všimnite si, že vyrobíme výstup v presne takej forme ako je v zadaní – teda vrcholy vo výstupe budú bez značiek a vybraté hrany budú mať každá svoju pôvodnú váhu.)

Poznámka na záver: Autorom úlohy nie je známe riešenie s lepšou ako lineárnou časovou zložitosťou. Je pravdepodobné, že zvolený model grafového počítača takéto riešenie neumožňuje zostrojiť. V jemne upravenom modeli by

sa ale takýto algoritmus zostrojiť dal. Existujú totiž aj iné algoritmy na zostrojenie najlacnejšej kostry a niektoré z nich sa napríklad dobre dajú paralelizovať. Príkladom je najstarší známy efektívny algoritmus, ktorý v roku 1926 navrhol Otakar Borůvka pri návrhu rozvodov elektriny na Morave. Borůvkov algoritmus je založený na nasledujúcom pozorovaní: Nech všetky hrany grafu majú navzájom rôzne ceny. V každom vrchole grafu si vyberieme najlacnejšiu hranu, ktorá z neho vychádza. (Uvedomte si, že niektoré hrany, vrátane tej úplne najlacnejšej, takto vyberieme dvakrát.) Z lemy dokázanej pri Jarníkovom-Primovom algoritme vyplýva, že všetky vybrané hrany patria do kostry. Tak ich tam všetky pridáme. Tým nám vznikne niekoľko disjunktných komponentov. Každý z nich nahradíme jedným novým vrcholom a proces opakujeme, až kým všetko nespojíme.

### Listing programu:

```
function kostra(G : Graph) : Graph;
var n, i : Integer;
    vystup, cesta, vyber : Graph;
begin
   { oznacime povodne vrcholy cislami }
   n := CountV(G);
   for i:=1 to n do SetV(G,i,i);

   { pridame nove vrcholy predstavujuce komponenty }
   AddV(G,n+1); AddV(G,n+2);
   AddE(G,1,n+1,0); for i:=2 to n do AddE(G,i,n+2,0);

   { vyrobime si cestu s tromi hranami, ktoru budeme vyhladavat v G }
   cesta := EmptyG;
   AddV(cesta,n+1); AddV(cesta,undef); AddV(cesta,undef); AddV(cesta,n+2);
   AddE(cesta,1,2,undef); AddE(cesta,2,3,undef); AddE(cesta,3,4,undef);

   { vyrobime si vystupny graf s n izolovanymi vrcholmi }
   vystup := EmptyG;
   for i:=1 to n do AddV(vystup,undef);

   { hladame a pridavame hrany }
   for i:=1 to n-1 do begin
      { najdeme hranu a pridame ju do vystupu }
      vyber := Find( G, cesta, IM_value, IM_any );
      AddE( vystup, GetV(vyber,2), GetV(vyber,3), GetE(vyber,2,3) );
      { upravime mnoziny spojenych a nespojenych vrcholov }
      DelE( G, n+2, GetV(vyber,3) );
      AddE( G, n+1, GetV(vyber,3), 0 );
   end;
   kostra := vystup;
end;
```

## A-III-4 Asfaltistan

Zo zadanej mapy územia si najprv vytvoríme graf; jeho vrcholmi budú jednotlivé štvorcové políčka. Medzi dvoma vrcholmi bude viesť neorientovaná hrana, ak ich políčka môžeme spojiť diaľnicou, mostom alebo tunelom. Hranám priradíme ohodnotenie podľa ceny za prepojenie ich políčok.

Našou úlohou je potom nájsť najkratšiu cestu medzi vrcholmi $(0, 0)$ a $(r - 1, s - 1)$. Na to použijeme Dijkstrov algoritmus.

**Konštrukcia grafu:**

Hrany reprezentujúce spojenie diaľnicou pridáme do grafu jednoducho. Stačí pre každú dvojicu susedných políčok overiť, či rozdiel ich nadmorských výšok nepresahuje 1. Keďže políčko môže mať nanajvýš štyroch susedov, časová zložitosť tejto časti aj počet pridaných hrán je $O(rs)$.

S ostatnými typmi hrán je to trochu zložitejšie. Zamerajme sa na hľadanie mostov, ktoré spájajú dve políčka v rovnakom riadku (pre vertikálny smer aj pre tunely použijeme analogický postup).

Ak chceme pre políčko $(i, j)$ nájsť most, ktorý má na ňom svoj pravý koniec (uvedomte si, že taký môže byť najviac jeden), môžeme postupne prechádzať od neho smerom doľava, kým je nadmorská výška políčok menšia ako $h_{i,j}$. Ak narazíme na políčko s rovnakou nadmorskou výškou, našli sme most. Tento postup ale vyžaduje $O(s)$ času pre každé políčko, dokopy teda $O((r + s)rs)$.

Namiesto toho budeme spracovávať každý riadok zľava doprava (aktuálne políčko si označme $(i, j)$). Počas toho si budeme udržiavať zoznam tých políčok, ktoré sú z aktuálnej pozície „viditeľné“ – teda ktoré by mohla trafiť vodorovne letiaca vrana, začínajúca na aktuálnom políčku.

Napríklad ak už spracované výšky boli 100, 70, 10, 20, 20 a 14, tak viditeľné sú políčka v stĺpcoch 0, 1, 4 a 5 (výšky 100, 70, 20 a 14).

Formálne, viditeľné sú tie políčka $(k, j)$, že $k < i$ a všetky políčka medzi $(k, j)$ a aktuálnym majú nadmorskú výšku menšiu ako $h_{k,j}$. Všimnite si, že ak si budeme políčka v tomto zozname pamätať v poradí zľava doprava, budú ich výšky tvoriť klesajúcu postupnosť. Tiež si všimnime, že akonáhle políčko nie je viditeľné, nemôže už nikdy byť začiatkom mostu – to, že nie je viditeľné, totiž znamená, že ho zakrylo iné, aspoň rovnako vysoké políčko.

Pri spracovaní políčka $(i, j)$ budeme postupne z konca zoznamu odstraňovať políčka, ktoré sme ním zakryli – teda všetky políčka, ktoré majú nadmorskú výšku menšiu ako $h_{i,j}$. Ak následne narazíme na políčko s výškou $h_{i,j}$, našli sme most končiaci na políčku $(i, j)$. V opačnom prípade na aktuálnom políčku horizontálny most nekončí. Pridáme $(i, j)$ na koniec zoznamu a ideme na nasle-

dujúce políčko. (Keďže pridávame aj uberáme len na konci zoznamu, môžeme ho implementovať v poli ako zásobník.)

Keďže každé políčko pridáme aj odstránime zo zoznamu najviac raz, časová zložitosť spracovania jedného riadka je $O(s)$, spolu teda $O(rs)$. V tejto fáze pribudne v grafe $O(rs)$ hrán.

**Dijkstrov algoritmus:**

V tejto časti popíšeme štandardný algoritmus na hľadanie najkratších ciest z daného začiatočného vrcholu $z$ do všetkých ostatných vrcholov v grafe bez záporných hrán – Dijkstrov algoritmus. Označme $V$ množinu vrcholov grafu $G$, množinu hrán označme $E$.

Počas výpočtu si budeme v poli $D$ na pozícii $i$ pamätať dĺžku najkratšej zatiaľ nájdenej cesty zo $z$ do vrcholu $i$. Na začiatku hodnoty $D$ inicializujeme na $\infty$, iba $D[z] = 0$. Navyše si budeme pre každý vrchol $i$ pamätať, či je už hodnota $D[i]$ finálna.

V každom kroku algoritmu nájdeme taký vrchol $v$, ktorý má najmenšiu hodnotu $D[v]$, ale táto hodnota ešte nebola vyhlásaná za finálnu. Keďže žiadna hrana grafu nemá zápornú dĺžku, už sa nám nikdy nepodarí $D[v]$ zmenšiť, a preto je $D[v]$ finálna. Ďalej skúsime všetkým susedom $i$ vrcholu $v$ zlepšiť hodnotu $D[i]$ – vezmeme najkratšiu cestu z $s$ do $v$, predĺžime ju do $i$ a porovnáme jej dĺžku ($D[v]$+ dĺžka hrany $vi$) s momentálnou hodnotou $D[i]$.

Po najviac $|V|$ krokoch už budeme poznať skutočné vzdialenosti všetkých vrcholov, do ktorých sa dá zo $z$ dostať.

Časová zložitosť algoritmu závisí od spôsobu, akým hľadáme vrchol $v$. Pri jednoduchom prechode všetkými vrcholmi dostaneme zložitosť $O(|V|^2)$. Ak si ale budeme hodnoty $D[i]$, ktoré ešte nie sú finálne, ukladať do minimovej haldy, najmenšiu z nich nájdeme v čase $O(\log |V|)$.

Pri zmene $D[i]$ sa však potrebujeme zbaviť starej hodnoty, ktorú máme v halde. Jedna možnosť je pamätať si, kde sa v halde ktorá hodnota nachádza, a pomocou tejto informácie ju v prípade potreby nájsť a odstrániť. Ale jednoduchším a v praxi stále dostatočne rýchlym riešením je tieto staré hodnoty v halde nechať. Nová hodnota je totiž menšia, preto ju z haldy vytiahneme skôr; keď potom niekedy vytiahneme starú hodnotu, jednoducho ju zahodíme.

Takto sa môže naraz v halde nachádzať $O(|E|)$ prvkov, čo môže byť až $O(|V|^2)$. Keďže ale $\log x^2 = 2 \log x$, časová zložitosť sa nezhoršila. Dokopy dostávame zložitosť $O(|E| \log |V|)$.

V našom konkrétnom prípade máme riedky graf ($O(rs)$ vrcholov aj hrán), preto použijeme implementáciu s haldou. Dostávame tak riešenie pôvodnej úlohy

s časovou zložitosťou $O(rs \log(rs))$.

(Namiesto haldy sa dá použiť aj vyvažovaný binárny vyhľadávací strom. V ňom je ľahké meniť záznamy – stači vždy zmazať starý a následne pridať nový.)

## Listing programu:

```
#include <cstdio>
#include <cstdlib>
#include <vector>
#include <stack>
#include <queue>
#include <functional>
using namespace std;

struct Edge {
    int x, y;
    long long c;
    Edge(int ix, int iy, long long ic): x(ix), y(iy), c(ic) {}
};

bool operator < (const Edge &a, const Edge &b) { return a.c > b.c; }

int m, n, cD, cM, cT;
vector<vector<int> > H;
vector<vector<vector<Edge> > > G;
vector<vector<long long> > D;
priority_queue<Edge> Q;

template<typename Compare>
void vertical(int x, long long cost, Compare cmp) {
    stack<int> S;
    for (int y = 0; y < n; ++y) {
        while (!S.empty() && cmp(H[x][S.top()], H[x][y])) S.pop();
        if (!S.empty() && H[x][S.top()] == H[x][y]) {
            long long c = cD + cost * (y - S.top() - 1);
            G[x][y].push_back(Edge(x, S.top(), c));
            G[x][S.top()].push_back(Edge(x, y, c));
            S.pop();
        }
        S.push(y);
    }
}

template<typename Compare>
void horizontal(int y, long long cost, Compare cmp) {
    stack<int> S;
    for (int x = 0; x < m; ++x) {
        while (!S.empty() && cmp(H[S.top()][y], H[x][y]))
            S.pop();
        if (!S.empty() && H[S.top()][y] == H[x][y]) {
            long long c = cD + cost * (x - S.top() - 1);
            G[x][y].push_back(Edge(S.top(), y, c));
            G[S.top()][y].push_back(Edge(x, y, c));
            S.pop();
        }
        S.push(x);
    }
}

int main() {
    scanf("%d%d%d%d%d", &m, &n, &cD, &cM, &cT);
```

```
    H.resize(m, vector<int>(n));
    for (int y = 0; y < n; ++y)
        for (int x = 0; x < m; ++x)
            scanf("%d", &H[x][y]);

    int dx[4] = {0, 1, 0, -1}, dy[4] = {-1, 0, 1, 0};
    G.resize(m, vector<vector<Edge> >(n));
    for (int x = 0; x < m; ++x)
        for (int y = 0; y < n; ++y)
            for (int d = 0; d < 4; ++d) {
                int nx = x + dx[d], ny = y + dy[d];
                if (0 <= nx && nx < m && 0 <= ny && ny < n
                    && abs(H[x][y] - H[nx][ny]) <= 1)
                        G[x][y].push_back(Edge(nx, ny, cD));
            }
    for (int x = 0; x < m; ++x) {
        vertical(x, cM, less<int>());
        vertical(x, cT, greater<int>());
    }
    for (int y = 0; y < n; ++y) {
        horizontal(y, cM, less<int>());
        horizontal(y, cT, greater<int>());
    }

    D.resize(m, vector<long long>(n, -1));
    D[m - 1][n - 1] = 0;
    Q.push(Edge(m - 1, n - 1, 0));
    while (!Q.empty()) {
        Edge e = Q.top(); Q.pop();
        if (D[e.x][e.y] < e.c)
            continue;
        vector<Edge>::iterator i;
        for (i = G[e.x][e.y].begin(); i != G[e.x][e.y].end(); ++i)
            if (D[i->x][i->y] == -1 || D[i->x][i->y] > e.c + i->c) {
                D[i->x][i->y] = e.c + i->c;
                Q.push(Edge(i->x, i->y, e.c + i->c));
            }
    }

    if (D[0][0] == -1)
        printf("NEEXISTUJE\n");
    else {
        printf("%lld\n", D[0][0] + cD);
        int x = 0, y = 0;
        while (x != m - 1 || y != n - 1) {
            printf("%d %d\n", x, y);
            vector<Edge>::iterator i;
            for (i = G[x][y].begin(); i != G[x][y].end(); ++i)
                if (D[i->x][i->y] + i->c == D[x][y]) {
                    x = i->x;
                    y = i->y;
                    break;
                }
        }
        printf("%d %d\n", x, y);
    }
}
```

## A-III-5 Romantické básničky II

Pre jednoduchosť najskôr skúsime vynechať požiadavku, aby sa rýmoval začiatok a koniec básničky. Keďže je poradie slôh pevne dané, dá sa takéto zjednodušené zadanie riešiť pomocou dynamického programovania. Môžeme totiž využiť to, že ak máme množinu všetkých slôh $R_i$, na ktoré by báseň mohla končiť, keby mala iba $i$ slôh, tak sme schopný ľahko nájsť množinu všetkých slôh $R_{i+1}$, na ktoré môže končiť $(i+1)$-slohová báseň.

Postup je jednoduchý: zo všetkých variantov slohy $i+1$, ktoré označme ako $V_{i+1}$, vyberieme tie, ktoré sa rýmujú, s niektorou slohou z $R_i$. Množina takto vybraných slôh potom tvorí množinu $R_{i+1}$. Ak sa na slohy budeme pozerať ako na dvojice prirodzených čísel, môžeme tento fakt zapísať matematicky ako

$$R_{i+1} = \Big\{(a,b) \;\Big|\; (a,b) \in V_{i+1} \;\wedge\; \exists x : (x,a) \in R_i\Big\}.$$

Pomocou tejto myšlienky vieme ľahko riešiť zjednodušenú úlohu. Množinu $R_1$ získame priamo ako množinu všetkých variantov prvej slohy, teda $R_1 = V_1$. Báseň potom postupne predlžujeme, až dostaneme množinu $R_n$. Keďže cieľom úlohy je vypísať vybrané varianty, musíme si zároveň s každou slohou $x$ v $R_{i+1}$ pamätať jej predchodcu (vďaka ktorému variantu predchádzajúcej slohy sme $x$ vybrali do množiny $R_{i+1}$). Ak je takých variantov viac, stačí si pamätať ľubovoľný z nich. Pomocou tejto dodatočnej informácie môžeme ľahko odzadu zrekonštruovať niektorú z hľadaných postupností.

(Na takéto riešenie sa môžeme dívať aj ako na prehľadávanie do šírky na priestore všetkých možných „stavov počas písania básničky“ – každý stav, teda vrchol prehľadávaného grafu, vieme popísať číslom slohy, ktorú sme poslednú spracovali, a číslom jej variantu, ktorý sme si vybrali. Rovnako dobre sa dalo použiť prehľadávanie do hĺbky, je však o niečo menej názorné.)

**Pridáme cyklickosť rýmov:**

Požiadavka na cyklickosť rýmov nám situáciu mierne skomplikuje. Nestačí totiž nájsť slohu v množine $R_n$, ktorá sa rýmuje s niektorou prvou slohou! Napríklad ak je prvá sloha buď $(1,2)$ alebo $(2,3)$ a druhá sloha je $(3,1)$, vieme nájsť necyklickú básničku $(2,3),(3,1)$, ale nemôžeme vziať $(1,2)$ ako prvú slohu.

Môžeme si ale napríklad s každou slohou $x$ z množiny $R_i$ navyše pamätať, na ktoré slohy báseň musí začínať, aby mohla končiť slohou $x$. Inou, trochu lepšou možnosťou je spustiť výpočet zvlášť pre každý variant prvej slohy. Ak na konci niektorá sloha z $R_n$ končí na rým, ktorým začína aktuálne skúšaná prvná sloha, tak sme našli riešenie. Oba postupy majú rovnakú časovú zložitosť. Líšia sa ale

pamäťovou zložitosťou, keďže v prvom prípade si musíme ukladať až $s_1$-krát viac informácií pre rekonštrukciu básničky.

Výsledný algoritmus je teraz už ľahký. Pre každý variant prvej slohy vyrobíme množinu $R_1$, čo je jednoprvková množina, obsahujúca vybranú prvú slohu. Použijeme výššie popísaný algoritmus pre necyklickú báseň. Ak sa v množine $R_n$ nachádza sloha, ktorá končí na rým, na ktorý začína aktuálna prvá sloha, tak sme našli riešenie, ktoré ľahko zrekonštruujeme, vypíšeme a skončíme. V opačnom prípade pokračujeme ďalším variantom prvej slohy.

**Pomalá ale funkčná implementácia:**

Každú množinu $R_i$ môžeme reprezentovať booleovským poľom. To bude pre každý variant $i$-tej slohy určovať, či sa nachádza v množine $R_i$ alebo nie. Pre výpočet $R_{i+1}$ si pre každý variant vo $V_{i+1}$ zistíme, či existuje sloha v $R_i$, s ktorou se rýmuje. Ak áno, vložíme tento variant do $R_{i+1}$.

Nech má každá sloha najviac $s$ variantov. Potom vieme množinu $R_{i+1}$ z množiny $R_i$ zostrojiť v čase $O(s^2)$ – stačí každý variant vo $V_{i+1}$ priamo porovnať s každým v $R_i$. Toto celé budeme robiť $O(sn)$-krát: pre každý z $O(s)$ možných začiatkov potrebujeme postupne prejsť všetkých $n$ slôh. Celé riešenie má teda časovú zložitosť $O(ns^3)$.

**Ľahká lenivá efektívna implementácia:**

V predchádzajúcom riešení vieme zlepšiť efektívnosť generovania množiny $R_{i+1}$ z množiny $R_i$.

Ľahký spôsob je použiť hešovanie. V našom programe namiesto množiny $R_i$ zostrojíme množinu čísel veršov, ktorými môže končiť $i$-ta sloha. Pre každé z nich si v hešovacej tabuľke zapamätáme číslo variantu $i$-tej slohy, ktorým sme ho dosiahli.

Predstavme si, že sme práve spracovali $i$-tu slohu. Ako dlho nám bude trvať spracovať nasledujúcu? Prejdeme všetky varianty v $V_{i+1}$. Každý z nich spracujeme (v očakávanom prípade) v konštantnom čase, a to nasledovne: Majme variant $(a, b)$. Pozrieme sa do hešovacej tabuľky pre $i$-tu slohu. Ak sa $a$ nedalo dosiahnuť po $i$-tej slohe, tento variant je nám naničˇ. Ak sa dalo, tak si v novej hešovacej tabuľke zaznačíme pre hodnotu $b$ číslo aktuálneho variantu.

Takto teda každú slohu spracujeme nie v čase $O(s^2)$, ale v očakávanom čase $O(s)$. Celková časová zložitosť tohto riešenia je teda $O(ns^2)$ v očakávanom prípade. (Najhorší možný prípad je $O(ns^3)$, teda rovnaký ako pri pomalom riešení. Pri použití vhodnej hešovacej funkcie je ale prudko nepravdepodobné, že takýto zlý prípad nastane.)

Prípadne vieme dosiahnuť zaručený čas $O(ns^2 \log s)$, ak by sme namiesto hešovacej tabuľky použili asociatívne pole implementované ako vyvažovaný strom. (Teda v našom programe zmenili `unordered_map` na `map`.)

### Listing programu:

```
#include <cstdio>
#include <algorithm>
#include <vector>
#include <set>
#include <tr1/unordered_map>
using namespace std;
using namespace std::tr1;

int N;
vector<int> S;
vector< vector< pair<int,int> > > verse;

void load() {
    scanf("%d",&N);
    S.resize(N);
    verse.resize(N);
    for (int n=0; n<N; ++n) {
        scanf("%d",&S[n]);
        for (int i=0; i<S[n]; ++i) {
            int x,y; scanf("%d%d",&x,&y);
            verse[n].push_back( make_pair(x,y) );
        }
    }
}

bool solve(int kde) {
    vector< unordered_map<int,int> > ako;
    ako.resize(N+1);
    ako[0][kde] = -1;
    // postupne spracuvame slohy
    for (int n=0; n<N; ++n) {
        for (int i=0; i<S[n]; ++i)
            if (ako[n].count( verse[n][i].first ))
                ako[n+1][ verse[n][i].second ] = i;
        if (ako[n+1].empty()) return false;
    }
    // ak sme sa dostali, kam sme chceli, vypiseme poradie
    if (!ako[N].count(kde)) return false;
    vector<int> answers;
    for (int n=N; n>0; --n) {
        answers.push_back( ako[n][kde]+1 );
        kde = verse[n-1][ ako[n][kde] ].first;
    }
    for (int n=0; n<N; ++n) printf("%d\n",answers[N-1-n]);
    return true;
}

int main() {
    load();
    set<int> candidates;
    for (int i=0; i<S[0]; ++i) candidates.insert( verse[0][i].first );
    for (set<int>::iterator it=candidates.begin(); it != candidates.end(); ++it)
        if (solve(*it)) return 0;
    printf("NEEXISTUJE\n");
}
```

**Zaručene efektívna implementácia:**

Upravíme spracovanie slohy na $O(s)$ v najhoršom prípade. To urobíme tak, že si v rámci predvýpočtu zoskupíme slohy z $V_i$ do skupín tak, aby v každej boli iba slohy končiace na rovnaký rým. Teraz každú slohu $x$ z $V_{i+1}$ prepojíme so skupinou, ktorá zodpovedá rýmu, na ktorý $x$ začína, prípadne si zapamätáme, že taká skupina neexistuje. Teraz môžeme jedným prechodom cez $V_i$ pre každú skupinu zistiť, či sa v nej nachádza aspoň jedna sloha, ktorá patrí do $R_i$. Druhým prechodom cez $V_{i+1}$ potom ľahko zistíme, ktoré slohy patria do $R_{i+1}$ a ktoré nie tým, že se pozrieme na výsledok odpovedajúci skupine z predchádzajúceho prechodu.

Predvýpočet prevedieme tak, že si slohy z $V_i$ usporiadame podľa druhého rýmu a slohy z $V_{i+1}$ podľa prvého rýmu. Potom lineárnym prechodom očíslujeme skupiny číslami 0, 1, ..., počet skupín a pre každú množinu $V_i$ si napr. v obyčajnom poli zapamätáme, do ktorej skupiny jednotlivé varianty patria. Druhým lineárnym prechodom potom pre slohy z $V_{i+1}$ určíme, s ktorou skupinou sú prepojené.

Časová zložitosť sa zmení na $O(ns \log s)$ na predspracovanie vstupu a $O(ns^2)$ pre samotný výpočet.

Druhé urýchlenie (ktoré ale pre dosiahnutie plného počtu bodov nebolo nutné implementovať) spočíva v myšlienke, že v skutočnosti nezáleží na tom, ktorou slohou začíname. Môžeme teda začať hľadať báseň od slohy, ktorá má najmenej variantov.

**Listing programu:**

```
#include <cstdio>
#include <vector>
#include <algorithm>

struct Sloka {
    Sloka() {}
    Sloka(int prvni, int druhy, int poradi)
        : PrvniRym(prvni), DruhyRym(druhy), Poradi(poradi),
          Skupina(-1), Propojeni(-1), Predchozi(-1) {}
    int PrvniRym, DruhyRym;  // cislo prvniho a druheho rymu sloky
    short int Poradi;        // poradi sloky mezi variantami
    short int Skupina;       // skupina, do ktere sloka patri
    short int Propojeni;     // skupina, s niz je sloka spojena; -1 = s zadnou
    short int Predchozi;     // predchozi sloka (pro rekonstrukci reseni)
};

// porovnani podle prvniho a podle druheho rymu
bool PodlePrvniho(const Sloka &leva, const Sloka &prava) {
    return leva.PrvniRym < prava.PrvniRym;
}
bool PodleDruheho(const Sloka &leva, const Sloka &prava) {
```

```
    return leva.DruhyRym < prava.DruhyRym;
}

int main() {
    int N;
    scanf("%d", &N);
    std::vector<std::vector<Sloka> > V(N); // seznam vsech variant

    // nacteni variant
    int maxS = 0, min = 0;
    for (int i=0; i<N; i++) {
        int S;
        scanf("%d", &S);
        for (int j=0; j<S; j++) {
            int a, b;
            scanf("%d%d", &a, &b);
            V[i].push_back(Sloka(a, b, j));
        }
        // prubezne hledane cislo sloky s nejmensim poctem variant
        if (V[i].size() < V[min].size()) min = i;
        // a take maximalni velikost sloky
        if (S > maxS) maxS = S;
    }

    // otocime vstup tak, abychom zacinali slokou s nejmensim poctem variant
    std::rotate(V.begin(), V.begin() + min, V.end());

    // predzpracovani
    for (int i=0; i<N-1; i++) {
        // setridime varianty
        std::sort(V[i].begin(), V[i].end(), PodleDruheho);
        std::sort(V[i+1].begin(), V[i+1].end(), PodlePrvniho);

        // nejdrive zaradime sloky z V[i] do skupin
        int rym = -1, skupina = -1;
        for (int j=0; j<V[i].size(); j++) {
            if (V[i][j].DruhyRym != rym) {
                V[i][j].Skupina = ++skupina;
                rym = V[i][j].DruhyRym;
            } else {
                V[i][j].Skupina = skupina;
            }
        }

        // nyni spojime sloky z V[i+1] s odpovidajicimi skupinami
        int k = 0, l = 0;
        while (k < V[i].size() && l < V[i+1].size()) {
            if (V[i][k].DruhyRym == V[i+1][l].PrvniRym) {
                V[i+1][l].Propojeni = V[i][k].Skupina;
                ++l;
            } else if (V[i][k].DruhyRym < V[i+1][l].PrvniRym) {
                ++k;
            } else {
                ++l;
            }
        }
    }

    std::vector<short int> skupiny(maxS, -1); // vysledky pro jednotlive skupiny
    std::vector<bool> R(maxS, false);         // mnozina R

    for (int prvni=0; prvni<V[0].size(); prvni++) {
```

```
        std::fill(R.begin(), R.begin() + V[0].size(), false);
        R[prvni] = true; // postupne zkousime jednotlive varianty prvni sloky

        for (int i=0; i<V.size()-1; i++) {
            std::fill(skupiny.begin(), skupiny.begin()+V[i].size(), -1);

            // spocitame, ktere skupiny obsahuji aspon jednu sloku,
            // ktera je v R, a rovnou si jednu z nich zapamatujeme
            for (int k=0; k<V[i].size(); k++) if (R[k])
                skupiny[V[i][k].Skupina]=k;

            std::fill(R.begin(), R.begin()+V[i+1].size(), false);

            // do R dame ty sloky, ktere jsou spojeny se skupinou,
            // ktera obsahuje aspon jednu sloku z minule verze R
            for (int k=0; k<V[i+1].size(); k++) {
                if (V[i+1][k].Propojeni>=0 && skupiny[V[i+1][k].Propojeni]>=0) {
                    V[i+1][k].Predchozi = skupiny[V[i+1][k].Propojeni];
                    R[k] = true;
                }
            }
        }

        // nyni mame R spocitanou a zjistime, zda lze navazat na prvni sloku
        for (int i=0; i<V[N-1].size(); i++) {
            if (R[i] && V[N-1][i].DruhyRym == V[0][prvni].PrvniRym) {

                // nasli jsme reseni, nyni ho zrekonstruujeme
                std::vector<short int> vysledek(N);
                short int aktualni = i;
                for (int k = N-1; k >= 0; --k) {
                    // kvuli trideni slok se puvodni poradi mohlo zmenit
                    vysledek[k] = V[k][aktualni].Poradi;
                    aktualni = V[k][aktualni].Predchozi;
                }

                // vyslednou posloupnost musime otocit zpatky tak,
                // aby zacinala opet puvodni prvni slokou
                std::rotate(vysledek.begin(), vysledek.begin() +
                    ((vysledek.size() - min) % vysledek.size()), vysledek.end());

                for (int k = 0; k < N; k++) printf("%d\n", vysledek[k] + 1);
                return 0; // stacilo najit libovolne reseni, takze muzeme skoncit
            }
        }
    }

    printf("NEEXISTUJE\n"); // pokud se program dostal az sem, nenalezl reseni
}
```

# Výsledky krajských kôl kategórie B

V kategórii B súťaž končí krajským kolom. Na nasledujúcich stranách uvádzame výsledky tohto kola v jednotlivých krajoch. Výsledky neboli koordinované, je teda možné, že v rôznych krajoch boli použité mierne odlišné bodovacie škály. Úspešnými riešiteľmi tohto krajského kola sú tí riešitelia, ktorí získali aspoň 10 bodov.

**Banskobystrický kraj:**

| Meno | Ročník, škola | 1. | 2. | 3. | 4. | Σ |
|---|---|---|---|---|---|---|
| 1. Vladimír Macko | 2 Gym. Ľ. Štúra Zvolen | 9 | 10 | 9 | 7 | 35 |
| 2. Roderik Ploszek | 2 Gym. Tajovského B. Bystrica | – | 5 | 9 | – | 14 |

**Bratislavský kraj:**

| Meno | Ročník, škola | 1. | 2. | 3. | 4. | Σ |
|---|---|---|---|---|---|---|
| 1. Askar Gafurov | 2 Gym. Grösslingová | 10 | 8 | 10 | 9 | 37 |
| Mário Lipovský | 1 Gym. Jura Hronca | 10 | 10 | 10 | 7 | 37 |
| 3. Viktória Vozárová | 1 Gym. Jura Hronca | 10 | 10 | 6 | 7 | 33 |
| 4. Michal Fikar | 2 Gym. Jura Hronca | 10 | 5 | 10 | 7 | 32 |
| 5. Matej Badin | 1 Gym. Jura Hronca | 10 | 5 | 7 | 7 | 29 |
| 6. Dusan Kavicky | 2 Gym. Jura Hronca | 8 | 1 | 9 | 10 | 28 |
| Lukáš Ivan | 1 Gym. Jura Hronca | 9 | 4 | 8 | 7 | 28 |
| Matej Krajčovič | 2 Gym. Jura Hronca | 8 | 10 | 10 | 0 | 28 |
| 9. Martin Fikar | 2 Gym. Jura Hronca | 9 | 4 | 5 | 7 | 25 |
| 10. Michal Bock | 2 Gym. Grösslingová | 5 | 4 | 9 | 5 | 23 |
| Peter Beňuš | 2 Gym. Jura Hronca | 5 | 4 | 7 | 7 | 23 |
| Peter Ralbovsky | -1 Škola pre mim. nadané deti | 9 | 4 | 7 | 3 | 23 |
| 13. Peter Hraška | 2 Gym. Grösslingová | 6 | 4 | 10 | 2 | 22 |
| 14. Richard Kakaš | 2 Gym. Jura Hronca | 9 | 4 | 7 | 1 | 21 |
| 15. Matúš Bezek | 2 Gym. Košická | – | 3 | 7 | 10 | 20 |
| 16. Ján Ondráš | 1 Gym. Grösslingová | 0 | 0 | 9 | 7 | 16 |
| 17. Michal Hledík | 2 Gym. Jura Hronca | 9 | 4 | 0 | 0 | 13 |
| 18. Jerguš Jalovecký | 1 Gym. Jura Hronca | 1 | 0 | 4 | 4 | 9 |

**Košický kraj:**

| Meno | Ročník, škola | 1. | 2. | 3. | 4. | Σ |
|---|---|---|---|---|---|---|
| 1. Gabriela Sklenčárová | 2 Gym. Sečovce | 5 | 5 | 10 | 5 | 25 |
| 2. Martin Bajtoš | 2 Gym. Školská Spiš. Nová Ves | 8 | 3 | 8 | 4 | 23 |
| 3. Vladimír Hvostaľ | 2 Gym. P. Horova Michalovce | | 2 | 9 | 3 | 14 |
| 4. Pavol Šalata | 2 Gym. P. Horova Michalovce | | 5 | 5 | 1 | 11 |
| 5. Peter Bočan | 2 Gym. P. Horova Michalovce | | 4 | 4 | 1 | 9 |
| 6. Ľubomír Jesze | 2 Gym Javorová Spiš. Nová Ves | | 5 | 2 | | 7 |

**Nitriansky kraj:**

| Meno | Ročník, škola | 1. | 2. | 3. | 4. | Σ |
|---|---|---|---|---|---|---|
| 1. Michal Porubský | -1 SKŠ Nitra, Gym. sv. Cyrila a Metoda | 6 | 8 | 9 | 1 | 24 |
| 2. Marek Šuppa | 2 SKŠ Nitra, Gym. sv. Cyrila a Metoda | 7 | 5 | 8 | 2 | 22 |
| 3. Attila Pivoda | 2 SPŠ Komárno | 2 | 3 | 6 | | 11 |

**Prešovský kraj:**

| Meno | Ročník, škola | 1. | 2. | 3. | 4. | Σ |
|---|---|---|---|---|---|---|
| 1. Miroslav Kravec | 2 SPŠE Prešov | 9 | 4 | 9 | 6 | 28 |
| 2. Juraj Joščák | 2 Gym. sv. Moniky Prešov | 1 | 5 | 9 | 6 | 21 |
| 3. Pavel Richtarik | 2 Gym. sv. Moniky Prešov | 8 | 3 | 8 | 1 | 20 |
| Róbert Eckhaus | 1 Gym. Konštantínova Prešov | 5 | 5 | 4 | 6 | 20 |
| 5. Tomáš Turlík | 2 Gym. Mudroňova Prešov | 1 | 5 | 9 | 2 | 17 |
| 6. Lukáš Richtarik | 0 Gym. sv. Moniky Prešov | 0 | 5 | 9 | 1 | 15 |
| 7. Damián Feťko | 2 SPŠE Prešov | | 5 | 3 | 4 | 12 |
| 8. Maroš Špak | 2 Gym. Kežmarok | 1 | | 4 | 2 | 7 |

**Trenčiansky kraj:**

| Meno | Ročník, škola | 1. | 2. | 3. | 4. | Σ |
|---|---|---|---|---|---|---|
| 1. Vladan Glončák | 2 Gym. Ľudovíta Štúra Trenčín | 6 | | 9 | 5 | 20 |
| 2. Peter Balčirák | 2 Gym. Nedožerského Prievidza | 3 | | 4 | | 7 |

**Trnavský kraj:**

| Meno | Ročník, škola | 1. | 2. | 3. | 4. | Σ |
|---|---|---|---|---|---|---|
| 1. Jakub Strapko | 2 SPŠ dopravná Trnava | 2 | 3 | 7 | 2 | 14 |
| 2. Michal Krátky | 2 Gym. Sereď | 2 | 2 | 9 | 0 | 13 |
| Márton Bartal | 2 Súkromné gym. s vjm. Dunajská Streda | 3 | 0 | 4 | 6 | 13 |
| 4. Radoslav Karlík | 2 Súkromná SOŠ Humanus Via | 2 | 0 | 4 | 1 | 7 |

**Žilinský kraj:**

| Meno | Ročník, škola | 1. | 2. | 3. | 4. | Σ |
|---|---|---|---|---|---|---|
| 1. Jozef Marko | 2 Gym. Lettricha Martin | 8 | 10 | 10 | 3 | 31 |
| 2. Jakub Záthurecký | 1 Gym. Malá Hora Martin | 9 | 5 | 4 | 7 | 25 |

# Výsledky celoštátneho kola kategórie A

Celoštátne kolo kategórie A sa v 26. ročníku Olympiády v informatike uskutočnilo v dňoch 30. marca až 2. apríla 2011 v Drienici, okres Sabinov. Praktickú časť súťažiaci riešili v priestoroch Fakulty humanitných a prírodných vied Prešovskej univerzity v Prešove. Netradične si prvé miesto delili medzi sebou hneď traja súťažiaci.

| | Meno | Ročník, škola | 1. | 2. | 3. | 4. | 5. | Σ |
|---|---|---|---|---|---|---|---|---|
| 1. | Anderle Michal | 4. Gym. B. S. Timravy, Lučenec | 7 | 2 | 10 | 10 | 11 | 40 |
| | Balog Matej | 4. Gym. Grösslingová 18, Bratislava | 8 | 3 | 10 | 10 | 9 | 40 |
| | Hozza Ján | 4. Gym. J. Hronca, Bratislava | 9 | 6 | 6 | 6 | 13 | 40 |
| 4. | Horňák Marián | 3. Gym. Párovská 1, Nitra | 6 | 5 | 8 | 12 | 0 | 31 |
| 5. | Večerík Matej | 4. ŠpMNDaG, Teplická 7, Bratislava | 7 | 4 | 3 | 4 | 12 | 30 |
| 6. | Brandys Jozef | 3. Gym. A. Bernoláka, Námestovo | 7 | 2 | 3 | 5 | 12 | 29 |
| 7. | Mariš Andrej | 2. Piaristické gym., Nitra | 10 | 6 | 7 | 2 | 1 | 26 |
| 8. | Greššák Jerguš | 2. Gym. J. A. Raymana, Prešov | 10 | 6 | 6 | 0 | 0 | 22 |
| 9. | Plavák Dušan | 4. Gym. M. Hattalu, Trstená | 4 | 1 | 0 | 3 | 13 | 21 |
| | Špano Marek | 4. Gym. J. Hronca, Bratislava | 9 | 6 | 5 | 0 | 1 | 21 |
| 11. | Součková Kamila | 2. Ev. lýceum Vranovská, Bratislava | 7 | 2 | 6 | 4 | 1 | 20 |
| 12. | Babej Tomáš | 4. Gym. Poštová 9, Košice | 8 | 5 | 4 | 0 | 0 | 17 |
| | Rabatin Rastislav | 2. Gym. J. Hronca, Bratislava | 9 | 3 | 5 | 0 | 0 | 17 |
| 14. | Jančo Tomáš | 4. Gym. Ľ. Štúra, Trenčín | 4 | 4 | 0 | 2 | 5 | 15 |
| | Sebechlebský Ján | 4. Gym. M. Rúfusa, Žiar n. Hronom | 7 | 5 | 2 | 0 | 1 | 15 |
| 16. | Beták Martin | 4. Gym. Nedožerského, Prievidza | 7 | 1 | 0 | 0 | 5 | 13 |
| 17. | Hrubý Tomáš | 4. Gym. J. Hronca, Bratislava | 3 | 5 | 2 | 2 | 0 | 12 |
| | Mrocková Mária | 4. Gym. J. Hronca, Bratislava | 7 | 1 | 0 | 2 | 2 | 12 |
| 19. | Dresslerová Anna | 4. Gym. J. Hronca, Bratislava | 8 | 3 | 0 | 0 | 0 | 11 |
| | Šafin Jakub | 2. Gym. P. Horova, Michalovce | 4 | 1 | 6 | 0 | 0 | 11 |
| 21. | Jurových Jakub | 4. Gym. Okružná 2469, Zvolen | 7 | 1 | 2 | 0 | 0 | 10 |
| | Molnár Richard | 2. Ev. spojená škola, Lipt. Mikuláš | 4 | 3 | 3 | 0 | 0 | 10 |
| | Pulmann Ján | 4. Gym. Grösslingová 18, Bratislava | 5 | 1 | 4 | 0 | 0 | 10 |
| | Strapko Martin | 4. Gym. J. Hronca, Bratislava | 3 | 0 | 2 | 0 | 5 | 10 |
| 25. | Sabo Matúš | 4. Gym. J. Hronca, Bratislava | 5 | 3 | 1 | 0 | 0 | 9 |
| 26. | Livora Tomáš | 4. Gym. Javorová 16, Spiš. N. Ves | 4 | 1 | 2 | 0 | 0 | 7 |
| 27. | Marko Jozef | 4. SPŠ, SNP 8, Myjava | 3 | 1 | 2 | 0 | 0 | 6 |

# Výsledky výberového sústredenia

V dňoch 26. apríla až 2. mája 2011 sa v Bratislave konalo výberové sústredenie. Na toto sústredenie boli pozvaní najlepší riešitelia celoštátneho kola OI, kategórie A. Štyria najlepší riešitelia výberového sústredenia majú možnosť reprezentovať Slovensko na Medzinárodnej olympiáde v informatike. Na základe výberového sústredenia taktiež vyberá SK OI reprezentačný tím pre Stredoeurópsku olympiádu v informatike a pre prípravné sústredenia.

Výberového sústredenia sa v tomto roku ako pozvaní hostia zúčastnili aj štyria súťažiaci zo Švajčiarska.

V nasledujúcej tabuľke sú postupne uvedené body za celoštátne kolo OI (resp. švajčiarskej olympiády v informatike), za „domáce úlohy“ a za sedem súťažných dní výberového sústredenia.

| | Meno | $\sum$ | CK | d.ú. | ut | st | št | pi | so | ne | po |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. | Ján Hozza | 664.5 | 40 | 17.5 | 50 | 148 | 20.5 | 123 | 125 | 74.5 | 66 |
| 2. | Matej Balog | 658.5 | 40 | 23 | 38 | 79 | 69 | 138 | 84.5 | 90 | 97 |
| | Nikola Djokić | 651.5 | 50 | 25 | 16 | 80 | 113 | 137 | 64 | 134 | 32.5 |
| 3. | Michal Anderle | 575.2 | 40 | 18.5 | 60 | 96.5 | 79 | 100.2 | 33 | 64 | 84 |
| 4. | Marián Horňák | 542.5 | 31 | 2.5 | 34 | 65 | 81 | 114 | 47 | 74.5 | 93.5 |
| | Lazar Todorović | 415 | 44.5 | 25 | 25 | 61.5 | 49 | 63 | 57 | 60 | 30 |
| 5. | Matej Večeřík | 413.5 | 30 | 5 | 11.5 | 81.5 | 25 | 76.5 | 47 | 43 | 94 |
| 6. | Jozef Brandys | 401 | 29 | 13 | 30.5 | 57.5 | 58.5 | 60.5 | 54 | 26 | 72 |
| 7. | Andrej Mariš | 383 | 26 | 16 | 41.5 | 43 | 53 | 58.5 | 22.5 | 54 | 68.5 |
| | Cyril Frei | 351.5 | 44.5 | 25 | 7 | 32.5 | 56 | 43.5 | 44 | 80.5 | 18.5 |
| 8. | Marek Špano | 347 | 21 | 11 | 50 | 40.5 | 36 | 48 | 28.5 | 74.5 | 37.5 |
| 9. | Jerguš Greššák | 303 | 22 | 13 | 9 | 22.5 | 40.5 | 43 | 27 | 31.5 | 94.5 |
| | Stefan Lippuner | 281 | 42 | 25 | 18 | 42 | 16 | 62.5 | 15.5 | 30 | 30 |
| 10. | Kamila Součková | 183.5 | 20 | 6 | 17.5 | 47.5 | 13 | 25 | 15 | 10.5 | 29 |
| 11. | Dušan Plavák | 158.5 | 21 | 5 | 6.5 | 25 | 10 | 24.5 | 8 | 26.5 | 32 |

## Medzinárodné prípravné sústredenie v Davose

Vďaka blízkej spolupráci medzi národnými olympiádami v informatike na Slovensku a vo Švajčiarsku mali vybraní riešitelia KSP a OI možnosť zúčastniť sa prípravného sústredení vo švajčiarskom Davose v dňoch 7. až 10. februára 2011. Toto sústredenie malo tento rok skutočne medzinárodnú účasť: okrem domácich a našich súťažiacich sa ho zúčastnili aj delegácie z Ruska a Hongkongu. Výsledky sústredenia uvádzame v nasledujúcej tabuľke.

| Meno | kraj. | day1 | day2 | day3 | day4 | $\sum$ |
|---|---|---|---|---|---|---|
| 1. Egor Suvorov | RUS | 398 | 400 | 336 | 320 | 1454 |
| 2. Igor Pyshkin | RUS | 292 | 300 | 300 | 315 | 1207 |
| 3. Nikola Djokić | SUI | 249 | 290 | 270 | 320 | 1129 |
| 4. Chan Pak Hay | HKG | 324 | 300 | 300 | 200 | 1124 |
| 5. Ján Hozza | SVK | 318 | 208 | 294 | 290 | 1110 |
| 6. Matej Balog | SVK | 332 | 181 | 220 | 256 | 989 |
| 7. Michal Anderle | SVK | 287 | 182 | 228 | 211 | 908 |
| 8. Marián Horňák | SVK | 245 | 87 | 216 | 200 | 748 |
| 9. Yik Wai Pan | HKG | 139 | 153 | 200 | 255 | 747 |
| 10. Johannes Kapfhammer | SUI | 210 | 60 | 218 | 215 | 703 |
| 11. Lazar Todorović | SUI | 199 | 70 | 204 | 192 | 665 |
| 12. Kwok Cheuk Hang | HKG | 121 | 100 | 240 | 196 | 657 |
| 13. Wong Chi Wai | HKG | 133 | 109 | 220 | 180 | 642 |
| 14. Dmitry Philippov | RUS | 196 | 40 | 216 | 184 | 636 |
| 15. Johannes Wüthrich | SUI | 147 | 120 | 224 | 136 | 627 |
| 16. Luzi Sennhauser | SUI | 148 | 93 | 222 | 140 | 603 |
| 17. Marco Keller | SUI | 179 | 90 | 202 | 116 | 587 |
| 18. Cyril Frei | SUI | 133 | 55 | 208 | 188 | 584 |
| 19. Thomas Leu | SUI | 124 | 74 | 200 | 144 | 542 |
| 20. Stefan Lippuner | SUI | 173 | 0 | 212 | 140 | 525 |
| 21. Timon Stampfli | SUI | 100 | 0 | 160 | 156 | 416 |
| 22. Martin Jehli | SUI | 100 | 10 | 178 | 116 | 404 |
| 23. Peter Müller | SUI | 100 | 40 | 178 | 82 | 400 |
| 24. Andre Ryser | SUI | 100 | 0 | 172 | 100 | 372 |

# Česko-poľsko-slovenské prípravné sústredenie

V poradí už trináste súťažné stretnutie najlepších stredoškolákov z Čiech, Poľska a Slovenska sa uskutočnilo (opäť po troch rokoch) na Slovensku. Pôvodné plány organizátorov narušila povodeň, a tak sústredenie začínalo v núdzovom režime v Bratislave. Až po odstránení následkov rozbúreného živlu sa mohli účastníci presunúť do prírody – do školiaceho strediska v Modre-Piesku. A keďže sa nakoniec počasie umúdrilo, okrem programovania si účastníci užili aj na pekný výlet na Veľkú Homolu.

V súťaži sa netradične na prvom mieste podarilo umiestniť súťažiacemu z Českej republiky. Najlepší z našich súťažiacich sa tento rok umiestnil až na siedmom mieste.

| meno | kraj. | day1 | day2 | day3 | day4 | $\sum$ |
|---|---|---|---|---|---|---|
| 1. Filip Hlásek | CZE | 176.0 | 90.0 | 284.0 | 260.0 | 810.0 |
| 2. Piotr Bejda | POL | — | 270.0 | 242.0 | 255.5 | 767.5 |
| 3. Krzysztof Pszeniczny | POL | 159.5 | 200.0 | 232.0 | 175.0 | 766.5 |
| 4. Krzysztof Kiewicz | POL | 44.0 | 180.0 | 209.0 | 251.5 | 684.5 |
| 5. Jan Kanty Milczek | POL | 130.5 | 160.0 | 208.0 | 170.0 | 668.5 |
| 6. Krzysztof Leszczńyski | POL | — | 240.0 | 252.0 | 172.0 | 664.0 |
| 7. Ján Hozza | SVK | 60.5 | 210.0 | 188.0 | 180.5 | 639.0 |
| 8. Hynek Jemelík | CZE | 124.0 | 80.0 | 229.0 | 160.0 | 593.0 |
| 9. Matej Balog | SVK | 49.0 | 90.0 | 202.0 | 203.5 | 544.5 |
| 10. Bartosz Tarnawski | POL | 127.0 | 50.0 | 182.0 | 160.0 | 519.0 |
| 11. Michal Anderle | SVK | 88.5 | 120.0 | 118.0 | 174.5 | 501.0 |
| 12. Ondřej Hübsch | CZE | 70.0 | 120.0 | 91.5 | 100.0 | 381.5 |
| 13. Marián Horňák | SVK | 10.0 | 90.0 | 126.5 | 88.0 | 314.5 |
| 14. Jakub Zíka | CZE | 0 | 60.0 | 83.5 | 170.0 | 313.5 |
| 15. Lukáš Folwarczný | CZE | 92.5 | 110.0 | 46.0 | 0 | 248.5 |
| 16. Jozef Brandys | SVK | 27.0 | 80.0 | 91.0 | 4.0 | 202.0 |
| 17. Martin Zikmund | CZE | 54.0 | 40.0 | 91.0 | 9.0 | 194.0 |
| 18. Andrej Mariš | SVK | 70.5 | 80.0 | — | — | 150.5 |

# Stredoeurópska olympiáda v informatike

V roku 2011 sa Stredoeurópska olympiáda v informatike (CEOI) konala v poľskej Gdyni v dňoch 7. až 12. júla 2011. Zúčastnili sa jej žiaci z tradičných siedmich krajín organizujúcich túto súťaž: Českej republiky, Chorvátska, Maďarska, Nemecka, Poľska, Slovenska a Rumunska. Okrem nich boli ako hostia pozvaní súťažiaci zo Slovinska a Švajčiarska.

Slovensko na CEOI 2011 reprezentovali štyria stredoškoláci: Jozef Brandys (Gym. Námestovo), Marián Horňák (Gym. Párovská, Nitra), Andrej Mariš (Piar. gym., Nitra) a Matej Večerík (Š. pre mim. nadané deti, Bratislava). Našu delegáciu na tejto súťaži viedli doc. RNDr. Gabriela Andrejková, CSc. a RNDr. Rastislav Krivoš-Belluš (obaja z Ústavu informatiky PF UPJŠ v Košiciach).

Napriek tomu, že na CEOI tradične posielame hlavne mladších žiakov, ktorí tak môžu získať skúsenosti pred reprezentáciou na Medzinárodnej olympiáde, sa tento rok podarilo dosiahnuť významný úspech – priniesť na Slovensko jednu z piatich udelených zlatých medailí.

Podrobné výsledky našich súťažiacich uvádzame v tabuľke.

| Meno | 1. deň | | | 2. deň | | | $\sum$ | medaila |
|---|---|---|---|---|---|---|---|---|
| Matej Večerík | 100 | 36 | 20 | 20 | 100 | 32 | 308 | zlatá |
| Marián Horňák | 100 | – | – | 80 | 90 | – | 270 | strieborná |
| Andrej Mariš | 40 | 36 | 20 | 0 | 40 | 32 | 168 | bronzová |
| Jozef Brandys | 20 | 27 | 0 | 0 | 10 | 32 | 89 | |

# Medzinárodná olympiáda v informatike

V dňoch 22. až 29. júla hostilo mesto Pattaya v Thajsku účastníkov 23. Medzinárodnej olympiády v informatike (IOI). Celkovo 302 stredoškolákov z 80 krajín si zmeralo sily v riešení informatických úloh. Výpravy pozostávli z dvoch lídrov a štyroch súťažiacich. Slovensko na tejto akcii reprezentovali ako vedúci tímu RNDr. Andrej Blaho, PhD. a Mgr. Marek Zeman (obaja Fakulta matematiky, fyziky a informatiky UK v Bratislave). Ako súťažiaci do Thajska cestovali títo stredoškoláci: Michal Anderle z Gymnázia Haličská v Lučenci, Matej Balog z Gymnázia Grösslingová v Bratislave, Marián Horňák z Gymnázia Párovská v Nitre a Ján Hozza z Gymnázia Jura Hronca v Bratislave.

Súťaž bola rozdelená do dvoch dní. V každý deň súťažiaci riešili 3 informatické úlohy algoritmickej povahy. Na navrhnutie a napísanie programov mali k dispozícii 5 hodín času. Sadu úloh zostavila medzinárodná odborná komisia (ISC). V tomto sedemčlennom orgáne má Slovensko svojho zástupcu. Je ním RNDr. Michal Forišek, PhD. (FMFI UK). Súťaž vďaka úsiliu všetkých, ktorí sa na jej organizácii podieľali, prebehla hladko a bezproblémovo.

Okrem samotnej súťaže pripravili domáci organizátori veľmi zaujímavý kultúrny program, takže pre celú slovenskú výpravu bol pobyt v Thajsku pôsobivým zážitkom. Do programu bola zaradená napríklad návšteva botanickej záhrady Nong Nooch alebo celodenný výlet do Bangkoku. V rámci tohto výletu mohli zúčastnení vidieť napríklad Kráľovskéhý paláca, ale aj bývalé kráľovské sídlo s chrámom smaragdového Budhu (Wat Phra Keaw) a iné zaujímavé lokality v hlavnom meste Thajska.

Hoci slovenská výprava zlatú medailu nepriniesla, za naše výsledky sa rozhodne nemusíme hanbiť. Z našich súťažiacich traja už dokončili strednú školu, len Marián Horňák má ešte štvrtú triedu pred sebou. Bude teda mať možnosť zúčastniť sa tejto súťaže aj o rok, keď sa bude konať v Taliansku. Veríme, že skúsenosti nadobudnuté na tejto IOI zúročí do čo najlepšieho výsledku. Výsledky našich súťažiacich uvádzame v tabuľke:

| Meno | 1. deň | | | 2. deň | | | $\sum$ | medaila |
|---|---|---|---|---|---|---|---|---|
| Ján Hozza | 100 | 21 | 100 | 100 | 26 | 81 | 428 | 46. miesto, striebro |
| Matej Balog | 49 | 43 | 100 | 100 | 26 | 98 | 416 | 52. miesto, striebro |
| Michal Anderle | 49 | 21 | 100 | 46 | 26 | 52 | 294 | 134. miesto, bronz |
| Marián Horňák | 0 | 0 | 17 | 100 | 50 | 81 | 248 | 171. miesto |

(správu spísal Marek Zeman)

RNDr. Michal Forišek PhD., Slovenská komisia OI
Dvadsiaty šiesty ročník Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava, 2011
134 strán, náklad 300 výtlačkov
Neprešlo jazykovou úpravou
ISBN 978-80-8072-117-6