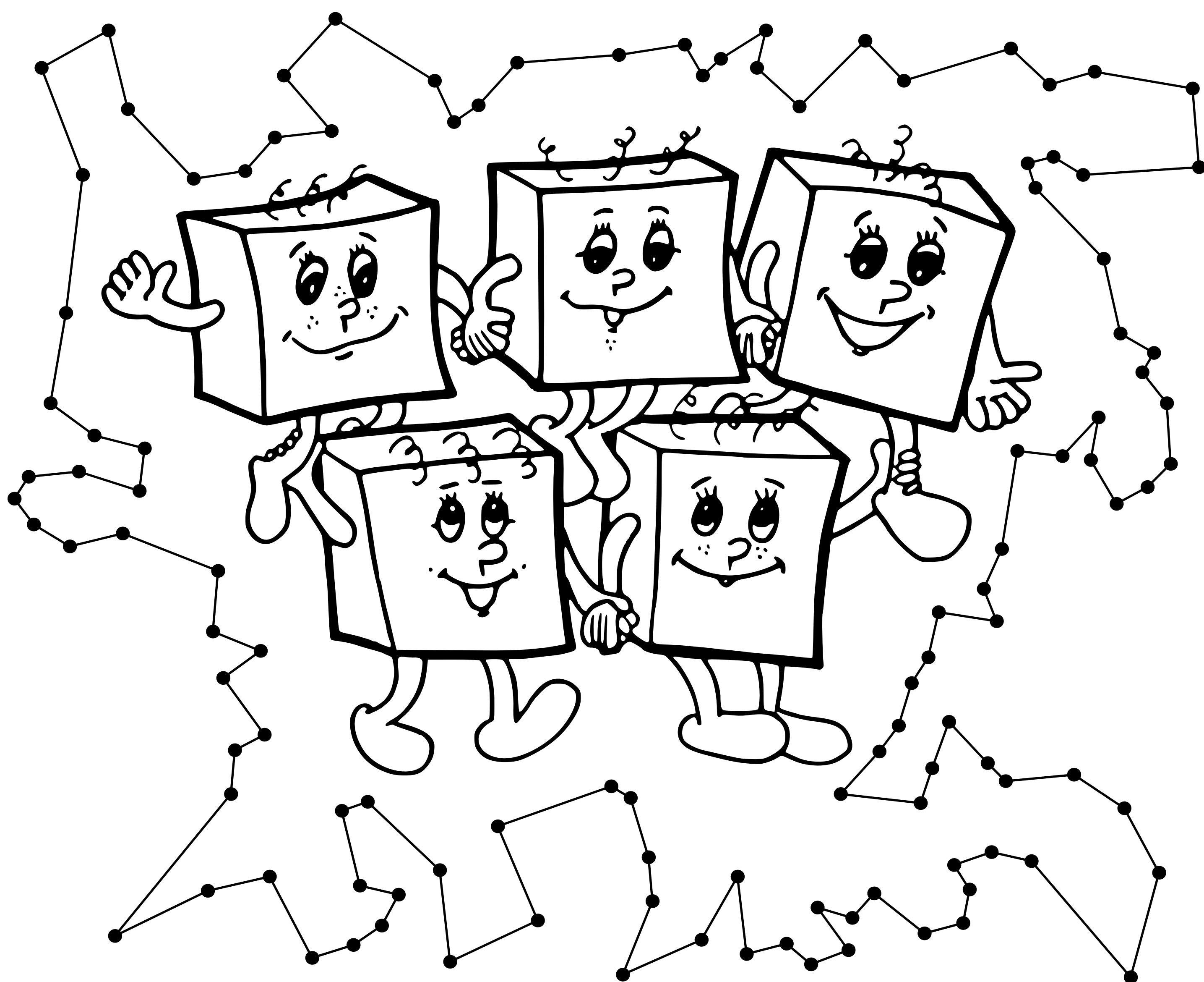
# OLYMPIÁDA V INFORMATIKE



**29. ročník** **2013/14**

# Dvadsiaty deviaty ročník Olympiády v informatike

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

Na obálke ročenky je vyznačených 114 bodov a tzv. Hamiltonovská kružnica – teda najkratšia cesta, ktorá ich postupne navštívi všetky a vráti sa na miesto odkiaľ začínala. Ide o veľmi dôležitý optimalizačný problém, ktorý súvisí s mnohými pre prax zaujímavými úlohami. Zároveň je však už dobre preskúmaný z teoretického hľadiska – lenže bohužiaľ s negatívnym výsledkom: vieme o ňom dokázať, že je veľmi ťažký. Nepoznáme žiaden polynomiálny algoritmus, ktorý by túto úlohu riešil, a máme dobré dôvody domnievať sa, že taký algoritmus ani neexistuje. Aj napriek tomu, že sme použili jeden z najefektívnejších známych všeobecných algoritmov (postupnú optimalizáciu pomocou celočíselného lineárneho programovania), trval výpočet riešenia, ktoré vidíte na obálke, približne dva dni.

---

RNDr. Michal Forišek PhD., Slovenská komisia OI
Dvadsiaty deviaty ročník Olympiády v informatike
ISBN 978-80-8072-156-5

# Obsah

O priebehu 29. ročníka Olympiády v informatike . . . . . . . . . . . . . 3

Zadania domáceho kola kategórie A . . . . . . . . . . . . . . . . . . . . 4
Študijný text: zoznam problémov (1) . . . . . . . . . . . . . . . . . . . 10
Študijný text: konkrétne semiorganické počítače . . . . . . . . . . . . . 12
Zadania domáceho kola kategórie B . . . . . . . . . . . . . . . . . . . . 15
Zadania krajského kola kategórie A . . . . . . . . . . . . . . . . . . . . 22
Študijný text: zoznam problémov (2) . . . . . . . . . . . . . . . . . . . 28
Zadania krajského kola kategórie B . . . . . . . . . . . . . . . . . . . . 30
Zadania celoštátneho kola kategórie A . . . . . . . . . . . . . . . . . . 39

Riešenia domáceho kola kategórie A . . . . . . . . . . . . . . . . . . . 50
Riešenia domáceho kola kategórie B . . . . . . . . . . . . . . . . . . . 73
Riešenia krajského kola kategórie A . . . . . . . . . . . . . . . . . . . 86
Riešenia krajského kola kategórie B . . . . . . . . . . . . . . . . . . . 100
Riešenia celoštátneho kola kategórie A . . . . . . . . . . . . . . . . . 111

Výsledky krajských kôl kategórie B . . . . . . . . . . . . . . . . . . . 139
Výsledky celoštátneho kola kategórie A . . . . . . . . . . . . . . . . . 141
Výsledky výberového sústredenia . . . . . . . . . . . . . . . . . . . . . 143

Medzinárodné prípravné sústredenie v Davose . . . . . . . . . . . . . . 144
1. vyšehradské prípravné sústredenie . . . . . . . . . . . . . . . . . . . 146
Stredoeurópska olympiáda v informatike . . . . . . . . . . . . . . . . . 147
Medzinárodná olympiáda v informatike . . . . . . . . . . . . . . . . . . 148

# O priebehu 29. ročníka Olympiády v informatike

V školskom roku 2013/14 na Slovensku prebehol už dvadsiaty deviaty ročník Olympiády v informatike (OI). Do súťaže sa zapojilo 76 žiakov v kategórii A (starší) a 32 v kategórii B (mladší). Najlepších 32 riešiteľov kategórie A sa zúčastnilo celoštátneho kola, ktoré sa tohto roku konalo v Bratislave.

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. Vladimír Boža, FMFI UK, Bratislava
- Bc. Michal Anderle, 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
- Mgr. Ing. Roman Horváth, KMI PedF TU, krajský predseda pre TT
- RNDr. Peter Varša, PhD., KI FRI ŽU, krajský predseda pre ZA

Zástupcom IUVENTY zabezpečujúcim organizačnú stránku súťaže bola Mgr. Mária Gajarová.

Michal Forišek, podpredseda SK OI

# Zadania domáceho kola kategórie A

## A-I-1 Taká zima...

„Teda, taká zima ako dnes bola naposledy na Štedrý deň..." povzdychol si nad pivom dedo Bonifác. „To nič nie je, včera bola taká zima, že podobná bola naposledy druhého decembra!" chcel ho tromfnúť dedo Ignác. No na Bonifáca nemal. „Pche, to zabúdaš na zimu čo bola piateho januára, tá bola od tej včera menšia, ale od druhého decembra väčšia!"

Ignác celú noc nemohol zaspať, spomínal na teploty a snažil sa prísť na výrok, ktorý už Bonifác nedokáže vyvrátiť ani prekonať. Chcel by povedať výrok nasledovného typu: „pred $x$ dňami bolo tak teplo, že podobne bolo naposledy predtým pred $y$ dňami".

Nech $T[x]$ a $T[y]$ sú teploty v dotyčné dva dni. Aby Bonifác Ignácov výrok nevedel vyvrátiť, musí platiť, že v žiaden z dní medzi $x$ a $y$ nemohla byť teplota medzi $T[x]$ a $T[y]$, vrátane. No a aby Ignácov výrok bol čo najohurúcejší, musí byť $y - x$ (teda počet dní, ktoré medzi spomínanými udalosťami uplynuli) čo najväčšie.

**Súťažná úloha:**

Dané je pole $T[0..n-1]$, pričom $T[i]$ je priemerná teplota pred $i$ dňami. Nájdite indexy $x < y$ také, že platí:

- Pre každé $i$ také, že $x < i < y$, platí buď $T[i] < \min(T[x], T[y])$ alebo $T[i] > \max(T[x], T[y])$.
- Rozdiel $y - x$ je najväčší možný.
- Ak je stále takých dvojíc $(x, y)$ viac, tak chceme tú, kde je $y$ najväčšie možné.

**Formát vstupu a výstupu:**

V prvom riadku vstupu je počet dní $n$. V druhom riadku vstupu je $n$ medzerami oddelených celých čísel $T[0]$ až $T[n-1]$. Na výstup vypíšte jediný riadok a v ňom hľadané dve celé čísla $x$ a $y$, oddelené medzerou.

**Obmedzenia a hodnotenie:**

Vždy bude platiť $n \geq 2$ a pre všetky $i$ bude $-10^9 \leq T[i] \leq 10^9$.

Riešenia budú testované na desiatich sadách testovacích vstupov; za každú z nich sa dá získať 1 bod. Maximálnu hodnotu $n$ v jednotlivých sadách udávame v nasledujúcej tabuľke.

| sada | #1 | #2 | #3 | #4 | #5 | #6 | #7 | #8 | #9 | #10 |
|---|---|---|---|---|---|---|---|---|---|---|
| max. $n$ | 10 | 20 | 100 | 1500 | 6000 | 7000 | 50 000 | 65 000 | 85 000 | 100 000 |

Navyše bude platiť: V sadách #1, #4 a #7 obsahuje pole $T$ práve raz každú z hodnôt od 1 po $n$. V sadách #2, #5 a #8 obsahuje pole $T$ len hodnoty od 1 po $n$ (ale každú ľubovoľne veľakrát).

**Príklady:**

**Vstup**

```
8
-5 10 32 17 24 -12 13 19
```

**Výstup**

```
1 6
```

*Pred $x = 1$ dňom bola teplota 10 stupňov, pred $y = 6$ dňami to bolo 13 stupňov. Žiadna z teplôt medzi nimi (32, 17, 24 ani -12) neleží v intervale $\langle 10, 13 \rangle$, takže tento výrok Bonifác skutočne nevie vyvrátiť.*

*No a žiaden dlhší interval už nevyhovuje – napr. nemôže byť $(x, y) = (1, 7)$, lebo $T[3] = 17$ leží medzi $T[1] = 10$ a $T[7] = 19$. Na záver si všimnite, že dvojica $(x, y) = (0, 5)$ síce vyhovuje a má rovnakú dĺžku, ale nami vypísaná dvojica má väčšiu hodnotu $y$.*

**Vstup**

```
5
7 7 7 7 7
```

**Výstup**

```
3 4
```

*Tu sa nedá nič robiť, musíme zvoliť $y = x + 1$. Následne zvolíme najväčšie možné $y$.*

## A-I-2 Dva ploty

Marika má na dedine babku. Marikina babka má sad a v ňom tie najlepšie ovocné stromy: hrušky maslovky. Lenže na hrušky sa naučila chodiť celá dedina, a tak keď Marika príde babku pozrieť cez jesenné prázdniny, po hruškách už ani chýru, ani slychu.

Peťko jej chce pomôcť. Zohnal si nejaké koly, laty, klince a kladivo a ide okolo stromov postaviť plot. Už-už sa chcel pustiť do práce, keď sa zháčil. Čo tak postaviť tie ploty dva? Nebolo by na ne treba menej materiálov?

**Súťažná úloha:**

Daných je $n \geq 3$ bodov v rovine. Hľadáme jeden alebo dva mnohouholníky také, že:

- každý z $n$ daných bodov leží vo vnútri alebo na obvode aspoň jedného z mnohouholníkov,
- každý vrchol každého mnohouholníka je v niektorom z daných bodov,
- žiadny mnohouholník nemá nulový obsah.

Vypočítajte najmenšiu možnú hodnotu celkového obvodu nájdených mnohouholníkov.

**Formát vstupu a výstupu:**

V prvom riadku vstupu je číslo $n$. V každom z nasledujúcich $n$ riadkov sú dve celé čísla $x_i$, $y_i$: súradnice jedného z bodov. Platí $1 \leq x_i, y_i \leq 10^6$.

Na výstup vypíšte jeden riadok a v ňom jedno reálne číslo: najmenšiu celkovú dĺžku plotu. Vypíšte aspoň 6 miest za desatinnou bodkou. Odpovede s relatívnou chybou nanajvýš $10^{-6}$ budú považované za správne.

**Obmedzenia a hodnotenie:**

Riešenia budú testované na desiatich sadách testovacích vstupov; za každú z nich sa dá získať 1 bod.

Maximálne $n$ v jednotlivých sadách vstupov bude nasledovné: $(3, 5, 8, 18, 18, 50, 50, 200, 300, 300)$.

V sadách #1, #2, #4, #6 a #9 bude platiť, že žiadne tri body neležia na jednej priamke. V sadách #1, #6 a #8 budú použité len vstupy, pre ktoré je optimálnym riešením postaviť jeden plot.

**Príklady:**

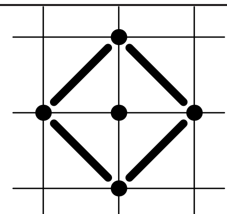
**Vstup**

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

**Výstup**

```
5.65685425
```

*Oplatí sa postaviť štvorcový plot. Jeho presná dĺžka obvodu je $4\sqrt{2}$.*



*Za správne považujeme odpovede cca. z rozsahu* $[5.6568486, 5.6568599]$.

**Vstup**

```
6
2 1
8 1
7 3
1 2
3 3
8 3
```

**Výstup**

```
11.12241749487
```

## A-I-3 Kaviarne

Keď sa Dávidko sťahoval do New Yorku, potreboval si nájsť niekde na Manhattane bývanie. Niektorí ľudia sa snažia hľadať bývanie blízko pri práci, iní v peknom prostredí, no Dávidkova priorita bola jasná: *káva*. Plánuje pravidelne navštevovať aspoň $k$ kaviarní v okolí svojho bytu.

Pre jednoduchosť si Manhattan predstavíme ako štvorcovú sieť, po ktorej sa dá pohybovať len v štyroch základných smeroch. Na niektorých mrežových bodoch (križovatkách) sú kaviarne; na každom najviac jedna.

**Súťažná úloha:**

Na vstupe je číslo $k$ a mapa Manhattanu. Pre každú križovatku spočítajte najmenšie $d$, pre ktoré platí: ak by Dávidko býval na tejto križovatke a bol ochotný prejsť vzdialenosť $d$, mal by na výber aspoň $k$ kaviarní, do ktorých sa vie dostať.

**Formát vstupu:**

V prvom riadku je číslo $k$.

V druhom riadku je rozmer $n$ štvorcovej siete predstavujúcej Manhattan: tvorí ho $n$ vodorovných a $n$ zvislých ciest. Máme teda presne $n^2$ križovatiek.

Zvyšok vstupu tvorí $n$ riadkov, v každom z nich je $n$ čísel: 1 predstavuje križovatku s kaviarňou, 0 križovatku bez kaviarne. Dokopy je v Manhattane aspoň $k$ kaviarní.

**Formát výstupu:**

Vypíšte $n$ riadkov a v každom z nich $n$ čísel: pre každú križovatku vzdialenosť $d$ takú, že do vzdialenosti $d$ vrátane od dotyčnej križovatky leží aspoň $k$ kaviarní.

**Hodnotenie:**

Plných 10 bodov dostanete za riešenie s asymptoticky optimálnou časovou zložitosťou. Pomalšie korektné riešenia môžu dostať 4 až 8 bodov podľa konkrétnej časovej zložitosti.

Ak úlohu neviete riešiť, môžete dostať 3 body za vyriešenie špeciálneho prípadu $k = 1$.

**Príklady:**

**Vstup**

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

**Výstup**

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

*Pre $k = 1$ hľadáme vzdialenosť do najbližšej kaviarne.*

**Vstup**

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

**Výstup**

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

## A-I-4 Mimozemské počítače

*Študijný text k tejto úlohe nájdete na strane 10 tejto ročenky.*
*Jednotlivé podúlohy spolu nesúvisia, môžete ich riešiť v ľubovoľnom poradí.*
*Na časovej zložitosti vašich algoritmov pri hodnotení nezáleží – len musí byť polynomiálna.*

**Podúloha A (3 body):**

Mimozemšťania nám dodali sálový KSP, ktorý rozhoduje problém existencie Hamiltonovskej kružnice v danom jednoduchom neorientovanom grafe.

Tento KSP má teda funkciu `kruznica(n,E)`. Táto funkcia čaká ako parametre počet $n$ vrcholov grafu a zoznam $E$ jeho hrán. Ak v danom grafe existuje Hamiltonovská kružnica, KSP rozsvieti zelené svetlo, inak rozsvieti červené.

Na vstupe dostanete jednoduchý neorientovaný graf $G$. Napíšte program s polynomiálnou časovou zložitosťou, ktorý rozsvieti zelené alebo červené svetlo podľa toho, či $G$ obsahuje aspoň jednu Hamiltonovskú cestu.

**Podúloha B (3 body):**

A teraz naopak. Mimozemšťania nám dodali sálový KSP, ktorý rozhoduje problém existencie aspoň jednej Hamiltonovskej cesty v danom jednoduchom neorientovanom grafe.

Tento KSP má teda funkciu `cesta(n,E)`. Táto funkcia čaká ako parametre počet $n$ vrcholov grafu a zoznam $E$ jeho hrán (každá hrana je dvojica čísel od 0 po $n-1$). Ak v danom grafe pre nejaké $u$ a $v$ existuje Hamiltonovská cesta z $u$ do $v$, KSP rozsvieti zelené svetlo, inak rozsvieti červené.

Na vstupe dostanete jednoduchý neorientovaný graf $G$. Napíšte program s polynomiálnou časovou zložitosťou, ktorý rozsvieti zelené alebo červené svetlo podľa toho, či $G$ obsahuje Hamiltonovskú kružnicu.

**Podúloha C (4 body):**

Mimozemšťania nám dodali kufríkový KSP, ktorý rozhoduje problém existencie 3-farbenia v zadanom grafe.

Tento KSP má teda funkciu `je_trojfarbitelny(n,E)`. Táto funkcia čaká ako parametre počet $n$ vrcholov grafu a zoznam $E$ jeho hrán. Na výstupe táto funkcia vracia `True` ak sa dá tento graf ofarbiť tromi farbami a `False`, ak sa ofarbiť nedá. Túto funkciu môžeme v našom programe volať ľubovoľne veľakrát pre ľubovoľné grafy.

Na vstupe dostanete jednoduchý neorientovaný graf $G$, ktorý sa dá ofarbiť tromi farbami. Napíšte program s polynomiálnou časovou zložitosťou, ktorý jedno takéto ofarbenie nájde.

**Programovacie jazyky:**

Vo svojich riešeniach môžete používať ľubovoľný štrukturovaný programovací jazyk. Vhodne si zvoľte potrebné dátové štruktúry. Napr. v Pascale by funkcia z podúlohy A mohla vyzerať nasledovne:

```
type hrana = array[0..1] of longint;
procedure kruznica(n : longint; m : longint; E : array of hrana);
```

a v C++ môžeme použiť buď nízkoúrovňové polia:

```
void kruznica(int n, int m, int E[][2]);
```

alebo trebárs aj vektor dvojíc:

```
void kruznica(int n, vector< pair<int,int> > E);
```

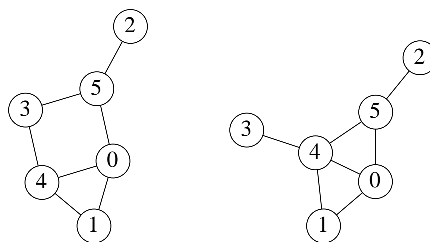
## Študijný text: zoznam problémov (1)

*Hlavný študijný text, uvedený na ďalších stranách, sa odkazuje na niekoľko problémov, ako napr. „existencia Hamiltonovskej kružnice“. V nasledujúcom texte uvádzame stručné definície týchto problémov.*

Jednoduchý neorientovaný graf je usporiadaná dvojica $(V, E)$. $V$ je konečná množina objektov nazývaných vrcholy. $E$ je konečná množina dvojíc vrcholov; jej prvky voláme hrany. Takýto graf si môžeme predstaviť napríklad ako cestnú sieť: $V$ je množina miest a $E$ je množina dvojíc miest spojených cestami. Počet vrcholov budeme označovať $n$, počet hrán bude $m$. Jednotlivé vrcholy budeme číslovať od 0 po $n-1$.

---

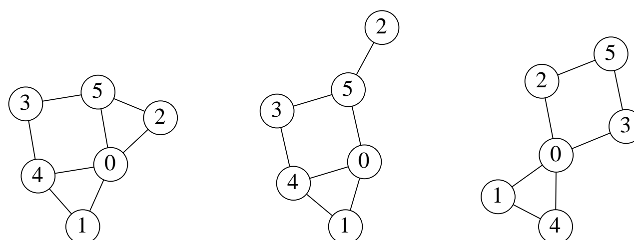
**Hamiltonovská cesta z u do v:**
Daný je jednoduchý neorientovaný graf $G$ a jeho dva rôzne vrcholy $u$ a $v$. Existuje v grafe $G$ cesta, ktorá začína vo vrchole $u$, končí vo vrchole $v$ a navštívi každý vrchol grafu $G$ práve raz?




*Graf vľavo obsahuje Hamiltonovskú cestu z 1 do 2:*
*dá sa ísť postupne cez vrcholy 1, 0, 4, 3, 5 a 2.*
*Graf vpravo Hamiltonovskú cestu z 1 do 2 neobsahuje:*
*ak chceme navštíviť vrchol 3, pôjdeme* $2\times$ *cez vrchol 4.*

---

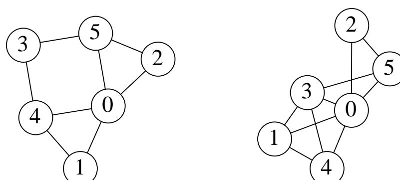
**Hamiltonovská kružnica:**
Daný je jednoduchý neorientovaný graf $G$ s $n \geq 3$ vrcholmi. Existuje v grafe $G$ kružnica, ktorá navštívi každý vrchol grafu $G$ práve raz?



*Graf vľavo obsahuje Hamiltonovskú kružnicu:*
*napr. začneme v 1 a postupne ideme do 0, 2, 5, 3, 4 a späť do 1.*
*Graf v strede ani graf vpravo Hamiltonovskú kružnicu neobsahujú.*

---

**Farbenie grafu k farbami:**
Daný je jednoduchý neorientovaný graf $G$. Každému vrcholu grafu $G$ chceme priradiť jednu z $k$ možných farieb (pre naše pohodlie označených číslami od 0 po $k-1$). Pritom ale musíme dodržať pravidlo, že žiadna hrana nesmie spájať dva vrcholy rovnakej farby.



*Pre graf vľavo a* $k = 2$ *ofarbenie neexistuje.*
*Napr. vrcholy 0, 1 a 4 musia zjavne mať navzájom rôzne farby.*
*Pre graf vľavo a* $k = 3$ *môžeme napr. dať vrcholom 1, 2, 3 farbu 0, vrcholom 4 a 5 farbu 1 a vrcholu 0 farbu 2.*
*Pre graf vpravo a* $k = 3$ *žiadne prípustné ofarbenie vrcholov neexistuje.*

---

**k-partícia postupnosti:**
Daná je postupnosť $n$ kladných celých čísel. Dá sa ich rozdeliť do $k$ disjunktných skupín (partícií) tak, aby čísla v každej skupine mali presne rovnaký súčet?

*Pre postupnosť* $(3, 1, 4, 1, 5, 8)$ *a* $k = 2$ *je odpoveď „áno“:*
*jedno vyhovujúce delenie je na* $3 + 8$ *a* $1 + 4 + 1 + 5$.
*Pre postupnosť* $(3, 1, 4, 1, 5, 9)$ *a* $k = 2$ *je odpoveď „nie“.*
*Pre postupnosť* $(3, 1, 4, 1, 5, 1)$ *a* $k = 3$ *je odpoveď „áno“:*
*jedno vyhovujúce delenie je na* $3 + 1 + 1$, $1 + 4$ *a* $5$.
*Pre postupnosť* $(2, 2, 2, 2, 2, 2, 12)$ *a* $k = 3$ *je odpoveď „nie“.*

## Študijný text: konkrétne semiorganické počítače

*V tomto študijnom texte v listingoch programov používame Python. Na adrese* `http://oi.sk/archiv/2013/a14.html` *nájdete ekvivalentné listingy v C, C++ a Pascale.*

Píše sa rok 2113. Zem pred pár rokmi kontaktovala vyspelá mimozemská rasa z planéty Žblnk. Títo mimozemšťania nás zásobujú novými konkrétnymi semiorganickými počítačmi (KSP), ktoré vedia riešiť rôzne konkrétne výpočtové úlohy. Vy ste členmi elitného výskumného tímu, ktorý má jediný cieľ: integrovať tieto KSP do normálnych počítačov a prinútiť ich riešiť naše problémy.

Mimozemským počítačom však vôbec nerozumieme, všetky snahy o ich rozobranie sa stretajú s neúspechom. A teda jediný spôsob, ako ich vieme využiť, je zadať im vstup a počkať si na výstup. Tu je prvá zaujímavosť: u KSP nezáleží na veľkosti vstupu ani na probléme, ktorý daný KSP rieši. Keď ľubovoľný KSP spustíme na ľubovoľnom vstupe, odpoveď vždy dostaneme o presne 47 stotín sekundy. (Pri odhade časovej zložitosti programov toto považujeme za konštantu.)

Mimozemšťania nám dodávajú dva druhy KSP: *kufríkové* a *sálové*.

Kufríkový KSP má tvar kufríka, ktorý má na sebe dva porty. Do jedného vieme pripojiť kábel so vstupom, do druhého zas kábel, po ktorom nám KSP pošle výstup. Kufríkový KSP teda vieme používať v našom programe ľubovoľne veľakrát, s navzájom rôznymi vstupmi.

**Ukážkové zadanie 1.** Mimozemšťania nám dodali kufríkový KSP, ktorý rozhoduje problém existencie Hamiltonovskej cesty z $u$ do $v$. Tento KSP teda počíta funkciu `cesta(n,E,u,v)`. Táto funkcia čaká ako parametre počet $n$ vrcholov grafu, zoznam $E$ jeho hrán a dve čísla vrcholov $u$ a $v$. ($E$ je zoznam $m$ dvojíc čísel, každé z rozsahu od 0 po $n-1$.) Na výstupe funkcia `cesta` vráti `True` alebo `False` podľa toho, či dotyčný graf obsahuje dotyčnú Hamiltonovskú cestu.

Našou úlohou je napísať program, ktorý bude v polynomiálnom čase riešiť problém existencie Hamiltonovskej kružnice. Na vstupe teda dostaneme jednoduchý neorientovaný graf s $n \geq 3$ vrcholmi, o ktorom máme zistiť, či obsahuje Hamiltonovskú kružnicu.

**Analýza zadania.** Musíme teda napísať program, ktorý spraví nasledovné:

1. Na vstupe dostane $n$ (počet vrcholov grafu) a $E$ (zoznam hrán grafu)

2. Urobí nejaké výpočty, počas ktorých môže ľubovoľne veľakrát používať funkciu `cesta`.
3. Vráti na výstup `True` alebo `False` podľa toho, či vstupný graf obsahuje Hamiltonovskú kružnicu.

**Riešenie.**

```
def kruznica(n,E):
    # pre kazdu hranu (x,y) v zozname hran E:
    for (x,y) in E:
        # newE = E okrem hrany (x,y)
        newE = [ (u,v)   for (u,v) in E   if (u,v) != (x,y) ]
        if cesta(n,newE,x,y): return True
    return False
```

**Popis riešenia.** Postupne pre každú hranu $(x, y)$ zadaného grafu si položíme otázku: „Existuje Hamiltonovská kružnica prechádzajúca hranou $(x, y)$?“ Kedy takáto kružnica existuje? Práve vtedy, keď vo zvyšku grafu existuje Hamiltonovská cesta z $x$ do $y$. Vyrobíme si teda nový zoznam hrán `newE`, do ktorého dáme všetky hrany okrem $(x, y)$, a na takýto graf zavoláme funkciu `cesta` nášho kufríkového KSP.

Ak v pôvodnom grafe nejaká Hamiltonovská kružnica existuje, náš program ju nájde a dá odpoveď `True`, len čo vyskúšame niektorú z jej hrán ako $(x, y)$. A naopak, ak náš program niekedy dá odpoveď `True`, tak v pôvodnom grafe existuje Hamiltonovská cesta z nejakého $x$ do nejakého $y$, no a tá spolu s hranou $(x, y)$ tvorí hľadanú kružnicu. Náš program teda naozaj vždy robí to, čo má.

Časová zložitosť nášho programu je $\Theta(m^2)$, kde $m$ je počet hrán zadaného grafu. Keďže v jednoduchom neorientovanom grafe platí $m \leq n(n-1)/2$, zhora môžeme našu časovú zložitosť odhadnúť ako $O(n^4)$.

**Poznámka.** Existuje aj efektívnejšie riešenie. Stačí si uvedomiť, že pred hľadaním Hamiltonovskej cesty z $x$ do $y$ vôbec netreba hranu $(x, y)$ z grafu odstraňovať. Hamiltonovská cesta z $x$ do $y$ ju v grafe s $n \geq 3$ vrcholmi aj tak nemôže obsahovať. Stačí teda pre každú hranu $(x, y)$ zistiť, či platí `cesta(n,E,x,y)`. Ďalej, Hamiltonovská kružnica musí prechádzať vrcholom 0, stačí teda namiesto každej hrany skúšať len hrany idúce z vrcholu 0.

Sálový KSP je obrovský a jeho jediným výstupom sú dve svetlá: červené a zelené. S týmto výstupom ďalej nepracujeme.

**Ukážkové zadanie 2.** Mimozemšťania nám dodali sálový KSP, ktorý rozhoduje problém 3-partície. Tento KSP má teda funkciu `tri_particia(X)`, ktorá rozsvieti zelené alebo červené svetlo podľa toho, či postupnosť $X$ má 3-partíciu – teda či sa jej prvky dajú rozdeliť do *troch* skupín s navzájom rovnakým súčtom.

Na vstupe dostanete postupnosť kladných celých čísel $A$. Napíšte program s polynomiálnou časovou zložitosťou, ktorý rozsvieti zelené alebo červené svetlo podľa toho, či má postupnosť $A$ 2-partíciu (teda podľa toho, či vieme prvky $A$ rozdeliť do *dvoch* skupín s rovnakým súčtom).

**Analýza zadania.** Musíme teda napísať program, ktorý spraví nasledovné:

1. Na vstupe dostane postupnosť čísel $A$.
2. Urobí nejaké výpočty a vyrobí nejakú novú postupnosť čísel $X$.
3. Na konci (každej možnej vetvy) výpočtu raz zavolá funkciu `tri_particia(X)`, ktorá rozsvieti správne svetlo.

**Riešenie.**

```
def dva_particia(A):
    s = sum(A)
    if s%2 == 0:                # s%2 je zvysok cisla s po deleni 2
        X = A + [ s//2 ]        # // je celociselne delenie
    else:
        X = [1]                 # [1] je postupnost ktora urcite 3-particiu nema
    tri_particia(X)
```

**Popis riešenia.** Nech sme na vstupe dostali postupnosť $A = (a_1, \ldots, a_n)$. Spočítame si jej súčet $s$.

Ak je $s$ párne, pridáme na koniec vstupnej postupnosti ešte jeden prvok s hodnotou $s/2$. Výslednú postupnosť pošleme do KSP. Ten nám odpovie, či má táto nová postupnosť 3-partíciu. Lenže táto nová postupnosť má 3-partíciu vtedy a len vtedy, keď mala naša pôvodná postupnosť 2-partíciu. KSP teda za nás práve vyriešil našu úlohu.

(Prečo platí vyššie spomenutá ekvivalencia? V novej postupnosti $X$ je súčet všetkých prvkov rovný $3s/2$. Ak teda existuje 3-partícia $X$, tak každá zo skupín, na ktoré $X$ rozdelíme, má súčet presne $s/2$. Ale potom zjavne jednu z týchto troch skupín tvorí samotný nami pridaný prvok s hodnotou $s/2$. A teda 3-partícia našej novej postupnosti existuje práve vtedy, keď sa dá ostatné prvky rozdeliť na dve skupiny s rovnakým súčtom – teda práve vtedy, keď pôvodná postupnosť $A$ mala 2-partíciu.)

No a ak je $s$ nepárne, vieme, že je odpoveď *nie*. Do KSP preto chceme poslať ľubovoľný vstup, pre ktorý vopred vieme, že KSP dá zápornú odpoveď. Takýmto vstupom je napr. $X = (1)$ alebo $X = (1, 1, 2)$.

Časová zložitosť tohto programu je lineárna od dĺžky postupnosti $X$.

# Zadania domáceho kola kategórie B

## B-I-1 Klince v doske

Kleofáš má dlhú hrubú latu. A do tej laty má (do rôznej hĺbky) zatlčených $n$ klincov, očíslovaných od 1 po $n$. Dĺžku tej časti klinca $i$, ktorá ešte trčí von z laty, označíme $h_i$.

Kleofáš má kladivo. Nemá ale ani kliešte, ani žiadne ďalšie klince. Preto jediné, čo teraz vie robiť, je zatĺcť nejaké klince *hlbšie* do laty (zmenšiť ich $h_i$). Kleofáš je majster v narábaní s kladivom: keď si povie ľubovoľnú novú hodnotu $h_i$ (menšiu ako súčasná), vie ju vždy jediným úderom kladiva presne dosiahnuť.

Onedlho príde Kleofáša navštíviť Marienka. Kleofáš by jej chcel ukázať latu s klincami. Tá by bola najkrajšia vtedy, ak by všetky klince z nej trčali presne rovnako. Na to ale nie je čas, Marienka už je takmer za dverami!

**Súťažná úloha:**

Na vstupe dostanete číslo $n$, čislo $u$ a postupnosť $h_1, \ldots, h_n$ výšiek klincov. Číslo $u$ udáva, že Kleofáš už má čas len na nanajvýš $u$ úderov kladivom. Nájdite najväčšie $m$ také, že je možné, aby Kleofáš pobúchal po klincoch tak, že $m$ z nich bude z laty trčať rovnako. Tiež nájdite (jednu možnú) spoločnú výšku $h$ ktorú bude mať po úprave laty týchto $m$ klincov. (Môže byť aj $h = 0$. Klince zarovnané na výšku $h$ nemusia susediť.)

**Formát vstupu a výstupu:**

Dostanete od nás 5 vstupných súborov, označených `1.txt` až `5.txt`. Každý z nich obsahuje 6 testovacích vstupov. Každý testovací vstup je tvorený dvomi riadkami. V prvom riadku sú čísla $n$ a $u$, pričom $0 \leq u \leq n$. V druhom riadku sú čísla $h_1, \ldots, h_n$, pričom $0 \leq h_i \leq 10^9$.

V jednotlivých vstupných súboroch platí $n \leq 10$, $n \leq 1000$, $n \leq 20\,000$, $n \leq 250\,000$ a $n \leq 1\,000\,000$.

V súboroch číslo 1, 2 a 4 navyše platí, že všetky výšky $h_i$ sú z rozsahu $0 \leq h_i \leq 10^6$.

Pre každý testovací vstup vypíšte do príslušného výstupného súboru jeden riadok s dvomi číslami: optimálnym počtom zarovnaných klincov $m$ a ich optimálnou výslednou výškou $h$. (Ak existuje viac možností pre $h$, vypíšte ľubovoľnú celočíselnú.) Správny výstupný súbor má teda obsahovať presne 6 riadkov.

**Príklad:**

| Vstup | Výstup |
|---|---|
| 4 2<br>23 20 10 47<br>7 1<br>10 9 10 9 10 9 10<br>... (4 ďalšie vstupy) ... | 3 10<br>4 10<br>... (4 ďalšie výstupy) ... |

*(V prvom príklade môže Kleofáš napr. klinec 1 a klinec 4 zarovnať na výšku 10. V druhom príklade je optimálne nič nerobiť. Iný správny výstup pre druhý príklad by bol „4 9“. Ten zodpovedá tomu, že Kleofáš jeden z klincov zníži z 10 na 9, čím dostane štyri klince trčiace po 9.)*

## B-I-2 Prvočísla z magnetiek

Malá Paulínka má chladničku. Presnejšie, Paulínkina mama má chladničku. Paulínka má magnetky, ktoré na tú chladničku rada lepí. Na každej z magnetiek je jedna cifra (od 0 po 9).

Donedávna si Paulínka z cifier skladala čísla len tak, ako ju napadlo. No nedávno sa dozvedela o tom, že existujú prvočísla. Rada by si nejaké na chladničke zložila. Ešte však nevie, ako spoznať, či je nejaké číslo prvočíslo. Pomôžte jej!

**Súťažná úloha:**

My vám povieme, aké cifry má Paulínka na magnetkách. Vašou úlohou je nájsť *všetky možné* prvočísla, ktoré sa dajú zložiť, ak použijeme *úplne všetky* cifry. (Cifru 0 nie je dovolené použiť na začiatku čísla.)

Prvočíslo je prirodzené číslo, ktoré má medzi prirodzenými číslami práve dva rôzne delitele. Najmenšie prvočísla sú teda 2, 3, 5, 7, 11, 13, atď.

**Formát vstupu a výstupu:**

Zaujíma nás 5 rôznych sád cifier:

- 3 4 5 7
- 1 1 3 4 7
- 0 1 2 7 7 8 9
- 0 1 2 2 3 5 7 8 9

- 1 2 2 4 4 4 4 7 7 7 7

Každú sadu cifier vám dáme aj v súbore. Ten má v prvom riadku počet cifier a v druhom riadku ich zoznam.

Za správne riešenie pre každú sadu cifier sú 2 body. Správnym riešením je súbor, ktorý obsahuje zoznam čísel, ktorý má nasledovné vlastnosti:

- každé číslo v zozname je prvočíslom ktoré sa dá vyrobiť z daných cifier
- každé prvočíslo, ktoré sa dá vyrobiť z daných cifier, je v zozname práve raz
- zoznam je usporiadaný v rastúcom poradí

**Príklad:**

**Vstup**

```
4
0 1 1 8
```

**Výstup**

```
1801
8011
8101
```

*(Všimnite si, že každé z čísel 1801, 8011 a 8101 je vo výstupe len raz a že čísla 0181 a 0811 vo výstupe nie sú.)*

**Dobrá rada:**
Čísla, ktoré sa dajú vyrobiť z piatej sady cifier, sú už pomerne veľké. V niektorých programovacích jazykoch si treba dať pozor na to, aby sa vám zmestili do číselnej premennej. Napr. v jazyku C++ preto odporúčame používať typ `long long`, v Jave typ `long` a vo FreePascale typ `int64`.

## B-I-3 Pády domín

Janko raz ležal v nemocnici a strašne sa nudil. Keď to videla Peťka, doniesla mu sáčok plný rôznych domín. Janko sa veľmi potešil a hneď ich začal ukladať do rady a potom do nich strkal aby popadali. Dominá majú však rôzne váhy, preto nie vždy popadajú všetky. Presnejšie, $i$-te domino v rade má váhu $v_i$.

Janko môže strčiť ľubovoľné domino buď doprava alebo doľava. Ak strčí domino s váhou $v$, dominá padajú, až kým nenarazia na domino s váhou väčšou ako $v$. Vtedy sa pád zastaví.

Predstavme si, že Janko má rad domín s váhami: 2 4 3 8 1 3 9. Ak postrčí štvrté domino (s váhou 8) doprava, toto domino zhodí dve dominá – 1, 3 a zastaví sa pri narazení do 9. Ak strčí štvrté domino doľava, zhodí tri dominá, kým sa nedostane na koniec radu, kde sa pád zastaví.

Samozrejme sa Jankovi táto deštrukcia veľmi páči a páči sa mu tým viac, čím viac domín pri tom popadá. Pomôžte Jankovi zistiť, do ktorej strany má strčiť $i$-te domino, aby popadalo čo najviac domín.

**Súťažná úloha:**

Dostanete popis radu domín, ktorý Janko postavil – pre každé domino jeho hmotnosť. Pre každé $i$ zistite: Ak strčím domino $i$ ako prvé, do ktorej strany ho mám strčiť, aby popadalo viac domín.

**Formát vstupu a výstupu:**

Na vstupe dostanete popis radu domín. Na prvom riadku bude číslo $n$ – počet domín v rade. Na druhom riadku bude $n$ čísiel $v_i$ – váhy domín v rade.

Na výstup vypíšte jeden riadok, ktorý obsahuje $n$ znakov: pre každé domino jeden. Ak je lepšie $i$-te domino strčiť doľava, má byť $i$-ty znak „<“. Ak je lepšie strčiť ho doprava, má byť $i$-ty znak „>“. A ak sú obe možnosti rovnako dobré, vypíšte „=“.

**Hodnotenie:**

Za riešenie podobne efektívne ako vzorové môžete získať plných 10 bodov. Za ľubovoľné správne riešenie získate od 3 po 6 bodov, v závislosti od časovej zložitosti daného riešenia.

**Príklad:**

| Vstup | Výstup |
|---|---|
| 7<br>2 4 3 8 1 3 9 | ===<=<< |

*Ak domino 4 (s váhou 8) zhodíme doľava, padnú dokopy 4 dominá. Ale ak ho zhodíme doprava, padnú len 3. Preto je 4. znak výstupu* <.

## B-I-4 Hľadáme v poli

Keď v počítači spracúvame dáta, často v nich potrebujeme vyhľadávať. V tejto úlohe sa budeme zaoberať jednou z najjednoduchších situácií, kedy v dá-

tach vyhľadávame: Máme *neprázdne* pole $A$ *navzájom rôznych* celých čísel, ktoré sú *usporiadané podľa veľkosti* v rastúcom poradí. A ďalej máme dané celé číslo $x$, o ktorom nás zaujíma, či sa v poli $A$ nachádza, a ak áno, *na ktorom indexe leží.*

Túto úlohu sa samozrejme dá riešiť hrubou silou: Postupne porovnáme $x$ s každým prvkom poľa $A$. Ak niekedy nastane rovnosť, vrátime na výstup zodpovedajúci index, ak nikdy rovnosť nenastane, podáme správu, že $x$ sa v poli $A$ nenachádza. Existujú však aj omnoho efektívnejšie spôsoby riešenia tejto úlohy.

Vy túto úlohu však riešiť nebudete. Pre vás sme si pripravili niečo úplne iné. Trom študentom sme zadali nasledovnú úlohu: „Naprogramujte čo najefektívnejšiu funkciu, ktorá na vstupe dostane pole $A$ a číslo $x$ spĺňajúce vyššie popísané podmienky a na výstupe vráti buď číslo $-1$, ak sa $x$ v poli $A$ nenachádza, alebo index $i$ taký, že $A[i] = x$.“ Každý z našich troch študentov stvoril nejaké riešenie. No a vašou úlohou bude teraz tieto tri riešenia skontrolovať a zistiť, či fungujú.

Ak nejaké riešenie nefunguje, stačí nájsť jeden konkrétny protipríklad: teda jedno pole $A$ a číslo $x$, pre ktoré naprogramovaná funkcia nespraví to, čo sme chceli. Pole $A$ vo vašom protipríklade nesmie mať viac ako 16 prvkov a tieto prvky musia byť z rozsahu od 0 po 1 000 000. Ak nejaké riešenie funguje, zdôvodnite, prečo to tak je, a odhadnite jeho časovú zložitosť. Pozor, hodnotiť máte funkčnosť konkrétnej implementácie, nie len správnosť hlavnej myšlienky riešenia.

V nasledujúcom texte nájdete tri spomínané programy v jazyku Pascal. Na webe OI na adrese `http://oi.sk/archiv/2013/b14.html` si môžete v prípade potreby pozrieť ekvivalentné programy v C++ a v Pythone.

Andrej vám k svojmu programu zanechal nasledovný popis:

Pozriem sa na prvý a na posledný prvok. Z nich si spočítam, kde vychádza, že bude $x$. Napríklad ak je prvý prvok 100, posledný prvok 1000 a ja hľadám 400, tak bude asi niekde v tretine poľa.

Presnejšie to funguje takto: nech práve hľadám v poli na pozíciách $p$ až $q$. Pozrieme sa na hodnoty $A[p]$ a $A[q]$. Teraz si predstavme, že všetky prvky na pozíciách $p$ až $q$ tvoria aritmetickú postupnosť. Aké by mali hodnoty? Označme $d = (A[q] - A[p])/(q - p)$. Potom tieto prvky majú hodnoty $A[p]$, $A[p] + d$, $A[p] + 2d$, a tak ďalej. A my teraz hľadáme nejaké $x$, hej? No tak si nájdeme také $i$, aby $A[p] + id$ bolo čo najbližšie ku $x$, a budeme ho hľadať tam.

Jasne že sa nemusím vždy na prvý krát trafiť presne. Ale aj keď sa netrafím presne, tak aspoň viem nejaké prvky zahodiť a ďalej hľadať v kratšom úseku poľa. A keď to celé pre istotu zopakujem 10-krát tak už to určite sadne.

## Listing programu (Pascal)

```
{ hľadáme hodnotu X v poli A[0..N-1],
  o ktorom predpokladáme, že je usporiadané vzostupne }

function andrej_hlada(var A : array of longint; N : longint; X : longint) : longint;

var kolo, odpoved, prvy, posledny, kde : longint;

begin
    prvy := 0; posledny := N-1;
    odpoved := -2;
    for kolo := 1 to 10 do begin
        if (X < A[prvy]) then odpoved := -1;
        if (X = A[prvy]) then odpoved := prvy;
        if (X = A[posledny]) then odpoved := posledny;
        if (X > A[posledny]) then odpoved := -1;
        if (odpoved = -2) and (prvy + 1 >= posledny) then odpoved := -1;
        if (odpoved <> -2) then break;

        kde := prvy + round((posledny-prvy) * (X-A[prvy]) / (A[posledny]-A[prvy]));
        if (kde < prvy+1) then kde := prvy+1;
        if (kde > posledny-1) then kde := posledny-1;

        if (X < A[kde]) then posledny := kde;
        if (X = A[kde]) then begin odpoved := kde; break; end;
        if (X > A[kde]) then prvy := kde;
    end;
    if (odpoved = -2) then odpoved := -1;
    andrej_hlada := odpoved;
end;
```

Boris len naškrabal na kus papiera stručný odkaz: „To je ľahké! Použijem binárne vyhľadávanie. Kým mám viac možností, tak sa vždy pozriem do stredu a podľa toho viem, či hľadať $x$ vľavo alebo vpravo. A keď mi už ostane len jedna možnosť, tak pozriem, či je to presne to čo hľadám alebo nie.“

## Listing programu (Pascal)

```
function boris_hlada(var A : array of longint; N : longint; X : longint) : longint;

var prvy, posledny, stredny : longint;

begin
    prvy := 0; posledny := N-1;
    while prvy < posledny do begin
        stredny := (prvy + posledny) div 2;
        if (A[stredny] <= X) then prvy := stredny else posledny := stredny;
    end;
    if (A[prvy] = X) then boris_hlada := prvy else boris_hlada := -1;
end;
```

Cilka vám ku svojmu programu nenapísala vôbec nič.

## Listing programu (Pascal)

```
function cilka_hlada(var A : array of longint; N : longint; X : longint) : longint;

var kde, krok : longint;

begin
    cilka_hlada := -1;
    if (X < A[0]) then exit;

    krok := 0;
    while krok*krok < N do inc(krok);
    kde := 0;
    while (kde+krok<N) and (A[kde+krok] <= X) do kde := kde+krok;
    krok := 1;
    while (kde+krok<N) and (A[kde+krok] <= X) do kde := kde+krok;
    if (A[kde] = X) then cilka_hlada := kde;
end;
```

# Zadania krajského kola kategórie A

## A-II-1 Fľaštičková úloha

*Medzičasom v paralelnom vesmíre.*

Jano Hrnčiar vstúpil do ďalšej miestnosti – pred sebou videl dvere, ktoré ho povedú ďalej, za sebou mal dvere, ktorými vošiel. V okamihu oboje dvere zachvátili čierne plamene a Jano tam zostal uväznený. Nezostávalo mu teda nič iné, ako prejsť ku stolu v strede miestnosti. Na ňom stál rad roztodivných fľaštičiek a vedľa neho ležal kus pergamenu. Jano zdvihol pergamen a pozorne si ho prečítal:

*„Ak chceš prejsť cez čierne plamene, musíš sa napiť z mojich fľaštičiek. Môžeš ich vypiť koľko len chceš, avšak fľaštičky, ktoré vypiješ, musia tvoriť* ***súvislý úsek****. Každá fľaštička má na obale napísané, akú veľkú odolnosť proti plameňom ti poskytne. Ak ich vypiješ viacero, tvoja odolnosť bude* ***priemerom*** *odolností, ktoré ti dávajú vypité fľaštičky. Na prechod plameňmi potrebuješ mať odolnosť* ***presne k****. A aby to nebolo také ľahké, musíš vypiť* ***najväčší možný počet*** *fľaštičiek. Splň túto neľahkú úlohu alebo zhyň.“*

Jano si povzdychol. Takže teraz je to logická hádanka… Čo však s ňou? Peťa by si vedela rady, bohužiaľ pohrýzol ju had pri hre Hady a rebríky. A Žaba, na toho zase padol rebrík. Sám si však Jano neporadí. Nemohli by ste mu pomôcť vy?

**Súťažná úloha:**

Dostanete číslo $k$ a postupnosť $n$ čísel. Nájdite najdlhšiu súvislú podpostupnosť týchto čísel, ktorej priemer je presne rovný $k$.

**Formát vstupu:**

V prvom riadku dostanete čísla $n$ a $k$. V druhom riadku je $n$ kladných celých čísel: prvky postupnosti.

Môžete predpokladať, že vstup obsahuje aspoň jednu súvislú podpostupnosť s priemerom prvkov presne rovným $k$. Tiež môžete pri písaní programu predpokladať, že sa vám súčet celej postupnosti pohodlne zmestí do bežnej číselnej premennej.

**Formát výstupu:**

Na výstup vypíšte dve celé čísla – pozíciu začiatku a pozíciu konca najdlhšej vhodnej podpostupnosti. (Pozície číslujeme od 1 po $n$.) Ak je najdlhších postupností viac, vyberte si ľubovoľnú jednu z nich.

**Hodnotenie:**

Plných 10 bodov môže dostať riešenie, ktoré efektívne vyrieši ľubovoľný vstup s $n \leq 200\,000$.

Najviac 6 bodov dostane riešenie, ktoré efektívne vyrieši ľubovoľný vstup s $n \leq 5\,000$.

Za ľubovoľné funkčné riešenie sa dajú získať 4 body.

**Príklady:**

**Vstup**

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

**Výstup**

```
4 7
```

*Existujú tri podpostupnosti s priemerom presne 4, a to* $(1, 3, 8)$, $(3, 8, 1)$ *a* $(8, 1, 5, 2)$. *Tretia je najdlhšia, vypíšeme teda jej index začiatku a konca.*

**Vstup**

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

**Výstup**

```
3 3
```

## A-II-2 Nájazd na jablone

Vďaka vašim riešeniam domáceho kola boli hrušky úspešne oplotené. Dedinčanom sa to nepáči, a tak naplánovali pomstu. Keď už sa nedostanú k hruškám, pripravia babičku aspoň o jablká. Susedka Mária, cvičiteľka sysľov, každú štvrťhodinu vystrelí ponad záhradu starú raketu a v nej cvičeného sysľa s padákom. Raketa sa niekde vo vzduchu rozpadne, syseľ dopadne babke do záhrady a poďho k jabloniam.

Babka sa našťastie o pláne dozvedela a našla si jednu dlhočiznú latu. Tú chce pohodiť medzi sysľa a jablone. Ak takto zamedzí sysľovi prístup ku všetkým jabloniam, ten zmätený zastane a babka ho ľahko zneškodní.

**Súťažná úloha:**

V babkinej záhrade je $n$ jabloní, tie budeme považovať za body v rovine. Postupne sa zjaví $q$ sysľov. Pre každého sysľa je vašou úlohou zistiť, či existuje spôsob, ako vhodne pohodiť latu medzi neho a stromy. Formálne, vašou úlohou je zistiť, či existuje priamka taká, že všetky stromy ležia na priamke alebo na jednej jej strane, zatiaľ čo bod dopadu sysľa leží ostro na druhej strane priamky. Ak áno, treba jednu takú priamku aj nájsť.

Pri písaní svojho riešenia môžete pre jednoduchosť predpokladať, že všetky výpočty a porovnávania sú presné (t.j. nenastávajú zaokrúhľovacie chyby).

**Formát vstupu:**

V prvom riadku je kladné celé číslo $n \geq 3$, označujúce počet stromov. V každom z ďalších $n$ riadkov sa nachádzajú súradnice jedného stromu. Ďalej nasleduje číslo $q$, počet útočiacich sysľov. V každom z nasledujúcich $q$ riadkov sú súradnice dopadu jedného zo sysľov. Môžete predpokladať, že žiadne tri stromy neležia na priamke a že žiaden sysel nedopadne na strom.

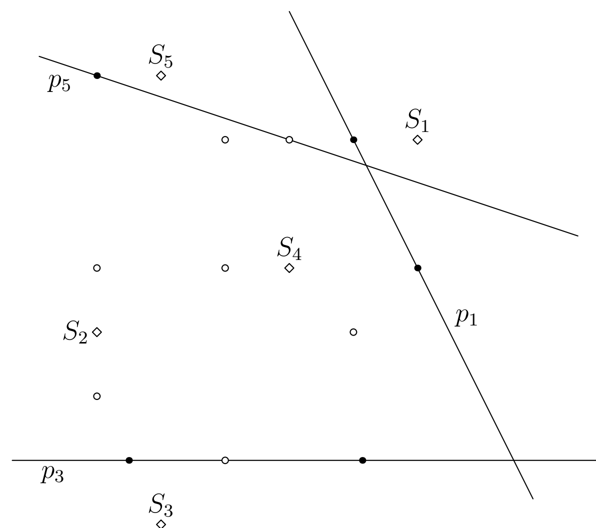
**Príklad:**

**Vstup**

```
7
-1 -1
2 3
1 -2
1 1
3 0
1 3
-1 1
5
4 3
-1 0
0 -3
2 1
0 4
```

výstup

```
4 1 3 3
Sysel vitazi!
-0.5 -2 3.14 -2
Sysel vitazi!
2 3 -1 4
```




*Biele krúžky označujú stromy, diamanty sú jednotlivé sysle. Riešenie z príkladu výstupu znázorňujú priamky a čierne krúžky (vypísané body).*

**Formát výstupu:**

Vypíšte $q$ riadkov, v každom z nich buď text „`Sysel vitazi!`“, ak neexistuje priamka vyhovujúca zadaniu, alebo 4 reálne čísla $x_1, y_1, x_2, y_2$: súradnice dvoch rôznych bodov na jednej z vyhovujúcich priamok.

**Hodnotenie:**

Plných 10 bodov môže dostať riešenie, ktoré efektívne vyrieši ľubovoľný vstup s $n \leq 100\,000$, $q \leq 100\,000$. Iné riešenia s polynomiálnou časovou zložitosťou dostanú 5-7 bodov podľa efektívnosti. Ak vaše riešenie správne zistí, či syseľ víťazí, ale nenájde správnu priamku, strhneme mu 1-2 body.

## A-II-3 Tajná misia

Špióna Jamesa Bonda poslali do ruskej základne v Jekaterinburgu na tajnú misiu. Má infiltrovať ruskú armádu a zničiť ich počítačovú sieť.

**Súťažná úloha:**

Počítačová sieť sa skladá z $n$ počítačov, ktoré sú spojené presne $n$ káblami – teda káblov je rovnako ako počítačov. Každé dva počítače vedia pomocou týchto káblov medzi sebou komunikovať (možno nie priamo, ale cez niekoľko iných počítačov). Žiadne dva počítače nie sú spojené viac ako jedným káblom.

James Bond chce prestrihnúť čo najviac káblov. Avšak akonáhle prestrihne nejaký kábel, aktivuje tým ochranné mechanizmy oboch počítačov, ktoré tento kábel spája. Ak by sa následne pokúsil prestrihnúť iný kábel vedúci do niektorého z týchto počítačov, určite by ho chytili.

Zistite, koľko najviac káblov môže James Bond postupne prestrihnúť.

**Formát vstupu:**

V prvom riadku je celé číslo $n$ ($n \geq 3$), označujúce aj počet počítačov, aj počet káblov. Počítače majú čísla 1 až $n$. V každom z ďalších $n$ riadkov sú dve čísla $a$ a $b$ ($1 \leq a, b \leq n$; $a \neq b$), označujúce, že počítače $a$ a $b$ sú spojené káblom.

**Formát výstupu:**

Vypíšte jeden riadok a v ňom jedno celé číslo: najväčší možný počet prestrihnutých káblov.

**Hodnotenie:**

Plných 10 bodov môže dostať riešenie, ktoré efektívne vyrieši ľubovoľný vstup s $n \leq 1\,000\,000$. Do 7 bodov dostane riešenie, ktoré efektívne vyrieši

ľubovoľný vstup s $n \leq 100$. Za ľubovoľné funkčné riešenie sa dá získať 4 body.

**Príklady:**

**Vstup**

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

**Výstup**

```
2
```

*James Bond môže napr. prestrihnúť káble 2-6 a 4-5.*

**Vstup**

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

**Výstup**

```
3
```

*Tentoraz sa dá prestrihnúť káble 1-5, 4-3 a 2-6.*

## A-II-4 Mimozemské počítače

*K tejto úlohe patrí aj zoznam problémov, uvedený na nasledujúcich stranách, a študijný text z domáceho kola, ktorý nájdete na strane 12 tejto ročenky. V zozname problémov nájdete dva nové problémy:* ***kostru-húsenicu*** *a* ***pokrytie podmnožinami****.*

*Jednotlivé podúlohy môžete ich riešiť v ľubovoľnom poradí, každá je hodnotená samostatne. Na časovej zložitosti vašich algoritmov pri hodnotení nezáleží – len musí byť polynomiálna.*

**Podúloha A (3 body):**

Mimozemšťania nám dodali sálový KSP, ktorý rozhoduje problém existencie kostry-húsenice v danom jednoduchom neorientovanom grafe.

Tento KSP má teda funkciu `husenica(n,E)`. Táto funkcia čaká ako parametre počet $n$ vrcholov grafu a zoznam $E$ jeho hrán. Ak v danom grafe existuje kostra-

húsenica, KSP rozsvieti zelené svetlo, inak rozsvieti červené. Túto funkciu môžete použiť **len raz** a jej volaním výpočet vášho programu končí.

Na vstupe dostanete jednoduchý neorientovaný graf $G$. Napíšte program s polynomiálnou časovou zložitosťou, ktorý rozsvieti zelené alebo červené svetlo podľa toho, či $G$ obsahuje aspoň jednu Hamiltonovskú *cestu*.

**Podúloha B (3 body):**

Na vstupe opäť dostanete jednoduchý neorientovaný graf $G$. Použijúc ten istý sálový KSP ako v podúlohe A, napíšte program s polynomiálnou časovou zložitosťou, ktorý rozsvieti zelené alebo červené svetlo podľa toho, či $G$ obsahuje aspoň jednu Hamiltonovskú *kružnicu*.

**Podúloha C (4 body):**

Mimozemšťania nám dodali kufríkový KSP, ktorý rozhoduje problém existencie dostatočne dobrého pokrytia podmnožinami.

Tento KSP má teda funkciu `existuje_pokrytie(k,n,m,Z)`. Táto funkcia má nasledujúce parametre: $k$ je nejaké nezáporné celé číslo, $n$ udáva počet prvkov množiny, ktorú chceme celú pokryť, $m$ udáva počet podmnožín, ktoré máme na výber, a $Z$ je zoznam týchto podmnožín (teda napr. dvojrozmerné pole, ktorého každý riadok popisuje jednu podmnožinu).

Na výstupe táto funkcia vracia `True` alebo `False` podľa toho, či sa spomedzi zadaných $m$ podmnožín dá vybrať nanajvýš $k$ tak, aby ich zjednotenie malo všetkých $n$ prvkov.

Túto funkciu môžeme v našom programe volať ľubovoľne veľakrát pre ľubovoľné vstupy.

Uvažujme nasledovný problém: V škole je $z$ žiakov (očíslovaných 0 až $z-1$) a škola pre nich organizuje dokopy $k$ krúžkov. Pre každý krúžok je daný neprázdny zoznam žiakov, ktorí ho navštevujú. Každý krúžok si spomedzi žiakov, ktorí ho navštevujú, zvolil svojho predsedu. Jeden žiak pritom mohol byť zvolený predsedom viacerých krúžkov. Radu predsedov tvoria predsedovia všetkých krúžkov. Koľko najmenej členov môže mať táto rada?

Vyriešte tento problém pomocou vyššie popísaného kufríkového KSP. Počet bodov, ktoré získate, bude závisieť od počtu použití funkcie `existuje_pokrytie`. (Menej je samozrejme lepšie.)

**Programovacie jazyky:**

Vo svojich riešeniach môžete používať ľubovoľný štruktúrovaný programovací jazyk. Vhodne si zvoľte potrebné dátové štruktúry. Napr. v Pascale by

funkcia z podúlohy A mohla vyzerať nasledovne:

```
type hrana = array[0..1] of longint;
procedure husenica(n : longint; m : longint; E : array of hrana);
```

a v C++ môžeme použiť buď nízkoúrovňové polia, alebo aj vektor dvojíc:

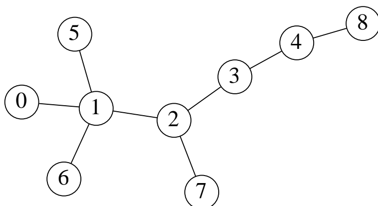
```
void husenica(int n, int m, int E[][2]);          // prva moznost
void husenica(int n, vector< pair<int,int> > E);  // druha moznost
```

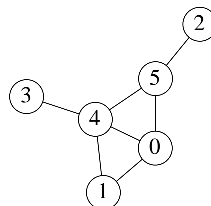
## Študijný text: zoznam problémov (2)

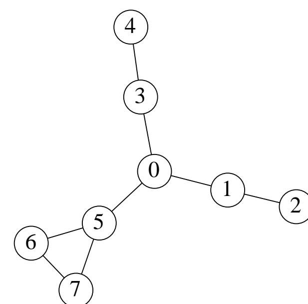
Jednoduchý neorientovaný graf je usporiadaná dvojica $(V, E)$. $V$ je konečná množina objektov nazývaných vrcholy. $E$ je konečná množina dvojíc vrcholov; jej prvky voláme hrany. Takýto graf si môžeme predstaviť napríklad ako cestnú sieť: $V$ je množina miest a $E$ je množina dvojíc miest spojených cestami. Počet vrcholov budeme označovať $n$, počet hrán bude $m$. Jednotlivé vrcholy budeme číslovať od 0 po $n - 1$.

---

**Kostra-húsenica:**
Húsenica je graf, ktorý vieme vyrobiť tak, že zoberieme nejakú cestu (tvorenú aspoň dvomi vrcholmi) a následne pridáme niekoľko nových vrcholov a každý z nich pripojíme hranou ku jednému vrcholu cesty. Inými slovami, ak máme $n$-vrcholový graf, tak je to húsenica vtedy a len vtedy, ak má presne $n - 1$ hrán a vieme nájsť nejakú cestu takú, že každý vrchol, ktorý nie je na tej ceste, má jediného suseda a ten leží na ceste.
Problém kostry-húsenice je potom definovaný nasledovne: Daný je ľubovoľný neorientovaný $n$-vrcholový graf. Dá sa zmazať niektoré jeho hrany tak, aby sme dostali $n$-vrcholovú húsenicu?








*Graf vľavo je húsenica, jednu možnú cestu tvoria vrcholy 0, 1, 2, 3, 4.*
*Graf v strede obsahuje húsenicu.*
*Aby sme jednu vyrobili, stačí zmazať napr. hrany 1–4 a 4–5.*
*Z grafu vpravo sa len mazaním hrán húsenica nijak vyrobiť nedá.*

**Pokrytie množinami:**
Dané sú čísla $n$ a $k$ a sada množín. Každá množina je podmnožinou množiny $\{0, 1, 2, \ldots, n-1\}$. Úlohou je zistiť, či sa z danej sady množín dá vybrať nanajvýš $k$ tak, aby ich zjednotením bola celá množina $\{0, 1, 2, \ldots, n-1\}$.
*Pre $n = 5$, $k = 2$ a množiny $\{1, 3, 4\}$, $\{0, 3\}$, $\{0, 1\}$ a $\{0, 1, 2, 4\}$ je odpoveď „áno“: môžeme napr. zobrať druhú a štvrtú množinu.*
*Pre $n = 5$, $k = 2$ a množiny $\{0, 1, 3\}$, $\{4\}$, $\{0, 1\}$ a $\{0, 1, 2\}$ je odpoveď „nie“.*
*Pre $n = 5$, $k = 4$ a množiny $\{0, 1, 3\}$, $\{4\}$, $\{0, 1\}$ a $\{0, 1, 2\}$ je odpoveď „áno“: môžeme napr. zobrať prvú, druhú a štvrtú množinu (alebo všetky štyri).*

# Zadania krajského kola kategórie B

## B-II-1 Lokálne ochladzovanie

Odkedy sa Frico presťahoval na Island za štúdiom fyziky, prešlo už mnoho rokov. Medzičasom sa stal doktorom, naučil sa plynule po islandsky a vyskúšal všetky termálne kúpaliská v krajine. Teraz však chce zažiť niečo nové – zaplávať si v Severnom ľadovom oceáne.

Frico by si rád výlet naplánoval na niekoľko po sebe idúcich dní. Pretože by toho za jediný deň veľa nestihol, musí celý výlet trvať **aspoň dva dni**. No a keďže Frico nie je nijaký otužilec, priemerná teplota počas jeho výletu nesmie byť nižšia ako $k$.

Teraz však sedí nad predpoveďou počasia a nevie si rady: Existuje vôbec vhodný termín výletu?

**Súťažná úloha:**

Daná je hodnota $k$ a teploty počas nasledujúcich $n$ dní: $t_1, t_2, \ldots, t_n$. Nájdite nejaký súvislý úsek dní dĺžky aspoň dva, ktorého priemerná teplota je aspoň $k$ (alebo rozhodnite, že taký úsek neexistuje).

**Formát vstupu:**

V prvom riadku vstupu sú dve celé čísla – počet dní $n$ ($n \geq 2$) a obmedzenie na priemernú teplotu $k$ ($-10^9 \leq k \leq 10^9$).

Druhý riadok obsahuje $n$ celých čísel $t_1$ až $t_n$ ($-10^9 \leq t_i \leq 10^9$).

**Formát výstupu:**

Ak neexistuje termín výletu, ktorý by vyhovoval Fricovym požiadavkám, vypíšte „`nemozne`“. Inak si vyberte niektorý vyhovujúci termín a vypíšte čísla jeho prvého a posledného dňa.

**Hodnotenie:**

Plných 10 bodov môže dostať riešenie, ktoré efektívne vyrieši ľubovoľný vstup s $n \leq 1\,000\,000$. Do 6 bodov dostane pomalšie riešenie, ktoré efektívne vyrieši ľubovoľný vstup s $n \leq 1\,000$. Najviac 3 body môže dostať riešenie, ktoré efektívne rieši len vstupy s $n \leq 100$.

**Príklady:**

**Vstup**

```
5 20
-25 7 5 21 18
```

**Výstup**

```
nemozne
```

*Najlepší termín by začal v štvrtý deň a skončil v piaty. Žiaľ, jeho priemerná teplota je iba* 19.5.

**Vstup**

```
5 10
-25 7 5 21 18
```

**Výstup**

```
2 4
```

*Jedným z možných riešení sú dni 2, 3 a 4 (priemerná teplota* 11*).*

## B-II-2 Magnetické písmenká

Ferko má doma niekoľko magnetických písmeniek, ktoré s obľubou ukladá na chladničku a vytvára tak rôzne slová a nápisy. Najviac sa mu páčia také nápisy, ktoré sa čítajú rovnako odpredu aj odzadu, teda takzvané palindrómy. Napr. „kobylamamalybok“ je palindróm, „abba“ je palindróm ale „ferko“ už nie je palindróm.

Ferka by zaujímalo, či sa z jeho písmeniek (ak musí použiť úplne všetky) dá poskladať palindróm, koľko takých palindrómov existuje a ktoré to sú.

**Súťažná úloha:**

Napíšte program, ktorý bude riešiť nasledovné úlohy. Každá úloha sa hodnotí samostatne, takže body sa dajú získať aj vyriešením len jednej alebo dvoch z nich. Na vstupe dostanete zoznam Ferkovych písmeniek. Ferko ich chce pouklada do radu tak, aby použil každé práve raz a aby výsledný text bol palindróm.

a) Dá sa to? (2 body)
b) Koľkými spôsobmi sa to dá? (4 body)
c) Vypíšte všetky slová, ktoré môže Ferko vyrobiť na chladničke. (4 body)

**Formát vstupu:**

V prvom riadku vstupu je jedno celé číslo $n$, počet Ferkovych písmeniek.

V druhom riadku je reťazec zložený z $n$ malých písmen anglickej abecedy.

**Formát výstupu:**

a) Program ma vypísať „`ano`“, ak existuje palindróm zložený zo všetkých Ferkovych písmen, „`nie`“ v opačnom prípade.

b) Program má vypísať jedno číslo, počet reťazcov, ktoré sa skladajú zo všetkých Ferkovych písmen a sú to palindrómy. (Pri písaní programu môžete predpokladať, že sa vám všetky medzivýsledky zmestia do bežných celočíselných premenných.)

c) Program má vypísať všetky možné palindrómy z Ferkovych písmen. Každý palindróm má byť na samostatnom riadku. Palindrómy môžu byť vypísané v ľubovoľnom poradí, ale každý tam musí byť práve raz.

**Hodnotenie:**

a) 2 body môže dostať riešenie, ktoré efektívne vyrieši ľubovoľný vstup s $n \leq 1\,000\,000$; 1 bod môže dostať ľubovoľné funkčné riešenie.

b) 4 body môže dostať ľubovoľné riešenie, ktoré efektívne vyrieši ľubovoľný vstup s $n \leq 1\,000\,000$; 3 body sú pre ľubovoľné riešenie, ktoré zvládne $n \leq 1000$; 2 body sa dajú získať za riešenie, ktorého časová zložitosť je približne priamo úmerná výsledku; 1 bod sa dá získať za hocičo funkčné.

c) 4 body sú za riešenie, ktorého časová zložitosť je približne priamo úmerná dĺžke výstupu; najviac 2 body za pomalšie funkčné riešenia.

**Príklady:**

**Vstup**

```
5
korok
```

**Výstup**

```
a)
ano

b)
2

c)
korok
okrko
```

**Vstup**

```
8
bacabaca
```

**Výstup**

```
a)
ano

b)
12

c)
aabccbaa
aacbbcaa
abaccaba
abcaacba
acabbaca
acbaabca
baaccaab
bacaacab
bcaaaacb
caabbaac
cabaabac
cbaaaabc
```

**Vstup**

```
4
ryba
```

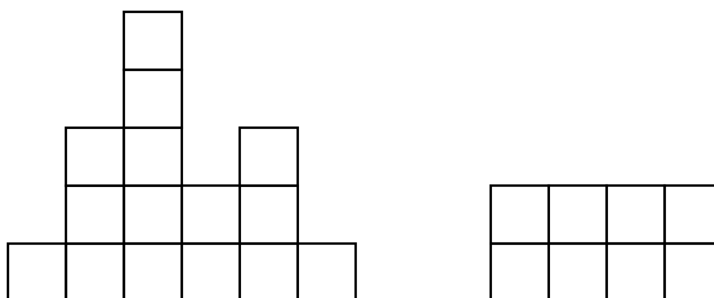
**Výstup**

```
a)
nie

b)
0

c)
```

## B-II-3 Búranie

Na Kvetinkovej ulici je rad domov, ktoré sú tesne pri sebe. Každý dom je široký 10 metrov, ale počet poschodí majú rôzny. Na každom poschodí domu je jeden byt. Príklad toho ako môže vyzerať Kvetinková ulica nájdete na obrázku vľavo.

Starostovi sa ale vzhľad Kvetinkovej ulice nepáči a rozhodol sa, že niektorým domom zbúra zhora niektoré poschodia tak, aby všetky domy na Kvetinkovej ulici, ktoré zostanú stáť, boli pri sebe a mali rovnakú výšku. Zároveň, ale starosta chce, aby sa zachovalo čo najviac bytov. Príklad takého zbúrania nájdete na obrázku vpravo.



**Súťažná úloha:**

Pre zadaný popis ulice napíšte program, ktorý určí, ktoré domy úplne zbúrať a ktorým zbúrať len nejaké poschodia tak, aby ponenchané domy boli pre sebe a mali rovnakú výšku a aby počet zachovaných bytov bol čo najväčší.

**Formát vstupu a výstupu:**

V prvom riadku vstupu je jedno celé číslo $n$, počet domov na Kvetinkovej ulici. V druhom riadku je $n$ celých čísel – výšky jednotlivých domov v poradí zľava doprava. Môžete predpokladať, že výška každého domu je kladné celé číslo menšie ako $2 \cdot 10^9$.

Vypíšte $n$ čísel. Pre každý dom vypíšte počet zachovaných poschodí.

**Hodnotenie:**

- 10 bodov dostane riešenie, ktoré efektívne vyrieši vstupy s $n \leq 1\,000\,000$.
- 7 bodov dostane riešenie, ktoré efektívne vyrieši vstupy s $n \leq 5\,000$.
- 4 body dostane riešenie, ktoré efektívne vyrieši vstupy s $n \leq 500$.
- 2 body dostane ľubovoľné korektné riešenie.

Pri riešení sa môžete odvolávať na vzorové riešenia úloh z domáceho kola.

**Príklad:**

**Vstup**

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

**Výstup**

```
0 2 2 2 2 0
```

## B-I-4 Kino

Po tom, ako sa naši študenti dozvedeli, že ich programy na vyhľadávanie nefungovali úplne najlepšie, boli smutní a začali sa sťažovať. Vraj vymýšľať postupy je ťažké, ale povedať, či je program dobrý, je už ľahšie. Tak sme im zadali nie len úlohu, ale a aj štyri možné riešenia s popismi. Poradíte si s nimi aj vy?

**Zadanie úlohy:**

V kine hrajú $n$ rôznych filmov. Je nám jedno, na aké filmy pôjdeme, ale radi by sme ich videli čo najviac.

Filmy očíslujeme od 1 po $n$, čas začiatku $i$-teho filmu označíme $z_i$ a čas jeho konca $k_i$. Ak chceme film vidieť, musíme byť po celý čas v správnej sále, a to vrátane okamihov začiatku a konca. Film si teda môžeme predstaviť ako uzavretý interval $[z_i, k_i]$ na časovej osi. Napíšte program, načíta číslo $n$ a všetky čísla $z_i$ a $k_i$ a následne zistí, koľko najviac filmov vieme vybrať tak, aby žiadne dva nemali neprázdny prienik.

Príklad: Ak by filmom zodpovedali intervaly $[1, 3]$, $[3, 5]$, $[4, 7]$ a $[5, 6]$, tak vieme stihnúť najviac 2 filmy. (Môžeme ísť napríklad na prvý a tretí, alebo na prvý a štvrtý film. Všimnite si, že sa nedá stihnúť prvý a druhý, keďže v čase 3 by sme museli byť na oboch filmoch.)

**Popis skúmaných riešení:**

K tejto úlohe sme vymysleli štyri možné riešenia. Vašou súťažnou úlohou bude tieto štyri riešenia skontrolovať a zistiť, ktoré z nich fungujú a ktoré nie. Zaujíma nás len korektnosť, ich časovou zložitosťou sa zaoberať nemusíte.

Ak riešenie vždy funguje, stručne zdôvodnite, prečo je to tak. Ak nejaké riešenie nefunguje, nájdite jeden konkrétny malý protipríklad: teda konkrétne $n \leq 20$ a konkrétnych $n$ časových intervalov, pre ktoré príslušný algoritmus nenájde optimálne riešenie.

Všetky riešenia budú založené na rovnakom princípe: Zoberiem si zoznam všetkých filmov. Vyberiem si nejaký film, na ktorý chcem ísť, a ten si zakrúžkujem a škrtnem všetky filmy, ktoré sa s ním prekrývajú. Vyberiem si (spomedzi nevybratých a nevyškrtnutých) ďalší film, ten si zakrúžkujem a škrtnem všetky filmy, ktoré sa s ním prekrývajú. A tak ďalej, až kým každý film nie je buď vybratý alebo vyškrtnutý. Meniť sa bude len kritérium, podľa ktorého si filmy vyberám. (V situácii, kedy je podľa zvoleného kritéria viacero filmov rovnocenných, si vždy vyberiem ten z nich, ktorý má najmenšie poradové číslo.)

**Riešenie A:** Vždy si vyberiem ten film, ktorý *začína najskôr.*
**Riešenie B:** Vždy si vyberiem ten film, ktorý *je najkratší.*
**Riešenie C:** Vždy si vyberiem ten film, ktorý *končí najskôr.*
**Riešenie D:** Vždy si vyberiem ten film, *pre ktorý by som následne vyškrtol najmenej iných.*

**Príklad výpočtu pre jednotlivé riešenia:**
Majme štyri filmy, ktorým zodpovedajú intervaly $[1,3]$, $[3,5]$, $[4,7]$ a $[5,6]$. Toto vypočítajú jednotlivé riešenia:

**Riešenie A**
– Prvý začína film 1 (v čase 1).
  Vyberiem ten a škrtnem film 2, ktorý sa s ním prekrýva.
– Spomedzi ostatných filmov (t.j. filmov 3 a 4) začína skôr film 3.
  Vyberiem ten, musím škrtnúť film 4 a končím.
**Riešenie B**
– Najkratší je film 4. Vyberiem ten a škrtnem filmy 2 a 3, s ktorými sa prekrýva.
– Ostal mi len film 1, vyberiem teda aj ten a končím.
**Riešenie C**
– Prvý končí film 1 (v čase 1).
  Vyberiem ten a škrtnem film 2, ktorý sa s ním prekrýva.
– Spomedzi ostatných filmov (t.j. filmov 3 a 4) končí skôr film 4.
  Vyberiem ten, musím škrtnúť film 3 a končím.
**Riešenie D**
– Film 1 sa prekrýva len s jedným iným, film 2 s tromi, filmy 3 a 4 každý s dvoma inými. Vyberiem teda film 1 a škrtnem film 2.
– Každý z filmov 3 a 4 sa teraz prekrýva s jedným iným filmom.
  Vyberieme teda film 3 lebo má menšie číslo.

V tomto konkrétnom prípade teda všetky štyri postupy našli niektoré optimálne riešenie.

**Príklad implementácie riešení:**

V tejto časti uvádzame konkrétny Pascalovský program, ktorý obsahuje implementáciu všetkých štyroch vyššie popísaných algoritmov. Vôbec ho nepotrebujete čítať, celá súťažná úloha sa dá pohodlne vyriešiť podľa slovných popisov algoritmov. Napriek tomu sme ho sem dali, aby ste si mohli v prípade záujmu pozrieť, ako sa dotyčné algoritmy dajú implementovať. O programe môžete predpokladať, že je korektný – chyby treba hľadať len v myšlienke jednotlivých algoritmov, nie v našej implementácii.

## Listing programu (Pascal)

```
var N : longint;
    Z, K : array of longint;
    skrtnute : array of boolean;

{ filmy a,b sa neprekrývajú <=> jeden začne neskôr ako druhý skončí }
function prekryvaju_sa(a,b : longint) : boolean;
begin prekryvaju_sa := not ( (Z[a]>K[b]) or (Z[b]>K[a]) ); end;

(* ----------- POROVNÁVACIE FUNKCIE PRE JEDNOTLIVÉ RIEŠENIA ------------------ *)

{ porovnávanie pre riešenie A: začína film a skôr ako film b? }
function skor_zacina(a,b : longint) : boolean;
begin skor_zacina := (Z[a] < Z[b]); end;

{ porovnávanie pre riešenie B: je film a kratší ako film b? }
function je_kratsi(a,b : longint) : boolean;
begin je_kratsi := (K[a]-Z[a]) < (K[b]-Z[b]); end;

{ porovnávanie pre riešenie C: končí film a skôr ako film b? }
function skor_konci(a,b : longint) : boolean;
begin skor_konci := (K[a] < K[b]); end;

{ porovnávanie pre riešenie D:
  ak vyberieme film a, škrtneme menej iných ako ak vyberieme film b? }
function skrtne_menej(a,b : longint) : boolean;
var i, skrtne_a, skrtne_b : longint;
begin
    skrtne_a := 0; skrtne_b := 0;
    for i:=1 to N do
        if (i<>a) and (not skrtnute[i]) and prekryvaju_sa(a,i) then inc(skrtne_a);
    for i:=1 to N do
        if (i<>b) and (not skrtnute[i]) and prekryvaju_sa(b,i) then inc(skrtne_b);
    skrtne_menej := (skrtne_a < skrtne_b);
end;

(* ----------- IMPLEMENTÁCIA VŠEOBECNÉHO RIEŠENIA ---------------------------- *)

{ porovnavacia_funkcia povie, či je film a lepší ako film b }
type porovnavacia_funkcia = function(a,b : longint) : boolean;

{ ako "lepsi" budeme postupne používať všetky štyri
  vyššie implementované porovnávacie funkcie }
function riesenie (lepsi : porovnavacia_funkcia) : longint;
var i, dalsi : longint;
begin
    for i:=1 to N do skrtnute[i] := false;
    riesenie := 0;
    while true do begin
        dalsi := -1;
        for i:=1 to N do
            if (not skrtnute[i]) and ( (dalsi=-1) or lepsi(i,dalsi) ) then
                dalsi := i;
        if (dalsi = -1) then exit; { už sa nám minuli všetky filmy }
        inc(riesenie);
        skrtnute[dalsi] := true;
        for i:=1 to N do if prekryvaju_sa(dalsi,i) then skrtnute[i] := true;
    end;
end;

(* ----------- HLAVNÝ PROGRAM ------------------------------------------------ *)
```

```
var i : longint;
begin
    read(N);
    setlength(Z,N+1); setlength(K,N+1); setlength(skrtnute,N+1);
    for i:=1 to N do read(Z[i],K[i]);
    writeln( riesenie( @skor_zacina  ) );
    writeln( riesenie( @je_kratsi    ) );
    writeln( riesenie( @skor_konci   ) );
    writeln( riesenie( @skrtne_menej ) );
end.
```

# Zadania celoštátneho kola kategórie A

## A-III-1 Pán Buridan a kaviarne

*Buridanov osol* je známy filozofický myšlienkový experiment. Predstavte si, že hladného osla postavíte presne do stredu medzi dve kôpky sena. Osol by si síce dal seno, ale keďže je situácia dokonale symetrická, nemá sa podľa čoho rozhodnúť, či ísť doľava alebo doprava. A tak ostane na mieste, až chudák zdochne od hladu.

Dávidka už poznáte z domáceho kola. Stále si chudák hľadá bývanie v Manhattane. Teraz si však uvedomil, že mu hrozí rovnaký problém ako Buridanovmu oslovi: ak by existovali dve kaviarne, ktoré sú od jeho bytu rovnako ďaleko, mohlo by sa mu stať, že sa medzi nimi nebude vedieť rozhodnúť a nedopadne to dobre.

Pre jednoduchosť si Manhattan predstavíme ako štvorcovú sieť, po ktorej sa dá pohybovať len v štyroch základných smeroch. Na niektorých mrežových bodoch (križovatkách) sú kaviarne; na každom najviac jedna.

**Súťažná úloha:**

Na vstupe je mapa Manhattanu. Pre každú križovatku zistite, či tam Dávidko môže bývať – teda či neexistujú žiadne dve kaviarne, ktoré by boli od danej križovatky rovnako ďaleko.

**Formát vstupu:**

V prvom riadku je rozmer $n$ štvorcovej siete predstavujúcej Manhattan: tvorí ho $n$ vodorovných a $n$ zvislých ciest. Máme teda presne $n^2$ križovatiek. Zvyšok vstupu tvorí $n$ riadkov, v každom z nich je $n$ čísel: 1 predstavuje križovatku s kaviarňou, 0 križovatku bez kaviarne.

**Formát výstupu:**

Vypíšte $n$ riadkov a v každom z nich $n$ znakov: pre každú križovatku buď '`A`' ak tam Dávidko bývať môže, alebo '`N`' ak nie.

**Hodnotenie:**

Plných 10 bodov dostanete za riešenie s asymptoticky optimálnou časovou zložitosťou. Pomalšie korektné riešenia môžu dostať 3 až 9 bodov podľa konkrétnej časovej zložitosti.

**Príklady:**

**Vstup**

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

**Výstup**

```
A A A A A
A A A A N
N N N N A
A A A A A
A A A A A
```

**Vstup**

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

**Výstup**

```
N N N
N N N
N N N
```

**Vstup**

```
2
0 1
0 0
```

**Výstup**

```
A A
A A
```

## A-III-2 Preťahovanie lanom

V Dolebrežnej Vieske sa koná tradičný turnaj v preťahovaní lanom. Keďže v tejto obci nie je nikde žiadna rovina, hrá sa turnaj na úbočí kopca. Sklonu kopca zodpovedá koeficient $q > 1$. Do turnaja sa prihlásilo $n$ hráčov. V poradí od najmladšieho po najstaršieho dostali čísla 1 až $n$. Silu hráča $i$ označíme $s_i$.

Na začiatku turnaja sú všetci hráči aktívni. Turnaj pozostáva z viacerých zápasov. V každom zápase hrajú všetci aktívni hráči. Na zápas sa aktívni hráči rozdelia do dvoch tímov. Horný tím (na vrchu kopca) tvorí $k$ najstarších aktívnych hráčov, dolný (na jeho spodku) tvoria všetci ostatní. Tím ťahajúci dohora má zjavne nevýhodu: vyhrajú práve vtedy, ak ich súčet síl je ostro väčší ako $q$-násobok súčtu síl tímu ťahajúceho dodola. Turnaj končí, akonáhle v niektorom zápase vyhrá horné družstvo.

Po každom zápase jeden z hráčov prestane byť aktívny. Domorodci si všimli, že je to vždy buď najmladší aktívny hráč (ktorý už musí ísť domov), alebo najstarší aktívny hráč (ktorý odíde do bufetu). Hodnota $k$ sa nemení, teda ak po zápase odišiel najstarší hráč, v nasledujúcom zápase bude za horný tím hrať aj najstarší z hráčov, ktorí doteraz hrali za dolný tím.

**Súťažná úloha:**

Na vstupe dostanete počet hráčov $n$, ich sily $s_1, \dots, s_n$, koeficient strmosti kopca $q$ a veľkosť horného tímu $k$. Napíšte program, ktorý vypočíta, koľko najviac zápasov sa môže v danom turnaji odohrať. Inými slovami, váš program by mal nájsť také poradie odchádzajúcich hráčov, pri ktorom prvá výhra horného tímu nastane čo najneskôr.

**Formát vstupu a výstupu:**

V prvom riadku vstupu sú celé čísla $n$ a $k$ a racionálne číslo $q$. Môžete predpokladať, že $1 \leq k \leq n$, že $q > 1$ a že všetky vaše výpočty s číslom $q$ dávajú presné výsledky.

V riadku druhom sú sily $s_1, \dots, s_n$. Môžete predpokladať, že sú to kladné celé čísla, ktorých súčet sa zmestí do bežnej číselnej premennej.

Na výstup vypíšte jediný riadok a v ňom jedno celé číslo: maximálny možný počet zápasov.

**Hodnotenie:**

Riešenia s časovou zložitosťou exponenciálnou od $n$ získajú nanajvýš 3 body. Ľubovoľné riešenie s časovou zložitosťou polynomiálnou od $n$ môže (v prípade, že je kompletné a korektné) získať aspoň 5 bodov. Časovú zložitosť optimálneho riešenia vám úmyselne neprezradíme. Nezabudnite zdôvodniť správnosť vášho algoritmu!

**Príklad:**

**Vstup**

```
5 2 1.01
40 70 1000 30 20
```

**Výstup**

```
4
```

*Tento kopec je skoro rovina, horný tím takmer nemá výhodu. Hore začínajú hráči so silou 30+20 = 50 a dole hráči so silou 40+70+1000 = 1110. Keďže* $1110 \times 1.01 > 50$, *dolný tím prvý zápas poľahky vyhrá.*

*Turnaj bude najdlhšie trvať vtedy, keď po prvom zápase odíde najmladší hráč (so silou 40) a po druhom zápase odíde opäť najmladší hráč (so silou 70).*

*Po treťom zápase je jedno kto odíde – na štvrtý zápas tak či onak ostanú hore dvaja hráči a dole nikto, a teda horný tím poľahky vyhrá.*

**Vstup**

```
12 7 2.0
2 4 9 5 3 3 4 8 8 6 1 6
```

**Výstup**

```
4
```

*Jeden možný optimálny priebeh zápasov: po prvom zápase odíde najmladší (sila 2), po druhom najstarší (sila 6), po treťom zase najmladší (sila 4). Na štvrtý zápas teraz na vrchu kopca nastúpia hráči so silou* $3+3+4+8+8+6+1=33$, *zatiaľ čo na spodku kopca už budú len hráči so silou* $9+5=14$. *No a keďže* $14 \times 2.0 < 33$, *štvrtý zápas už vyhrá horný tím a turnaj končí.*

## A-III-3 Mimozemské počítače

*K tejto úlohe patrí aj študijný text z domáceho kola, ktorý nájdete na strane 12 tejto ročenky.*

*Jednotlivé podúlohy môžete ich riešiť v ľubovoľnom poradí, každá je hodnotená samostatne. Na časovej zložitosti vašich algoritmov pri hodnotení nezáleží – len musí byť polynomiálna.*

**Podúloha A (1 bod):**

Mimozemšťania nám dodali sálový KSP, ktorý rozhoduje problém existencie Hamiltonovskej cesty obsahujúcej predpísanú hranu. Tento KSP má teda funkciu `cesta_s_hranou(n,E,u,v)`. Táto funkcia čaká ako parametre počet $n$ vrcholov grafu, zoznam $E$ jeho hrán, a dve čísla vrcholov $u$ a $v$. Ak v danom grafe existuje Hamiltonovská cesta, na ktorej $u$ a $v$ nasledujú bezprostredne po sebe, KSP rozsvieti zelené svetlo, inak rozsvieti červené. Túto funkciu môžete použiť **len raz** a jej volaním výpočet vášho programu končí.

Na vstupe dostanete jednoduchý neorientovaný graf $G$. Napíšte program s polynomiálnou časovou zložitosťou, ktorý rozsvieti zelené alebo červené svetlo podľa toho, či $G$ obsahuje aspoň jednu Hamiltonovskú cestu.

**Podúloha B (5 bodov):**

Mimozemšťania nám dodali kufríkový KSP, ktorý rozhoduje problém existencie Hamiltonovskej cesty. Tento KSP má teda funkciu `cesta(n,E)`. Táto funkcia čaká ako parametre počet $n$ vrcholov grafu a zoznam $E$ jeho hrán. Na

výstupe táto funkcia vracia `True` alebo `False` podľa toho, či v dotyčnom grafe existuje aspoň jedna Hamiltonovská cesta. Funkciu `cesta` môžete volať aj viackrát.

Na vstupe dostanete jednoduchý neorientovaný graf $G$ ktorý obsahuje Hamiltonovskú cestu. Napíšte pomocou tohto kufríkového KSP program, ktorý v polynomiálnom čase (nerátajúc volania funkcie `cesta`) v $G$ jednu Hamiltonovskú cestu nájde.

**Pozor!** Počet bodov, ktoré dostanete, bude závisieť od toho, rádovo koľko volaní funkcie `cesta` vaše riešenie v najhoršom prípade potrebuje.

**Podúloha C (4 body):**

Mimozemšťania nám dodali sálový KSP, ktorý rozhoduje problém existencie Hamiltonovskej cesty. Tento KSP má teda funkciu `cesta(n,E)`. Táto funkcia čaká ako parametre počet $n$ vrcholov grafu a zoznam $E$ jeho hrán. Ak v danom grafe existuje Hamiltonovská cesta, KSP rozsvieti zelené svetlo, inak rozsvieti červené. Túto funkciu môžete použiť **len raz** a jej volaním výpočet vášho programu končí.

Na vstupe dostanete *orientovaný* graf $G$. Každá hrana tohto grafu má teda určený jeden smer, v ktorom sa po nej smie ísť. Napíšte program s polynomiálnou časovou zložitosťou, ktorý rozsvieti zelené alebo červené svetlo podľa toho, či $G$ obsahuje aspoň jednu *orientovanú* Hamiltonovskú cestu. (Teda Hamiltonovskú cestu, ktorá každú svoju hranu použije v správnom smere.)

**Programovacie jazyky:**

Vo svojich riešeniach môžete používať ľubovoľný štrukturovaný programovací jazyk. Vhodne si zvoľte potrebné dátové štruktúry. Napr. v Pascale by funkcia z podúlohy A mohla vyzerať nasledovne:

```
type hrana = array[0..1] of longint;
procedure cesta_s_hranou(n : longint; m : longint; E : array of hrana; u,v : longint);
```

a v C++ môžeme použiť buď nízkoúrovňové polia, alebo aj vektor dvojíc:

```
void cesta_s_hranou(int n, int m, int E[][2], int u, int v);         // prva moznost
void cesta_s_hranou(int n, vector< pair<int,int> > E, int u, int v); // druha moznost
```

## A-III-4 Hore a dole

Baška mala postupnosť $n$ celých čísel, každé z rozsahu od 0 po $m$. Pre každé dva po sebe idúce členy postupnosti si zapísala znak <, =, alebo > podľa toho, či

bol skorší z nich menší, rovnaký, alebo väčší ako neskorší. Napr. pre postupnosť $(3, 1, 7, 7, 4)$ platí $3 > 1 < 7 = 7 > 4$, takže Baška by zapísala reťazec `><=>`.

Olívia Baškinu postupnosť nepozná. Pozná len $n$, $m$ a postupnosť znakov, ktoré si Baška zapísala. Olívia by teraz chcela Baškinu postupnosť (v rámci možností) zrekonštruovať.

Časť bodov za túto úlohu môžete získať tak, že napíšete program, ktorý zrekonštruuje ľubovoľnú takú postupnosť (ak nejaká existuje). Plný počet bodov dostanete za program, ktorý navyše vždy nájde to riešenie, v ktorom je *súčet členov Baškinej postupnosti najmenší možný.*

**Formát vstupu a výstupu:**

Vstup má dva riadky. V prvom z nich sú čísla $n$ a $m$. V druhom z nich je reťazec $n - 1$ znakov, ktoré si Baška zapísala. Na výstup vypíšte jeden riadok a v ňom $n$ medzerami oddelených celých čísel: zrekonštruovanú postupnosť. Ak žiadna postupnosť zodpovedajúca vstupu neexistuje, vypíšte $n$-krát hodnotu $-1$.

**Obmedzenia a hodnotenie:**

Pripravených je 10 sád vstupov, očíslovaných od 01 do 10. Za správne vyriešenie každej sady získate 1 bod. V sadách 01 až 05 akceptujeme ľubovoľnú korektnú postupnosť, v sadách 06 až 10 len postupnosť s najmenším možným súčtom členov. Veľkosti a typ testovacích dát sú popísané v nasledujúcej tabuľke:

| číslo sady | obmedzenia | | |
|---|---|---|---|
| 01, 06 | $2 \leq n \leq 10$, | $m = 10^9$, | niektorý jeden znak je `>` a všetky ostatné sú `<` |
| 02, 07 | $2 \leq n \leq 20$, | $m = 10^9$ | |
| 03, 08 | $2 \leq n \leq 1000$, | $m = 998$ | |
| 04, 09 | $2 \leq n \leq 75\,000$, | $0 \leq m \leq 10^9$, | v reťazci nie je znak `=` |
| 05, 10 | $2 \leq n \leq 250\,000$, | $0 \leq m \leq 10^9$ | |

**Príklady:**

**Vstup**

```
5 1000000000
><=>
```

**Výstup**

```
1 0 1 1 0
```

*Tento vstup by mohol patriť do sady 02 alebo sady 07. Ak by patril do sady 02, akceptovali by sme aj odpoveď „3 1 7 7 4“, ale v sade 07 je jedinou správnou odpoveďou práve „1 0 1 1 0“.*

**Vstup**

```
4 2
>>>
```

**Výstup**

```
-1 -1 -1 -1
```

*Reťazec* `>>>` *popisuje štvorprvkovú klesajúcu postupnosť. Keďže ale $m = 2$, my môžeme použiť len hodnoty 0, 1 a 2, a teda takúto postupnosť vyrobiť nevieme.*

## A-III-5 Vyvážené reťazce

Reťazec je *vyvážený* vtedy, keď sa v ňom všetky jeho písmená vyskytujú rovnako veľa ráz. Príklady vyvážených reťazcov: `aaaaa`, `badcx`, `bbaaab`. Reťazec `abacb` vyvážený nie je – napríklad preto, že sú v ňom dve `a` ale len jedno `c`.

V danom reťazci nájdite najdlhší súvislý podreťazec, ktorý je vyvážený.

Hľadaný podreťazec nemusí obsahovať všetky druhy písmen, ktoré sa vyskytujú v pôvodnom reťazci. Teda napr. pre reťazec `cbababac` je správnym riešením podreťazec `bababa`.

**Formát vstupu a výstupu:**

Vstup má jediný riadok a v ňom reťazec tvorený malými písmenami anglickej abecedy. Dĺžku reťazca si označíme $n$, znaky reťazca očíslujeme zľava doprava od 0 po $n - 1$.

Na výstup vypíšte jeden riadok a v ňom dve celé čísla: index znaku, ktorým hľadaný podreťazec začína, a index znaku, ktorým tento podreťazec končí. Ak existuje viacero optimálnych riešení, nájdite to, ktoré začína najskôr.

**Obmedzenia a hodnotenie:**

Pripravených je 10 sád vstupov, očíslovaných od 01 do 10. Za správne vyriešenie každej sady získate 1 bod.

Veľkosti a typ testovacích dát sú popísané v nasledujúcej tabuľke:

| číslo sady | obmedzenia | |
|---|---|---|
| 01 | $1 \leq n \leq 20$, | reťazec obsahuje len písmená `ab` |
| 02 | $1 \leq n \leq 1000$, | reťazec obsahuje len písmená `ab` |
| 03 | $1 \leq n \leq 1000$, | reťazec obsahuje len písmená `abcdefgh` |
| 04 | $1 \leq n \leq 1000$, | reťazec obsahuje len písmená `abcdefgh` |
| 05 | $1 \leq n \leq 100\,000$, | reťazec obsahuje len písmená `ab` |
| 06 | $1 \leq n \leq 100\,000$, | reťazec obsahuje len písmená `abc`, v optimálnom riešení nie sú použité všetky tri |

| číslo sady | obmedzenia |
|---|---|
| 07 | $1 \leq n \leq 100\,000$, reťazec obsahuje len písmená `abc` |
| 08 | $1 \leq n \leq 50\,000$, reťazec obsahuje len písmená `abcde` |
| 09 | $1 \leq n \leq 50\,000$ reťazec obsahuje len písmená `abcde` |
| 10 | $1 \leq n \leq 50\,000$ reťazec obsahuje len písmená `abcde` |

**Príklady:**

**Vstup**

```
baab
```

**Výstup**

```
0 3
```

*Optimálnym podreťazcom je celý reťazec.*

**Vstup**

```
baaab
```

**Výstup**

```
1 3
```

*Optimálnym podreťazcom je reťazec* `aaa`.

**Vstup**

```
cbababac
```

**Výstup**

```
1 6
```

*Príklad zo zadania.*

**Vstup**

```
cbabadbabae
```

**Výstup**

```
1 4
```

*Skorší výskyt je ten správny.*

## A-III-6 ACGT

Usámec sa snaží z fragmentov DNA skladať genóm. Fragment DNA je postupnosť znakov `ACGT`. Usámec už našiel $n$ fragmentov a označil ich $F_1, \ldots, F_n$. Navyše zistil, ktoré dvojice fragmentov môžu nasledovať bezprostredne za sebou.

**Poskladanie DNA.** Poskladaním DNA nazývame postupnosť fragmentov $a_1, a_2, \ldots, a_k$ ($1 \leq a_i \leq n$, niektoré fragmenty sa môžu aj opakovať) takú,

že každé dva po sebe idúce fragmenty po sebe smú nasledovať. Poskladaniu DNA zodpovedá reťazec znakov `ACGT`, ktorý dostaneme pospájaním príslušných fragmentov.

Príklad: Nech $n = 3$, $F_1 =$ `CG`, $F_2 =$ `AT` a $F_3 =$ `TCC`. Po sebe nech môžu nasledovať dvojice fragmentov (1,2), (1,3) a (2,1). Korektným poskladaním DNA pre tento prípad je napr. postupnosť 2,1 (ktorej zodpovedá reťazec `ATCG`) a tiež postupnosť 1,2,1,3 (reťazec `CGATCGTCC`). Postupnosť 3,1 ani postupnosť 1,1,2 nie sú korektné.

**Vzdialenosť reťazcov.** Pre dané dva reťazce $p$ a $q$ definujeme ich vzdialenosť ako najmenší počet zmien, ktoré treba postupne spraviť, aby sme z $p$ vyrobili $q$ (resp. naopak). Povolené zmeny sú troch typov: pridanie znaku (kamkoľvek do reťazca), zmazanie znaku a zmena jedného znaku na iný.

Príklad: Nech $p =$ `ATTA` a $q =$ `AGA`. Zjavne nevieme $p$ na $q$ prerobiť jednou zmenou, ide to však dvomi: najskôr zmažeme jedno `T` a potom zmeníme druhé `T` na `G`. Preto majú tieto dva reťazce vzdialenosť 2. Ľubovoľný reťazec má sám od seba vzdialenosť 0.

**Súťažná úloha:**

Na vstupe je popis Usámcovych fragmentov a toho ako môžu na seba nadväzovať. Ďalej je na vstupe reťazec $R$ (opäť tvorený znakmi `ACGT`), *veľmi malé* nezáporné celé číslo $d$ a dva indexy fragmentov $u$ a $v$ (pričom $u \neq v$). Vašou úlohou je nájsť (jedno ľubovoľné) poskladanie DNA, ktoré začína fragmentom $u$, končí fragmentom $v$, a reťazec, ktorý mu zodpovedá, má od reťazca $r$ vzdialenosť nanajvýš $d$.

Môžete predpokladať, že žiaden fragment nesmie nasledovať sám po sebe – teda že v zozname prípustných dvojíc nebudú záznamy tvaru $(x, x)$. Tiež môžete predpokladať, že ak po nejakom fragmente $x$ môže nasledovať aj fragment $y$ aj fragment $z$, tak sa prvé písmená $F_y$ a $F_z$ líšia.

**Formát vstupu a výstupu:**

V prvom riadku vstupu je počet fragmentov $n$ a počet prípustných dvojíc $m$. V nasledujúcich $n$ riadkoch sú jednotlivé fragmenty $F_1$ až $F_n$. V ďalších $m$ riadkoch sú dvojice čísel $a_i$, $b_i$, ktoré vyjadrujú, že po fragmente $a_i$ môže nasledovať fragment $b_i$. Predposledný riadok obsahuje čísla $d$, $u$ a $v$; posledný riadok obsahuje reťazec $R$.

Na výstup vypíšte jeden riadok obsahujúci $-1$, pokiaľ žiadne vhodné poskladanie DNA neexistuje. Ináč vypíšte jeden riadok, ktorý obsahuje jedno vhodné

poskladanie DNA – teda postupnosť indexov fragmentov oddelených medzerami. (Nie je potrebné mať najmenšiu možnú vzdialenosť od $R$.)

**Hodnotenie:**

Pripravených je 10 sád vstupov, očíslovaných od 01 do 10. Na sadách 01 až 08 bude vaše riešenie otestované na testovači. Vstupy 09 a 10 si stiahnite z testovača, spustite na nich váš program a odovzdajte iba výstup.

Za správne vyriešenie každej sady získate 1 bod. V nasledujúcej tabuľke $f$ označuje súčet dĺžok fragmentov DNA a $r$ dĺžku reťazca $R$. Pokiaľ nie je uvedené inak, tak v sadách 01 až 08 platí $n \leq 1000$, $m \leq 4000$, $f \leq 50\,000$, $r \leq 50\,000$ a $d \leq 5$. V sadách 09 a 10 platí $n \leq 1000$, $m \leq 1000$, $f \leq 1\,000\,000$, $r \leq 1\,000\,000$ a $d \leq 5$. Navyše tieto vstupy sú z reálnych dát, čo napríklad znamená, že medzi fragmentmi na vstupe nenájdete veľa podobných. Dodatočné obmedzenia pre jednotlivé sady sú nasledovné:

| sada | obmedzenia | komentár |
|---|---|---|
| 01 | $d = 0$ | fragmenty musia naskladať presne $R$ |
| 02, 03 | $n = 2$, $m = 1$, $f \leq 1000$, $r \leq 1000$ | dva fragmenty a jeden spôsob ich zreťazenia |
| 04, 05 | $n = 2$, $m = 1$ | dva fragmenty a jeden spôsob ich zreťazenia |
| 06 | $d = 1$ | smieme mať nanajvýš jednu chybu |
| 07, 08 | $f \leq 1000$, $r \leq 1000$ | |

**Príklady:**

**Vstup**

```
3 3
CG
AT
TCC
1 2
2 1
1 3
0 1 3
CGATCGTCC
```

**Výstup**

```
1 2 1 3
```

*Toto je príklad zo zadania, hľadaný reťazec vieme naskladať presne.*

**Vstup**

```
4 4
CCC
A
T
GGG
1 2
2 3
3 2
2 4
1 1 4
CCCATTGGG
```

**Výstup**

```
1 2 3 2 4
```

*Toto poskladanie nám dá reťazec* `CCCATAGGG`*, ktorý sa od reťazca na vstupe líši jednou zmenou.*

**Vstup**

```
4 4
CCC
A
T
GGG
1 2
2 3
3 2
2 4
1 1 4
CCCAAAAGGG
```

**Výstup**

```
-1
```

*Treba aspoň dve zmeny, povolenú však máme nanajvýš jednu.*

# Riešenia domáceho kola kategórie A

## A-I-1 Taká zima...

Poďme sa vrhnúť rovno na riešenie. Prvé, čo by sme mali spraviť, keď neviem nič lepšie, je napísať si priamočiare, aj keď pomalé riešenie. V tomto prípade stačí skúšať všetky možné dvojice začiatkov a koncov. Keď už máme zvolený konkrétny začiatok a koniec, tak ešte v jednom cykle overíme, či žiadne z čísel ležiacich medzi nimi neporušuje podmienku zo zadania. Takto dostaneme riešenie so zložitosťou $O(n^3)$, za ktoré pravdepodobne získame 3 body. Na začiatok dobré, ale ešte stále je čo zlepšovať.

Pri riešeniach, kde sa snažíme hľadať nejaký súvislý úsek v poli, sa často dá použiť nasledovný prístup. Znovu si generujeme všetky dvojice (začiatok,koniec), robíme to však v určitom poradí a snažíme sa znížiť čas, ktorý potrebujeme na overenie, či je táto konkrétna dvojica indexov vhodná.

Zafixujeme si $x$ (teda prvý index, ktorý je v poli viac naľavo) a postupne budeme zvyšovať $y$. Začneme na $y = x + 1$ a skončíme na $y = n$. Keď toto celé spravíme pre $x = 1$, začneme odznova s $x = 2$, a tak ďalej. V čom je výhoda takéhoto prístupu? Ak si zoberieme dve za sebou idúce kontroly, dostaneme dvojicu $(x, y)$ a $(x, y+1)$. Vidíme, že úsek medzi týmito číslami sa rozšíril iba o číslo $T[y]$. Ak sme teda niečo zistili pre dvojicu $(x, y)$, možno nám stačí pridať jediný prvok a budeme vedieť odpoveď aj pre $(x, y + 1)$.

Podmienku zo zadania si môžeme preformulovať nasledovne: Ak si zoberieme všetky prvky na pozíciách medzi $x$ a $y$, nesmie medzi nimi ležať žiadne z intervalu $\big[\min(T[x], T[y]),\ \max(T[x], T[y])\big]$. Aby sme toto vedeli efektívne overiť, môžeme si napríklad udržiavať usporiadanú množinu čísel, ktoré ležia ostro medzi indexami $x$ a $y$. Do tejto množiny budeme postupne pridávať nové čísla a pozerať sa, či sa medzi nimi vyskytuje nejaké, ktoré leží medzi aktuálnymi hodnotami $T[x]$ a $T[y]$.

Všetky tieto veci zvláda robiť vyvažovaný binárny vyhľadávací strom, v C++ implementovaný v dátovej štruktúre `set`. Riešenie je teda nasledovné: Postupne skúšame všetky možnosti pre ľavý index $x$. Pre pevne zvolené $x$ od neho ideme pravým indexom $y$ doprava, pričom si v `sete` udržujem množinu čísel, ktoré sa vyskytujú ostro medzi pozíciami $x$ a $y$. (Napriek tomu, že sa na vstupe môže vyskytnúť tá istá hodnota viackrát, nie je v tomto riešení potrebné použiť `multiset`, keďže nám je jedno, koľkokrát sa daná hodnota v spracúva-

nom úseku vyskytuje.) Pri spracúvaní prvku $T[y]$ sa najskôr pozriem, či nie je rovnaký ako $T[x]$. Ak je, toto je vyhovujúca dvojica, ale žiadna ďalšia $(x, z)$, kde $y < z$ vyhovovať nebude, preto môžeme následne prejsť rovno na ďalšie $x$. A ak sú $T[x]$ a $T[y]$ rôzne, tak sa pozrieme do nášho `setu` a nájdeme si (v logaritmickom čase) začiatok a koniec úseku, v ktorom ležia prvky z intervalu $\left[\min(T[x], T[y]),\ \max(T[x], T[y])\right]$. Ak je tento úsek prázdny, tvoria $x$ a $y$ vyhovujúcu dvojicu, inak nie.

Riešenie, ktoré sme práve popísali, má časovú zložitosť $O(n^2 \log n)$, lebo pre každú z $n^2$ dvojíc indexov urobíme niekoľko (konštantne veľa) operácií so `setom`.

Celého `setu` sa však vieme ľahko zbaviť. Stačí si uvedomiť jednu vec: počas toho, ako postupne zvyšujeme $y$, sa hodnota $T[x]$ nemení. Predstavme si teraz, že práve spracúvame nejaké $y$ také, že $T[x] \leq T[y]$. Potom nás z prvkov ležiacich medzi indexami $x$ a $y$ zaujímajú len tie, ktoré sú väčšie alebo rovné $T[x]$. Presnejšie, zaujíma nás len to, či je niektorý z týchto prvkov zároveň menší alebo rovný $T[y]$. Na efektívne zodpovedanie tejto otázky si stačí pamätať *najmenší* z prvkov väčších alebo rovných $T[x]$: ak tento prvok neleží v intervale od $T[x]$ po $T[y]$, tak tam neleží žiaden.

No a symetricky: kvôli indexom $y$ takým, že $T[x] \geq T[y]$, si budeme zase pamätať najväčší z prvkov menších alebo rovných $T[x]$. Takto sme celý `set` nahradili dvomi premennými. Toto zlepšenie nám zaručí časovú zložitosť $O(n^2)$. Tomu by mal zodpovedať zisk šiestich bodov.

**Vzorové riešenie:**

Pozrime sa lepšie na predchádzajúce riešenie. Kedy sme hodnotu $y$ vyhodnotili ako prípustnú a vyskúšali zlepšiť doteraz nájdené optimum? Práve vtedy, keď sme tým zmenili aktuálnu hodnotu jednej z dvoch pamätaných premenných.

Pre konkrétne $x$ nás v skutočnosti zaujíma len *posledná* z prípustných hodnôt $y$. A vďaka práve spravenému pozorovaniu vieme, že pre ňu sú len dve možnosti: buď je ňou index najmenšieho prvku napravo od $x$ ktorý je väčší alebo rovný $T[x]$, alebo je ňou index najväčšieho prvku napravo od $x$ ktorý je menší alebo rovný $T[x]$. Oba tieto indexy (ak aspoň jeden taký prvok existuje) vždy zodpovedajú prípustným riešeniam a ten z nich, ktorý je ďalej od $x$, je zjavne najlepším jemu zodpovedajúcim $y$.

(V prípade, že sa v poli môžu prvky opakovať, ide o *najbližší* výskyt prvku s danou hodnotou – vtedy sa príslušná premenná zmení. Ďalšie výskyty rovnakej hodnoty už nebudú prípustné.)

V našej implementácii robíme presne toto isté, len ako vonkajšiu premennú

cyklu máme $y$. (Zjednoduší to podmienku pri porovnávaní či je nové riešenie lepšie ako doteraz nájdené.) Postupne pre každé $y$ nás zaujímajú dva indexy: kde sa nachádza veľkosťou najbližšia väčšia alebo rovná hodnota (ak tam nejaká je) a kde sa nachádza veľkosťou najbližšia menšia alebo rovná hodnota (opäť, ak tam nejaká je). V C++ na toto môžeme použiť dátovú štruktúru `map`, v ktorej si pre každú hodnotu pamätáme index jej doteraz posledného výskytu.

Pomocou `mapy` vieme pre dané $y$ zistiť hľadané dva indexy $x_1, x_2$ v čase $O(\log n)$. Celkovo má teda toto riešenie časovú zložitosť $O(n \log n)$.

### Listing programu (C++)

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

void update(int &bx, int &by, int x, int y) { if (y-x >= by-bx) bx=x,by=y; }

int main() {
    int best_x=-1, best_y=-1;
    int N; cin >> N;

    map<int,int> last_seen;
    // pridáme zarážky, ktoré nikdy nevyrobia dobré riešenie
    last_seen[ -1<<30 ] = last_seen[ 1<<30 ] = N;
    for (int y=0; y<N; ++y) {
        int t; cin >> t;
        auto geq = last_seen.lower_bound(t);
        update( best_x, best_y, geq->second, y );
        auto leq = last_seen.upper_bound(t); --leq;
        update( best_x, best_y, leq->second, y );
        last_seen[t]=y;
    }
    cout << best_x << " " << best_y << endl;
}
```

**Alternatívne vzorové riešenie – navzájom rôzne prvky:**

Vyššie popísané vzorové riešenie sa dá implementovať aj bez použitia `mapy`: usporiadame všetky hodnoty v poli $T$, odstránime duplikáty, a následne si budeme ku každej hodnote jej posledný výskyt pamätať v obyčajnom poli. Na prístup ku konkrétnej hodnote použijeme vždy binárne vyhľadávanie, takže časová zložitosť ostane $O(n \log n)$. (Ide vlastne o jednoduché použitie univerzálnejšej techniky, tzv. *kompresie súradníc.*)

V tejto časti si ukážeme iné riešenie, ktoré si vystačí bez zložitých dátových štruktúr. Jediné, čo budeme potrebovať, bude obyčajné triedenie.

Najskôr si toto riešenie ukážeme pre postupnosti, v ktorých sa prvky neopakujú. Pomôže nám to spraviť si správnu predstavu, ktorú následne upravíme pre všeobecný prípad. Majme teda pole $T$, v ktorom je $n$ navzájom rôznych hodnôt. Predstavme si, že sme toto pole usporiadali podľa veľkosti. Zjavne platí, že ak

nejaké dva prvky, ktoré pôvodne boli na pozíciách $a$ a $b$, skončili po usporiadaní poľa $T$ bezprostredne vedľa seba, tak indexy $a$ a $b$ tvoria jedno možné riešenie – keďže vôbec neexistuje prvok s hodnotou medzi $T[a]$ a $T[b]$, tak zjavne nemôže taký prvok ležať medzi pozíciami $a$ a $b$. To teda znamená, že každá dvojica po sebe nasledujúcich hodnôt určuje jedno potenciálne riešenie.

My teraz tvrdíme, že medzi týmito $n-1$ riešeniami sa vždy nachádza aj to optimálne. Presnejšie, dokonca všetky optimálne riešenia musia byť tohto typu. Toto si teraz dokážeme.

Chceme dokázať, že optimálne riešenie $(x, y)$ nemôžu tvoriť také prvky $T[x]$ a $T[y]$, ktoré neležia vedľa seba po usporiadaní celého poľa. Pre spor, nech nejaká takáto dvojica optimálne riešenie tvorí. Keďže po usporiadaní neležia zodpovedajúce prvky pri sebe, existuje pre pôvodné pole $T$ index $z$ taký, že $\min(T[x], T[y]) < T[z] < \max(T[x], T[y])$. Keďže však $(x, y)$ je prípustné riešenie, tak nesmie byť $x < z < y$. Preto platí buď $z < x$ alebo $y < z$. Bez ujmy na všeobecnosti, nech $z < x$. Ak je takýchto možných hodnôt $z$ viac, nech je $z$ najväčšia z nich. Potom ale $(z, y)$ je zjavne tiež prípustné riešenie a je lepšie ako $(x, y)$, čo je hľadaný spor.

Stačí teda usporiadať pole usporiadaných dvojíc $(T[i], i)$. Následne už optimálne riešenie nájdeme jedným prechodom, v ktorom pre každé dve po sebe nasledujúce hodnoty spočítame rozdiel indexov na ktorých ležia.

**Alternatívne vzorové riešenie – ľubovoľné prvky:**

Ako posledné nám zostáva vyriešiť, čo s rovnakými prvkami. Keď skúsime zovšeobecniť vyššie uvedenú úvahu, vynoria sa zrazu dva rôzne prípady: buď bude optimálne riešenie predstavovať interval medzi dvoma výskytmi tej istej hodnoty, alebo pôjde o interval medzi výskytmi dvoch po sebe idúcich hodnôt.

Výskyty tej istej hodnoty sú ľahké: o prípustné riešenie ide vtedy a len vtedy, keď ide o dva po sebe idúce výskyty danej hodnoty.

A vlastne to presne rovnako musí byť aj v prípade, že berieme dve po sebe idúce hodnoty. Nech teda chceme nájsť všetky prípustné riešenia, pri ktorých $T[x] = \alpha$ a $T[y] = \beta$, kde $\alpha \neq \beta$ sú dve bezprostredne po sebe nasledujúce hodnoty. Predstavme si, že sme zobrali všetky indexy, kde sa vyskytujú hodnoty $\alpha$ a $\beta$. Potom prípustné riešenia opäť zodpovedajú práve dvojiciam po sebe idúcich indexov – pre ľubovoľnú inú dvojicu hneď vidíme, že aspoň jeden iný prvok leží v zakázanom rozsahu.

Budeme teda postupovať nasledovne. Pole si usporiadame ako v predchádzajúcom riešení a rozdelíme ho na úseky rovnakých prvkov (tie volajme buckety). Následne budeme mať funkciu, ktorá nájde spomedzi prvkov, ktoré dostane na

vstupe, optimálne riešenie. Tejto funkcii práve raz dáme na vstup každý bucket, a tiež práve raz každú dvojicu po sebe idúcich bucketov. No a spomínaná funkcia vždy spraví to, že prvky, ktoré dostala na vstupe, preusporiada podľa ich pôvodného indexu, potom prejde všetky po sebe idúce dvojice indexov a vyberie najlepšiu z nich.

Každý prvok usporiadame najviac 4-krát, čo znamená, že riešenie bude mať časovú zložitosť $O(n \log n)$. Pamäťová zložitosť bude zjavne $O(n)$.

## A-I-2 Dva ploty

Najskôr si ukážeme, ako riešiť úlohu pre jeden plot, potom toto riešenie zovšeobecníme na dva ploty.

**Konvexný obal:**

Pre ľubovoľnú konečnú množinu bodov v rovine je jednoznačne určený útvar s najmenším obvodom, ktorý ich všetky obsahuje. Je ním ich *konvexný obal*. (Dôkaz tohto tvrdenia uvádzame nižšie.)

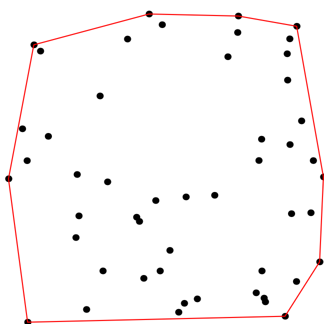
Čo je to konvexný obal?

Útvar $\mathcal{U}$ v rovine voláme *konvexný*, ak má nasledovnú vlastnosť: pre ľubovoľné dva body $A, B \in \mathcal{U}$ aj celá úsečka $AB$ patrí do útvaru $\mathcal{U}$. Ľudovo povedané, konvexný útvar nemá žiadne diery ani záhyby dovnútra.

Prienik ľubovoľného (dokonca aj nespočítateľne nekonečného!) množstva konvexných útvarov je zjavne opäť konvexný útvar. (Totiž nech $A, B$ sú ľubovoľné dva body z toho prieniku. Potom tieto body sú v každom z pôvodných útvarov. A keďže všetky tie útvary sú konvexné, každý obsahuje celú úsečku $AB$. A teda táto úsečka patrí aj do ich prieniku.)

Konvexný obal danej množiny bodov teda môžeme formálne definovať ako prienik *úplne všetkých* konvexných útvarov, ktoré danú množinu bodov obsahujú. Menej formálne ale omnoho názornejšie je povedať, že konvexný obal danej množiny bodov je *najmenší* zo všetkých konvexných útvarov, ktoré danú množinu obsahujú.

A nie je ťažké si uvedomiť, ako tento konvexný obal vyzerá: konvexným obalom danej konečnej množiny bodov v rovine je vždy mnohouholník, ktorého vrcholmi sú niektoré spomedzi zadaných bodov. (Spolu s danými bodmi musí konvexný obal obsahovať aj všetky úsečky spájajúce dva z daných bodov. Tie z úsečiek, ktoré ležia „na obvode“, tvoria hranicu hľadaného konvexného mnohouholníka.)



Konvexný obal danej množiny bodov.

A prečo má teda práve konvexný obal zo všetkých možných útvarov, ktoré našu množinu bodov obsahujú, najmenší obvod?

Intuitívne, predstavme si naše body ako klince a hľadaný útvar ako gumičku natiahnutú okolo nich. Keď gumičku pustíme, tá sa začne sťahovať (a teda skracovať). Najradšej by sa celá skrátila na nulovú dĺžku, to jej ale nedovolia klince v jej vnútri. Časom sa teda ustáli v polohe kedy je najkratšia ako sa dá – bude napnutá okolo všetkých „vonkajších“ klincov. Inými slovami, bude tvoriť práve obvod vyššie popísaného konvexného obalu.

Formálny matematický dôkaz by sa dal spraviť napr. sporom. Nech teda existuje útvar $\mathcal{U}$, ktorý obsahuje všetky dané body a má menší obvod ako ich konvexný obal. Zoberme ľubovoľnú stranu $AB$ konvexného obalu, ktorá nie je (celá) súčasťou obvodu útvaru $\mathcal{U}$. Predĺžme $AB$ na priamku a označme $A'$ a $B'$ „najľavejší“ a „najpravejší“ z bodov tejto priamky, ktoré ešte patria do útvaru $\mathcal{U}$. (Tieto body vždy existujú, lebo $A$ a $B$ patria do $\mathcal{U}$.) Ak ale teraz zahodíme časť obvodu $\mathcal{U}$ medzi $A'$ a $B'$ a nahradíme jú úsečkou $A'B'$, dostaneme nový útvar, ktorý tiež obsahuje všetky dané body a má obvod ostro kratší ako $\mathcal{U}$. A to je hľadaný spor.

**Konštrukcia konvexného obalu:**

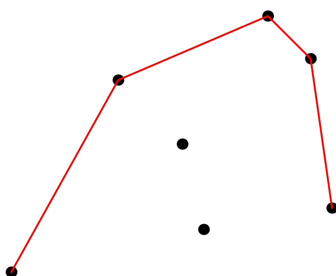
Ukážeme si Grahamov algoritmus, ktorý konvexný obal $n$ bodov zostrojí v čase $\Theta(n \log n)$. Toto je takmer optimálne. (Existujú komplikovanejšie algoritmy s o chlp lepšou časovou zložitosťou $\Theta(n \log h)$, kde $h$ je počet bodov zostrojeného konvexného obalu.)

Začneme tým, že dané body usporiadame zľava doprava a v rámci rovnakej x-ovej súradnice zdola hore. Následne konvexný obal zostrojíme v dvoch prechodoch: v prvom jeho hornú a v druhom jeho dolnú „polovicu“. Presnejšie, prvý bod aj posledný bod z usporiadaného poradia (t.j. najspodnejší z najľavejších a najhornejší z najpravejších) určite oba ležia na konvexnom obale a delia ho na dva úseky: „horný“ a „dolný“. V každom prechode zostrojíme jeden z nich.

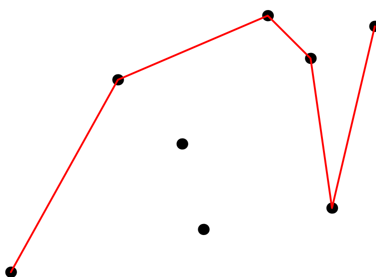
Popíšeme prvý prechod, teda zostrojenie horného obalu. (Druhý prechod je potom takmer identický, až na znamienka.)

Budeme postupne spracúvať body v usporiadanom poradí, ktoré sme si pred chvíľou vyrobili. Po spracovaní každého bodu bude platiť, že práve poznáme horný konvexný obal doteraz spracovaných bodov. To je nejaká lomená čiara, ktorá začína v prvom spracovanom bode, niekoľkokrát (možno nulakrát) zatočí doprava a skončí v poslednom spracovanom bode.

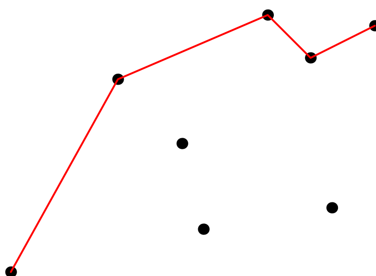
Príklad takejto situácie:



Čo sa teraz stane, keď nám pribudne nasledujúci bod? Body spracúvame usporiadané, nasledujúci bod teda pribudne napravo od doterajšieho konca horného obalu. Najjednoduchšie čo môžeme teraz spraviť je jednoducho obal poň predĺžiť:

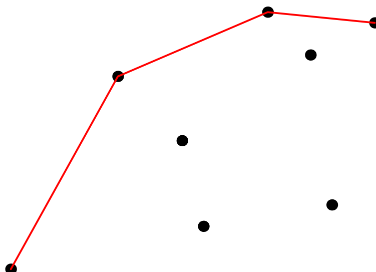


Občas nám však môže nastať práve taká situácia ako na našom obrázku: tým, že sme obal predĺžili, zrazu zatáča do nesprávnej strany a vznikla v ňom „preliačenina“. Toto potrebujeme napraviť. Ako? Jednoducho. Problémy vždy nutne spôsobuje práve predposledný bod. Ten sa nachádza na našej lomenej čiare, v skutočnosti však už má ležať pod horným obalom. Z obalu ho preto vyhodíme:



Ale čo to? Aj nasledujúci bod (ktorý je teraz novým predposledným bodom)

ešte robí problémy. Aj v ňom chce naša lomená čiara zatáčať doľava, a to nám jednoznačne hovorí, že aj tento bod patrí pod nový horný obal:



A už je všetko v poriadku a máme hotový horný obal novej množiny bodov.

Akú má toto riešenie časovú zložitosť? Na začiatku potrebujeme usporiadať $n$ bodov, čo spravíme v čase $\Theta(n \log n)$. Následne robíme dva prechody a pri každom vyššie popísaným spôsobom udržiavame časť konvexného obalu. Na prvý pohľad by sa mohlo zdať, že časová zložitosť každého prechodu je kvadratická – pri pridaní nového bodu môže byť treba veľa nových vyhodiť. Toto sa ale nemôže stávať často. Dobrý pohľad na časovú zložitosť je napr. nasledovný: každý bod na konvexný obal raz pridáme (keď ho spracúvame) a následne ho z obalu nanajvýš raz vyhodíme. Dokopy má celý prechod teda lineárnu časovú zložitosť. (Keďže vždy vyhadzujeme body len z konca zostrojovaného konvexného obalu, stačí si pri implementácii jeho vrcholy pamätať v zásobníku.)

**Dva konvexné obaly:**

Teraz už teda vieme, ako k danej množine bodov efektívne nájsť jej konvexný obal. Potrebujeme sa už len zamyslieť nad tým, kedy je vhodné spraviť ich dva.

Predstavme si, že máme danú množinu bodov a už poznáme jej optimálne rozdelenie na dve podmnožiny. Samozrejme, pre každú podmnožinu je optimálnym riešením spraviť jej konvexný obal.

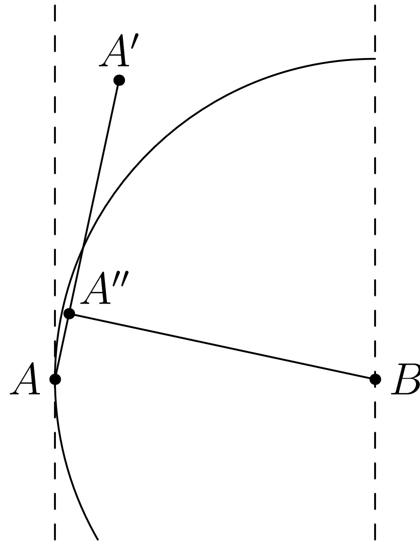
Až 7 bodov sa dalo získať za nasledovný algoritmus: „Nájdi konvexný obal všetkých bodov. Ak $n \leq 18$, vyskúšaj následne všetky rozdelenia bodov na dve podmnožiny a nájdi konvexné obaly tých.“

Aby sme ale našli polynomiálne riešenie našej úlohy, budeme sa ešte musieť trochu zamyslieť. Prvé, čo si musíme uvedomiť, je nasledovná skutočnosť: Ak je optimálne spraviť konvexné obaly dva, tak sú nutne disjunktné. Totiž ak by disjunktné neboli, tak zoberme ich zjednotenie $\mathcal{Z}$. Toho obvod je nanajvýš tak dlhý ako súčet obvodov našich dvoch konvexných obalov. No a zároveň platí, že $\mathcal{Z}$ je súvislý útvar, ktorý obsahuje všetky dané body, a teda je jeho obvod aspoň tak dlhý ako obvod konvexného obalu všetkých daných bodov.

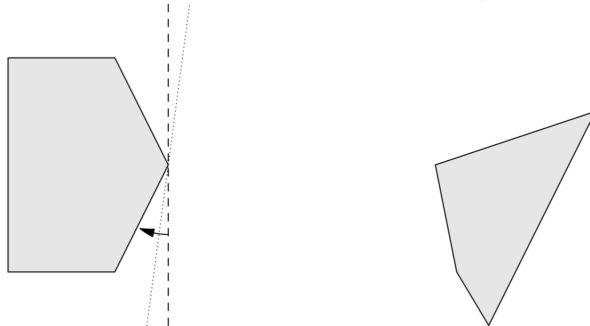
Rozdeliť dané body na dve podmnožiny sa nám teda oplatí len vtedy, ak ich

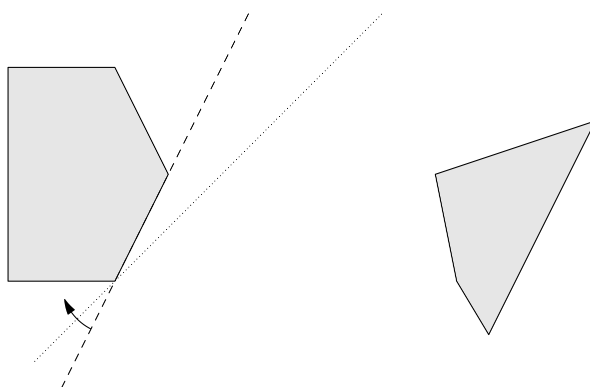
konvexné obaly budú disjunktné. Ako ale takéto rozdelenia nájsť? A koľko ich vlastne môže byť?

Tu nám pomôže ďalší známy výsledok z geometrie: veta o separujúcej priamke. Pre ľubovoľné dva konečné konvexné útvary v rovine existuje priamka, ktorá ich od seba oddeľuje. Jeden možný dôkaz tohto tvrdenia: Označme naše útvary $\mathcal{A}$ a $\mathcal{B}$. Nech $A \in \mathcal{A}$ a $B \in \mathcal{B}$ je konkrétna dvojica bodov zvolená tak, aby dĺžka $AB$ je najmenšia možná. Potom hľadanou priamkou je napríklad os úsečky $AB$, alebo aj ľubovoľná iná priamka kolmá na úsečku $AB$ a prechádzajúca jej vnútorným bodom. Sporom: nech napr. nejaký iný bod $A' \in \mathcal{A}$ leží na jednej z týchto priamok. Potom v $\mathcal{A}$ leží aj celá úsečka $AA'$. Označme $A''$ pätu kolmice z $B$ na $AA'$. Zjavne platí $|AB| > |A''B|$, čo je hľadaný spor.

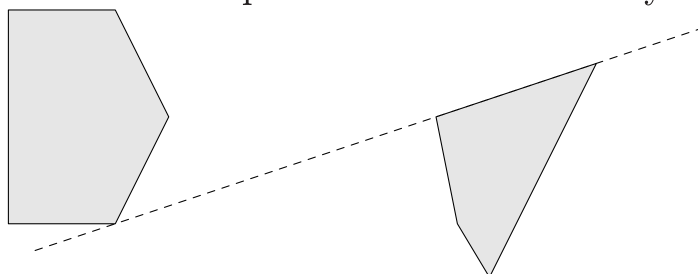



Dobre, teda už vieme, že ak máme dva disjunktné konvexné obaly, tak určite existuje priamka, ktorá ich od seba separuje. Ako nám má toto pomôcť? Predstavme si, že už máme dva konvexné mnohouholníky a priamku, ktorá ich separuje. V prvom kroku túto priamku posunieme tak, aby sa dotýkala jedného z mnohouholníkov. A keď už toto máme, začneme ju „kotúľať“ po jeho obvode.





Skôr alebo neskôr musí naša priamka naraziť na druhý z mnohouholníkov:



Týmto argumentom sme teda dokázali nasledovnú skutočnosť: pre ľubovoľné dva disjunktné konvexné mnohouholníky existuje priamka, ktorá separuje ich vnútra a oboch sa dotýka. Pritom zjavne platí, že na tejto novej separujúcej priamke leží aspoň jeden z vrcholov každého z mnohouholníkov.

A toto je nesmierne užitočné pozorovanie. Totiž takýchto separujúcich priamok je nanajvýš toľko ako dvojíc bodov daných na vstupe – teda nanajvýš rádovo $n^2$. Pre obmedzenia dané v zadaní ($n \leq 300$) si môžeme dovoliť všetky potenciálne separujúce priamky vyskúšať.

Výsledný algoritmus bude teda v preudokóde vyzerať nasledovne:

```
pre každú dvojicu bodov A, B:
    ak existuje bod C na úsečke AB, preskoč túto dvojicu
    roztrieď body na dve množiny:
        M1 = body naľavo od priamky AB + body na priamke AB ktoré sú bližšie ku A
        M2 = body napravo od priamky AB + body na priamke AB ktoré sú bližšie ku B
    nájdi konvexné obaly M1 a M2
    pozri, či si nenašiel lepšie riešenie ako doteraz optimálne
```

Zopár technických detailov na záver. Všimnite si, že nie je potrebné testovať disjunktnosť obalov `M1` a `M2`, tá je zaručená ich výberom (a aj keby nebola, pre nedisjunktné by nám proste vyšiel väčší súčet dĺžok obvodov ako pre optimálne riešenie). Treba si ešte dať pozor na degenerované prípady – podľa zadania ak delíme body na dve časti, tak konvexný obal každej musí mať neprázdny obsah. Toto v našej implementácii testujeme priamo: spočítame obsahy oboch zostrojených konvexných obalov a zarátame ich len ak sú oba kladné.

V našej implementácii neignorujeme tie dvojice `A`, `B`, pre ktoré nejaký `C` leží na úsečke `AB` – jednoduchšie bolo takéto body pridať do množiny `M2` a vyskúšať aj túto možnosť, nič to nepokazí. Mohlo by sa zdať, že treba ešte skúšať aj možnosť, kde `M1 = body naľavo od priamky AB + body na priamke AB ktoré sú bližšie ku *B*`, toto však nie je potrebné – vždy vieme nájsť takú separujúcu priamku, ktorej zodpovedá nami skúšané delenie.

Časová zložitosť tohto algoritmu je $O(n^3 \log n)$: pre každú z $O(n^2)$ dvojíc bodov zostrojíme v celkovom čase $O(n \log n)$ dva konvexné obaly. Takýto algoritmus stačil na získanie 10 bodov. Jeho časová zložitosť sa dá ďalej zlepšiť na $O(n^3)$: stačí si uvedomiť, že keď na začiatku raz usporiadame všetky body, už ich potom pri počítaní menších konvexných obalov netreba usporadúvať znova.

## Listing programu (C++)

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

// bod
struct point {
    long long x, y;
    point(long long x=0, long long y=0) : x(x), y(y) {}
};

// operátor < na usporiadanie bodov
bool operator< (const point &A, const point &B) {
    return A.x < B.x || ( A.x == B.x && A.y < B.y );
}

// vektorový súčin: vráti >0 / 0 / <0 podľa toho,
// či OAB zatáča proti smeru ručičiek / ide rovno / zatáča v smere
long long cross(const point &O, const point &A, const point &B) {
    return (A.x-O.x)*(B.y-O.y)-(A.y-O.y)*(B.x-O.x);
}

// skalárny súčin: pre O, A, B na priamke vráti
// >0 ak sú A a B na tej istej strane O, <0 ak sú na opačných stranách
long long dot(const point &O, const point &A, const point &B) {
    return (A.x-O.x)*(B.x-O.x)+(A.y-O.y)*(B.y-O.y);
}

// konvexný obal pomocou Grahamovho algoritmu; P musí byť usporiadané zľava doprava
vector<point> convex_hull(vector<point> P) {
    int n = P.size(), k = 0;
    vector<point> H(2*n);
    for (int i = 0; i < n; i++) {
        while (k >= 2 && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
        H[k++] = P[i];
    }
    for (int i = n-2, t = k+1; i >= 0; i--) {
        while (k >= t && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
        H[k++] = P[i];
```

```
    }
    H.resize(k-1);
    return H;
}

// obvod mnohouholníka
double circumference(const vector<point> &P) {
    int N = P.size();
    double answer = 0.;
    for (int n=0; n<N; ++n)
        answer += hypot( P[(n+1)%N].x - P[n].x, P[(n+1)%N].y - P[n].y );
    return answer;
}

// dvojnásobok obsahu mnohouholníka
// používame to ako test, či nejde o degenerovaný prípad
long long twice_area(const vector<point> &P) {
    int N = P.size();
    long long answer=0;
    for (int n=0; n<N; ++n) answer += cross( point(0,0), P[n], P[(n+1)%N] );
    return abs(answer);
}

int main() {
    int N; cin >> N;
    vector<point> P(N); for (int n=0; n<N; ++n) cin >> P[n].x >> P[n].y;
    sort(P.begin(), P.end());

    double answer = circumference(convex_hull(P));

    for (int a=0; a<N; ++a) for (int b=0; b<a; ++b) {
        // vyskúšame separovať body podľa priamky P[a]--P[b]
        vector<point> P1, P2;
        for (int i=0; i<N; ++i) {
            if (i==a) { P1.push_back(P[i]); continue; }
            if (i==b) { P2.push_back(P[i]); continue; }
            long long cp = cross(P[a],P[b],P[i]);
            if (cp <  0) P1.push_back(P[i]);
            if (cp >  0) P2.push_back(P[i]);
            if (cp == 0) {
                if (dot(P[a],P[b],P[i]) < 0) P1.push_back(P[i]);
                else P2.push_back(P[i]);
            }
        }
        vector<point> H1 = convex_hull(P1), H2 = convex_hull(P2);
        long long ta1 = twice_area(H1), ta2 = twice_area(H2);
        if (ta1>0 && ta2>0)
            answer = min( answer, circumference(H1)+circumference(H2) );
    }
    cout << setprecision(15) << answer << endl;
}
```

## A-I-3 Kaviarne

Pre $k = 1$ je riešenie ľahké: stačí spustiť prehľadávanie do šírky, začínajúc naraz vo všetkých možných kaviarňach. Takto nájdeme v čase $O(n^2)$ pre každú križovatku jej vzdialenosť od najbližšej kaviarne.

Pre všeobecné $k$ má úloha viacero možných riešení „hrubou silou“. Uvedieme dva príklady.

Pri jednom možnom riešení si na začiatku zostrojíme zoznam kaviarní. Následne pre každú križovatku $X$ prejdeme tento zoznam a pre každú kaviareň si (v konštantnom čase) spočítame jej vzdialenosť od $X$. Tieto vzdialenosti následne usporiadame a nájdeme $k$-tu najmenšiu z nich. Ak v tomto riešení použijeme efektívne všeobecné triedenie, dostávame časovú zložitosť $O(n^4 \log n)$. Ak namiesto neho použijeme CountSort (alebo prípadne priamo lineárny algoritmus na nájdenie $k$-teho najmenšieho prvku), dostávame riešenie s časovou zložitosťou $O(n^4)$.

V inom riešení každú križovatku spracujeme nasledovne: začneme s hodnotou $d = 0$, postupne ju zvyšujeme a zakaždým prezrieme všetky nové políčka, ktoré práve začali byť v dosahu. Prestaneme, keď nájdeme prvé $d$ pre ktoré v zodpovedajúcej oblasti leží dostatočne veľa kaviarní. Takéto riešenie má tiež časovú zložitosť $O(n^4)$: v najhoršom možnom prípade sa nám môže stať to, že pre každú z $n^2$ križovatiek musíme na nájdenie $k$ kaviarní prezrieť celú mapu.

Lepšie riešenia budú založené na tom, že nebudeme zbytočne zas a znova prezerať celú mapu políčko po políčku. V nasledujúcom texte si ukážeme jeden možný spôsob, ako si mapu Manhattanu *predspracovať*, aby sme potom pre ľubovoľnú konkrétnu križovatku a ľubovoľné $d$ vedeli v konštantnom čase povedať, koľko kaviarní je v dosahu.

**Prefixové súčty v 1D:**

Predstavme si, že máme obyčajné pole $A[0..n-1]$, ktoré obsahuje nejaké 0 a nejaké 1. Chceli by sme si ho predspracovať tak, aby sme následne pre každý úsek vedeli v konštantnom čase povedať, koľko jednotiek obsahuje.

Hľadané predspracovanie bude vyzerať nasledovne: Vyrobíme si nové pole $S[0..n]$. (Všimnite si, že $S$ má o prvok viac ako $A$.) Jeho obsah v lineárnom čase vypočítame nasledovne: začneme s $S[0] = 0$ a potom každé ďalšie $S[i+1]$ spočítame ako $S[i] + A[i]$. Pre takto vypočítané pole $S$ zjavne platí, že $S[i]$ je súčet prvých $i$ prvkov poľa $A$. Preto pole $S$ voláme poľom prefixových súčtov pre pole $A$.

Príklad:

```
index:   0   1   2   3   4   5   6
----------------------------------
    A:   1   2   3   4   4   5
    S:   0   1   3   6  10  14  19
```

Pomocou poľa $S$ teraz vieme v konštantnom čase určiť súčet ľubovoľného úseku poľa $A$: súčet prvkov na indexoch z polootvoreného intervalu[1] $[x, y)$ je rovný $S[y] - S[x]$.

Prečo je tomu tak? Lebo $S[y]$ je súčet prvkov na pozíciách z intervalu $[0, y)$, a od nich odčítame súčet tých, ktoré nechceme – teda prvkov na pozíciách z intervalu $[0, x)$.

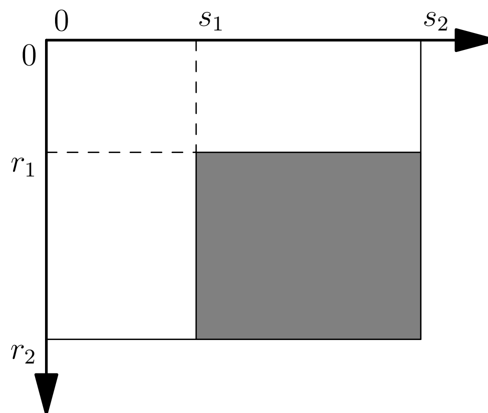
Príklad: Chceme zistiť súčet $A[2] + A[3] + A[4]$. Tomuto úseku poľa $A$ zodpovedá interval indexov $[2, 5)$, tento súčet je teda rovný $S[5] - S[2]$. Pre vyššie uvedené pole $A$ a jemu zodpovedajúce $S$ máme $A[2]+A[3]+A[4] = 3+4+4 = 11$ a zároveň $S[5] - S[2] = 14 - 3 = 11$.

**Prefixové súčty v 2D:**

Veľmi podobným spôsobom si vieme aj dvojrozmerné pole čísel $A[0..r - 1][0..s - 1]$ predspracovať tak, aby sme vedeli v konštantnom čase určiť súčet čísel v ľubovoľnej obdĺžnikovej oblasti.

To, čo si predpočítame, budú opäť v istom zmysle prefixové súčty: vyrobíme si pole $S[0..r][0..s]$, pre ktoré bude platiť, že $S[i][j]$ je súčet prvkov, ktoré ležia v prvých $i$ riadkoch a zároveň v prvých $j$ stĺpcoch poľa $A$. Inými slovami, $S[i][j]$ bude súčet tých $A[x][y]$, pre ktoré $x \in [0, i)$ a zároveň $y \in [0, j)$.

Ako pomocou takéhoto poľa $S$ určiť súčet čísel v ľubovoľnej obdĺžnikovej oblasti? To si najľahšie ukážeme na obrázku:




Zaujíma nás súčet vyfarbenej oblasti. To, čo priamo poznáme, je súčet od rohu $(0, 0)$ po pravý dolný roh vyfarbenej oblasti: ten udáva hodnota $S[r_2][s_2]$. Od nej ale potrebujeme odčítať súčet v nevyfarbených častiach. Súčet v ľavej časti (obdĺžniku s $r_2$ riadkami a $s_1$ stĺpcami) udáva hodnota $S[r_2][s_1]$. Podobne $S[r_1][s_2]$ je súčet v hornej časti. Všimnime si teraz hodnotu $S[r_2][s_2] - S[r_2][s_1] -$

[1] Číslo $x$ je najmenšie číslo, ktoré patrí do polootvoreného intervalu $[x, y)$; číslo $y$ je najmenšie číslo, ktoré doň už nepatrí.

$S[r_1][s_2]$. To je skoro presne to, čo chceme: od súčtu celého obdĺžnika sme odčítali tie časti, ktoré nechceme. Až na jeden problém: malý obdĺžnik vľavo hore sme odčítali nie raz ale dvakrát. To ešte potrebujeme napraviť – pripočítať ho k výsledku. Správny vzorec pre súčet vyfarbenej oblasti je teda nasledovný: $S[r_2][s_2] - S[r_2][s_1] - S[r_1][s_2] + S[r_1][s_1]$.

Druhou otázkou je, ako vlastne hodnoty v poli $S$ spočítať. A odpoveď je jednoduchá: presne rovnako! Ako by sme pomocou vyššie uvedeného vzorca určili hodnotu na políčku $(r, s)$?

$$A[r][s] = S[r+1][s+1] - S[r+1][s] - S[r][s+1] + S[r][s]$$

No ale toto môžeme upraviť do nasledovnej podoby:

$$S[r+1][s+1] = A[r][s] + S[r+1][s] + S[r][s+1] - S[r][s]$$

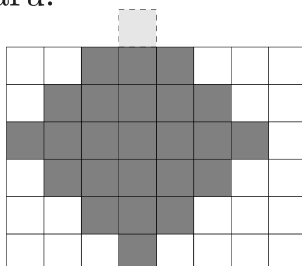
A to nie je nič iné ako vzorec, pomocou ktorého vypočítame $S[r+1][s+1]$ z predchádzajúcich hodnôt. (Skúste si nakresliť podobný obrázok ako sme ukázali vyššie a rozmyslieť si, čo počíta tento nový vzorec – z akých častí „naskladáme“ hodnotu $S[r+1][s+1]$?)

Ak teda máme pole rozmerov $r \times s$, takto ho vieme v čase $\Theta(rs)$ predspracovať a následne vieme v konštantnom čase určiť súčet ľubovoľnej obdĺžnikovej oblasti.

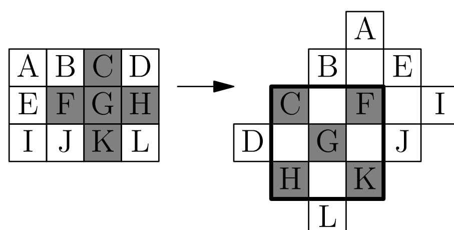
**Zrotujme si rovinu:**

V našej úlohe potrebujeme niečo podobné ako sme si práve ukázali: pre danú križovatku a dané $d$ potrebujeme vedieť rýchlo povedať, koľko kaviarní leží v príslušnom okolí danej križovatky.

Majme teda dvojrozmerné pole. Každé políčko tohto poľa bude zodpovedať jednej križovatke a bude tam zapísaná 0 alebo 1 podľa toho, či je na danej križovatke kaviareň. Keď si zvolíme konkrétne $d$ (v príklade nižšie je $d = 3$) a ideme spočítať kaviarne v dosahu z konkrétnej križovatky, potrebujeme zistiť súčet oblasti nasledovného tvaru:



Táto oblasť má síce v princípe tvar štvorca, len jeho strany bohužiaľ nie sú rovnobežné so súradnicovými osami. Vzorce pre dvojrozmerné prefixové súčty sa samozrejme dajú vhodne upraviť aj pre takto otočené štvorce. My si ale teraz ukážeme iný trik: otočíme si vhodne celý Manhattan a potom použijeme obyčajné dvojrozmerné prefixové súčty. Políčko, ktoré je teraz na súradniciach $(p, q)$, umiestnime namiesto toho na súradnice $(p+q, p-q)$. To, čo dostaneme, bude vyzerať nasledovne:




A ľahko nahliadneme, že po tejto jednoduchej transformácii konkrétnemu $d$ zodpovedá obyčajný štvorec rozmerov $(2d+1) \times (2d+1)$.

Na uloženie transformovanej tabuľky $n \times n$ potrebujeme približne pole rozmerov $(2n) \times (2n)$. Takéto predspracovanie má teda časovú zložitosť aj pamäťovú $O(n^2)$.

**Lepšie riešenia pôvodnej úlohy:**

Pomocou vyššie popísaného (alebo jemu podobného) predspracovania vieme teraz ľahko zlepšiť naše prvé riešenie súťažnej úlohy.

Pre každú križovatku platí, že optimálne $d$ je menšie ako $2n$. Uvažujme teraz pôvodné riešenie, ktoré pre každú križovatku postupne zväčšuje $d$. Ak namiesto hrubej sily použijeme naše predspracovanie, budeme konkrétne $d$ vedieť vyhodnotiť v konštantnom čase. Takéto riešenie bude mať časovú zložitosť $O(n^3)$, lebo pre každú z $n^2$ križovatiek spravíme nanajvýš $2n$ krokov. (Časová zložitosť predspracovania je zanedbateľná oproti časovej zložitosti druhej časti algoritmu.)

Toto riešenie môžeme ďalej vylepšiť. Stačí si uvedomiť, že namiesto sekvenčného skúšania všetkých možných $d$ môžeme optimálnu hodnotu určiť binárnym vyhľadávaním. Takéto riešenie teda získa odpoveď pre konkrétnu križovatku v čase $O(\log n)$. Jeho celková časová zložitosť je teda $O(n^2 \log n)$.

## Listing programu (C++)

```
// riešenie s časovou zložitosťou O( n^2 log n )
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
```

```
int K, N;
vector<vector<int> > A, SA, ans;

inline int sucet(int r, int s, int d) {
    int r1 = max(0,r+s-d),        c1 = max(0,r-s+N-1-d),
        r2 = min(2*N-1,r+s+d+1),  c2 = min(2*N-1,r-s+N-1+d+1);
    return SA[r2][c2] - SA[r2][c1] - SA[r1][c2] + SA[r1][c1];
}

int main() {
    cin >> K >> N;
    // načítame vstup, rovno počas načítania transformujeme súradnice
    A.resize( 2*N-1, vector<int>(2*N-1,0) );
    for (int r=0; r<N; ++r) for (int s=0; s<N; ++s) cin >> A[r+s][r-s+N-1];

    // spočítame 2D prefixové súčty
    SA.resize( 2*N, vector<int>(2*N,0) );
    for (int r=0; r<2*N-1; ++r)
        for (int s=0; s<2*N-1; ++s)
            SA[r+1][s+1] = SA[r+1][s] + SA[r][s+1] - SA[r][s] + A[r][s];

    // pre každú križovatku určíme odpoveď v log. čase binárnym vyhľadávaním
    ans.resize( N, vector<int>(N,0) );
    for (int r=0; r<N; ++r) for (int s=0; s<N; ++s) {
        if (A[r+s][r-s+N-1] >= K) continue; // pre toto políčko je odpoveď 0
        int lo=0, hi=2*N; // invariant počas vyhľadávania: lo < správna odpoveď <= hi
        while (hi - lo > 1) {
            if (sucet(r,s,(lo+hi)/2) >= K) hi=(lo+hi)/2;
            else lo=(lo+hi)/2;
        }
        ans[r][s] = hi;
    }
    for (int r=0; r<N; ++r)
        for (int s=0; s<N; ++s) cout << ans[r][s] << (s==N-1 ? "\n" : " ");
}
```

**Vzorové riešenie:**

Posledné z vyššie popísaných riešení je takmer optimálne. Teraz si ale ukážeme ešte lepšie riešenie. Jeho časová zložitosť bude $\Theta(n^2)$, a teda bude zjavne optimálna – vždy musíme aspoň na začiatku načítať celý vstup a na konci vypísať výstup, a už len na to potrebujeme $\Theta(n^2)$ krokov.

Oproti predchádzajúcim riešeniam budeme potrebovať ešte jedno nové pozorovanie. Predstavme si, že pre nejakú križovatku už vieme, že na to, aby sme mali v dosahu aspoň $k$ kaviarní, potrebujeme vzdialenosť $d$. Teraz nás bude zaujímať odpoveď pre susednú križovatku. Nakoľko sa môže líšiť od $d$?

Pomerne zjavný je nasledujúci argument: pre susednú križovatku nám určite stačí dosah $d+1$. Totiž môžeme prvým krokom prejsť na pôvodnú križovatku a odtiaľ na ďalších $d$ dosiahnuť ľubovoľnú z aspoň $k$ kaviarní. Teda nová hľadaná hodnota je nanajvýš o 1 väčšia ako tá, ktorú sme práve vypočítali.

Rovnakú úvahu však vieme spraviť aj opačným smerom – odpoveď pre pôvodnú križovatku je nanajvýš o 1 väčšia od odpovede pre novú, s ňou susediacu

križovatku. Dostávame teda nasledovný záver: v poli, ktoré máme vypísať na výstup, sa ľubovoľné dve susedné hodnoty líšia nanajvýš o 1.

A toto pozorovanie nám pomôže spracúvať križovatky v konštantnom čase. Ak už pre nejakú križovatku vieme, že odpoveď je $d$, tak pre jej susednú križovatku nemá zmysel binárne vyhľadávať odpoveď na celom intervale $[0, 2n)$. Namiesto toho stačí postupne vyskúšať odpovede $d-1$, $d$ a $d+1$.

Celkovo si teda toto nové riešenie môžeme zhrnúť nasledovne: Na začiatku nejako (pokojne aj vhodnou hrubou silou) zistíme správnu odpoveď pre križovatku v ľavom hornom rohu. Potom postupne po riadkoch spracúvame ostatné križovatky. Pre každú z nich sa pozrieme na susednú už spracovanú križovatku, čím získame odhad, akú odpoveď hľadať, a potom v konštantnom čase (pomocou 2D prefixových súčtov) dopočítame jej presnú hodnotu. Takto naozaj dostávame riešenie so sľubovanou časovou zložitosťou $O(n^2)$.

### Listing programu (C++)

```
// zmena na riešenie s časovou zložitosťou O( n^2 )

// pre každú križovatku okrem prvej určíme odpoveď v konštantnom čase
for (int r=0; r<N; ++r) for (int s=0; s<N; ++s) {
    if (r==0 && s==0) {
        while (sucet(r,s,ans[r][s])<K) ++ans[r][s];
    } else {
        int prevd = (s==0 ? ans[r-1][s] : ans[r][s-1]);
        for (int d=prevd-1; d<=prevd+1; ++d)
            if (d>=0 && sucet(r,s,d)>=K) { ans[r][s]=d; break; }
    }
}
```

### Prídavok na záver: obdĺžnikové vstupy:

V tejto úlohe bol Manhattan úmyselne štvorcový, aby sme vám uľahčili návrh algoritmu aj analýzu jeho časovej zložitosti. Úlohu však v rovnakej časovej zložitosti (lineárnej od veľkosti vstupu) vieme riešiť pre vstupy ľubovoľného obdĺžnikového tvaru.

Vysvetlime si najskôr, v čom je problém. Predpokladajme, že sme na vstupe dostali obdĺžnik rozmerov $r \times s$. Ak by sme implementovali vyššie popísané riešenie, dostali by sme časovú aj pamäťovú zložitosť $\Theta((r+s)^2)$. A teda pre ešte stále rozumne veľké, ale „podlhovasté“ vstupy (napr. pre $10^6 \times 10$) je toto riešenie už prakticky nepoužiteľné: $\Theta((r+s)^2)$ je rádovo horšia zložitosť ako teoreticky optimálna $\Theta(rs)$.

Existuje však aj viacero riešení, ktoré túto optimálnu časovú zložitosť dosahujú aj pre vstupy ľubovoľného obdĺžnikového tvaru. Jedna možnosť je predchádzajúce riešenie upraviť tak, aby sme si z tabuľky 2D prefixových súčtov

pamätali len podstatné časti. (Rozmyslite si, že pri tom záleží na tom, či vstup pri transformácii otočíme „o 45°“ doľava alebo doprava. Treba si vybrať tú správnu možnosť.)

Iné možné riešenie: Vystačíme si s jednorozmernými prefixovými súčtami, ale zato dvoch typov: budeme mať zvlášť prefixové súčty pre každú uhlopriečku idúcu doľava dodola a tiež zvlášť pre každú uhlopriečku idúcu doprava dodola. Takéto prefixové súčty vieme predpočítať v čase $\Theta(rs)$. Pomocou takýchto prefixových súčtov vieme v konštantnom čase ľubovoľný „šikmý štvorec“ o jedno zmenšiť, zväčšiť, posunúť doprava, aj posunúť dodola – a samozrejme si zakaždým prepočítať, koľko kaviarní teraz obsahuje.

## A-I-4 Mimozemské počítače

**Podúloha A (3 body):**

Mimozemšťania nám dodali sálový KSP, ktorého funkcia `kruznica(n,E)` rozhoduje problém existencie Hamiltonovskej kružnice v danom jednoduchom neorientovanom grafe. My pomocou nej chceme rozhodnúť, či daný graf $G$ obsahuje nejakú Hamiltonovskú cestu.

Najskôr sa zamyslime nad jednoduchšou verziou úlohy: ako pomocou nášho sálového KSP zistiť, či v grafe $G$ existuje Hamiltonovská cesta, ktorá začína v konkrétnom vrchole $u$ a končí v konkrétnom vrchole $v$?

Najskôr nesprávne riešenie: do grafu $G$ pridáme hranu $uv$ (ak tam ešte nie je) a na nový graf zavoláme funkciu `kruznica`.

Prečo je toto riešenie nesprávne? Preto, že v grafe $G$ pokojne mohla existovať už aj predtým iná Hamiltonovská kružnica, ktorá hranu $uv$ neobsahuje. Sálový KSP by pre takéto grafy vždy rozsvietil zelené svetlo, čo je zle – zďaleka nie každý takýto graf obsahuje Hamiltonovskú cestu z $u$ do $v$. (Napr. nech $G$ je kružnica idúca postupne cez vrcholy 0 až 5 a nech $u = 0$ a $v = 3$.)

Teraz opravené riešenie. Nech má $G$ práve $n$ vrcholov, očíslovaných 0 až $n - 1$. Pridáme do grafu nový vrchol $n$ a hrany $nu$ a $nv$. Na tento nový graf zavoláme funkciu `kruznica`.

Ľahko nahliadneme, že toto riešenie je skutočne korektné. Ak je v pôvodnom grafe Hamiltonovská cesta z $u$ do $v$, tak spolu s novými hranami $nu$ a $nv$ dostávame Hamiltonovskú kružnicu v novom grafe. A naopak, ak je v novom grafe Hamiltonovská kružnica, tá ide cez všetky vrcholy, a teda aj cez vrchol $n$. No

a keďže z toho idú len dve hrany ($nu$ a $nv$), musia obe ležať na tejto kružnici. No a zvyšok tejto Hamiltonovskej kružnice zodpovedá v pôvodnom grafe práve Hamiltonovskej ceste z $u$ do $v$.

Toto riešenie teraz ľahko upravíme na riešenie našej prvej súťažnej úlohy. Ak nám je jedno, medzi ktorými dvoma vrcholmi grafu $G$ Hamiltonovská cesta vedie, jednoducho nový vrchol $n$ spojíme so všetkými z nich.

Zdôvodnenie správnosti je rovnaké ako v minulom riešení: Ak pôvodný graf obsahoval Hamiltonovskú cestu, nový zjavne obsahuje Hamiltonovskú kružnicu. A naopak, ak nový obsahuje kružnicu, odstránime z nej vrchol $n$ a vidíme, že pôvodný graf musel obsahovať cestu.

```
def cesta(n,E):
    # na vstupe dostaneme graf G ako zoznam hran
    # pridame don hrany medzi povodnymi vrcholmi a novym vrcholom n
    E += [ (n,i) for i in range(n) ]
    # a zavolame funkciu kruznica(), ktora rozsvieti spravne svetlo
    kruznica(n+1,E)
```

**Podúloha B (3 body):**

A teraz naopak. Mimozemšťania nám dodali sálový KSP, ktorého funkcia `cesta(n,E)` rozhoduje problém existencie aspoň jednej Hamiltonovskej cesty v danom jednoduchom neorientovanom grafe. My pomocou nej chceme rozhodnúť, či daný graf $G$ obsahuje Hamiltonovskú kružnicu.

V študijnom texte sme si túto úlohu vyriešili na kufríkovom KSP. To bolo celkom pohodlné: postupne pre každú hranu sme sa kufríka na niečo opýtali, dozvedeli sme sa odpoveď a podľa tej sme pokračovali ďalej. Teraz ale tento luxus nemáme, funkciu `cesta` smieme zavolať len raz. Čo s tým?

Hamiltonovská kružnica je skoro to isté ako cesta, len navyše začína a končí v tom istom vrchole. My by sme sa teda radi opýtali nášho sálového KSP otázku typu: „Existuje v tomto grafe Hamiltonovská cesta z vrcholu 0 do vrcholu 0?“ To síce nevieme urobiť priamo, ale potrebný trik nebude vôbec zložitý.

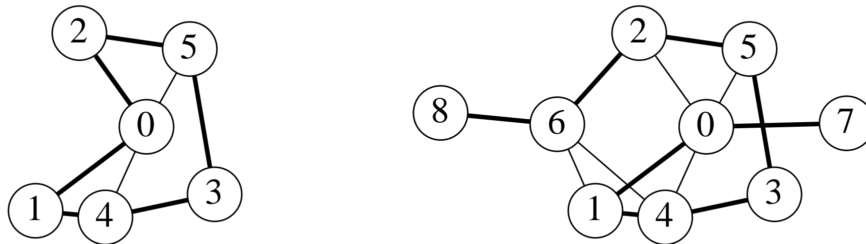
Najskôr si opäť ukážeme riešenie, ktoré síce ide správnym smerom, ale ešte nie je korektné. Do grafu $G$ pridáme nový vrchol $n$. Tento nový vrchol bude kópiou vrcholu 0: teda spojíme ho hranami s práve tými vrcholmi, s ktorými je spojený vrchol $n$. Na nový graf zavoláme funkciu `cesta`.

V čom je chyba tohto riešenia? V tom, že sme si nijak nevynútili, aby tá nájdená cesta viedla z vrcholu 0 do vrcholu $n$. Zoberme napríklad graf $G$, ktorý má $n = 3$ vrcholy a len dve hrany: 01 a 02. Tento graf zjavne Hamiltonovskú kružnicu neobsahuje, no keď zdvojíme vrchol 0, dostaneme graf obsahujúci Hamiltonovskú cestu (ba dokonca aj kružnicu).

Ako vyššie uvedené riešenie opraviť? Tak, že si vynútime, aby Hamiltonovská cesta viedla z 0 do $n$. Na to pridáme ešte dva nové vrcholy: $n+1$ a $n+2$. Vrchol $n+1$ spojíme s vrcholom 0 a vrchol $n+2$ s vrcholom $n$.

Čo sme takto dostali? V upravenom grafe $G'$ isto máme dva vrcholy stupňa 1: sú to $n+1$ a $n+2$. Keďže takýmto vrcholom nemôže Hamiltonovská cesta *prechádzať*, musí v jednom z nich začínať a v druhom končiť. Ak teda v $G'$ existuje nejaká Hamiltonovská cesta, tak na jej koncoch sú nutne hrany z $n+1$ do 0 a z $n+2$ do $n$. Zvyšok tejto Hamiltonovskej cesty teda tvorí cestu z vrcholu 0 do vrcholu $n$, idúcu práve raz cez každý z vrcholov 1 až $n-1$. A táto cesta zodpovedá Hamiltonovskej kružnici v pôvodnom grafe $G$.

Opačná implikácia je zjavná: ak $G$ obsahoval Hamiltonovskú kružnicu, tak z nej ľahko zostrojíme Hamiltonovskú cestu v nami zostrojenom $G'$.




*Vľavo graf $G$ s Hamiltonovskou kružnicou.*
*Vpravo z neho vyrobený $G'$ a zodpovedajúca Hamiltonovská cesta.*

```
def kruznica(n,E):
    # na vstupe dostaneme graf G ako zoznam hran
    # zostrojime si zoznam susedov vrcholu 0
    susedia0 = []
    for x,y in E:
        if x==0: susedia0.append(y)
        if y==0: susedia0.append(x)
    # do grafu pridame novy vrchol n, ktory je kopiou vrcholu 0
    E += [ (n,x) for x in susedia0 ]
    # a este dva nove vrcholy ktore sluzia ako konce cesty
    E += [ (0,n+1), (n,n+2) ]
    # na konci zavolame funkciu cesta(), ktora rozsvieti spravne svetlo
    cesta(n+3,E)
```

**Podúloha C (4 body):**

Mimozemšťania nám dodali kufríkový KSP, ktorého funkcia `je_trojfarbitelny(n,E)` rozhoduje problém existencie 3-farbenia v zadanom grafe. My sme na vstupe dostali 3-farbiteľný graf $G$ a chceme v polynomiálnom čase jedno platné ofarbenie zostrojiť.

Začneme riešením, ktoré je trošku náročnejšie na implementáciu, ale názornejšie; neskôr si ukážeme, ako podobné riešenie ľahšie implementovať.

Pridajme do grafu $G$ tri nové vrcholy, nazvime ich $a$, $b$, $c$. Tieto tri nové vrcholy spojíme každý s každým. Toto zjavne 3-farbiteľnosť $G$ nepokazilo. A zároveň vieme, že $a$, $b$ a $c$ musia mať navzájom rôzne farby. Bez ujmy na všeobecnosti nech $a$ dostane farbu 0, $b$ farbu 1 a $c$ farbu 2.

Zoberme teraz ľubovoľný vrchol $v$ grafu $G$. Vyskúšame postupne tri možnosti:

- $v$ spojíme hranami s vrcholmi $a$ a $b$
- $v$ spojíme hranami s vrcholmi $a$ a $c$
- $v$ spojíme hranami s vrcholmi $b$ a $c$

Pre každú možnosť zavolaním funkcie `je_trojfarbitelny` zistíme, či je dotyčný graf 3-farbiteľný. Keďže v pôvodnom grafe 3-farbenie existovalo, bude nutne existovať aspoň pre jeden z nových grafov. Ten si teda necháme (čím je zafixovaná farba vrcholu $v$ v každom platnom ofarbení) a pokračujeme ďalej.

Keď takto postupne spracujeme všetky vrcholy grafu $G$, máme zostrojené jeho platné ofarbenie. Dokopy sme potrebovali $3n$ volaní funkcie `je_trojfarbitelny` a pre každé z nich sme vstup zostrojili v polynomiálnom čase (pri vhodnej implementácii dokonca v konštantnom).

A teraz už to spomínané riešenie s ľahšou implementáciou. Tri pomocné vrcholy vôbec nepotrebujeme. Jednoducho stačí postupne pre každú dvojicu $x, y$ vrcholov grafu $G$ (a to v ľubovoľnom poradí) vykonať nasledujúcu operáciu: „ak pridanie hrany $xy$ do $G$ nepokazí jeho 3-farbiteľnosť, tak ju tam pridaj".

Keď celý tento cyklus dobehne, nutne skončíme s grafom $G'$, ktorý má (až na zámenu čísel farieb) jediné platné ofarbenie a platí: dva vrcholy majú rovnakú farbu vtedy a len vtedy, ak nie sú spojené hranou.

Prečo je to tak? Pozrime si ľubovoľnú dvojicu vrcholov, ktoré na konci nie sú spojené hranou. Túto dvojicu sme niekedy počas behu algoritmu spracúvali. Ak sme vtedy hranu medzi nimi do grafu nepridali, znamená to, že už vtedy vo všetkých prípustných ofarbeniach mali dotyčné dva vrcholy rovnakú farbu. No a jediné, čo robíme, je, že pridávame do grafu nové hrany, teda nové obmedzenia na ofarbenia vrcholov. Pri tom nám môžu platné ofarbenia len ubúdať, nikdy nepribudnú žiadne nové. Takže ani na konci behu algoritmu nesmú mať dotyčné dva vrcholy rôznu farbu.

Implementácia je skutočne jednoduchá:

```
def ofarbi(n,E): # na vstupe dostaneme graf G ako zoznam hran

    # pre kazdu dvojicu vrcholov skusime pridat hranu medzi nimi
    for x in range(n):
```

```
        for y in range(x):
            if je_trojfarbitelny(n,E + [(x,y)]):
                E += [(x,y)]

    # hrubou silou rozdelime vrcholy na farbu 0 a ostatne
    farba0 = [ x for x in range(n) if (not (0,x) in E) and (not (x,0) in E) ]
    farby12 = [ x for x in range(n) if not x in farba0 ]

    # a znova hrubou silou rozdelime ostatne na farbu 1 a 2
    if len(farby12) > 0:
        v1 = farby12[0]
        farba1 = [ x for x in farby12 if (not (v1,x) in E) and (not (x,v1) in E) ]
        farba2 = [ x for x in farby12 if not x in farba1 ]
    else:
        farba1, farba2 = [], []

    return (farba0,farba1,farba2)
```

(Pre názornosť je táto implementácia úmyselne neefektívna, lebo čo sa hodnotenia súťažnej úlohy týka, je to jedno. Uvedomte si ale napríklad, že v Pythone má pre zoznamy operátor `in` lineárnu časovú zložitosť. Tiež sa pri každom volaní funkcie `je_trojfarbitelny` zbytočne vytvára kópia celého zoznamu hrán. Rozmyslite si, ako by ste toto isté riešenie implementovali efektívnejšie.)

# Riešenia domáceho kola kategórie B

## B-I-1 Klince v doske

Položme si najskôr ľahšiu otázku: keby sme už poznali výšku $h$, ktorú majú mať zarovnané klince, ako spočítame, koľko takých klincov vieme vyrobiť?

Odpoveď je jednoduchá: Určite necháme na pokoji všetky klince, ktoré majú teraz výšku presne $h$. Klince, ktoré už teraz majú výšku menšiu ako $h$, použiť nemôžeme. No a spomedzi klincov, ktoré sú väčšie ako $h$, vieme na výšku $h$ zarovnať nanajvýš $u$ klincov. Dokopy teda bude mať výšku $h$ nasledovný počet klincov:

$$(\text{počet klincov výšky presne } h) + \min(\ u,\ (\text{počet klincov výšky väčšej ako } h)\ )$$

Toto nám dáva prvé riešenie našej úlohy: pre každú výšku od 0 po $\max h_i$ spočítame, koľko klincov takejto výšky vieme vyrobiť. Takéto riešenie zrejme získalo 4 body.

Do vzorového riešenia nám chýba spraviť dve pozorovania, ktoré nám pomôžu naše prvé riešenie zefektívniť. Prvým pozorovaním obmedzíme množinu výšok klincov, ktoré budeme skúšať. Ukáže sa totiž, že optimálna výška je určite rovná niektorej z výšok, ktoré majú klince na vstupe.

Prečo je to tak? Ukážeme si to najskôr na príklade. Predpokladajme, že optimálnym riešením pre nejaký vstup je zobrať klince výšok 10, 11, 12 a 13, a všetky tieto klince zarovnať na výšku 7. Rovnako dobre však môžeme tieto isté klince zarovnať na výšku 10 (a dokonca nám na to stačia 3 údery kladivom namiesto 4). Tento istý argument ľahko zovšeobecníme: keď si vyberieme, ktoré klince ideme zarovnať, optimálnym riešením je zarovnať ich všetky na výšku najkratšieho spomedzi nich.

Namiesto skúšania všetkých výšok od 0 po $h_i$ teda stačí vyskúšať tých (nanajvýš) $n$ výšok, ktoré sa naozaj vyskytujú v testovacom vstupe. Takéto riešenie teda spraví na vstupe s $n$ klincami rádovo $n^2$ krokov. Ak ste toto riešenie naprogramovali (a neboli ste ochotní nechať ho bežať dňom i nocou), zrejme ste získali 6 bodov.

Druhé pozorovanie nám pomôže zrýchliť výpočet toho, koľko klincov vieme zarovnať na danú výšku. Vo vyššie uvedenom riešení sme pre každú skúšanú

výšku zas a znova prešli celé pole klincov, aby sme zistili, koľko ich je rovnako vysokých a koľko vyšších. Toto však vieme zisťovať omnoho efektívnejšie. A nebude to ani tak náročné, úplne bude stačiť, keď budeme všetko robiť trochu systematickejšie.

Začneme tým, že si klince *usporiadame* podľa výšky. A v tomto poradí, začínajúc napríklad od najvyšších, budeme aj jednotlivé výšky kontrolovať. Pre každú výšku takto vieme v konštantnom čase povedať, koľko klincov je od nej vyšších. Príklad: nech usporiadané pole s výškami klincov vyzerá nasledovne:

```
index:  0  1  2  3  4  5  6  7  8  9 10
výška:  3  4  5  7  7  7  8  9  9 11 47
                  ^     ^
```

Šípkami je označený úsek klincov, ktoré majú výšku 7. V poli sa nachádzajú na indexoch 3 až 5. No a keďže úplne posledný klinec je na indexe 10, vieme rovno povedať, že v tomto poli je presne $10 - 5 = 5$ klincov vyšších ako 7.

Riešenie sa bude teda skladať z dvoch častí: najskôr pole výšok klincov usporiadame, a následne ho raz prejdeme a pre každú výšku spočítame, koľko klincov danej výšky vieme vyrobiť. Druhú časť riešenia vieme implementovať v čase priamo úmernom počtu klincov, teda lineárnom od $n$. Prvá časť je o čosi pomalšia: aj pri použití efektívneho algoritmu na triedenie (napr. Mergesort alebo Heapsort) bude náš program potrebovať spraviť rádovo $n \log n$ krokov. Samozrejme, to je ešte stále veľmi málo v porovnaní s $n^2$ krokmi – aj pre $n = 1\,000\,000$ triedenie s časovou zložitosťou $n \log n$ prebehne za pár sekúnd.

V nasledujúcej implementácii (v jazyku C++) používame na triedenie knižničnú funkciu `sort()`. Následne prejdeme klincami od konca, pričom pre každý klinec zistíme, koľko je v poli od neho doprava rovnako veľkých a koľko väčších klincov. Ušetríme si tak prácu s delením poľa na úseky klincov rovnakej výšky. Rozmyslite si, že optimálne riešenie nájdeme, keď budeme spracúvať najľavejší z klincov správnej výšky.

## Listing programu (C++)

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

int main() {
    for (int kolo=0; kolo<6; ++kolo) {
        // načítame a usporiadame vstup
        int N; cin >> N;
        int U; cin >> U;
```

```
        vector<int> H(N);
        for (int n=0; n<N; ++n) cin >> H[n];
        sort( H.begin(), H.end() );

        int vacsich = 0, rovnych = 0;
        int optimalny_pocet = 0, optimalna_vyska = 0;
        for (int n=N-1; n>=0; --n) {
            // postupne spracúvame klince od najväčšieho
            // pamätáme si, koľko väčších a koľko rovnakých sme videli
            if (n==N-1 || H[n]<H[n+1]) {
                vacsich+=rovnych; rovnych=1;
            } else {
                ++rovnych;
            }
            int pocet = rovnych + min( U, vacsich );
            if (pocet > optimalny_pocet) {
                optimalny_pocet=pocet;
                optimalna_vyska=H[n];
            }
        }
        cout << optimalny_pocet << endl;
    }
}
```

Na záver si ešte ukážeme inú implementáciu v jazyku C++. V tej použijeme pokročilú dátovú štruktúru (`map`) na to, aby sme si priamo ku každej výške zapamätali počet klincov, ktoré ju majú.

## Listing programu (C++)

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

int main() {
    for (int test=0; test<6; ++test) {
        // v mape si ku každej výške spočítame počet klincov, ktoré ju majú
        int N, U; cin >> N >> U;
        map<int,int> vysky;
        for (int n=0; n<N; ++n) { int h; cin >> h; ++vysky[h]; }

        // postupne od najmenšej po najväčšiu spracúvame všetky výšky klincov,
        // ktoré sa vyskytli na vstupe
        int primale = 0;
        int optimalny_pocet = 0, optimalna_vyska = 0;
        for (auto rec : vysky) {
            int vyska, rovnake;
            tie(vyska,rovnake) = rec;
            int vacsie = N - primale - rovnake;
            int pocet = rovnake + min( U, vacsie );
            if (pocet > optimalny_pocet) {
                optimalny_pocet=pocet;
                optimalna_vyska=vyska;
            }
            primale += rovnake;
        }
        cout << optimalny_pocet << " " << optimalna_vyska << endl;
    }
}
```

## B-I-2 Prvočísla z magnetiek

Riešenie tejto úlohy sa skladá z dvoch samostatných logických krokov: Za prvé, potrebujeme vedieť vygenerovať všetky čísla, ktoré sa z daných kartičiek dajú poskladať. A za druhé, potrebujeme o každom z nich vedieť zistiť, či ide o prvočíslo. Pozrime sa najskôr na druhú časť úlohy.

**Testovanie prvočíselnosti:**

Najjednoduchšia možnosť ako otestovať, či je číslo prvočíslom, vychádza priamo z definície: číslo $p$ je prvočíslom vtedy a len vtedy, ak ho nedelí žiadne z čísel 2 až $p-1$.

Takýto test je však pre väčšie čísla už príliš pomalý. Najväčšie číslo, ktoré vieme vytvoriť z magnetiek v poslednom vstupe, je číslo 77 774 444 221. Otestovanie tohto jediného čísla by nám na priemernom súčasnom počítači trvalo rádovo 15 minút. A to je predsa len vcelku dosť – hlavne preto, že my tých čísel budeme potrebovať otestovať veľa.

Našťastie tento test vieme veľmi výrazne urýchliť jednoduchým pozorovaním: delitele čísla vždy prichádzajú v pároch. Nech $d_1$ delí $n$. Označme $d_2 = (n/d_1)$. Keďže $d_1$ delí $n$, je aj $d_2$ prirodzené číslo. Jeho definíciu vieme zapísať aj v tvare $d_1 \cdot d_2 = n$. A z tohto zápisu je zjavné, že aspoň jeden z činiteľov $d_1$ a $d_2$ je menší alebo rovný $\sqrt{n}$. (Keby boli oba činitele väčšie ako $\sqrt{n}$, ich súčin by bol väčší ako $n$.)

Inými slovami, platí nasledujúca implikácia: „AK $n$ nie je prvočíslo, TAK existuje deliteľ čísla $n$, ktorý má veľkosť nanajvýš $\sqrt{n}$." Namiesto toho, aby sme skúšali delitele od 2 po $n-1$, stačí skúšať delitele od 2 po $\lfloor\sqrt{n}\rfloor$. Takto vylepšeným algoritmom už na bežnom počítači zvládneme za milisekundu otestovať prvočíselnosť ľubovoľného z čísel, ktoré môžeme v tejto úlohe stretnúť.

```
bool je_prvocislo(long long n) {
    if (n < 2) return false;
    for (long long d=2; d*d<=n; ++d)
        if (n%d==0) return false;
    return true;
}
```

(Všimnite si v programe, že sme nepočítali odmocninu z $n$ pomocou reálnych čísel. Namiesto toho sme použili ekvivalentnú podmienku `d*d<=n`, vyhodnotenie ktorej vieme spraviť presne v celých číslach.)

**Generovanie všetkých zložiteľných čísel:**

Ukážeme si niekoľko rôznych postupov.

Prvý bude založený na prístupe, kedy postupne po jednej pridávame magnetky. Budeme pri ňom pracovať s reťazcami, lebo tie sa ľahšie spájajú a roz-

pájajú. Začneme s tým, že zoberieme prvú magnetku. Pomocou tej vieme zložiť práve jeden možný reťazec: ak je napr. na magnetke cifra 7, vieme zložiť len reťazec `"7"`.

Teraz sa pozrime na všeobecný krok. Už sme spracovali niekoľko magnetiek a poznáme všetky reťazce, ktoré sa z nich dajú zložiť. Príklad: `"117"`, `"171"`, a `"711"`. Teraz zoberieme ďalšiu magnetku, napr. s cifrou 4. Ako sa zmení množina reťazcov, ktoré vieme zostrojiť? Musíme vyskúšať všetky možnosti: postupne zoberieme všetky reťazce, ktoré sme doteraz mali, a do každého z nich skúsime na každú pozíciu vložiť novú cifru. V našom príklade z prvého starého reťazca dostaneme nové reťazce `"1174"`, `"1147"`, `"1417"` a `"4117"`, z druhého reťazce `"1714"`, `"1741"`, `"1471"` a `"4171"`, a z tretieho starého reťazca nové reťazce `"7114"`, `"7141"`, `"7411"` a `"4711"`.

Ak sa nám na magnetkách opakujú cifry, vieme toto riešenie ešte zefektívniť tým, že po spracovaní každej novej cifry spomedzi vygenerovaných reťazcov vyhádžeme duplikáty. Napr. keď skúsime do reťazca `"111"` na všetky možné pozície pridať novú cifru 1, dostaneme štyri kópie reťazca `"1111"`. Nám ale stačí pamätať si každý možný reťazec len raz.

V nasledujúcej ukážke implementácie používame v C++ dátovú štruktúru `set`, ktorá duplikáty sama zahodí a navyše nám rovno všetky čísla (uložené ako reťazce) usporiada.

```
// v poli D[N] mame jednotlive cifry ako znaky
set<string> moznosti;
moznosti.insert( D[0] );

for (int n=1; n<N; ++n) {
    set<string> nove_moznosti;
    for (string cislo : moznosti) {
        for (unsigned i=0; i<=cislo.size(); ++i) {
            nove_moznosti.insert( cislo.substr(0,i) + D[n] + cislo.substr(i) );
        }
    }
    moznosti = nove_moznosti;
}
```

Druhou možnosťou je použiť rekurziu. Rekurzívna funkcia ako parameter dostane aktuálne poskladaný začiatok čísla. V globálnom poli si navyše budeme pamätať, koľko kusov ktorej cifry nám aktuálne ostáva nepoužitých. Ak už nemáme žiadne nepoužité cifry, práve sme vygenerovali jednu možnosť a spracujeme ju (t.j. otestujeme, či ide o prvočíslo). V opačnom prípade postupne skúsime na koniec aktuálneho čísla pridať každú z nepoužitých cifier, a zakaždým sa rekurzívne zavoláme s novým číslom. (Keď máme číslo $n$ a na jeho koniec pridáme cifru $d$, dostaneme nové číslo s hodnotou $10n + d$.)

```
int zostava[10];
```

```
void generuj(long long cislo) {
    bool koniec = true;
    for (int d=0; d<10; ++d)
        if (zostava[d]>0) {
            koniec=false;
            --zostava[d];
            generuj(10*cislo+d);
            ++zostava[d];
        }
    if (koniec && je_prvocislo(cislo)) cout << cislo << endl;
}

int main() {
    int N; cin >> N;
    for (int i=0; i<10; ++i) zostava[i] = 0;
    for (int n=0; n<N; ++n) {
        int d; cin >> d;
        ++zostava[d];
    }
    // v nasledujucom cykle zaciname s d=1, lebo na zaciatok cisla nemozeme dat nulu
    for (int d=1; d<10; ++d)
        if (zostava[d]>0) { --zostava[d]; generuj(d); ++zostava[d]; }
}
```

Treťou z mnohých možností je všetky možné permutácie cifier generovať postupne v tzv. *lexikografickom poradí.* (Inými slovami, generovať všetky čísla, ktoré sa z daných číslic dajú zložiť, v poradí od najmenšieho po najväčšie.) V niektorých jazykoch na to máme priamo funkciu: napr. v C++ existuje funkcia `next_permutation`. (V Pythone existuje `itertools.permutations`, tá ale nevie preskočiť duplikáty.) A ak ju aj nemáme, nie je vôbec ťažké si ju naprogramovať. Na príklade si ukážeme, ako na to.

Predstavme si napríklad, že sme práve vygenerovali číslo 19476313. Ako vyzerá najbližšie väčšie číslo, ktoré je tvorené tými istými ciframi? To je ešte zjavné: 19476331. Ale čo teraz, ako vyzerá to ďalšie v poradí? Radi by sme zachovali čo najviac cifier na začiatku nedotknutých. Lenže už zjavne neexistuje žiadne ďalšie takéto číslo, ktoré by začínalo 194..., lebo cifry na nasledujúcich miestach (76331) sú už tvoria najväčšie možné číslo – sú usporiadané od najväčšej po najmenšiu.

Hľadáme teda číslo tvaru $19x\dots$, kde $x > 4$. Samozrejme, $x$ musí byť niektorá z cifier, ktoré máme k dispozícii. Najmenšia z nich, ktorá je väčšia ako 4, je 6. Nasledujúce číslo bude teda začínať ciframi 196. Ostávajú nám nepoužité cifry 1, 3, 3, 4 a 7. Z tých už môžeme postaviť čo len chceme, výsledok bude vždy väčší ako 19476331. A keďže chceme najmenšie z týchto čísel, nepoužité cifry pridáme od najmenšej po najväčšiu.

Dostávame teda, že najbližšie číslo väčšie ako 19476331, ktoré je tvorené presne tými istými ciframi, je číslo 19613347.

Tu je ukážka, ako v C++ priamo použiť funkciu `next_permutation`:

```
// nacitame pocet cifier a jednotlive cifry (usporiadane od najmensej po najvacsiu) do pola
int N; cin >> N;
vector<int> D(N);
for (int n=0; n<N; ++n) cin >> D[n];

do {
    // ideme spracovat cislo reprezentovane polom D

    // cisla zacinajuce nulou preskocime
    if (D[0]==0) continue;

    // poskladame cislo z jeho cifier
    long long cislo = 0;
    for (int d : D) cislo = 10*cislo+d;

    // ak je prvocislom, vypiseme ho
    if (je_prvocislo(d)) cout << cislo << endl;

    // skusime vygenerovat nasledujucu permutaciu pola D,
    // ukoncime cyklus ked sa nam to uz nepodari

} while (next_permutation(D.begin(),D.end()));
```

## B-I-3 Pády domín

Skôr ako sa pustíme do riešenia tejto úlohy, urobme si jedno zjednodušenie. Aby sme nemuseli riešiť okraje poľa, teda čo sa stane, keď domino zhodí všetky ostatné, ktoré sú naľavo/napravo od neho, pridajme si na začiatok aj koniec poľa veľké číslo[2], ktoré určite zastaví pád ľubovoľného iného domina. Takto sa nám s tým bude robiť oveľa lepšie.

**Jednoduché riešenie:**

Skúsme teraz vymyslieť nejaké jednoduché riešenie, ktoré nám získa aspoň niekoľko bodov. Prvé čo nás napadne je skúsiť si pády domín simulovať. Ak chceme vedieť, koľko domín zhodí domino $i$ s váhou $v_i$, keď ho postrčíme doľava, môžeme sa pozerať, ktoré dominá zhodí. Najskôr sa pozrieme na domino $i-1$. Ak je $v_{i-1} > v_i$, teda domino naľavo je ťažšie ako zhodené domino, celý pád sa zastaví. V opačnom prípade spadne aj domino $i-1$, čo znamená, že sa musíme pozrieť na ďalšie domino a to je $i-2$. Postupne sa budeme pozerať na dominá čoraz viac naľavo až kým sa celý pád nezastaví na domine $j$ ($v_j > v_i$). Vieme, že všetky dominá medzi $j$ a $i$ museli byť ľahšie ako $v_i$ (inak by sa pád zastavil

[2]Ono by bolo pekné, keby sme si tam vedeli dať hodnotu nekonečno. Keďže však programovacie jazyky takú hodnotu zväčša nepoznajú, musíme sa uspokojiť s konečnou hodnotou. To zase nie je taký problém – ak poznáme rozsah spracúvaných hodnôt, môžeme ako nekonečno použiť hocijakú od neho väčšiu hodnotu. A ak aj nie, môžeme si nájsť maximum $m$ daného poľa a ako nekonečno použiť hodnotu $m+1$.

už skôr) a teda $i$-te domino zhodilo naľavo $i - j$ domín.

Túto hodnotu si viem takto vyrátať pre všetky hodnoty $i$ a rovnakým spôsobom, len v opačnom smere, si viem pre každé domino zistiť aj to, koľko domín zhodí, keď ho postrčíme doprava. Porovnaním týchto dvoch hodnôt vieme priamo zisťovať výsledok.

Jediný problém je časová zložitosť. Pre každý prvok poľa musíme ísť doľava aj doprava, až kým nenájdem väčší prvok, ktorý sa môže nachádzať až úplne na konci. Preto sa budeme musieť posunúť až $n$-krát pre každý prvok, z čoho dostávam časovú zložitosť $O(n^2)$. Toto nám však zaručí solídnych 5 bodov, čo je v pomere k vynaloženej námahe pomerne veľa.

**Vzorové riešenie:**

Samozrejme, s predchádzajúcim riešením sa neuspokojíme, pokúsme sa ho teda vylepšiť. Pozrime sa ako pracoval náš predchádzajúci program, keď sme hľadali počet zhodených domín po strčení doľava.

Ak sme zhodili domino $i$, to padalo a zhadzovalo ostatné dominá, až kým sa nezastavilo na domine $j$. Skúsme doľava zhodiť domino $i + 1$. Môže sa stať, že $v_i > v_{i+1}$. V tomto prípade sa pád zastaví okamžite a nezhodí sa už žiadne ďalšie domino.

Ak je však $i$-te domino ľahšie, spadne a my by sme mali pokračovať a pozerať sa na domino $i-1$. To však nie je potrebné, najbližšie domino, na ktoré sa musím pozrieť, je domino $j$. Totiž všetky dominá medzi $j$ a $i$ spadli, keď sme zhadzovali domino $i$. To znamená, že museli byť ľahšie (alebo rovnako ťažké) ako domino $i$. A to je predsa ľahšie ako domino $i + 1$. To znamená, že môžem preskočiť všetky tieto už spadnuté dominá a pozrieť sa priamo na domino $j$. Toto domino môže spadnúť tiež a budeme sa musieť pozerať ďalej, opäť však platí, že môžem preskakovať prvky, ktoré domino $j$ zhodilo. Takto sa dostanem až k dominu, ktoré je ťažšie.

Uvedomme si, že ak raz zhodím nejaké domino, už sa naň nikdy nemusím pozrieť, lebo domino, ktoré ho zhodilo buď ďalší pád zastaví a k nemu sa nedostaneme, alebo spadne aj ono, čo znamená, že spadne aj toto domino. To znamená, že mám časovú zložitosť $O(n)$, lebo každé domino spadne najviac raz a potom ho môžem ignorovať. Opäť si uvedomme, že tento prístup platí aj keď strkáme dominá doprava, akurát musíme spracúvať dominá v opačnom poradí.

Implementácia tohoto riešenia je opäť veľmi jednoduchá. Pre každé $i$ si budeme chcieť vyrátať, ktoré domino zastaví jeho pád doľava. Označme si túto hodnotu $l(i)$. Ak vieme tieto hodnoty pre všetky čísla menšie rovné $i$, ľahko to vieme zrátať pre $i+1$. Pozrieme sa, či $v_i \leq v_{i+1}$. Ak áno, môžeme preskočiť do-

miná až po pozíciu $l(i)$. Opäť sa pozrieme, či $v_{l(i)} \leq v_{i+1}$ a ak to platí, skočíme na $l(l(i))$, atď.. Prvá pozícia, na ktorej neprejde daná nerovnica bude hodnota $l(i+1)$. Zároveň sme si týmto vyrátali, koľko domín zhodí $i$-te domino naľavo – bude to hodnota $i - l(i)$. Tento postup potom zopakujeme aj pre pravé strany a na konci pre každé $i$ porovnám výsledné hodnoty.

Všimnite si, ako sa táto implementácia podobá na predchádzajúce triviálne riešenie. Hlavný rozdiel je len v menení premennej $j$.

### Listing programu (C++)

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

int main() {
    int n;
    cin >> n;
    vector<int> D(n+2);
    D[0] = D[n+1] = 1000000000; // pridaj nekonečno na začiatok a na koniec
    for(int i=1; i<=n; i++) cin >> D[i];

    //v L[i] sa zastaví pád i-teho domina doľava, v R[i] jeho pád doprava
    vector<int> L(n+2), R(n+2);
    for(int i=1; i<=n; i++) {
        //pozerám sa doľava
        int j=i-1;
        while(D[j]<=D[i]) j=L[j];
        L[i]=j;
    }
    for(int i=n; i>=1; i--) {
        //pozerám doprava
        int j=i+1;
        while(D[j]<=D[i]) j=R[j];
        R[i]=j;
    }
    for(int i=1; i<=n; i++)
        if(i-L[i]==R[i]-i) cout << "=";
        else if(i-L[i]>R[i]-i) cout << "<";
        else cout << ">";
    cout << endl;
}
```

### Alternatívna implementácia:

Existuje aj iný spôsob, ako sa dá naprogramovať vyššie spomínané vzorové riešenie a to pomocou dátovej štruktúry `stack` (po slovensky zásobník). Táto dátová štruktúra funguje veľmi jednoducho – keď chcem do nej prvok pridať, hodím ho na samý vrch, keď prvok odstraňujem, odstraňujem vždy prvok navrchu, takže ten, čo bol pridaný najneskôr.

V tomto stacku si budem udržiavať dominá, ktoré mi ešte nespadli, presnejšie ich váhu a pozíciu. Keď hľadám, pokiaľ bude padať $i+1$-vé domino, pozriem sa na vrch stacku. Ak obsahuje domino ťažšie, pád sa zastaví a viem odpoveď na

svoju otázku. V opačnom prípade vyhodím vrchný prvok zo stacku. Toto spraviť môžem, lebo to znamená, že dané domino bolo zhodené $i+1$-vým dominom a teda ho už nepotrebujem – neskončí na ňom žiaden ďalší pád.

Vrch stacku vyhadzujem, až kým neobjavím väčší prvok, vtedy prestanem, zaznačím si výsledok a na vrch stacku pridám $i+1$-vé domino. To totiž ešte nespadlo a potrebujem s ním do budúcna rátať. Uvedomte si, že prvky v stacku sú usporiadané klesajúco podľa váhy. Môžete si pozrieť moju implementáciu využívajúcu stack a sami posúďte, ktorá sa vám páči viac. (V listingu uvádzame len časť, ktorá sa zmenila oproti predchádzajúcemu riešeniu.)

## Listing programu (C++)

```
#include <stack>

//pozerám doľava
{
    stack<int> S;
    S.push(0);
    for(int i=1; i<=n; i++) {
        while(D[S.top()]<=D[i]) S.pop();
        L[i]=S.top();
        S.push(i);
    }
}
//pozerám doprava
{
    stack<int> S;
    S.push(n+1);
    for(int i=n; i>=1; i--) {
        while(D[S.top()]<=D[i]) S.pop();
        R[i]=S.top();
        S.push(i);
    }
}
```

**Iné riešenia:**

Úloha sa dá riešiť aj principiálne inými spôsobmi.

Prvé riešenie (to s kvadratickou časovou zložitosťou) vieme vylepšiť nasledovne: namiesto toho, aby sme skúšali zhadzovať dominá najskôr smerom doľava a potom smerom doprava si predstavíme, že oba smery začali padať naraz. Striedavo sa teda pozrieme na ďalšie domino doľava a doprava a so simuláciou prestaneme, akonáhle jeden smer zastane.

Takéto riešenie je zjavne nanajvýš také pomalé ako pôvodné kvadratické riešenie – môžeme oproti nemu len ušetriť. To sa nám podarí vždy, keď nebola odpoveď **=**, my totiž nebudeme tú stranu, na ktorú spadne viac domín, simulovať celú. Ešte stále existujú vstupy, pre ktoré je aj toto riešenie kvadratické – napríklad $n$ rovnako veľkých domín. Dá sa však (pomerne komplikovane) dokázať, že

ak sú všetky dominá navzájom rôzne veľké, tak sa toto jednoduché zlepšenie algoritmu zaručene výrazne prejaví na časovej zložitosti. Tá totiž klesne dokonca až na $O(n \log n)$.

Ešte úplne iné riešenie s časovou zložitosťou $O(n \log n)$ vieme dosiahnuť pomocou nasledovnej myšlienky: dominá si usporiadame od najmenšieho po najväčšie a následne ich v tomto poradí začneme ukladať na stôl (ktorý je na začiatku prázdny) na miesta, kam patria. Vždy, keď pridávame domino, pozrieme sa, koľko domín už stojí bezprostredne naľavo od jeho miesta (tie by nové domino zhodilo padajúc doľava) a koľko ich stojí bezprostredne napravo. Keď si tieto dĺžky úsekov vhodne pamätáme (v poli, pre každý úsek domín si pamätáme jeho dĺžku aj na pozícii, kde začína, aj na pozícii, kde končí), tak vieme každé domino pridať v konštantnom čase. Najpomalšou časťou tohto riešenia je teda usporiadanie domín.

## B-I-4 Hľadáme v poli

**Borisov program:**

Začneme od najľahšieho programu, v ktorom je dier viac ako vo švajčiarskom syre. Reč je o Borisovom programe. Ten si stačilo skúsiť spustiť na niekoľkých náhodných vstupoch a skoro určite ste pri tom natrafili na nejaký, na ktorom sa tento program zacyklil a nikdy neskončil.

Myšlienka tohto programu je pritom správna: Boris sa snažil naprogramovať *binárne vyhľadávanie*. Nepodarilo sa mu to ale. Binárne vyhľadávanie je notoricky známe tým, že keď si nedáte pozor, ľahko sa (tak ako Boris) pomýlite v niektorom indexe o $\pm 1$ a problém je na svete.

Už pre niektoré dvojprvkové polia Borisov program nefunguje. Zoberme si napríklad `A=(10,20)` a `X=15`. Čo sa stane v Borisovom programe? Premenné `prvy` a `posledny` inicializujeme na 0 a 1. Následne vypočítame hodnotu `stredny` pomocou vzťahu `stredny := (prvy + posledny) div 2`, čiže `stredny` bude tiež 0. Teraz sa pozrieme na prvok `A[stredny]` a porovnáme ho s `X`. Keďže je menší, vykoná sa príkaz `prvy := stredny`, čiže… čiže sa vlastne nestane nič. Naďalej bude `prvy` rovné nule. A potom začne nová iterácia while-cyklu: opäť spočítame, že `stredny` je nula, opäť spravíme to isté porovnanie, pochopiteľne s tým istým výsledkom, a tak dokola až do nekonečna.

**Cilkin program:**

Kontrast ku predchádzajúcemu riešeniu prináša Cilkin program. Ten je síce

založený na pomalšej myšlienke, ale zato je funkčný. Čo že to Cilka robí vo svojom programe? Na začiatku je podmienka `if (X < A[0]) then exit;` Ak toto nenastane, má Cilka istotu, že `X` je aspoň také veľké ako prvok na indexe 0.

Všimnime si najskôr, čo by sa stalo, keby sme spustili len druhú polovicu Cilkinho programu:

```
kde := 0;
krok := 1;
while (kde+krok<N) and (A[kde+krok] <= X) do kde := kde+krok;
```

Alebo teda ekvivalentne:

```
while (kde+1<N) and (A[kde+1] <= X) do kde := kde+1;
```

Toto je len obyčajný cyklus, ktorý prechádza všetkými prvkami poľa, kým sú menšie alebo rovné od hľadaného `X`. Na poslednom takomto prvku (alebo na konci poľa) tento cyklus zastane.

No a ak sa niekde v poli hodnota `X` nachádza, tak to musí byť práve na mieste, kde vyššie popísaný cyklus zastal – tie za ním sú priveľké, tie pred ním zase všetky od neho menšie, a teda ostro menšie ako `X`. Takže na konci už len skontrolujeme, či `A[kde] = X` a podľa toho buď povieme, že `X` leží na pozícii `kde`, alebo že sa v poli nenachádza.

Takýto program by teda bol funkčný, ale bol by pomalý – v najhoršom možnom prípade by postupne prvok po prvku prešiel celé pole `A`.

Teraz sa zamyslime, čo by sa stalo, keby sme zmenili hodnotu `krok` napríklad na 100:

```
kde := 0;
while (kde+100<N) and (A[kde+100] <= X) do kde := kde+100;
```

Do poľa `A` sa teraz budeme pozerať len na každý stý prvok. No a rovnako ako predtým, aj teraz tento cyklus zastane na poslednom z týchto prvkov, ktorý je menší alebo rovný `X`. Toto samozrejme nemusí byť prvok, ktorý hľadáme – ale vieme, že o 100 prvkov ďalej je už prvok väčší ako `X`. A teda nám už stačí pozrieť sa len na nasledujúcich nanajvýš 99 prvkov.

Celý funkčný program by teda mohol vyzerať napr. nasledovne:

```
kde := 0;
while (kde+100<N) and (A[kde+100] <= X) do kde := kde+100;
while (kde+1  <N) and (A[kde+1  ] <= X) do kde := kde+1;
```

Najskôr „ideme po poli s krokom 100“, a keď to už nejde, ideme ďalej s krokom 1, až kým nenájdeme naozaj posledný z prvkov neprevyšujúcich `X`.

Ak takýto program spustíme na $n$-prvkovom poli, pozrieme sa určite na menej ako $(n/100) + 100$ jeho prvkov.

Samozrejme, namiesto konštanty 100 si môžeme zvoliť nejakú inú, vhodnejšiu. Ak by sme napríklad mali $n = 1\,000\,000$, je 100 najlepšia možná konštanta? Zjavne nie: ak pôjdeme s krokom 100, spravíme v najhoršom možnom prípade približne $10\,000 + 100$ prístupov do poľa. Lepšie by bolo zobrať krok 1000 a spraviť len $1000 + 1000$ prístupov do poľa.

No a Cilka prišla na to, akú dĺžku kroku je najlepšie zobrať pre pole dĺžky $n$: ako prvú hodnotu premennej `krok` zoberie hodnotu $\lceil \sqrt{n}\, \rceil$. Takto aj v najhoršom možnom prípade Cilkin program spraví len nanajvýš $2\sqrt{n}$ prístupov do poľa `A`: najskôr nanajvýš $\sqrt{n}$ krokov dĺžky $\sqrt{n}$ a následne nanajvýš $\sqrt{n}$ krokov dĺžky 1.

**Andrejov program:**

Na záver sme si nechali Andrejov program. Toho myšlienka je tiež v princípe šikovná. Podobným spôsobom by sme napríklad kedysi, keď ešte boli telefónne zoznamy papierové, hľadali konkrétneho človeka: keď viete, že sa volá napríklad Wagner, nebudete zbytočne otvárať zoznam v strede, ale skúsite ho otvoriť niekde pri konci, kde odhadujete, že sa budú nachádzať mená začínajúce na W.

Takéto vyhľadávanie (odborne nazývané *interpolačné*) vie skutočne byť efektívne – ale jeho efektívnosť je založená na veľmi dôležitom predpoklade: že hodnoty prvkov v poli naozaj rastú približne rovnomerne.

To nám ale v našom prípade nik nezaručil. Práve naopak – okrem toho, že sú prvky usporiadané v rastúcom poradí, o nich vôbec nič nevieme. Medzi ich hodnotami pokojne môžu byť veľmi nerovnomerné skoky.

A práve tadiaľ viedla cesta k nájdeniu chyby v Andrejovom programe – teda takého vstupu, pre ktorý by ani 10 iterácií hľadania nestačilo. Čo teda potrebujeme? „Oklamať“ Andrejov program, aby prvok hľadal na opačnom konci poľa ako na tom, kde sa skutočne nachádza. Tu je príklad vstupu, ktorý takto Andrejov program oklame: $A = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 987654)$, $X = 14$.

Čo spraví Andrejov program pre tento vstup? Keďže na začiatku poľa je 0 a na konci 987654, hodnota 14 by podľa interpolácie mala ležať úplne na začiatku – čo sa zaokrúhli na index 1. Tam sa teda Andrejov program pozrie v prvej iterácii. V druhej iterácii si program z hodnôt 1 a 987654 vypočíta, že sa chce pozrieť na index 2. A tak ďalej. Našou voľbou vstupu sme teda Andrejov program donútili prechádzať poľom prvok po prvku, a na 10 iterácii sa takto k hodnote 14 dostať nestihne.

# Riešenia krajského kola kategórie A

## A-II-1 Fľaštičková úloha

Našou úlohou bolo nájsť najdlhší súvislý úsek zadanej postupnosti tak, aby priemer prvkov tohoto úseku bol práve $k$.

Prvé a najľahšie riešenie, ktoré nás napadne, je skúsiť postupne každý možný súvislý úsek a zrátať aký má priemer. Stačí teda použiť dva cykly, ktoré budú určovať začiatok a koniec úseku a tretím cyklom spočítať súčet prvkov v ňom. Následne súčet predelím počtom prvkov a pozriem sa, či výsledné číslo (priemer) je rovné $k$.

### Listing programu (C++)

```
//v poli A je uložená vstupná postupnosť
int n,k;
int A[500];
//uloženie výsledku
int vyszac=0,vyskon=-1;
//vyber si začiatok a koniec úseku
for(int zac=0; zac<n; zac++)
    for(int kon=zac; kon<n; kon++) {
        int sucet=0;
        //sčítaj prky úseku
        for(int i=zac; i<=kon; i++) sucet+=A[i];
        if(sucet == k*(kon-zac+1) && vyskon-vyszac<kon-zac) {
            vyszac=zac; vyskon=kon;
        }
    }
```

Ak sa však pozriete do programu vidíte, že som delenie vlastne nikde nepoužil. Delenie je totiž nepresné. My však vieme, že ak si označíme $s$ súčet prvkov v úseku a $p$ počet prvkov v úseku, tak rovnicu $s/p = k$ vieme upraviť na $s = k \cdot p$. Máme teda jednoduchý program, ktorý všetko počíta presne. Jeho časová zložitosť je $\Theta(n^3)$.

**Zlepšenie:**

Keď sa však pozrieme do limitov, tak za riešenie s takouto zložitosťou dostaneme najviac 3 body. Treba teda niečo zrýchliť. V predchádzajúcom programe sme robili zbytočnú robotu, keď sme rátali súčet prvkov v danom úseku. Vždy odznovu sme totiž sčítavali tie isté čísla.

Keď si pozrieme daný program, prvé dva cykly sa dajú interpretovať ako – najskôr zvolím začiatok úseku *zac* a potom pre tento začiatok postupne volím vzdialenejšie a vzdialenejšie konce *kon*. A premenná *kon* sa vždy zväčší o jedna,

čo znamená, že náš zvolený úsek vyzerá rovnako ako predtým, akurát sa k nemu na konci pridal jeden nový prvok, ktorý je na pozícii $kon$.

Postupovať teda môžeme nasledovne – najskôr si zvolíme pozíciu začiatku. Zatiaľ máme zvolený úsek dĺžky nula a teda aj jeho súčet je nulový. Postupne zvyšujeme premennú $kon$, ktorá nám bude ukazovať na koniec zvoleného úseku. No a vždy, keď zväčšíme $kon$ o 1, zväčšíme aj zapamätaný súčet úseku o hodnotu prvku, ktorý doň práve pribudol. Takto teda dostávame riešenie, ktoré postupne prejde všetky súvislé podpostupnosti a každú spracuje v konštantnom čase. Jeho časová zložitosť je teda $\Theta(n^2)$.

## Listing programu (C++)

```
//v poli A je uložená vstupná postupnosť
int n,k;
int A[5000];
//uloženie výsledku
int vyszac=0,vyskon=-1;
//vyber si začiatok a koniec úseku, priebežne sčituj
for(int zac=0; zac<n; zac++) {
    int sucet=0;
    for(int kon=zac; kon<n; kon++) {
        sucet += A[kon];
        if(sucet == k*(kon-zac+1) && vyskon-vyszac<kon-zac) {
            vyszac=zac; vyskon=kon;
        }
    }
}
```

**Vzorové riešenie:**

Ak sa chceme posunúť ďalej, musíme sa na našu úlohu pozrieť trochu ináč. Problém je totiž v tom, že hľadáme vec s príliš mnohými parametrami. Potrebujeme vedieť aj súčet úseku aj počet jeho prvkov. Oveľa lepšie by bolo, ak by sme sa mohli pozrieť len na súčet úseku a priamo z neho vidieť, či má daný úsek priemer práve $k$.

Položme si otázku, kedy má nejaký úsek priemer práve $k$. Je jasné, že tam nemôže byť príliš veľa prvkov menších ako $k$, ale zároveň ani veľa prvkov väčších ako $k$, teda musí byť určitá rovnováha medzi prvkami menšími a väčšími ako $k$. Napríklad ak si zoberieme postupnosť $(1, 3, 4, 4)$, tak má priemer 3, lebo dvom štvorkám sa podarí vyvážiť prítomnosť jednotky, ktorá je až o 2 menšia.

Napíšme si teraz našu rovnicu, ktorú sa snažíme overiť pre nejaký úsek čísiel, ktoré si označíme $a_1$ až $a_p$. $a_1 + a_2 + \cdots + a_p = pk$ čo si vieme prepísať ako $a_1+a_2+\cdots+a_p-pk = 0$. Nakoniec už len rozložme $-pk$ na $p$ samostatných $-k$ a priraďme jedno ku každému prvku a dostaneme $(a_1-k)+(a_2-k)+\dots(a_p-k) = 0$. Vidíme, že každý prvok sa zmenšil práve o $k$ a ak je súčet týchto prvkov 0,

tak úsek má priemer $k$.

Inými slovami, ak od každého prvku odčítame $k$, zmenšíme tým aj priemer ľubovoľného úseku o $k$. Teda úseky, ktoré mali v pôvodnej postupnosti priemer $k$, zodpovedajú úsekom, ktoré majú v novej postupnosti priemer 0. No a prečo nám toto pomôže? Lebo postupnosť má priemer 0 vtedy a len vtedy, keď má súčet 0. A pri hľadaní postupností ktoré majú súčet 0 už nemusíme prihliadať na ich dĺžku.

Našu postupnosť si teda upravíme tak, že od každého prvku odčítame $k$. Odteraz budeme slovom „postupnosť“ označovať túto novú postupnosť, v ktorej hľadáme úseky so súčtom 0.

Teraz potrebujeme spraviť ešte posledný myšlienkový krok. Súčet úsekov sa dá rátať aj iným spôsobom. Vyplňme si pole $P[0..n]$, kde prvok $P[i]$ je súčet prvých $i$ prvkov našej postupnosti. (Prvky poľa $P$ voláme prefixové súčty danej postupnosti.) Pole $P$ vieme ľahko vyplniť v čase $O(n)$ jedným prechodom: $P[0] = 0$ a každé ďalšie $P[i]$ spočítame ako súčet $P[i-1]$ a nasledujúceho prvku postupnosti. Ak potom chceme zistiť súčet úseku od pozície $zac$ po pozíciu $kon$, tak túto hodnotu vyrátame ako $P[kon] - P[zac-1]$.

Zoberme si pevný koniec $kon$. Ktoré začiatky pre tento koniec určujú postupnosti so súčtom 0? Začiatok $zac$ pre tento koniec je vhodný práve vtedy, keď $P[kon] - P[zac-1] = 0$, teda keď $P[zac-1] = P[kon]$. A aby sme dostali úsek čo najdlhší, chceme vybrať najmenšie $zac$ s danou vlastnosťou. No a zisťovať prítomnosť nejakého konkrétneho prvku vieme veľmi ľahko pomocou vyvažovaného binárneho stromu – v C++ implementovaný ako `set` alebo `map`.

Celý algoritmus bude prebiehať nasledovne. Najskôr odčítam od každého prvku pôvodnej postupnosti číslo $k$. Postupne si budem určovať stále väčšie a väčšie $kon$, ktoré bude určovať, kde má končiť náš úsek. Zároveň s tým si budem pamätať sumu všetkých už spracovaných prvkov, teda hodnotu $P[kon]$. V mape budem mať pre každú predchádzajúcu prefixovú sumu zapamätané najmenšie číslo $zac$ také, že prvých $zac$ prvkov má takúto sumu. Pre každé $kon$ sa do tejto mapy pozrieme, či sme aktuálny prefixový súčet $P[kon]$ už niekedy predtým videli. Ak áno, dozvedeli sme sa práve najlepší začiatok zodpovedajúci tomu koncu. A ak nie, tak si pre hodnotu $P[kon]$ zapamätáme, že prvýkrát bola videná na pozícii $kon$. Na záver už len vypíšeme najdlhší nájdený úsek.

Celková časová zložitosť programu bude $O(n \log n)$, lebo pre každý koniec sa pozriem zopár ráz do mapy, ktorej operácie trvajú $O(\log n)$ času. Pamäťová zložitosť je $O(n)$.

### Listing programu (C++)

```
#include <cstdio>
#include <map>
using namespace std;

#define For(i,n) for(int i=0; i<(n); i++)

int main() {
    // načítam vstup a rovno upravím postupnosť
    int n, k, A[100047];
    scanf("%d %d",&n,&k);
    For(i,n) { scanf("%d",&A[i]); A[i]-=k; }

    // prechádzam postupnosť
    map<int,int> M;
    int prefix=0,vyszac=0,vyskon=-1;
    // na začiatku je prvá prefixová suma 0
    M[0]=-1;

    For(i,n) {
        prefix+=A[i];
        // rovnaké číslo má byť na pozícii zac-1, takže pridávam ešte 1
        if(M.find(prefix)!=M.end() && vyskon-vyszac+1 < i-M[prefix]) {
            vyszac=M[prefix]+1;
            vyskon=i;
        }
        if(M.find(prefix)==M.end()) M[prefix]=i;
    }
    printf("%d %d\n",vyszac+1,vyskon+1);
    return 0;
}
```

### Alternatívne riešenie:

V poslednom kroku vôbec nebolo nutné používať pokročilú dátovú štruktúru. (Je to ale pohodlné, preto sme to riešenie uviedli ako prvé.) Namiesto mapy si vystačíme aj s obyčajným triedením. Zoberieme si pole, ktorého prvky sú usporiadané dvojice čísel: $(0,0), (P[1],1), (P[2],2), \ldots, (P[n],n)$. Tieto si usporiadame – primárne podľa prvej súradnice, teda zodpovedajúceho prefixového súčtu, a sekundárne podľa indexu, ktorý mu zodpovedá. V usporiadanom poli budú všetky indexy, ktorým zodpovedá tá istá hodnota prefixového súčtu, tvoriť vždy súvislý úsek. Najvzdialenejšiu dvojicu indexov, ktorým zodpovedá rovnaký prefixový súčet, vieme pomocou tohto usporiadaného poľa ľahko nájsť v lineárnom čase.

## A-II-2 Nájazd na jablone

### Základné pomalé riešenie:

Začnime s nejakým mierne pomalším ale zjavne funkčným riešením. Jedno-

duchou možnosťou by bolo skúsiť nájsť nejakú konečnú množinu kandidátov na vhodné deliace priamky. Potom by vždy, keď príde nový syseľ, stačilo vyskúšať všetkých kandidátov a zistiť, či niektorý z nich vyhovuje. A akí by mohli byť vhodní kandidáti? Napríklad priamky, na ktorých ležia dva stromy. Algoritmus by potom vyzeral tak, že pre každú dvojicu stromov vyskúšame, či syseľ leží na opačnej strane priamky ako všetky ostatné stromy. (Syseľ na priamke ležať nesmie, u stromov to testovať netreba, keďže podľa zadania žiadne tri stromy neležia na priamke.)

Testovanie, či nejaký bod $X$ leží na priamke $YZ$, napravo alebo naľavo od nej môžeme spraviť napríklad pomocou znamienka vektorového súčinu vektorov $\vec{YZ}$ a $\vec{YX}$. Takýto algoritmus by pre každú z rádovo $n^2$ dvojíc stromov testoval rádovo $n$ bodov, a to všetko by vykonal toľkokrát, koľko príde sysľov, čiže $q$-krát, takže výsledná časová zložitosť by bola $O(qn^3)$.

Súčasťou tohto riešenia by však mal byť aj dôkaz, že naša množina kandidátov naozaj stačí. Presnejšie, potrebujeme dokázať, že ak existuje (nejaká ľubovoľná) rozdeľujúca priamka, tak aj jedna z nami zvolených priamok bude vyhovovať. Vezmime si teda ľubovoľnú vyhovujúci priamku. Tú môžeme „posúvať“ smerom od sysľa, až kým sa naša priamka nedotkne nejakého stromu. (V tomto okamihu sme teda už dokázali, že vždy, keď existuje rozdeľujúca priamka, existuje rozdeľujúca priamka prechádzajúca niektorým stromom.)

Ak sme mali šťastie, práve nájdená priamka prechádza dvoma stromami. Ak nie, zoberieme túto priamku a začneme ju otáčať okolo toho stromu ktorým prechádza. Dajme tomu, že najprv v kladnom smere. Teraz máme opäť dve možnosti, čo sa môže stať. Buď najprv otáčaná priamka „narazí“ na druhý strom a našli sme vyhovujúcu priamku, alebo najprv narazí na sysľa, prípadne zároveň na strom a sysľa. V takomto prípade ju začneme otáčať do opačného smeru. A keďže otáčajúca sa priamka pokryje celý priestor skôr než znova narazí na sysľa, tak tentoraz musí najskôr naraziť na strom. A tým sme dôkaz ukončili.

**Prvé predpočítanie:**

Predchádzajúce riešenie robí veľa zbytočnej práce. Napríklad si môžeme všimnúť, že akonáhle má priamka stromy na obe strany od seba, zjavne nebude nikdy vyhovovať. Takéto priamky teda nemá vôbec zmysel skúšať zas a znova pre každého jedného sysľa.

Najjednoduchšie by bolo zbaviť sa týchto zlých priamok hneď na začiatku. Strávime teda na začiatku behu programu čas $O(n^3)$ tým, že pre každú priamku určenú dvoma stromami zistíme, či všetkých $n-2$ zvyšných stromov leží na jednej jej strane alebo nie. Takto nám ostane len $k \leq n^2$ priamok, ktoré majú

všetky stromy na jednej strane. No a teraz keď príde syseľ, už stačí skontrolovať pre každú priamku len polohu sysľa. Takto teda dostaneme riešenie s celkovou časovou zložitosťou $O(n^3 + qk)$.

Celé toto predpočítanie však vieme spraviť aj omnoho efektívnejšie, a navyše bude hneď jasné, že $k$ musí vždy byť malé. Totiž priamky, ktoré má zmysel testovať, budú práve zodpovedať jednotlivým stranám konvexného obalu všetkých stromov (a teda ich bude nanajvýš $n$). Prečo je to tak?

Nám by dokonca stačila jedna implikácia: Ak všetky ostatné stromy ležia na tej istej strane od priamky vedúcej cez stromy $A$ a $B$, tak úsečka $AB$ tvorí stranu ich konvexného obalu. Toto zjavne platí, a teda každá priamka v našej množine kandidátov je skutočne predĺžením niektorej strany konvexného obalu.

Platí však aj opačná implikácia, a teda naozaj každá strana konvexného obalu zodpovedá nejakej možnej rozdeľujúcej priamke. Totiž ak úsečka $AB$ tvorí stranu konvexného obalu, tak musia všetky ostatné stromy ležať na tej istej strane od nej. (Sporom, ak by stromy $C$ a $D$ ležali na jej opačných stranách, tak celý štvoruholník $CADB$ je súčasťou konvexného obalu, čo je spor s tým že $AB$ je jeho strana.)

V domácom kole ste sa naučili, ako spraviť konvevxný obal v čase $O(n \log n)$. Tieto znalosti môžeme teraz využiť k výraznému zlepšeniu časovej zložitosti nášho algoritmu. Ten sa teda bude skladať z dvoch fáz:

a) Zobeieme $n$ stromov a v čase $O(n \log n)$ nájdeme ich konvexný obal. Tým sme zostrojili množinu nanajvýš $n$ priamok, ktoré stačí testovať.

b) Pre každého sysľa postupne vyskúšame každú z priamok nájdených v prvom kroku: zoberieme ľubovoľný tretí strom a pozrieme sa, či leží na opačnej strane priamky ako syseľ. Každého sysľa teda spracujeme v čase $O(n)$.

Celková časová zložitosť tohto algoritmu je teda $O(n \log n + qn)$.

Alternatívna implementácia druhej časti: Vieme využiť to, že algoritmus pre tvorbu konvexného obalu nám jeho strany nájde v usporiadanom poradí po obvode. Keď takto napr. proti smeru hodinových ručičiek ideme po konvexnom obale, vieme, že stromy máme stále naľavo od testovanej priamky, a teda stačí testovať, či syseľ od nej leží napravo.

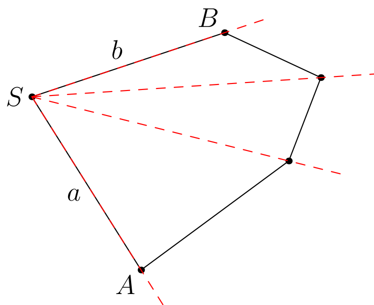
**Vzorové riešenie:**

Z pozorovania o konvexnom obale dokonca vyplýva, že rozdeľujúca priamka existuje práve vtedy, keď sa syseľ nachádza mimo konvexného obalu. (Akonáhle totiž syseľ leží mimo obalu, môžeme použiť vetu o separujúcej priamke z riešení

domáceho kola. Aj samotný syseľ je totiž konvexná množina. Opačná implikácia je zjavná.)

Ako zistiť, či syseľ leží v konvexnom útvare v lepšom než lineárnom čase? Dobré spôsoby sú opäť založené na vhodnom predpočítaní: Treba si daný útvar rozsekať na nejaké kúsky nejako systematicky naskladané vedľa seba a následne pre každého sysľa binárne vyhľadať ten správny kúsok, v ktorom jedinom možno leží. Konkrétnych implementácií existuje viacero, napríklad môžeme rozrezať náš útvar zvislými rezmi idúcimi cez všetky jeho vrcholy.

My si ukážeme jednu z implementačne najjednoduchších možností. Jeden vrchol nášho konvexného $k$-uholníka si označíme za špeciálny, $S$. Z neho vychádzajú dve strany: strana $a$ do bodu ktorý označíme $A$ a strana $b$ do bodu $B$. Predstavme si teraz $k-1$ polpriamok, ktoré idú z bodu $S$ cez každý z ostatných vrcholov (vrátane $A$ a $B$).




Tieto polpriamky nám rozdelia rovinu na $k-1$ uhlov: jeden vypuklý, ktorý vôbec neobsahuje vnútro nášho $k$-uholníka, a $k-2$ zvyšných. Vypuklý uhol si môžeme predstaviť ako zjednotenie dvoch polrovín: jednej určenej stranou $a$ a druhej určenej stranou $b$. V každom zo zvyšných uhlov tá časť, ktorá patrí do nášho $k$-uholníka, tvorí trojuholník. Dve strany trojuholníka ležia na našich polpriamkach, tretiu tvorí vždy jedna zo strán $k$-uholníka.

Ako teraz spracujeme konkrétneho sysľa? Začneme tým, že overíme, či leží v jednej z prázdnych polrovín určených stranami $a$ a $b$. Ak áno, našli sme rozdeľujúcu priamku a končíme. Ak nie, tak sa syseľ nachádza niekde v uhle $ASB$. Tento uhol máme rozdelený na $k-2$ menších. Pomocou $O(\log k)$ otázok tvaru „leží syseľ naľavo od tejto polpriamky?“ vieme binárnym vyhľadávaním nájsť ten uhol, v ktorom syseľ leží. No a následne už len stačí zobrať stranu nášho konvexného obalu, ktorá dotyčnému uhlu zodpovedá, a pozrieť sa, či syseľ leží na správnej strane od nej. (Ak sa náhodou stane, že syseľ leží presne na jednej z našich polpriamok, môžeme ho zaradiť do ľubovoľného z uhlov určených dotyčnou polpriamkou.)

Týmto sme našli algoritmus pracujúci v čase $O(n \log n + q \log k)$. A keďže $k \le n$, toto môžeme ďalej zhora odhadnúť ako $O((n+q) \log n)$.

### Listing programu (C++)

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

// definíciu bodu, operátor < na usporiadanie bodov,
// vektorový a skalárny súčin a konvexný obal
// nájdete v riešení domáceho kola

// binárne vyhľadanie: vráti poradové číslo úsečky, ktorú obsahuje lúč,
// v ktorom sa objaví sysel, resp. 0/10^8, ak je sysel príliš vľavo/vpravo
long long binsrch(point& sysel, vector<point>& hull, unsigned long long size) {
        if (cross(hull [0], hull [1], sysel) < 0) return 0;
        if (cross(hull [0], hull [size-1], sysel) > 0) return 10e8;
        long long lower = 1, upper = size - 1;
        while (upper - lower > 1) {
                if (cross(hull [0], hull [(lower+upper) / 2], sysel) > 0)
                    lower = (lower + upper) / 2;
                else
                    upper = (lower + upper) / 2;
        }
        return lower;
}

int main() {
    int N; cin >> N;
    vector<point> P(N); for (int n=0; n<N; ++n) cin >> P[n].x >> P[n].y;
    sort(P.begin(), P.end());
        vector <point> hull = convex_hull(P);
        int Q; cin >> Q;
        point sysel;
        long long answer;
        for (int i = 0; i < Q; ++i) {
                cin >> sysel.x >> sysel.y;
                answer = binsrch(sysel, hull, hull.size());
                if (answer == 0) {
                    cout << hull [0].x << ' ' << hull [0].y << ' ';
                    cout << hull [1].x << ' ' << hull [1].y << endl;
                } else if (answer == 10e8) {
                    cout << hull [0].x << ' ' << hull [0].y << ' ';
                    cout << hull.back().x << ' ' << hull.back().y << endl;
                } else if (cross(hull [answer], hull [answer + 1], sysel) < 0) {
                    cout << hull [answer].x << ' ' << hull [answer].y << ' ';
                    cout << hull [answer+1].x << ' ' << hull [answer+1].y << endl;
                } else {
                    cout << "Sysel vitazi!" << endl;
                }
        }
        return 0;
}
```

## A-II-3 Tajná misia

Máme súvislý graf, v ktorom je vrcholov rovnako veľa, ako hrán. Chceme vybrať čo najviac hrán tak, aby so žiadnym vrcholom nesusedili viaceré vybrané

hrany. Inými slovami, hľadáme v grafe maximálne párenie.

Keďže vrcholov a hrán je rovnako veľa, v grafe je určite práve jeden cyklus. (Totiž ľubovoľná kostra nášho grafu má $n-1$ hrán a nie sú na nej žiadne cykly. Keď následne pridáme chýbajúcu $n$-tú hranu, tá vytvorí ten spomínaný jediný cyklus.) Takže niekde v našom grafe je niekoľko vrcholov spojených do cyklu, a na každý môže byť pripojených niekoľko stromov, v ktorých už žiadne cykly nie sú.

Vyriešme najprv jednoduchšiu úlohu: predstavme si, že by sme mali iba strom – čiže súvislý graf s $n$ vrcholmi a $n-1$ hranami.

Na takto zjednodušenú úlohu sa dá použiť dynamické programovanie. Algoritmus najprv strom zakorení za ľubovoľný koreň, a potom sa bude zdola hore postupne pozerať na každý vrchol. Keď sa pozrie na nejaký vrchol $v$, bude ho zaujímať, ako vyzerá situácia v podstrome pod vrcholom $v$ (vrátane), a aké najlepšie párenie sa v tom podstrome dá dosiahnuť. Preto vypočíta dve čísla: $A(v)$ bude veľkosť najlepšieho párenia, ak môže použiť aj samotný vrchol $v$, a $B(v)$ bude veľkosť toho párenia, ak vrchol $v$ nejde použiť (lebo chceme do párenia vybrať hranu, ktorá ho spája s jeho rodičom).

Ako vypočítame číslo $B(v)$? Ak vieme, že vrchol $v$ bude spojený s jeho rodičom, určite nebude spojený so žiadnym synom, takže stačí sčítať $A(c)$ každého syna $c$.

Ako vypočítame číslo $A(v)$? Tu treba zobrať maximum z viacerých možností, ako to párenie bude vyzerať. Prvá možnosť je, že $v$ nespojíme s nikým, takže veľkosť párenia bude znovu súčet $A(c)$ pre každého syna $c$. Druhá možnosť je, že hranu medzi $v$ a niektorým synom $u$ vyberieme do párenia (prestrihneme medzi nimi kábel). Veľkosť už nebude súčet všetkých $A(c)$, ale odbudne $A(u)$ a pribudne $1 + B(u)$.

Ak pôjdeme po strome zdola hore a pre každý vrchol $v$ si zapamätáme čísla $A(v)$ a $B(v)$, všetko vybavíme v lineárnom čase. Výsledok potom bude jednoducho číslo $A(r)$ koreňa $r$.

Ako sa situácia zmení, ak nemáme $n-1$ hrán, ale $n$? Graf už nie je len strom, je v ňom aj jeden cyklus. Tak ten cyklus nájdime a pozrime sa na ľubovoľnú hranu na ňom. Táto hrana buď v párení bude, alebo nebude. Ak tam nebude, môžeme ju z grafu hneď vyhodiť a zostane nám strom, čo už vieme riešiť. Ak tam bude, dané dva vrcholy určite nebudú spárované s nikým iným, takže môžeme vyhodiť ostatné hrany, čo z nich vedú. Tým sa graf rozpadne na niekoľko stromov, čo už opäť dokážeme riešiť. Stačí vyskúšať tieto dva prípady a vybrať maximum. Keďže sú len dva, celé riešenie má stále lineárnu časovú aj pamäťovú zložitosť.

Iné korektné riešenie je založené na pažravej myšlienke: Kým máme v grafe vrchol $u$ stupňa 1, môžeme jedinú hranu $uv$ vedúcu z $u$ zobrať do riešenia (a následne zahodiť vrcholy $u$, $v$ a všetky ostatné hrany vedúce z $v$, ktoré už do riešenia zobrať nemôžeme).

Dôkaz vyššie uvedeného tvrdenia: Zoberme ľubovoľné optimálne párenie. Ak je v ňom použitá hrana $uv$, sme hotoví. Ak nie, musí byť použitá nejaká iná hrana $vw$ (inak by sme hranu $uv$ mohli pridať a mali by sme lepšie párenie). Potom ale môžeme hranu $vw$ zahodiť a namiesto nej zobrať práve hranu $uv$. Tým sme zjavne opäť dostali korektné párenie a navyše je rovnakej, teda tiež optimálnej veľkosti.

Vyššie uvedený postup teda len dookola opakujeme, až kým nedostaneme graf, ktorý už žiadne vrcholy stupňa 1 nemá. Tu sú teraz dva možné prípady: ak nám už neostalo nič, máme hotové optimálne riešenie. Mohol nám však ešte ostať graf, ktorý má všetky vrcholy stupňa aspoň 2. No a to zjavne mohlo pre náš pôvodný graf nastať len jediným spôsobom: to, čo nám na konci ostalo nevyriešené, je práve náš jediný cyklus (na ktorom majú všetky vrcholy stupeň presne 2). A cyklus už vyriešime jednoducho: ak má $k$ vrcholov, tak najlepšie párenie na ňom má zjavne veľkosť $\lfloor k/2 \rfloor$.

Pri implementácii si stačí pamätať stupne jednotlivých vrcholov a akonáhle stupeň vrchola klesne na 1, zaradiť ho do fronty na spracovanie.

## Listing programu (C++)

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

int N;
vector< vector<int> > susedia;
vector<int> stupen;
vector<bool> pouzil;

int main() {
    cin >> N;
    susedia.resize(N);
    stupen.resize(N,0);
    pouzil.resize(N,false);
    for (int i=0; i<N; ++i) {
        int a, b; cin >> a >> b; a--; b--;
        susedia[a].push_back(b); ++stupen[a];
        susedia[b].push_back(a); ++stupen[b];
    }

    queue<int> Q;
    for (int i=0; i<N; ++i) if (stupen[i]==1) Q.push(i);

    int riesenie = 0;
```

```
    while (!Q.empty()) {
        int kde = Q.front(); Q.pop();
        // nič nerobíme, ak sme medzičasom tento vrchol použili alebo zablokovali
        if (pouzil[kde] || stupen[kde]==0) continue;

        int sus = -1;
        for (int a : susedia[kde]) if (!pouzil[a]) sus=a;

        pouzil[kde] = pouzil[sus] = true;
        ++riesenie;
        for (int a : susedia[kde])
            if (!pouzil[a]) { --stupen[a]; if (stupen[a]==1) Q.push(a); }
        for (int a : susedia[sus])
            if (!pouzil[a]) { --stupen[a]; if (stupen[a]==1) Q.push(a); }
    }

    int na_cykle = 0;
    for (int i=0; i<N; ++i) if (!pouzil[i] && stupen[i]>1) ++na_cykle;
    riesenie += (na_cykle/2);

    cout << riesenie << endl;
}
```

## A-I-4 Mimozemské počítače

### Podúloha A (3 body):

Mimozemšťania nám dodali sálový KSP, ktorého funkcia `husenica(n,E)` rozhoduje problém existencie kostry-húsenice v danom jednoduchom neorientovanom grafe. My pomocou tejto funkcie chceme rozhodnúť, či daný graf $G$ obsahuje nejakú Hamiltonovskú cestu.

Vlastnosť „graf obsahuje kostru-húsenicu“ si môžeme ekvivalentne sformulovať nasledovne: „graf obsahuje nejakú cestu takú, že každý vrchol, ktorý na tej ceste neleží, s ňou susedí“. Takto povedané sa to už začína podobať na Hamiltonovskú cestu, s jediným rozdielom – tentokrát nemusíme prejsť cez všetky vrcholy, stačí ísť len popri niektorých.

Aby sme vyriešili zadanú úlohu, potrebujeme vymyslieť nejakú úpravu vstupného grafu $G$ s nasledovnou vlastnosťou: Ak pôvodný graf $G$ obsahoval Hamiltonovskú cestu, bude upravený graf $G'$ obsahovať kostru-húsenicu. A naopak, ak $G$ žiadnu Hamiltonovskú cestu nemá, upravený graf $G'$ kostru-húsenicu obsahovať nesmie.

Existuje veľmi jednoduchá transformácia, ktorá má práve túto vlastnosť: ku každému vrcholu $v$ v $G$ pridáme nový vrchol $v'$ a hranu $vv'$.

Prečo táto transformácia funguje?

Ak v pôvodnom grafe existovala Hamiltonovská cesta, v novom grafe zjavne existuje kostra-húsenica: stačí zmazať všetky hrany okrem jednej Hamiltonovskej cesty a tých hrán, ktoré sme do grafu $G$ pridali my.

A teraz naopak. Predpokladajme, že nami upravený graf $G'$ obsahuje húsenicu. Ako môže vyzerať jej „hlavná“ cesta? Vrcholy, ktoré sme pridali do $G$, majú všetky stupeň 1, preto sa takýto vrchol môže nachádzať len na začiatku alebo na konci cesty. Zvyšok cesty teda nutne tvoria *niektoré* vrcholy pôvodného grafu $G$. No a čo by sa stalo, ak by medzi nimi niektorý vrchol $v$ chýbal? Potom s ním susediaci vrchol $v'$ neleží ani na našej ceste, ani s ňou nemá ako susediť, a to je spor. Preto ak $G'$ obsahuje húsenicu, jej „hlavná“ cesta nutne zodpovedá Hamiltonovskej ceste v pôvodnom grafe $G$.

## Listing programu (Python)

```
def cesta(n,E):
    for i in range(n): E.append( (i,n+i) )
    husenica(2*n,E)
```

## Podúloha B (3 body):

Mimozemšťania nám dodali sálový KSP, ktorého funkcia `husenica(n,E)` rozhoduje problém existencie kostry-húsenice v danom jednoduchom neorientovanom grafe. My pomocou tejto funkcie chceme rozhodnúť, či daný graf $G$ obsahuje nejakú Hamiltonovskú kružnicu.

V podúlohe A sme si ukázali, ako funkciu `cesta(n,E)` naprogramovať pomocou funkcie `husenica(n,E)`. No a z domáceho kola vieme, ako funkciu `kruznica(n,E)` implementovať pomocou funkcie `cesta(n,E)`. Tieto dva postupy zložíme dokopy a máme podúlohu B vyriešenú.

Detailnejší popis začneme tým, že si pripomenieme, ako fungovala transformácia z domáceho kola. Z grafu $G$ vyrobíme nový graf $G'$ tak, že pridáme vrchol $n$, ktorý bude kópiou vrcholu 0; ďalej pridáme vrcholy $n+1$ a $n+2$ a hrany medzi 0 a $n+1$ a medzi $n$ a $n+2$. Takto zostrojený $G'$ obsahuje Hamiltonovskú cestu práve vtedy, keď $G$ obsahoval Hamiltonovskú kružnicu.

Ak nemáme k dispozícii funkciu `cesta(n,E)` ale len funkciu `husenica(n,E)`, musíme následne na graf $G'$ použiť konštrukciu z riešenia podúlohy A: každý vrchol $G'$ dostane nového kamaráta, ktorý s ním bude spojený hranou. Pre takto zostrojený graf $G''$ platí: $G''$ obsahuje kostru-húsenicu práve vtedy, keď $G'$ obsahoval Hamiltonovskú cestu.

Celé riešenie bude teda vyzerať nasledovne: zoberieme vstupný graf $G$, z neho vyrobíme $G'$, z toho ďalej vyrobíme $G''$, a ten zadáme do nášho sálového KSP. Ak KSP rozsvieti zelené svetlo, znamená to, že $G''$ obsahuje kostru-húsenicu, a teda $G$ obsahuje Hamiltonovskú kružnicu. A naopak, červené svetlo znamená, že $G$ Hamiltonovskú kružnicu neobsahuje. A presne to sme chceli dosiahnuť.

### Listing programu (Python)

```
def kruznica(n,E):
    # zostrojime si zoznam susedov vrcholu 0
    susedia0 = []
    for x,y in E:
        if x==0: susedia0.append(y)
        if y==0: susedia0.append(x)
    # do grafu pridame novy vrchol n, ktory je kopiou vrcholu 0
    E += [ (n,x) for x in susedia0 ]
    # a este dva nove vrcholy ktore sluzia ako konce cesty
    E += [ (0,n+1), (n,n+2) ]
    # na konci zavolame funkciu cesta(), ktora rozsvieti spravne svetlo
    # (tu volaná funkcia cesta() je naša funkcia z riešenia podúlohy A)
    cesta(n+3,E)
```

### Podúloha C (4 body):

Máme kufríkový KSP, ktorého funkcia `existuje_pokrytie(k,n,m,Z)` rozhoduje problém existencie pokrytia množinami, ktoré má veľkosť nanajvýš $k$. My chceme pomocou tohto kufríkového KSP nájsť najmenší možný počet žiakov, ktorí sú predsedom aspoň jedného krúžku.

Celé riešenie tejto úlohy je založené na jednoduchom pozorovaní: KSP, ktorý máme k dispozícii, vie takmer priamo riešiť našu úlohu, len si ju musíme vhodne sformulovať.

Na vstupe sme dostali ku každému krúžku zoznam žiakov, ktorí doň chodia. Túto istú informáciu si ale vieme reprezentovať aj opačne: pre každého žiaka $z$ si zostrojíme množinu krúžkov $K(z)$, do ktorých chodí.

Predstavme si teraz, že postupne vyberáme žiakov, ktorí budú patriť do rady predsedov. Vždy, keď nejakého vyberieme, spravíme ho predsedom všetkých krúžkov, do ktorých chodí a ktoré ešte nemajú predsedu. Akonáhle teda vyberieme hocijakú sadu žiakov takú, že každý krúžok v nej má zastúpenie, budeme mať platnú radu predsedov.

Preto otázku zo zadania môžeme položiť aj nasledovne: „Koľko najmenej spomedzi množín $K(z)$ musím vybrať, ak chcem aby ich zjednotenie obsahovalo všetky krúžky?“

A toto je presne otázka na veľkosť pokrytia množinami. Pomocou kufríkového KSP, ktorý máme k dispozícii, vieme správnu odpoveď nájsť na $O(\log z)$

otázok pomocou binárneho vyhľadávania.

## Listing programu (Python)

```
# načítame údaje o krúžkoch, prerobíme ich na údaje o žiakoch

Z = int(input()) # počet žiakov
K = int(input()) # počet krúžkov

kruzky = [ [] for z in range(Z) ] # pre každého žiaka zoznam jeho krúžkov

for k in range(K):
    kruzok = [ int(x) for x in input().split() ]
    for x in kruzok: kruzky[x].append(k) # žiak x navštevuje krúžok k

# ideme binárne vyhľadávať minimálny počet žiakov-predsedov
# na začiatku vieme, že 0 predsedov je primálo a že Z určite stačí

malo, dost = 0, Z
while dost > malo+1:
    stred = (malo + dost) // 2
    if existuje_pokrytie( stred, K, Z, kruzky ): dost = stred
    else: malo = stred

print(dost)
```

# Riešenia krajského kola kategórie B

## B-II-1 Lokálne ochladzovanie

Začnime úplne triviálnym riešením: Pre každý úsek dní dĺžky aspoň dva spočítajme priemernú teplotu a porovnajme ju so zadaným limitom $k$.

Koľko úsekov takto odskúšame? Ak výlet začne v prvý deň, môže skončiť v druhý, tretí, ... alebo až v posledný, $n$-tý deň. Máme teda na výber z $n-1$ možností. Ak by výlet začal v druhý deň, mali by sme $n-2$ možností kedy ho ukončiť. Podobne, ak výlet začne v $i$-ty deň, máme $n-i$ možností. Dokopy teda existuje $(n-1)+(n-2)+\cdots+3+2+1=(n^2-n)/2$ úsekov dĺžky aspoň dva. V $O$-notácii jednoducho zapíšeme, že úsekov vyskúšame $O(n^2)$.

Pre každý úsek potrebujeme vypočítať priemernú teplotu v ňom. Označme si súčet teplôt v úseku $s$ a jeho dĺžku (počet dní v ňom) $\ell$. Priemerná teplota sa potom samozrejme rovná $s/\ell$. V našom riešení však namiesto podmienky $s/\ell \geq k$ použijeme ekvivalentnú podmienku $s \geq k\ell$. Vyhneme sa tak počítaniu s nepresnými desatinnými číslami.

Takéto riešenie sa implementuje veľmi jednoducho: V dvoch vnorených cykloch vyskúšame všetky možnosti pre začiatok a koniec výletu a ďalším cyklom sčítame teploty počas výletu. Keďže prinajhoršom vyskúšame $O(n^2)$ úsekov a na sčítanie teplôt v jednom úseku vynaložíme $O(n)$ operácií, celková časová zložitosť tohto riešenia je $O(n^3)$.

Ak chceme predchádzajúce riešenie ľahko zlepšiť, stačí si všimnúť, že súčet teplôt v úseku sa dá počítať aj rýchlejšie: Hneď po úseku dní $i, i+1, \ldots, j$ totiž skúšame úsek $i, i+1, \ldots, j, j+1$. Súčet teplôt v tom dlhšom úseku nemusíme počítať odznova (v lineárnom čase) – vieme ho vypočítať okamžite (v konštantnom čase) ako súčet teplôt v kratšom úseku plus $t_{j+1}$. Namiesto opätovného sčítavania tak stačí pričítať len jednu hodnotu, čím sa nám riešenie zrýchli na $O(n^2)$.

Od vzorového riešenia nás delí už len jedno pozorovanie: Ak existuje nejaký vhodný úsek dní, potom existuje aj vhodný úsek dĺžky najviac tri. To znamená, že namiesto skúšania všetkých $O(n^2)$ možných úsekov nám postačia tie, ktoré majú dĺžku 2 alebo 3, čiže len $O(n)$ úsekov.

Prečo je tomu tak? Vezmime si nejaký úsek dní, ktorý má priemernú teplotu aspoň $k$. Ak tento úsek rozdelíme na dve kratšie časti, ľahko vidíme, že aspoň

jedna z nich musí mať priemernú teplotu aspoň $k$. (Totiž spojením dvoch úsekov, ktoré majú oba priemer menší ako $k$, vznikne vždy úsek, ktorý má priemer menší ako $k$.)

Nech teda existuje nejaký vyhovujúci úsek, ktorý má dĺžku $\ell \geq 4$. Takýto úsek môžeme rozdeliť na dve časti dĺžok 2 a $\ell - 2$ a aspoň jedna z nich musí tiež byť vyhovujúcim úsekom. Opakovaním tohto postupu dostávame čoraz kratšie úseky, ktoré všetky spĺňajú zadanie úlohy. Zastavíme sa až na úseku dĺžky 2 alebo 3, ktorý už nevieme rozdeliť na dve časti ktoré by obe mali dĺžku aspoň 2. Tým sme teda dokázali nasledovné tvrdenie: *Ak existuje ľubovoľný vyhovujúci úsek dĺžky aspoň 2, tak nutne existuje vyhovujúci úsek dĺžky presne 2 alebo presne 3.*

**Listing programu (Python)**

```
def vyries(N,K,T):
    for zaciatok in range(N-1):
        if sum( T[zaciatok:zaciatok+2] ) >= 2*K:
            return (zaciatok+1,zaciatok+2)
    for zaciatok in range(N-2):
        if sum( T[zaciatok:zaciatok+3] ) >= 3*K:
            return (zaciatok+1,zaciatok+3)
    return None
```

## B-II-2 Magnetické písmenká

**Podúloha a:**

Čo vieme usúdiť z toho, že vieme, že slovo na chladničke je palindróm?

Ak je celkový počet písmen párny, tak ich vieme rozdeliť na niekoľko dvojíc: prvé s posledným, druhé s predposledným. . . , vo všeobecnosti $i$-te s $(n+1-i)$-tym. Keďže naše slovo je palindróm, musia v každej dvojici obe písmenká byť rovnaké. Tým pádom z každého písmenka musíme mať párny počet kusov – teda musíme mať párny počet áčok, párny počet béčok, atď.

Ak je celkový počet písmen v slove nepárny, tak vieme spraviť presne tú istú úvahu, s jediným rozdielom: jedno písmenko v strede ostane spárované samé so sebou. Teda všetkých písmen okrem jedného musí byť párny počet, zatiaľ čo jedno konkrétne písmeno (to, ktoré je v strede) sa v našom slove vyskytuje nepárne veľa krát.

To, čo sme si práve odvodili, sa odborne volá *nutná podmienka*: *ak* je reťazec palindróm, *tak* každé písmeno okrem nanajvýš jedného *nutne musí* v ňom mať párny počet výskytov.

Ľahko ale nahliadneme, že táto podmienka je zároveň aj *postačujúca* na to, aby sa z danej sady písmen dal nejaký palindróm poskladať. Ak máme z nejakého jedného písmena nepárny počet kusov, jeden z nich dáme do stredu, inak začneme s prázdnym stredom. Následne rozdelíme ostatné písmená do rovnakých dvojíc a tie postupne, jednu po druhej, pridávame do reťazca – vždy jedno na začiatok a druhé na koniec.

Vďaka informáciám z predchádzajúcuch odsekov vieme teda ľahko zistiť, či sa dá poskladať aspoň jeden palindróm, teda riešiť podúlohu a). Prejdeme všetky písmená na vstupe a pre každé písmeno od a po z si spočítame, koľkokrát sa na vstupe vyskytlo. Ak máme viac ako jedno písmeno s nepárnym počtom výskytov, palindróm sa poskladať nedá, ak je také najviac jedno, tak to ide.

Podúlohu a) teda teda vieme riešiť v čase $O(n)$, kde $n$ je počet písmen na vstupe.

### Listing programu (Python)

```
n = int(input())
pocty = [0]*26
for znak in input():
    pocty[ord(znak)-ord('a')]+=1

neparnych = 0
for i in pocty:
    if i%2:
        neparnych+=1

if neparnych<2:
    print('ano')
else:
    print('nie')
```

### Podúloha b:

Keď zistíme, že nejaký palindróm existuje, ako zistiť počet palindromatických slov?

Ak je počet písmen nepárny, tak vieme aj povedať, ktoré písmeno je v strede – to, ktoré má jediné nepárny počet výskytov. Toto písmeno môžeme teda priamo umiestniť do stredu a odteraz ďalej už uvažovať len situáciu, kedy je každého písmena párny počet kusov.

Ďalej vieme povedať, z každého druhu písmen musí byť presne polovica medzi prvými $k$ znakmi a polovica medzi poslednými $k$. Tým pádom rozumný spôsob ako vyrábať palindrómy je zobrať z každého druhu písmen polovicu a nejako ich poukladať na prvých $k$ miest slova.

No a nech umiestnime prvých $k$ písmen akokoľvek, vždy vieme posledných $k$ jednoznačne doplniť – budú tvoriť „zrkadlový obraz“ prvých $k$. Všetkých

možných palindrómov bude teda presne toľko, koľkými spôsobmi vieme preusporiadať písmená patriace do prvej polovice slova.

Príklad: Nech má Ferko 3 áčka a 6 béčok. Začne tým, že jedno $a$ umiestni do stredu. Teraz platí, že aj v prvej, aj v druhej polovici sa má vyskytovať jedno $a$ a tri $b$. Sú teda štyri možnosti, ako môže vyzerať prvá polovica: *abbb*, *babb*, *bbab*, alebo *bbba*. Im potom prislúchajú celé palindrómy *abbbabbba*, *babbabbab*, *bbababab b* a *bbbaaabbb*.

Z predchádzajúcich úvah je teda jasné, že počet správnych palindrómov – odpoveď na podúlohu b) – je rovnaký, ako počet slov dĺžky $k$, ktoré obsahujú polovičné počty písmeniek ako boli na vstupe.

A aký je tento počet? Keby boli všetky písmená rôzne tak máme $k$ písmen, ktoré môžeme dať na prvú pozíciu. Zo zvyčných $k-1$ písmen si môžeme vybrať ľubovoľné písmeno, ktoré pôjde na druhú pozíciu. Na tretiu pozíciu máme $k-2$ možností… až na poslednú pozíciu máme len jednu možnosť, dáme tam písmenko, ktoré nám ostalo. Dokopy máme teda $k \cdot (k-1) \cdot (k-2) \cdot \cdots \cdot 2 \cdot 1 = k!$ možností.

My ale nemáme nutne všetky písmená rovnaké. Ako počet možností spočítať vtedy? Použijeme jednoduchý trik, ktorý si najskôr ukážeme na príklade.

Predpokladajme, že všetky písmená sú navzájom rôzne, až na to, že máme tri béčka. Očíslujme si ich teda $b_1$, $b_2$ a $b_3$, čím z nich vznikli tri rôzne písmená. Teraz vieme, že existuje presne $k!$ možností, ako všetkých $k$ písmen usporiadať. Lenže pre nás sú všetky tri béčka nerozlíšiteľné. A teda slovo, v ktorom sa napr. vyskytuje najskôr $b_1$, neskôr $b_2$ a ešte neskôr $b_3$ pre nás vyzerá úplne rovnako ako to isté slovo, kde je ale najskôr napr. $b_3$, potom $b_1$ a nakoniec $b_2$.

Presnejšie, medzi $k!$ reťazcami, ktoré sme vygenerovali, sa každý z naozaj rôznych reťazcov vyskytuje presne $3! = 6$-krát: raz pre každé z poradí $b_1b_2b_3$, $b_1b_3b_2$, $b_2b_1b_3$, $b_2b_3b_1$, $b_3b_1b_2$ a $b_3b_2b_1$. Takže v situácii, kedy je všetkých $k$ písmen navzájom rôznych až na to, že tri z tých $k$ písmen sú béčka, existuje presne $k!/3!$ rôznych usporiadaní.

No a tento trik môžeme použiť aj v situáciách, kedy sa opakuje viacero typov písmen. Majme napríklad štyri $a$, tri $b$ a dve $c$. Keď v takomto prípade vygenerujeme všetkých $k!$ permutácií, bude sa medzi nimi každý z navzájom rôznych reťazcov vyskytovať presne $(4! \cdot 3! \cdot 2!)$-krát.

Vo všeobecnosti teda platí, že ak máme dokopy $k$ písmen, pričom $p_a$ z nich sú $a$, $p_b$ sú $b$, atď., tak počet rôznych reťazcov vieme vyjadriť nasledujúcim

vzorcom:

$$\frac{k!}{p_a! \cdot p_b! \cdot \ldots \cdot p_z!}$$

Túto hodnotu vieme vypočítať v lineárnom čase od $k$: najskôr si vypočítame faktoriály pre čísla 0 až $k$ a potom pomocou nich vypočítame výslednú hodnotu vzorca.

### Listing programu (Python)

```
n = int(input())//2
pocty = [0]*26
for znak in input():
    pocty[ord(znak)-ord('a')]+=1

neparnych = 0
for i in range(26):
    if pocty[i]%2:
        neparnych+=1
    pocty[i]//=2

if neparnych>1:
    print(0)
else:
    faktorial = [1]
    for i in range(n):
        faktorial.append(faktorial[i] * (i+1))

    vysledok = faktorial[n]
    for i in pocty:
        vysledok //= faktorial[i]

    print(vysledok)
```

## Podúloha c:

A ako je to s generovaním všetkých týchto možností?

Našou hlavnou úlohou bude teda napísať program, ktorý dostane skupinu písmen a vygeneruje všetky možné poradia tých písmen. Každé z týchto poradí vieme potom ľahko doplniť na palindróm. No a túto úlohu sme už predsa riešili – v úlohe 2 domáceho kola!

Jednou z možností je implementovať úlohu pomocou rekurzie. Ak mi už neostali žiadne písmená, tak jediným riešením je prázdny reťazec. A ak ešte nejaké písmená mám, tak vyskúšam všetky možnosti pre prvé písmeno a zakaždým zavolám ten istý program na zvyšných $k-1$ písmen. A to je vlastne celé riešenie.

Ešte potrebujeme poriešiť trocha implementačných detailov. Aby bola naša rekurzívna procedúra dosť rýchla, nemôžeme strácať veľa času na udržiavanie informácie o počte zostávajúcich písmen. Celkom efektívna metóda je pamätať

si v globálnom poli, koľko kusov každého písmena ešte máme k dispozícii. Toto pole sa pri rekurzívnom volaní, aj pri návrate z neho, ľahko upravuje.

Časová zložitosť programu je približne priamo úmerná dĺžke výstupu. (Zdôvodnenie tejto skutočnosti nechávame na čitateľa.)

## Listing programu (Python)

```
def generuj(n):
    global znak, pocty, slovo
    if n==0:
        print(''.join(slovo + [znak] + slovo[::-1]))
    for i in range(26):
        if pocty[i]:
            pocty[i]-=1
            slovo.append(chr(ord('a')+i))
            generuj(n-1)
            slovo.pop()
            pocty[i]+=1

n = int(input())//2
pocty = [0]*26
for z in input():
    pocty[ord(z)-ord('a')]+=1

znak, slovo = '', []
for i in range(26):
    if pocty[i]%2:
        if znak:
            print(0)
            quit()
        znak = chr(ord('a')+i)
    pocty[i]//=2

generuj(n)
```

Druhá možnosť je použiť funkciu `next_permutation`, ktorá sa v C++ nachádza v knižnici `algorithm` a v Pascale si ju musíme naprogramovať.

`next_permutation` urobí z $k$-tice písmen novú $k$-ticu tých istých písmen, s tým že keď si možné $k$-tice usporiadame podľa abecedy, tak tá nová pôjde hneď po predchádzajúcej. Keď do funkcie dáme poslednú v abecede, vráti false. Takže keď začneme abecedne najmenšou $k$-ticou, vieme ľahko vygenerovať všetky.

Takže na začiatku cheme vyplniť pole dĺžky $k$ najprv $p_a$ áčkami, potom $p_b$ béčkami, a tak ďalej. Jedno volanie funkcie je lineárne od dĺžky poľa, takže celková zložitsť je priamo úmerná dĺžke výstupu.

No a ešte existuje veľa iných možných, podobne efektívnych implementácií. Jednu inú sme si ukázali v riešeniach domáceho kola.

## B-II-3 Búranie

Začnime jednoduchým pozorovaním: Pokiaľ si určíme, ktorý úsek domov ponecháme, tak najvýhodnejšie je búrať čo najmenej. Zistíme si teda výšku najmenšieho domu v danom úseku. Ak chceme celý tento úsek ponechať, určite musíme ostatné domy zarovnať aspoň na jeho výšku. No a takéto zarovnanie zároveň stačí, preto presne to urobíme.

Pomocou tohoto pozorovania vieme dosiahnúť pomerne jednoduché riešenie s časovou zložitosťou $O(n^3)$. Vyskúšame každý úsek, celý ho prejdeme, nájdeme najmenšiu výšku, a z tej a dĺžky úseku spočítame, koľko bytov by ostalo, ak by sme vybrali tento úsek. Spomedzi všetkých riešení vyberieme to najvýhodnejšie.

Toto riešenie sa dá ľahko urýchliť. Nepotrebujeme pre každý úsek prepočítavať jeho minimum. Pre všetky úseky, ktoré majú rovnaký začiatok, si vieme minimálnu výšku prepočítavať postupne. Vždy keď posunieme koniec úseku, tak prepočítame minimálnu výšku na jedno porovnanie – porovnáme pôvodnú minimálnu výšku a výšku budovy, ktorá práve do úseku pribudla. Takéto riešenie má časovú zložitosť $O(n^2)$.

Toto riešenie sa už ďalej veľmi vylepšovať nedá. K rýchlejšiemu riešeniu vedie iná cesta. Nebudeme si vyberať začiatok ani koniec úseku. Namiesto toho si vyberieme dom, na ktorého výšku zarovnáme okolité domy. Koľko najviac okolitých domov môžeme zobrať do takéhoto úseku? To je ľahké: zjavne sa oplatí do každej strany roztiahnuť čo najviac, teda brať všetky domy až kým nenarazíme na prvý menší od toho, ktorý sme na začiatku vybrali.

Priamou implementáciou tohto postupu by sme opäť dostali riešenie s časovou zložitosťou $O(n^2)$.

Otázka „Kde je vľavo/vpravo najbližší menší dom?“ by vám ale mala pripomínať úlohu o dominách z domáceho kola. Tam sme pre každé domino mali nájsť najbližšie ťažšie naľavo a napravo. Využijeme teda riešenie z domáceho kola, akurát v ňom otočíme smer porovnávania.

Celé riešenie sa dá zhrnúť do dvoch krokov. Najprv dvakrát prejdeme pole (raz zľava doprava a druhýkrát sprava doľava) a pomocou niektorého z algoritmov uvedených v riešeniach domáceho kola nájdeme ku každému domu najbližší menší naľavo aj napravo, to celé v čase $O(n)$. Následne vyskúšame každý dom ako dom, na ktorého výšku sa zarovná jeho okolie. To vieme urobiť v jednom ďalšom cykle, teda opäť v čase $O(n)$. Celkovo máme teda riešenie s optimálnou časovou zložitosťou $O(n)$.

## Listing programu (C++)

```
#include <cstdio>
#include <vector>
#include <algorithm>
#include <stack>

using namespace std;

int main() {
  int n;
  scanf("%d", &n);
  vector<int> vysky(n+2);
  // na zaciatok a koniec si pridame zarazku
  vysky[0] = vysky[n+1] = -1;
  for (int i = 0; i < n; i++) scanf("%d", &vysky[i+1]);

  stack<int> Sl;
  Sl.push(0);
  vector<int> vlavo(n+2);
  for (int i = 1; i < n+1; i++) {
    while (vysky[Sl.top()] >= vysky[i]) Sl.pop();
    vlavo[i] = Sl.top();
    Sl.push(i);
  }

  stack<int> Sp;
  Sp.push(n+1);
  vector<int> vpravo(n+2);
  for (int i = n; i >= 1; i--) {
    while (vysky[Sp.top()] >= vysky[i]) Sp.pop();
    vpravo[i] = Sp.top();
    Sp.push(i);
  }

  int best = 0, beststart = 0, bestend = 0, bestv = 0;
  for (int i = 1; i < n+2; i++) {
    int plocha = (vpravo[i] - vlavo[i] - 1) * vysky[i];
    if (plocha > best) {
      best = plocha;
      bestv = vysky[i];
      beststart = vlavo[i];
      bestend = vpravo[i]-1;
    }
  }
  for (int i = 0; i < beststart; i++) printf("0 ");
  for (int i = beststart; i < bestend; i++) printf("%d ", bestv);
  for (int i = bestend; i < n; i++) printf("0 ");
  printf("\n");
}
```

(Existujú aj iné efektívne riešenia, napr. riešenie v čase $O(n \log n)$, ktoré domy usporiada podľa výšky a postupne spracúva, začínajúc najvyšším. Počas spracovania potom potrebujeme efektívne vedieť povedať, koľko už spracovaných domov leží bezprostredne naľavo/napravo od práve spracúvaného. Toto vieme pomocou vhodnej dátovej štruktúry spraviť v konštantnom čase.)

## B-II-4 Kino

Ako riešiť takýto typ úloh? V princípe máme dve možnosti. Môžeme si sami najprv zadanú úlohu vyriešiť optimálne, teda nájsť skutočne korektný algoritmus, a potom skúmať, kde sa správne riešenie zachová inak ako to zo zadania. Nevýhodou tohto prístupu je, že pri niektorých problémoch nemusí byť optimálny algoritmus vôbec jednoduchý a jeho hľadaním možno zbytočne stratíme čas.

Druhý prístup je trochu lenivý a priamočiarejší. Zoberieme ľubovoľný vstup a pozrieme sa na to, čo preň spravia jednotlivé algoritmy. Ak je niektorý z nich horší od iného, vieme, že ten horší algoritmus nemôže byť optimálny. Takto by sa nám malo podariť aspoň o niektorých z algoritmov zistiť, že optimálne nie sú.

Ešte o niečo lepší (ale menej lenivý) je prístup, pri ktorom si zvolíme takú sadu intervalov, pre ktorú poznáme správnu odpoveď, alebo ju aspoň vieme nájsť. Potom môžeme porovnávať výstupy všetkých algoritmov s touto odpoveďou.

Jedna možná technika, ako rýchlo nejaké takéto sady vyrobiť, je začať so sadou zo zadania: $[1, 3]$, $[3, 5]$, $[4, 7]$ a $[5, 6]$. Tieto intervaly môžeme teraz meniť tak, aby sa správna odpoveď nezmenila, ale beh jednotlivých programov áno. Najľahšie bude, ak program donútime pokaziť už výber prvého intervalu.

**Berieme ten, ktorý začína najskôr:**

Na akom vstupe by tento program nedal správnu odpoveď? Keď ho chceme donútiť, aby zlyhal, musíme mu podhodiť taký interval, ktorý určite nie je v optimálnom riešení, ale začína čo najskôr. To je ale ľahké: nakoľko nás zaujíma len jeho začiatok, tak jeho koniec môžeme posunúť, ako sa nám páči. Aby sme toho pokazili čo najviac, tak stačí, aby sa tento film prekrýval so všetkými ostatnými. Zoberieme teda vstup zo zadania a pridáme ešte nový film $[0, 10]$. Práve skúmaný algoritmus zoberie tento film, lebo začína najskôr, a následne už nestihne žiadny iný. Stihneme teda iba jeden film, a to nie je optimálne.

**Berieme ten najkratší:**

Ako nám môže krátky film (nazvyme ho $F$) pokaziť správne riešenie? Znovu potrebujeme, aby sa prekrýval z niektorými filmami zo správneho riešenia. S koľkými sa ale môže prekrývať? Vieme, že v správnom riešení sa žiadne dva neprekrývajú. Ak by sa potom $F$ prekrýval aspoň s tromi, tak by musel aspoň jeden prekrývať úplne. To ale znamená, že by bol dlhší, a teda by sme ho nevybrali. Vieme ho teda prekryť s maximálne dvomi inými. Ak teda zoberieme

vstup, pre ktorý bude optimálne vybrať dva filmy a $F$ sa s obomi bude prekrývať a ostane najkratší, tak sme vyhrali. Stačí nám teda zobrať dva veľmi dlhé filmy, ktoré sú hneď za sebou. Tie potom nie je problém prekryť jedným krátkym. Napríklad môžeme teda uvažovať nasledujúce tri filmy: $[0, 10]$, $[11, 20]$ a $[10, 11]$. Náš algoritmus by si vybral iba tretí film, ale v skutočnosti je optimálne ísť na prvý a druhý.

(Ako zaujímavosť si môžete všimnúť, že vieme zaručiť, že počet filmov, ktoré navštívi tento algoritmus, je vždy aspoň polovicou optimálneho počtu.)

**Berieme ten, ktorý končí najskôr:**

Čo teraz? Ak všetky pokusy o nájdenie protipríkladu zlyhali, môžeme sa pokúsiť dokázať, že tento postup naozaj vedie k optimálnemu výberu. Ako sa niečo také ale niečo také dokazuje?

Predstavme si, že poznáme jedno optimálne riešenie pre nejaký vstup. Dokážeme, že aj to, čo vygeneroval náš program, je optimálne (teda presne rovnako dobré) riešenie. Usporiadajme si filmy v oboch riešeniach chronologicky. Takto dostaneme dve postupnosti filmov: jednu optimálnu a jednu našu.

Pozrime sa na film $F$, ktorý zo všetkých úplne prvý končí. Ten sme zobrali v našom riešení ako prvý. Ak optimálne riešenie tiež zobralo $F$, je všetko OK a môžeme pokračovať ďalej. Nech teda optimálne riešenie zobralo iný film $Q$. Podľa výberu $F$ vieme, že $F.koniec \leq Q.koniec$. To ale znamená, že v optimálnom riešení môžeme *namiesto* filmu $Q$ ísť na film $F$: akýkoľvek ďalší film, ktorý stíhame po skončení $Q$, stíhame aj po skončení $F$. Opakovaním tejto úvahy pre nasledujúce filmy postupne prerobíme pôvodné optimálne riešenie na to naše, a pritom sa nikdy nezmení celkový počet videných filmov. A teda aj naše riešenie muselo byť optimálne.

Dôkaz inými slovami: Nejaký film musíme vidieť ako prvý. No a film $F$ je najlepšou voľbou, pretože v porovnaní s ostatnými nám „vyškrtne“ najmenej iných filmov, ktoré už nebudeme stíhať. Samotný algoritmus len opakuje túto úvahu až kým sa mu neminú filmy.

**Berieme ten, ktorý nám vyškrtne najmenej iných:**

Tu to už nebude také priamočiare. Je to tak preto, lebo aj samotné optimálne riešenie vyberá také filmy, ktoré nám nechajú čo najväčšiu množinu nevyškrtnutých. Dokonca sme veľmi podobnú formuláciu uviedli aj v dôkaze. To je dôvod, prečo nájsť protipríklad pre tento algoritmus nie je jednoduché. Ukážeme ale, že takto formulovaný algoritmus nie je vždy optimálny. Problém bude v tom, že tu nemáme tú záruku, že je náš výber filmu lepší od všetkých ostatných. Uká-

žeme si, že nastane podobný problém ako keď sme vyberali od najkratšieho – výberom jedného filmu vyškrtneme dva iné, ktoré by bolo bývalo lepšie zobrať.

Podobne ako pri výbere najkratšieho si položme otázku: keď vyberieme film, ktorý sa prekrýva s najmenej inými, koľko najviac filmov nám mohol vyškrtnúť z optimálneho riešenia? Znovu platí, že odpoveď je „najviac dva“ – zjavne nemôže žiaden iný film prekrývať celý. A ak chceme nájsť protipríklad, tak potrebujeme práve situáciu, kde tento film prekrýva dva z optimálneho riešenia.

Keďže náš film prekrýva dva iné a máme ho vybrať ako prvý, tak aj každý iný film musí prekrývať aspoň dva iné. Poďme teda vyrobiť takú množinu filmov. Začnime tým, že si zvolíme (jediné) optimálne riešenie. Nech je napríklad tvorené štyrmi filmami: $[1, 3]$, $[4, 6]$, $[7, 9]$, $[10, 12]$. Náš film, ktorým chceme algoritmus oklamať, bude $[6, 7]$. (Prekrýva sa teda s druhým a tretím optimálnym filmom.) No a teraz už len potrebujeme zabezpečiť, aby sa všetky optimálne filmy prekrývali s viac ako dvoma inými. Pridáme teda tri filmy $[3, 4]$ a tri filmy $[9, 10]$.

Každý z optimálnych filmov sa teda prekrýva s aspoň tromi inými. Každý z filmov $[3, 4]$ a $[9, 10]$ sa prekrýva so štyrmi inými filmami. Náš algoritmus teda ako prvý naozaj vyberie film $[6, 7]$. No a následne sa mu už podarí vybrať len jeden film skôr a jeden neskôr, skončí teda s tromi filmami namiesto štyroch – nie je optimálny.

# Riešenia celoštátneho kola kategórie A

## A-III-1 Pán Buridan a kaviarne

Začnime triviálnym riešením: Pre každú z $n^2$ križovatiek overíme, či neexistujú dve kaviarne s rovnakou vzdialenosťou od tejto križovatky tak, že vyskúšame všetky dvojice kaviarní.

Vzdialenosť dvoch bodov $[x_1, y_1]$ a $[x_2, y_2]$ v štvorcovej mriežke vypočítame v konštantnom čase ako $|x_1 - x_2| + |y_1 - y_2|$. Označme počet kaviarní $k$, potom všetkých dvojíc kaviarní je $O(k^2)$. Naše prvé riešenie má teda časovú zložitosť $O(n^2k^2)$. Keďže kaviarní môže byť až toľko čo križovatiek, v najhoršom prípade dostávame zložitosť $O(n^6)$.

Skúšať všetky dvojice kaviarní je však zbytočné – zaujíma nás iba to, či sú nejaké dve rovnako vzdialené od vybranej križovatky. Stačilo by teda vygenerovať zoznam vzdialeností k jednotlivým kaviarniam, usporiadať ho a potom jedným prechodom overiť, či sa v zozname nenachádza nejaká vzdialenosť viackrát. Dostali by sme riešenie v čase $O(n^2k \log k)$, čo vieme zhora odhadnúť ako $O(n^4 \log n)$.

Pozrime sa teraz, aké vzdialenosti sa v tomto zozname môžu objaviť. Ak sa nejaká kaviareň nachádza priamo na vybranej križovatke, jej vzdialenosť je 0, čo je zrejme najmenšia možná hodnota. Naopak, najväčšiu vzdialenosť medzi sebou dosahujú dve križovatky v protiľahlých rohoch štvorcovej siete – sú vzdialené $2n-2$. Všetky vzdialenosti v zozname sú teda z rozsahu $0, 1, \ldots, 2n-2$, takže ich vieme usporiadať v lineárnom čase, napríklad CountSort-om, a tým zlepšiť časovú zložitosť na $O(n^2k)$.

V rovnakej zložitosti vieme úlohu vyriešiť aj jednoduchšie: Pre každú križovatku budeme postupne prechádzať všetky kaviarne, pre každú sa pozrieme na jej vzdialenosť a v poli `bool`-ov si zaznačíme, že sme dotyčnú vzdialenosť už videli. Akonáhle sa nám nejaká zopakuje, práve spracúvanú križovatku prehlásime za nevhodnú pre Dávidka.

Ako sme už spomínali, kaviarní môže byť až $n^2$, čiže toto riešenie má prinajhoršom zložitosť $O(n^4)$. Všimnime si ale, že ak je kaviarní priveľa (viac ako $2n-1$), žiadna križovatka v Manhattane nemôže Dávidkovi vyhovovať. Vyplýva to práve z toho, že v celom Manhattane existuje len $2n-1$ rôznych vzdialeností. Ak teda máme $2n$ a viac kaviarní, nemôže existovať žiadna dobrá križovatka –

vždy totiž máme viac kaviarní ako vzdialeností od nej, a teda musia niektoré dve kaviarne ležať v tej istej vzdialenosti.[3]

Toto pozorovanie nás vedie k jednoduchému vylepšeniu: Ak je $k$ väčšie ako $2n-1$, pre každú križovatku vypíšeme, že na nej Dávidko nemôže bývať. Iba v opačnom prípade spustíme naše riešenie s časovou zložitosťou $O(n^2k)$. Časovú zložitosť tohto vylepšeného riešenia teraz môžeme zhora odhadnúť ako $O(n^3)$.

Aj vo vzorovom riešení samostatne ošetríme prípad s $2n$ a viac kaviarňami. Ďalej však budeme postupovať akoby z opačnej strany: Pre každú dvojicu kaviarní nájdeme tie križovatky, ktoré sú od oboch kaviarní vzdialené rovnako.

Zafarbime si križovatky ako šachovnicu. Keďže sa v Manhattane môžeme pohybovať len o jednu ulicu v štyroch základných smeroch, každým krokom sa zmení farba križovatky, na ktorej stojíme. To ale znamená, že ak sú dve kaviarne rovnako vzdialené od nejakej križovatky, potom tieto kaviarne musia ležať na križovatkách tej istej farby.

| 5 | 4 | 3 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 4 | 3 | 2 | 1 | 2 | 3 | 4 |
| 3 | 2 | 1 | 0 | 1 | 2 | 3 |
| 4 | 3 | 2 | 1 | 2 | 3 | 4 |
| 5 | 4 | 3 | 2 | 3 | 4 | 5 |

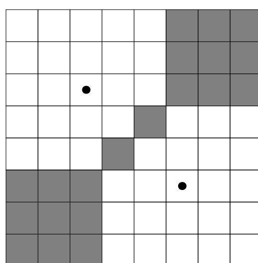
Obr. 1: Vzdialenosti od stredného políčka. Všimnite si, že políčka s rovnakou vzdialenosťou majú rovnakú farbu.

Stačí sa teda venovať len dvojiciam kaviarní s rovnakou farbou. Rozoberme dva prípady.

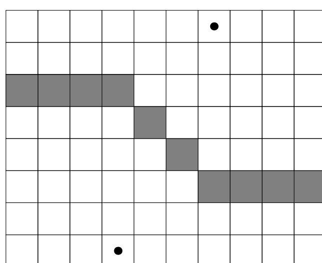
Prvý prípad: Kaviarne ležia na tej istej uhlopriečke, teda určujú štvorec. Ľahko sa presvedčíme, že hľadané križovatky (tie, ktoré sú rovnako vzdialené od oboch kaviarní) sú tie na opačnej uhlopriečke tohto štvorca a tiež v dvoch rohových oblastiach.

Druhý prípad: Kaviarne neležia na tej istej uhlopriečke, teda určujú obdĺžnik. Tentoraz sa hľadané križovatky nachádzajú na jednej lomenej čiare, znázornenej na nasledujúcom ilustračnom obrázku:

[3] Hodnota $2n-1$ je iba horné ohraničenie. Napríklad od stredu Manhattanu sa ostatné križovatky nachádzajú len v rádovo $n$ rôznych vzdialenostiach. Teda napr. už pre $n+47$ kaviarní vznikne v strede Manhattanu zóna križovatiek ktoré sú zaručene zlé. Toto ale nebudeme



Obr. 2: Sivou farbou sú označené križovatky s rovnakou vzdialenosťou od obidvoch kaviarní (čierne krúžky).



Obr. 3: Križovatky rovnako vzdialené od oboch kaviarní v druhom prípade.

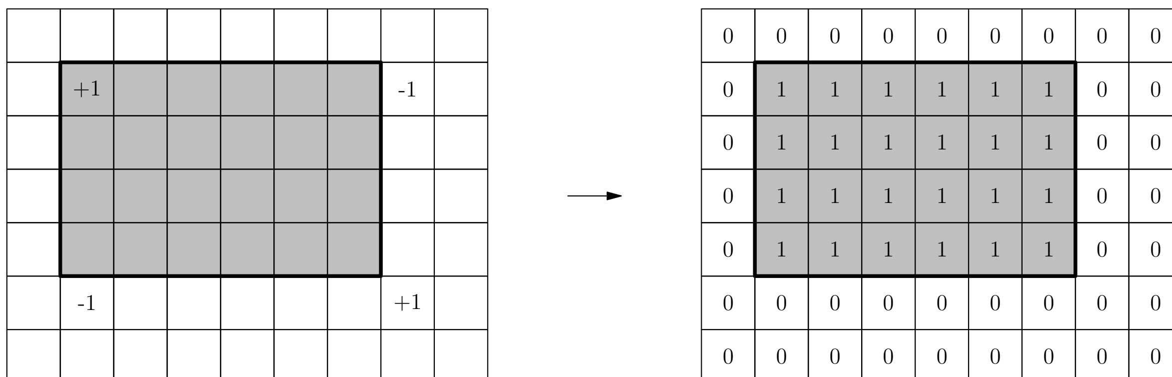
Zostáva nám nájsť zjednotenie týchto oblastí pre všetky dvojice kaviarní. Na to použijeme techniku z domáceho kola – prefixové súčty. Pre každú križovatku spočítame, kvôli koľkým dvojiciam kaviarní je táto križovatka pre Dávidka nevhodná.

Vezmime si najprv obdĺžnikové oblasti. Ak kvôli nejakej dvojici kaviarní nemôže Dávidko bývať v obdĺžniku s ľavým horným rohom na križovatke $[x_1, y_1]$ a pravým dolným rohom na križovatke $[x_2, y_2]$, potom si do pomocného poľa poznačíme nasledovné hodnoty:

- $+1$ na križovatku $[x_1, y_1]$
- $-1$ na križovatku $[x_1, y_2 + 1]$
- $-1$ na križovatku $[x_2 + 1, y_1]$
- $+1$ na križovatku $[x_2 + 1, y_2 + 1]$

Ak potom na tomto poli spočítame dvojrozmerné prefixové súčty, dostaneme jednotky práve vnútri nášho obdĺžnika:

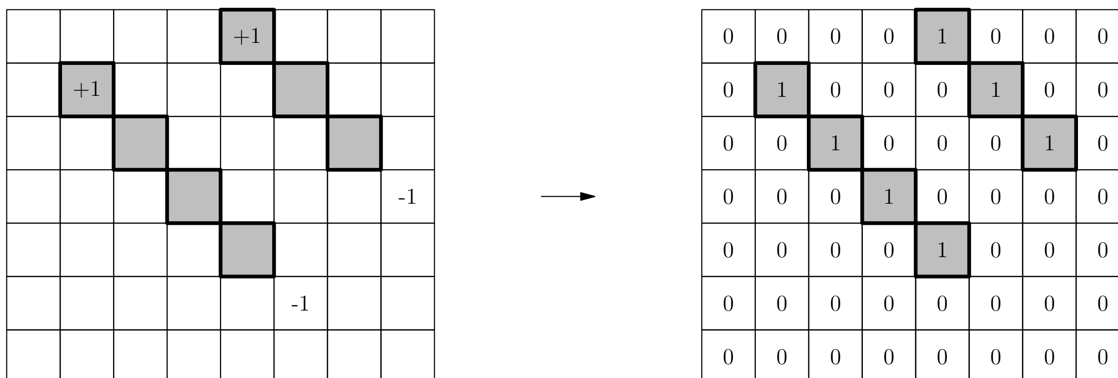
potrebovať, vystačíme si so slabším odhadom $2n - 1$ ktorý platí pre všetky políčka.




Obr. 4: Prefixové súčty pre obdĺžniky.

Rovnako to funguje aj s viacerými obdĺžnikmi – za každý najprv pričítame štyri hodnoty $+1/-1$ do pomocného poľa a potom prefixovými súčtami zistíme pre každú križovatku počet obdĺžnikov, ktoré ju prekrylo.

Okrem obdĺžnikových oblastí sa musíme vysporiadať aj s vodorovnými a zvislými čiarami – tie sú ale vlastne tiež obdĺžniky (s výškou/šírkou jeden), teda na nich funguje rovnaká technika. Zostali ešte šikmé čiary. Tie vyriešime podobne – jednorozmernými prefixovými súčtami v smere diagonál.




Obr. 5: Prefixové súčty pre uhlopriečky.

Na záver ešte zhrňme vzorové riešenie. Ak je kaviarní viac ako $2n - 1$, vypíšeme samé `N`. V opačnom prípade pre každú dvojicu kaviarní nájdeme križovatky, od ktorých sú tieto kaviarne vzdialené rovnako. Hranice útvarov, ktoré tieto križovatky tvoria, si poznačíme do pomocného poľa (pre každú dvojicu kaviarní nám to potrvá len $O(1)$). Nakoniec vypočítame prefixové súčty na pomocných poliach, na základe čoho budeme pre každú križovatku vedieť, či k nej

existuje dvojica rovnako vzdialených kaviarní.

Celková časová zložitosť vzorového riešenia je teda $O(n^2)$.

## Listing programu (Python)

```
n = int(input())
A = [[int(x) for x in input().split()] for i in range(n)]

kaviarne = []
for i in range(n):
    for j in range(n):
        if A[i][j]:
            kaviarne.append((i, j))
k = len(kaviarne)

def pridaj_obdlznik(S, x1, y1, x2, y2):
    if 0 <= x1 <= x2 < n and 0 <= y1 <= y2 < n:
        S[y1][x1] += 1
        S[y1][x2 + 1] -= 1
        S[y2 + 1][x1] -= 1
        S[y2 + 1][x2 + 1] += 1

def pridaj_kus_uhlopriecky(U, cislo, od, do):
    if 0 <= od <= do < len(U[cislo]):
        U[cislo][od] += 1
        U[cislo][do + 1] -= 1

R = [[False] * n for i in range(n)]
if k <= 2 * n - 1:
    # pomocné pole pre obdĺžniky
    S = [[0] * (n + 1) for i in range(n + 1)]
    # pomocné pole pre uhlopriečky zľava hore doprava dole
    UH = [[0] * (n + 1) for i in range(2 * n - 1)]
    # pomocné pole pre uhlopriečky zľava hore doprava dole
    UV = [[0] * (n + 1) for i in range(2 * n - 1)]
    for k1 in range(len(kaviarne)):
        for k2 in range(k1 + 1, len(kaviarne)):
            y1, x1 = kaviarne[k1]
            y2, x2 = kaviarne[k2]
            if x1 > x2:
                x1, y1, x2, y2 = x2, y2, x1, y1
            hlavna = (y1 <= y2)
            # ležia na rovnakej farbe?
            if (x1 + y1) % 2 != (x2 + y2) % 2:
                continue
            if abs(x1 - x2) == abs(y1 - y2):
                # kaviarne ležia na spoločnej uhlopriečke
                if hlavna:
                    pridaj_obdlznik(S, 0, y2, x1, n - 1)
                    pridaj_obdlznik(S, x2, 0, n - 1, y1)
                    pridaj_kus_uhlopriecky(UV, x1 + y2, y1, y2)
                else:
                    pridaj_obdlznik(S, 0, 0, x1, y2)
                    pridaj_obdlznik(S, x2, y1, n - 1, n - 1)
                    pridaj_kus_uhlopriecky(UH, x1 - y2 + n - 1, x1, x2)
            elif abs(x1 - x2) > abs(y1 - y2):
                # ... neležia a bounding box je širší ako vyšší
                posun = (abs(x1 - x2) - abs(y1 - y2)) // 2
                if hlavna:
                    pridaj_obdlznik(S, x1 + posun, y2 + 1, x1 + posun, n - 1)
                    pridaj_obdlznik(S, x2 - posun, 0, x2 - posun, y1 - 1)
                    pridaj_kus_uhlopriecky(UV, x1 + y1 + posun + abs(y1-y2),
```

```
                                           y1, y2)
                else:
                    pridaj_obdlznik(S, x1 + posun, 0, x1 + posun, y2 - 1)
                    pridaj_obdlznik(S, x2 - posun, y1 + 1, x2 - posun, n - 1)
                    pridaj_kus_uhlopriecky(UH, x1 - y1 + posun + abs(y1-y2) + n-1,
                                           x1 + posun, x2 - posun)
            else:
                # ... neležia a bounding box je vyšší ako širší
                posun = (abs(y1 - y2) - abs(x1 - x2)) // 2
                if hlavna:
                    pridaj_obdlznik(S, 0, y2 - posun, x1 - 1, y2 - posun)
                    pridaj_obdlznik(S, x2 + 1, y1 + posun, n - 1, y1 + posun)
                    pridaj_kus_uhlopriecky(UV, x1 + y1 + posun + abs(x1-x2),
                                           y1 + posun, y2 - posun)
                else:
                    pridaj_obdlznik(S, 0, y2 + posun, x1 - 1, y2 + posun)
                    pridaj_obdlznik(S, x2 + 1, y1 - posun, n - 1, y1 - posun)
                    pridaj_kus_uhlopriecky(UH, x1 - y1 + posun + abs(x1-x2) + n-1,
                                           x1, x2)
    for i in range(n):
        for j in range(n):
            if i > 0:
                S[i][j] += S[i - 1][j]
                UV[j + i][i] += UV[j + i][i - 1]
            if j > 0:
                S[i][j] += S[i][j - 1]
                UH[j - i + n - 1][j] += UH[j - i + n - 1][j - 1]
            if i > 0 and j > 0:
                S[i][j] -= S[i - 1][j - 1]
            R[i][j] = (S[i][j] == 0 and UH[j-i+n-1][j] == 0 and UV[j + i][i] == 0)

for x in R:
    print(' '.join(('A' if y else 'N') for y in x))
```

## A-III-2 Preťahovanie lanom

Stručný popis riešenia: Na polynomiálnu časovú zložitosť nám stačí použiť dynamické programovanie, pri ktorom si pamätáme, koľko mladých a koľko starých už odišlo. Lepšie riešenia sú založené na dodatočnom pozorovaní že vždy existuje optimálne riešenie, v ktorom najskôr odchádzajú starí a až potom mladí hráči. Pre konkrétny počet starých hráčov ktorí odídu vieme maximálny počet mladých nájsť binárnym vyhľadávaním. Existuje aj ešte šikovnejšie riešenie, ale pri ňom je treba dať dobrý pozor na detaily.

**Predpočítanie:**

Ľahko nahliadneme, že v ľubovoľnom okamihu tvorí každý z tímov súvislý úsek hráčov zo vstupu. Aby sme vedeli v konštantnom čase povedať súčet síl hráčov v ľubovoľnom takomto úseku, stačí, keď si predpočítame prefixové súčty síl hráčov. Formálne, nech $s_1, \ldots, s_n$ sú sily hráčov, ktoré sme dostali na vstupe (v poradí od najmladšieho po najstaršieho). Definujme $p_0 = 0$ a $\forall i : p_{i+1} =$

$p_i + s_{i+1}$. Tieto hodnoty vieme vypočítať v čase $O(n)$ a zjavne platí, že $p_i$ je súčet síl $i$ najmladších hráčov.

Uvažujme teraz úsek hráčov, ktorý začína $i$-tym a končí $j$-tym najmladším. Toho súčet síl môžeme zjavne pomocou predpočítaných prefixových súčtov vyjadriť ako $p_j - p_{i-1}$.

(Poznámka: Existujú aj riešenia, ktoré si vedia poradiť aj bez tohto predpočítania – ak vieme, aké boli sily oboch tímov, vieme v konštantnom čase prepočítať, ako sa zmenili odchodom jedného z hráčov. Keďže však toto predpočítanie vieme spraviť v lineárnom čase, teda v podstate zadarmo, budeme ho pre jednoduchosť používať aj v riešeniach, ktoré by sa bez neho zaobišli.)

**Skúšame všetky možnosti:**

Začneme jednoduchým rekurzívnym riešením, ktoré vyskúša všetky možné priebehy turnaja (teda všetky možné poradia odchádzajúcich hráčov) a vyberie najlepší z nich.

V každom kroku toto riešenie skontroluje, či ešte turnaj neskončil, a ak nie, tak postupne vyskúša obe možnosti pre nasledujúcu zmenu (t.j. raz pošle preč najmladšieho a raz najstaršieho z ešte hrajúcich), pre každú z nich rekurzívne nájde ako najdlhšie ešte mohol turnaj trvať, a následne vráti lepšiu z oboch možností (plus jedna za aktuálny zápas).

Akú má toto riešenie časovú zložitosť? V najhoršom možnom prípade môže turnaj mať až $n - k + 1$ zápasov. Ak by toto nastalo pre každé možné poradie odchodu hráčov, vyskúšali by sme až $2^{n-k}$ rôznych poradí, a aj naša časová zložitosť by teda bola $\Theta(2^{n-k})$. A tento najhorší možný prípad skutočne nastáva – napr. v situácii kedy je kopec taký strmý, že kým má dolný tím aspoň jedného hráča, vždy vyhrá.

**Memoizácia:**

V predchádzajúcom riešení je ľahko vidieť, prečo je neefektívne – zbytočne zas a znova počítame odpoveď na tie isté otázky. Rekurzívna funkcia `solve` počas výpočtu volá samú seba exponenciálne veľa ráz. My si však môžeme všimnúť, že v skutočnosti je *rôznych* volaní funkcie `solve` veľmi málo. Parametre `lo` a `hi` majú vždy hodnoty z rozsahu 0 až $n$. A navyše dokonca vždy platí, že `hi` je aspoň o $k$ väčšie ako `lo`, inak by už turnaj dávno skončil. Počas celého behu predchádzajúceho riešenia sa teda dokola počíta hodnota funkcie `solve` pre iba $O((n-k)^2)$ rôznych vstupov.

Štandardnou technikou, ako takýto algoritmus zefektívniť, je *memoizácia*: vždy, keď prvýkrát vypočítame nejakú návratovú hodnotu funkcie `solve`, zapa-

mätáme si ju. A vždy, keď v budúcnosti náš program zavolá funkciu `solve` s tými istými parametrami, namiesto toho, aby sme ju znova vyhodnotili (pod čím si treba predstaviť celý strom rekurzívnych volaní), jednoducho rovno v konštantnom čase dáme na výstup zapamätanú hodnotu.

Takto vylepšený algoritmus bude až prekvapivo efektívny. Jeho časovú zložitosť môžeme odhadnúť nasledovne: pre každú platnú kombináciu parametrov `lo` a `hi` sa telo funkcie `solve` (teda tá jej časť, ktorú sme mali už v predchádzajúcom riešení) vykoná najviac jedenkrát. A samotné vykonanie tela funkcie `solve` prebehne v konštantnom čase.[4] Preto je celková časová zložitosť nanajvýš priamo úmerná počtu rôznych vstupov, pre ktoré potrebujeme funkciu `solve` vyhodnotiť – a teda je časová zložitosť nášho algoritmu $O((n-k)^2)$.

Naša implementácia uvedená nižšie má o chlp horšiu časovú zložitosť $O(n^2)$ kvôli inicializácii tabuľky, v ktorej si pamätáme vypočítané hodnoty. Toto by sa pochopiteľne dalo spraviť aj lepšie, ale bolo by to na úkor čitateľnosti programu.

## Listing programu (Python)

```
data = input().split()
N, K, Q = int( data[0] ), int( data[1] ), float( data[2] )
S = [ int(x) for x in input().split() ]

P = [0]
for s in S: P.append( P[-1]+s )

memo = [ [ None for hi in range(N+1) ] for lo in range(N+1) ]

def solve(lo,hi):
    # vráti maximálny počet zápasov pre turnaj,
    # v ktorom ešte hrajú hráči s číslami lo..hi-1 (číslované od 0)

    # ak už poznáme odpoveď pre tieto vstupy, rovno ju vrátime
    if memo[lo][hi] is not None:
        return memo[lo][hi]

    # ak túto kombináciu (lo,hi) vidíme prvýkrát, vyriešime ju
    sucet_hore = P[hi] - P[hi-K]
    sucet_dole = P[hi-K] - P[lo]
    if sucet_hore > Q*sucet_dole + 1e-9: # týmto zápasom turnaj končí
        answer = 1
    else: # turnaj bude pokračovať, vyberieme lepšiu možnosť koho poslať preč
        a1 = solve(lo+1,hi)
        a2 = solve(lo,hi-1)
        answer = 1 + max(a1,a2)

    # a pred tým ako vrátime odpoveď si ju zapamätáme
    memo[lo][hi] = answer
    return answer

print( solve(0,N) )
```

[4]Všimnite si, že do tohto konštantného času nerátame prípadné rekurzívne volania. Tie totiž buď tiež vrátia výstup okamžite, alebo ich započítame inokedy.

**Dynamické programovanie:**

To isté riešenie ako v predchádzajúcej časti vieme implementovať aj v iteratívnej podobe: pôjdeme napríklad postupne od menších počtov hráčov k väčším. V okamihu, kedy potrebujeme zistiť, ako dlho môže trvať turnaj, ak ešte hrajú hráči s číslom 4 až 17, už vieme aj najdlhšie trvanie pre turnaj hraný hráčmi s číslom 5 až 17, aj trvanie pre turnaj s hráčmi 4 až 16. Vieme teda v konštantnom čase vypočítať optimálnu dĺžku trvania pre práve spracúvaný turnaj.

**Zjednodušenie úlohy:**

Skôr, než sa pustíme do lepších riešení, si trochu zjednodušíme problém, ktorý ideme riešiť.

Najskôr ošetríme špeciálny prípad, keď turnaj skončí hneď prvým zápasom. Odteraz teda budeme predpokladať, že existuje riešenie tvorené aspoň dvoma zápasmi.

Všimnime si teraz *predposledný* zápas v *optimálnom* riešení. Po tomto zápase musí byť jedno, ktorý hráč odíde – obe možnosti musia viesť k zápasu v ktorom horný tím vyhrá. Každé optimálne riešenie teda končí postupnosťou „predposledný zápas – hocikto odíde – posledný zápas – koniec“. Túto časť riešenia odteraz budeme ignorovať.

Formálne si náš problém upravíme nasledovne: Pre konkrétne $i$ a $j$ budeme hovoriť, že stav turnaja, v ktorom majú aktívni hráči čísla $i$ až $j$, je *živý*, ak by nasledujúcim zápasom ešte turnaj neskončil. Inými slovami, v živom stave je ešte horný tím prislabý v porovnaní s dolným.

Namiesto pôvodnej úlohy budeme teraz riešiť ekvivalentnú: nájsť najdlhšiu postupnosť odobratí hráča (vždy najmladšieho alebo najstaršieho) takú, že všetky stavy turnaja, ktoré počas odoberania nastanú, sú živé – a to vrátane stavu po odobratí posledného hráča.

**Skoro optimálne riešenie:**

Ak chceme ešte lepšie riešenie ako to, ktoré sme vyššie dosiahli použitím memoizácie, nemôžeme si dovoliť ani len sa pozrieť na všetky možné dosiahnuteľné stavy počas turnaja. Potrebujeme teda nájsť nejaké kritérium, ktoré nám umožní sústrediť sa len na niektoré z nich. Prípadne sa namiesto konkrétnych stavov môžeme zamerať na hľadanie konkrétnych *priebehov* turnaja. Naším cieľom je teda objavenie nejakého tvrdenia tvaru „Vždy bude existovať optimálny priebeh turnaja, ktorý spĺňa [túto dodatočnú vlastnosť].“

Poďme sa teda pozrieť na to, v akom poradí sa najviac oplatí hráčov odoberať. Predpokladajme, že sa niekedy počas riešenia našej novej úlohy odohrala

nasledovná postupnosť udalostí:

- Sme v živom stave $S_0$ tvorenom hráčmi $i$ až $j$.
- Odišiel najmladší hráč (číslo $i$).
- Sme v živom stave $S_1$ tvorenom hráčmi $i+1$ až $j$.
- Odišiel najstarší hráč (číslo $j$).
- Sme v živom stave $S_2$ tvorenom hráčmi $i+1$ až $j-1$.

Pozrime sa na stav $S_2$. Keďže je tento stav živý, hráči na vrchu kopca v nasledujúcom zápase ešte nevyhrajú. V stave $S_2$ sú na vrchu kopca hráči s číslami $j-k$ až $j-1$ a na jeho spodku hráči s číslami $i+1$ až $j-k-1$.

Predstavme si teraz, že by sme *najskôr* poslali preč hráča číslo $j$ a až potom hráča číslo $i$. Čo by sa tým zmenilo?

Stav $S_2$ by sa tým nezmenil vôbec – stále by sme v ňom mali hráčov s číslami $i+1$ až $j-1$. Zmenil by sa stav $S_1$. V tom by teraz dole stáli hráči s číslami $i$ až $j-k-1$ a hore hráči s číslami $j-k$ až $j-1$. No a teraz príde dôležité pozorovanie: *aj tento stav musí byť živý.* Prečo? Lebo je to ten istý stav ako $S_2$, len *navyše* máme dole aj hráča s číslom $i$. Tým skôr je teda dolný tím dostatočne silný.

Túto úvahu môžeme ľubovoľne veľakrát zopakovať. Ak teda začneme s ľubovoľným riešením, vieme postupne vymieňať kroky, kedy posielame preč mladých hráčov, s krokmi, kedy posielame preč starých. Na konci takto dostaneme *rovnako dobré* riešenie, v ktorom *najskôr* pošleme preč niekoľko starých hráčov a *až následne* niekoľko mladých. Platí teda nasledovné tvrdenie: **Vždy existuje optimálne riešenie** (našej upravenej úlohy), **v ktorom najskôr posielame preč najstarších hráčov a až potom najmladších.**

(Ten istý argument inými slovami: predstavme si, že už vieme, ktorých starých a ktorých mladých hráčov pošle preč optimálne riešenie. Potom určite môžeme poslať preč najskôr tých starých – zatiaľ totiž všetci mladí, ktorých časom chceme poslať preč, ešte stoja na spodku kopca a pomáhajú tak dolnému tímu. Ak by sme mladých poslali preč priskoro, len tým dolnému tímu uškodíme.)

Na základe práve dokázaného tvrdenia ľahko navrhneme riešenie s časovou zložitosťou $O((n-k)\log(n-k))$. V tomto riešení postupne vyskúšame všetky možnosti pre to, koľko najstarších hráčov postupne odoberieme. (Počas tohto skúšania nezabudneme kontrolovať, či sú všetky stavy, cez ktoré ideme počas odoberania najstarších hráčov, živé.)

Keď už máme pevne zvolený počet $p$ odobratých najstarších hráčov, máme vlastne pevne zvolenú $k$-ticu hráčov, ktorí budú stáť na vrchu kopca počas

toho ako my postupne odoberáme najmladších hráčov. Najmladších hráčov však nebudeme odoberať po jednom. Namiesto toho použijeme efektívnejšiu metódu: binárnym vyhľadávaním na intervale od 0 po $n-k-p$ nájdeme najväčšie $x$ také, že po odobratí $p$ najstarších a následne $x$ najmladších hráčov ešte stále budeme mať živý stav.

## Listing programu (Python)

```
# načítanie a predspracovanie vyzerá rovnako ako v predchádzajúcom riešení

def je_zivy(mladych,starych):
    sucet_hore = P[N-starych] - P[N-starych-K]
    sucet_dole = P[N-starych-K] - P[mladych]
    return sucet_hore <= Q*sucet_dole + 1e-9

# ošetríme špeciálny prípad keď turnaj končí prvým zápasom
if not je_zivy(0,0):
    print(1)
    from sys import exit
    exit()

odpoved = 0
for starych in range(0,N-K+1):
    if not je_zivy(0,starych):
        break # toľkoto ani viac starých hráčov už nemôžeme odobrať
    lo, hi = 0, N-K-starych
    # binárne vyhľadávame -- invariant: lo mladých ešte môžeme odobrať, hi už nie
    while hi-lo > 1:
        med = (lo+hi)//2
        if je_zivy(med,starych): lo = med
        else:                    hi = med
    odpoved = max( odpoved, lo+starych+2 )

print(odpoved)
```

### Nesprávna úvaha:

V tomto okamihu je veľmi ľahké nechať sa zlákať nasledujúcou nesprávnou úvahou: Začneme tým, že nájdeme optimálny počet najmladších pre 0 najstarších. Teraz budeme postupne, rovnako ako v predchádzajúcom riešení, počet najstarších postupne zväčšovať, a vždy, keď je to nutné, počet najmladších postupne zmenšovať. Takto dostaneme riešenie s lineárnou časovou zložitosťou.

Vyššie popísané riešenie síce má lineárnu časovú zložitosť, ale úvaha, ktorá k nemu vedie, je nesprávna. V nej totiž implicitne predpokladáme, že väčšiemu počtu odstránených starých hráčov musí nutne zodpovedať menší alebo rovný počet odstránených mladých hráčov. No a toto vôbec nie je pravda.

Ukážeme si to na konkrétnom príklade. Majme $q = 1.01$ (teda skoro rovinu) a zoberme napr. $n = 10$ ľudí, pričom $k = 3$ najstarší sú vždy na vrchu kopca. Sily (od najmladšieho) nech sú $(47, 1, 1, 1, 1, 1, 1, 1, 3, 1)$.

- Ak odíde 0 najstarších, vedia potom odísť 2 najmladší. Posledný živý stav má $1+1+1+1+1$ na spodku a $1+3+1$ na vrchu kopca.
- Ak odíde 1 najstarší, vie potom odísť **len 1** najmladší. Posledný živý stav má $1+1+1+1+1$ na spodku a $1+1+3$ na vrchu kopca.
- Ak ale odídu 2 najstarší, vedia potom odísť **znova až 2** najmladší. Posledný živý stav má $1+1+1$ na spodku a tiež $1+1+1$ na vrchu kopca.

**Optimálne riešenie:**

Ako opraviť úvahu z predchádzajúcej časti? Pôjdeme na to od konca. Začneme tým, že si zistíme, koľko najviac najstarších hráčov môžeme prípustným spôsobom odobrať. Následne sa pokúsime spraviť v podstate to isté ako v predchádzajúcom riešení: budeme postupne *zmenšovať* počet odobratých starých hráčov a zakaždým sa budeme snažiť *zväčšovať* počet odobratých mladých.

Ako sme ale videli vo vyššie uvedenom protipríklade, môžu nastať situácie, v ktorých by sme mali (na udržanie prípustnosti riešenia) počet odobratých mladých hráčov zmenšiť. Čo s takýmito situáciami? Jednoducho ich odignorujeme a počet mladých hráčov nezmenšíme. Takáto situácia totiž zjavne nemôže predstavovať optimálne riešenie – aj starých aj mladých hráčov by sme odobrali menej ako v inom riešení ktoré už poznáme.

Takto dostávame korektné riešenie s časovou zložitosťou $O(n)$.

### Listing programu (Python)

```
# predspracovanie, je_zivy() a ošetrenie špeciálneho prípadu sú rovnaké ako doteraz

mladych, starych = 0, 0
while je_zivy(mladych,starych+1): starych += 1
while je_zivy(mladych+1,starych): mladych += 1
odpoved = mladych+starych+2

while starych > 0:
    starych -= 1
    if not je_zivy(mladych,starych): continue # preskakujeme neoptimálnu možnosť
    while je_zivy(mladych+1,starych): mladych += 1
    odpoved = max( odpoved, mladych+starych+2 )

print(odpoved)
```

## A-III-3 Mimozemské počítače

V prvej podúlohe stačila drobná úprava zadaného grafu. V druhej sa dalo postupne po jednej odstraňovať hrany a overovať, či sme tak neodstránili hľadanú cestu. Efektívnejšie riešenie však vie vhodne odstrániť viac hrán naraz. V

poslednej, tretej podúlohe bolo treba každý vrchol pôvodného grafu nahradiť viacerými vrcholmi v novom grafe.

**Podúloha A (1 bod):**

V tejto podúlohe sme smeli raz zavolať funkciu `cesta_s_hranou(n,E,u,v)` a pomocou tohto volania sme chceli o našom grafe $G$ zistiť, či obsahuje Hamiltonovskú cestu.

Ak by sme funkciu `cesta_s_hranou` mohli volať viackrát, bolo by riešenie jednoduché: stačilo by túto funkciu postupne zavolať toľkokrát, koľko hrán má náš graf $G$, pričom ako $u$ a $v$ mu vždy dáme vrcholy spojené jednou z hrán. Alebo by stačilo zvoliť $u = 0$ a ako $v$ postupne vyskúšať všetky vrcholy, ktoré sú s 0 spojené. My však smieme funkciu `cesta_s_hranou` volať len raz. Ako na to?

Graf $G$ má vrcholy s číslami 0 až $n - 1$. My doň pridáme ešte dva vrcholy: $n$ a $n + 1$. Tieto spojíme medzi sebou, a navyše vrchol $n$ spojíme s každým z vrcholov 0 až $n - 1$.

Nech `E` je zoznam hrán takto upraveného grafu. Potom my zavoláme funkciu `cesta_s_hranou(n+2,E,n,n+1)`. Teda zistíme, či náš upravený graf obsahuje Hamiltonovskú cestu, na ktorej je hrana medzi vrcholmi $n$ a $n + 1$.

Správnosť algoritmu je zjavná, stačí si všimnúť, že v novom grafe *každá* Hamiltonovská cesta musí obsahovať hranu medzi $n$ a $n + 1$, a že nový graf má nejakú Hamiltonovskú cestu práve vtedy, ak mal nejakú Hamiltonovskú cestu pôvodný graf.

**Podúloha B (5 bodov):**

**Pomalšie riešenie.** Postupne prejdeme všetky hrany grafu $G$. Pre každú hranu zopakujeme nasledovný proces: Odoberieme ju z aktuálneho grafu, a následne volaním funkcie `cesta` overíme, či v grafe ešte ostala nejaká Hamiltonovská cesta. Ak ostala, práve spracúvanú hranu necháme odstránenú. Ak neostala, hranu vrátime späť.

Na konci tohto algoritmu nám zjavne musí ostať práve jediná Hamiltonovská cesta. (Je zjavné, že nám nejaká ostane, lebo po každom kroku máme graf, v ktorom aspoň jedna cesta existuje. Tiež je zjavné, že tam okrem nej už nemôžu byť žiadne iné hrany – každú takú hranu sme niekedy spracúvali, a keďže v danom okamihu existovala Hamiltonovská cesta na ktorej daná hrana neležala, nutne sme ju vyhodili.)

Tento algoritmus potrebuje presne $m$ volaní funkcie `cesta`. Pre husté grafy je $m$ rádovo rovné $n^2$.

**Lepšie riešenie.** Ukážeme si teraz riešenie, ktoré si vystačí s $O(n \log n)$ volaniami funkcie `cesta`. Existuje viacero podobne efektívnych riešení, my sme si vybrali jedno, ktoré sa ľahko implementuje. Hľadanú Hamiltonovskú cestu budeme zostrojovať postupne, vrchol za vrcholom.

Na začiatok by sme potrebovali vedieť, kde naša cesta začína. Toto vyriešime napríklad tak, že si graf upravíme rovnako ako v riešení podúlohy A. Teraz teda vieme, že prvé dva vrcholy na hľadanej Hamiltonovskej ceste sú vrcholy $n+1$ a $n$.

Nech $x$ je posledný vrchol na tej časti cesty, ktorú sme už zostrojili. Ako nájsť ďalší jej vrchol? Z vrcholu $x$ vedú v našom grafe $G$ nejaké hrany. Jedna z nich je tá, ktorou do $x$ príde nami zostrojovaná cesta. Ostatné vedú do nových, ešte nenavštívených vrcholov. (Toto platí na začiatku, keď $x = n$. Následne budeme priebežne odstraňovať nepotrebné hrany z $G$, takže to bude platiť aj v ďalších iteráciách.) No a my sa teraz potrebujeme rozhodnúť, ktorou z týchto hrán bude naša cesta pokračovať.

Toto by sme vedeli spraviť sekvenčne, podobne ako v predchádzajúcom riešení. Existuje ale aj efektívnejší spôsob, podobný binárnemu vyhľadávaniu. Kým budeme mať na výber viac ako jednu hranu, budeme opakovať nasledovný postup: Z $G$ odstránime polovicu spomedzi kandidátov na nasledujúcu hranu a zavoláme funkciu `cesta`. Ak graf stále obsahuje nejakú Hamiltonovskú cestu, pokračujeme priamo ďalej. Ak nám ale volanie funkcie `cesta` oznámi, že nový graf už Hamiltonovskú cestu nemá, znamená to, že (každá možná) hľadaná hrana z $x$ ďalej je medzi tými, ktoré sme práve odstránili. Vrátime ich teda späť do $G$ – a *namiesto nich* odstránime tú polovicu hrán z $x$, ktorú sme pôvodne v $G$ nechali!

Takto pre konkrétny vrchol $x$ jedným volaním funkcie `cesta` zmenšíme počet hrán vedúcich z $x$ približne na polovicu, pričom naďalej dostaneme graf obsahujúci aspoň jednu Hamiltonovskú cestu. Keď tento postup opakujeme, po $O(\log n)$ volaniach funkcie `cesta` nám ostane vo vrchole $x$ (okrem hrany, ktorou sme doň prišli) už len jediná hrana – a teda sme práve našli nasledujúcu hranu na zostrojovanej Hamiltonovskej ceste.

## Listing programu (Python)

```
def susedia(v,E):
    s1 = set( y for x,y in E if x==v )
    s2 = set( x for x,y in E if y==v )
    return [ z for z in s1+s2 ]

def najdi_cestu(n,E):
    E += [ (n,x) for x in range(n) ] + [ (n,n+1) ]
```

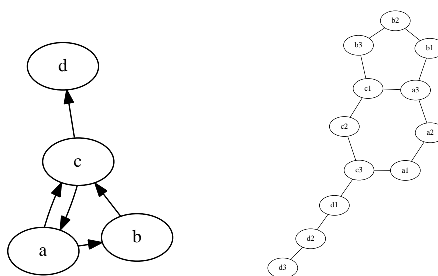
```
cesta = [ n+1, n ]
for kolo in range(n):
    kde = cesta[-1]
    kam_mozem = susedia( kde, E )
    kam_mozem.remove( cesta[-2] )
    ostatne_hrany = [x,y for x,y in E if x!=kde and y!=kde] + [(kde,cesta[-2])]
    while len(kam_mozem) > 1:
        cnt = len(kam_mozem)
        k1, k2 = kam_mozem[:cnt//2], kam_mozem[cnt//2:]
        if cesta( n+2, ostatne_hrany + [ (kde,x) for x in k1 ] ):
            kam_mozem = k1
        else:
            kam_mozem = k2
    vitaz = kam_mozem[0]
    cesta.append( vitaz )
    E = ostatne_hrany + [ (kde,vitaz) ]
return cesta[2:]
```

## Podúloha C (4 body):

Z pôvodného orientovaného grafu $G$ s $n$ vrcholmi spravíme nový neorientovaný graf s $3n$ vrcholmi – a to tak, že každý pôvodný vrchol nahradíme tromi novými.

Namiesto každého vrcholu $v$ teda budeme mať tri vrcholy: $v_1$ (tzv. vstupný), $v_2$ (tzv. stredný) a $v_3$ (tzv. výstupný). Dvojice $v_1v_2$ a $v_2v_3$ budú spojené hranou.

Orientované hrany z pôvodného grafu zmeníme v novom grafe na neorienované hrany z príslušného výstupného do príslušného vstupného vrcholu. Ak sme teda napr. v pôvodnom grafe mali hranu $u \to v$, budeme v novom grafe mať neorientovanú hranu $u_3v_1$.




*Vľavo: orientovaný graf. Vpravo: k nemu zostrojený neorientovaný graf.*

Zjavne ak v pôvodnom grafe existovala orientovaná Hamiltonovská cesta, existuje Hamiltonovská cesta aj v našom novom neorientovanom grafe – vždy, keď sme v pôvodnom grafe vošli do vrcholu $v$, vôjdeme v novom grafe postupne do vrcholov $v_1$, $v_2$ a $v_3$.

Ako je to s opačnou implikáciou? Predpokladajme, že v našom novom grafe existuje Hamiltonovská cesta. Čo o nej vieme povedať? V prvom rade to, že musí ísť cez všetkých $n$ stredných vrcholov. A keďže každý $v_2$ je spojený len s dvomi inými vrcholmi, vieme, že sa táto Hamiltonovská cesta musí skladať

z $n$ 3-vrcholových úsekov tvaru $v_1v_2v_3$. Ďalej, v našom grafe nemáme žiadnu hranu tvaru $u_1v_1$ ani $u_3v_3$, preto sa na našej Hamiltonovskej ceste musí dokola opakovať vstupný, stredný a výstupný vrchol. No a takejto Hamiltonovskej ceste zjavne zodpovedá orientovaná Hamiltonovská cesta v pôvodnom grafe.

(Rozmyslite si, kde by dôkaz tejto implikácie zlyhal, ak by sme spravili konštrukciu s $2n$ vrcholmi, pri ktorej vynecháme stredné vrcholy a namiesto toho spojíme hranou priamo každý vstupný vrchol so zodpovedajúcim výstupným.)

## A-III-4 Hore a dole

Stručný popis optimálneho riešenia: Nájdeme tie prvky postupnosti, ktoré nemusia byť od ničoho väčšie. Tie všetky nastavíme na nulu, a od nich následne odvodíme minimálne hodnoty pre všetky ostatné prvky postupnosti.

V nasledujúcom texte najskôr popíšeme jednoduchšie riešenia, ktoré mohli ľahko získať nejaké body, a následne podrobne rozoberieme prečo naše optimálne riešenie funguje a ako ho dobre implementovať.

**Ľahké riešenia za pár bodov:**

Platnú postupnosť *celých* čísel zostrojíme ľahko: začneme z ľubovoľnej hodnoty a potom po znakoch spracúvame daný reťazec. Keď vidíme znak `=` zopakujeme poslednú hodnotu, keď vidíme `<` alebo `>`, pridáme novú hodnotu, ktorá je o 1 väčšia, resp. menšia, ako predchádzajúca.

Napr. ak začneme od 10, tak pre reťazec `<=>><` takto dostaneme postupnosť $(10, 11, 11, 10, 9, 10)$.

V prvých dvoch testovacích vstupoch stačilo napr. začať od hodnoty $n$. (Alebo napr. od hodnoty 500 000 000, ktorá leží v strede povoleného rozsahu.) Toto nám zaručilo, že všetky hodnoty, ktoré použijeme v riešení, patria do povoleného rozsahu.

Na tretí testovací vstup stačilo vyššie riešenie drobne vylepšiť. Po tom, ako vyrobíme vyššie popísaným postupom *nejakú* postupnosť, môžeme nájsť jej minimum a to odčítať od všetkých jej členov. Vo vyššie použitom príklade by sme takto upravili našu postupnosť na $(1, 2, 2, 1, 0, 1)$.

Takýmto postupom v tretej sade vždy dostaneme platné riešenie – až na dve výnimky. Tými sú vstupy obsahujúce tisíc znakov `>`, resp. tisíc znakov `<`. Tieto dva vstupy nemajú platné riešenie, treba pre ne teda vypísať samé mínus jednotky.

**Myšlienka vzorového riešenia:**

Sústreďme sa teraz na hľadanie *optimálneho* riešenia, teda riešenia s najmenším možným súčtom členov. Radi by sme začali tým, že prestaneme uvažovať znak `=`. Na prvý pohľad si to môžeme dovoliť, lebo predsa `=` vždy vieme vyriešiť tak, že nájdeme riešenie pre vstup bez znakov `=` a potom len zduplikujeme tie hodnoty, ktoré zodpovedajú výskytu `=`. Toto bude skutočne fungovať – no uvedomte si, že toto nie je niečo, čo automaticky muselo platiť! Keby sme použili iné kritérium optimálnosti, už by takéto riešenie vôbec nemuselo fungovať. Prečo? Preto, že nič nám (zatiaľ) nezaručuje, že ak zoberieme optimálne riešenie pre vstup bez `=`, dostaneme z neho vyššie popísaným postupom *optimálne* riešenie pre vstup s `=`. („Keby som vedel, že túto hodnotu mám sedemkrát zopakovať, dal by som ju menšiu a radšej by som namiesto toho dal väčšiu túto inú, ktorú budem mať v riešení len raz!“ mohol by sa sťažovať imaginárny riešiteľ inej, podobnej úlohy.)

Zatiaľ teda uvažujme vstupy obsahujúce všetky tri druhy znakov. Nebojte sa, časom si ukážeme, že v našej úlohe naozaj môžeme `=` spokojne odignorovať. Pre niektoré vstupy je optimálne riešenie zjavné. Napríklad pre postupnosť `<<<` je to zjavne $(0,1,2,3)$, pre `>>>` je to $(3,2,1,0)$, pre `<=<=<` je to $(0,1,1,2,2,3)$ a pre `<<>>` je to $(0,1,2,1,0)$. Teraz sa zamyslime, ako bude vyzerať optimálne riešenie pre vstup `<<<<<>>`.

Hľadáme teda postupnosť celých čísel $(x_0, x_1, \dots, x_7)$ takú, že platí

$$0 \le x_0 < x_1 < x_2 < x_3 < x_4 < x_5 > x_6 > x_7 \ge 0.$$

Pozerajúc sa zľava si vieme postupne odvodiť $x_0 \ge 0$, $x_1 \ge 1$, a tak ďalej až po $x_5 \ge 5$. Pozerajúc sa sprava si podobne odvodíme $x_7 \ge 0$, $x_6 \ge 1$ a $x_5 \ge 2$. Pre $x_5$ sme teda dostali dva rôzne odhady; silnejší z nich je $x_5 \ge 5$. Pre každé iné číslo sme dostali odhad len jeden. Zvoľme teraz postupnosť, pre ktorú bude *naraz vo všetkých odhadoch* (samozrejme okrem toho slabšieho pre $x_5$) platiť rovnosť. Touto postupnosťou je v našom prípade postupnosť $(0,1,2,3,4,5,1,0)$. Táto postupnosť je nutne optimálna – za prvé spĺňa všetky požadované nerovnosti, a za druhé o každom jej člene vieme dokázať že menší už byť nemôže.

Vyššie popísaný postup zjavne vieme spraviť aj pre úplne všeobecnú postupnosť znakov. Na papieri by sme takto už asi vedeli k ľubovoľnej postupnosti znakov ručne nájsť zodpovedajúcu *optimálnu* postupnosť hodnôt, aj dokázať jej optimálnosť. My však potrebujeme popísať exaktný (a pokiaľ možno ľahko implementovateľný) algoritmus, ako tú postupnosť hodnôt zostrojiť.

V čom môže byť pri jeho implementácii problém? No, napríklad s tými nešťastnými znakmi rovná sa. Ak vidím postupnosť znakov `><`, dostanem postupnosť nerovností $x_{n-1} > x_n < x_{n+1}$, z ktorej je zjavné, že $x_n = 0$ nič nepokazí. Ale ako zabezpečiť, že ak program uvidí postupnosť znakov `>===<`, priradí zodpovedajúcim premenným nuly, ale zároveň nespraví to isté pre postupnosť `>===>`?

**Prvá možná implementácia vzorového riešenia:**

*Kopcom* nazveme postupnosť znakov, v ktorej sú najskôr používané znaky `<` a `=`, a následne sú používané znaky `>` a `=`. Inými slovami, postupnosť znakov je kopec práve vtedy, ak sa v nej nevyskytuje žiadne `>` naľavo od nejakého `<`. Kopcu zodpovedá postupnosť, ktorá najskôr neklesá a potom nerastie.

Optimálne riešenie pre kopec zostrojíme ľahko: až na špeciálne prípady (viď nižšie) budú obe premenné na jeho koncoch nuly, a od nich dovnútra postupne zvyšujeme hodnoty. Zároveň s touto konštrukciou priamo dostávame aj dôkaz optimálnosti.

No a keď dostaneme ľubovoľný všeobecný vstup, môžeme ho jednoduchým pažravým algoritmom rozbiť na postupnosť kopcov – kým môže nasledujúci znak patriť do aktuálneho kopca, pridáme ho tam, a ak už doň patriť nemôže, začneme robiť nový kopec. Rozmyslite si, že nový kopec začneme robiť práve vtedy, keď sme práve stretli znak `<` a aktuálny kopec už obsahoval aspoň jeden znak `>`.

Príklad: vstup `>><=<<><<>=>==<<` by sme takto rozbili na kopce `>>`, `<=<<>`, `<<>=>==` a `<<`.

Ľahko teraz nahliadneme, že optimálne riešenie pre celý vstup môžeme zostrojiť tak, že nájdeme optimálne riešenie pre každý kopec zvlášť, a následne tieto riešenia pospájame do jedného. Na to si stačí všimnúť, že optimálne riešenie každého kopca bude začínať aj končiť premennou, ktorá dostane hodnotu 0, takže pri ich spájaní nenastane žiaden konflikt. (Možnou výnimkou bude len začiatok prvého a koniec posledného kopca, tie však s ničím nespájame.)

**Pozorovanie o znakoch „rovná sa":**

Vyššie popísané riešenie nám okrem iného ukazuje, že znaky `=` naozaj môžeme najskôr odignorovať, vyriešiť vstup bez nich a následne ich pridať späť a zduplikovať zodpovedajúce prvky zostrojenej postupnosti. Odstránenie/pridanie znakov `=` totiž nezmení miesta, na ktorých náš pažravý algoritmus rozdelí vstup na jednotlivé kopce.

**Druhá možná implementácia vzorového riešenia:**

Ak už vieme, že = môžeme odstrániť, dostávame jednoduchšie riešenie. Najskôr teda odstránime všetky =. V druhom kroku každej premennej ktorá má byť menšia od všetkých svojich susedov, priradíme hodnotu 0. V treťom kroku ideme od premenných, ktoré dostali hodnotu 0, doľava aj doprava „do kopca“ a priraďujeme premenným postupne rastúce hodnoty. Na záver pridáme späť = a zduplikujeme zodpovedajúce prvky.

Príklad:

```
     vstup bez =:       >   <   <   <   <   >   >   >   <   <
         2. krok:         0                           0
 3. krok, prvá 0:     1   0   1   2   3   4           0
3. krok, druhá 0:     1   0   1   2   3   4   2   1   0   1   2
```

(Všimnite si, že pri spracúvaní druhej nuly sme „na vrchu kopca“ hodnotu 4 neprepísali menšou hodnotou 3.)

**Tretia možná implementácia vzorového riešenia:**

Predchádzajúce riešenie sa dá ešte o chlp pohodlnejšie implementovať. Začneme tým, že rovnako ako v ňom odstránime =. Následne použijeme algoritmus, ktorý sme si popísali úplne na začiatku, aby sme zostrojili *nejakú* postupnosť, ktorá spĺňa zadané nerovnosti. Následne už len túto postupnosť dvakrát prejdeme – raz zľava doprava a raz sprava doľava. Pri každom prechode sa postupne pozrieme na každú hodnotu a zmenšíme ju najviac ako to len ide bez porušenia zadaných podmienok. Pridať späť = už vieme triviálne pri výpise riešenia.

Rozmyslite si, že takto zostrojíme presne to isté riešenie ako predchádzajúcim postupom. (Kedy dostanú hodnotu 0 premenné, ktoré ju dostať majú? Čo sa udeje vo zvyšku prechodu zľava doprava? A čo v následnom prechode sprava doľava?)

Všetky tri vyššie popísané riešenia sa dajú implementovať s optimálnou časovou zložitosťou $\Theta(n)$.

Existujú aj pomalšie riešenia. Napríklad v treťom riešení sa dá namiesto prechodu sprava doľava robiť prechody zľava doprava dovtedy, kým sa počas niektorého prechodu už žiadna hodnota nezmení. Takéto riešenie má v najhoršom prípade časovú zložitosť $\Theta(n^2)$ a malo by získať 6 bodov.

Body sa dalo získať aj za riešenia založené na iných myšlienkach. Napríklad sa dalo využiť zadanú hranicu $m$ a namiesto celých kopcov si rozdeliť postupnosť na „svahy dohora“ a „svahy dodola“. Ak existuje platná postupnosť, môžeme

jednu zostrojiť tak, že po svahu dohora robíme kroky o 1, až na to že v úplne posledom kroku dohora skočíme na $m$. A naopak, idúc dodola o vždy keď máme klesnúť, klesneme o 1, až na úplne posledné klesnutie, kedy skočíme na 0. Ak existuje spôsob, ako udržať všetky hodnoty postupnosti v rozsahu 0 až $m$, takto sa nám to určite podarí. Takto implementované riešenie malo dostať aspoň 5 bodov.

## Listing programu (C++)

```
// optimálne riešenie treťou z metód popísaných vo vzorovom riešení
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

inline void fix(int z, vector<int> &A, const string &nS) {
    // čo najviac zmenši hodnotu na indexe z
    A[z] = 0;
    if (z>0              && nS[z-1]=='<') A[z] = max( A[z], A[z-1]+1 );
    if (z<int(A.size()) && nS[z]  =='>') A[z] = max( A[z], A[z+1]+1 );
}

int main() {
    int N, M; cin >> N >> M;
    string S; cin >> S;

    // Odstránime znaky '=' a namiesto nich si o každej hodnote zapamätáme,
    // koľkokrát ju vypísať. Zároveň zostrojíme nejakú korektnú postupnosť.
    vector<int> A(1,N);
    vector<int> pocet(1,1);
    string nS;
    for (char c:S) {
        if (c=='<') { A.push_back( A.back()+1 ); pocet.push_back(1); nS += c; }
        if (c=='=') ++pocet.back();
        if (c=='>') { A.push_back( A.back()-1 ); pocet.push_back(1); nS += c; }
    }

    // prejdeme postupnosť zľava doprava a následne sprava doľava
    int Z = A.size();
    for (int z=0; z<Z; ++z) fix(z,A,nS);
    for (int z=Z-1; z>=0; --z) fix(z,A,nS);

    // zistíme, či sa postupnosť zmestí do zadaného rozsahu
    // podľa toho buď vypíšeme ju alebo chybovú správu
    bool fits = true;
    for (int i=0; i<Z; ++i) if (A[i]>M) fits = false;
    if (fits) {
        for (int i=0; i<Z; ++i) for (int j=0; j<pocet[i]; ++j) cout << (i+j?" ":"") << A[i];
    } else {
        for (int n=0; n<N; ++n) cout << (n?" -1":"-1");
    }
    cout << endl;
}
```

## A-III-5 Vyvážené reťazce

Úlohy kde hľadáme súvislú postupnosť s nejakou konkrétnou vlastnosťou sú pomerne časté a aj v tomto ročníku olympiády sa nejaké vyskytli. A preto sa aj k tejto úlohe dalo pristupovať veľmi klasicky: základným nástrojom bude predpočítanie vhodnej informácie pre všetky prefixy nejakého reťazca.

**Pomalé riešenie:**

Prvá možnosť je samozrejme postupne prechádzať všetkými možnými podpostupnosťami a každú z nich overiť zvlášť. A rozumny spôsob prechádzadania je taký, kde si určíme začiatok a k nemu skúšame každý možný koniec, pričom postupne zväčšujeme dĺžku vybranej podpostupnosti. Týmto spôsobom sa nový podreťazec líši od toho predchádzajúceho len v jednom písmene, takže väčšinu predrátanej informácie si môžeme ponechať.

Najjednoduchší spôsob bude pamätať si pre každé písmeno z našej abecedy, koľkokrát sa dosiaľ vyskytlo v našej podpostupnosti. Vždy keď posúvam začiatok, musím si pole v ktorom si to pamätám vynulovať. Keď posuniem koniec, len pripočítam písmeno, ktoré som pridal. Následne už len skontrolujem, či mám všetkých písmen rovnako veľa. Treba si uvedomiť, že vo výsledku sa nemusia nachádzať všetky písmená z abecedy, takže ak som niektorých písmen videl 0, je to v poriadku. Takéto riešenie má (pre $n$-písmenný reťazec a abecedu tvorenú $k$ písmenami) časovú zložitosť $O(kn^2)$.

Predchádzajúce riešenie vieme ešte o trochu zlepšiť. Namiesto toho, aby sme pre každý koniec kontrolovali v čase $\Theta(k)$, ktoré písmeno máme koľkokrát, vieme túto kontrolu spraviť v konštantnom čase. Rovnako ako v predchádzajúcom riešení si budeme pamätať, koľkokrát sme v aktuálnom úseku videli ktoré písmeno. Navyše si budeme pamätať niekoľko pomocných premenných: maximum $m$ zo zapamätaných počtov písmen, počet písmen ktoré majú práve $m$ výskytov, a počet písmen ktoré majú aspoň jeden výskyt. Takto vieme dostať riešenie s časovou zložitosťou $O(n^2)$.

**Prefixy a dvojpísmenová abeceda:**

Predchádzajúce riešenie nám zaručuje zisk zhruba 4 bodov. Pozrime sa teda na ďalšiu testovaciu sadu. Tá obsahuje abecedu zloženú len z dvoch písmen – $a$ a $b$. Výrazne sa však zväčšila dĺžka reťazca. V podobných úlohách sa často oplatí zamyslieť sa, či nevieme vyrátať nejakú užitočnú informáciu pre každý prefix. Prefixov je totiž len $n$ (lineárne veľa) a ľubovoľný súvislý úsek vieme vyskladať odčítaním dvoch prefixov.

Najdlhší úsek tvorený opakovaním jedného písmena vieme nájsť triválne. Čo

ale s úsekmi, ktoré majú rovnako veľa $a$ a $b$? Pamätať si samostatne aj počet $a$ aj počet $b$ v každom prefixe nám príliš nepomôže. Ak totiž máme prefix, ktorý obsahuje $a$ 4-krát a $b$ obsahuje 5-krát (skrátene $(4,5)$), nevieme aký druhý prefix k nemu priradiť. Lebo ak odčítame prefix $(3,4)$ dostaneme úsek s jedným $a$ a jedným $b$ čo je dobré. Takisto je však vyhovujúci prefix $(2,3)$ alebo $(0,1)$.

Čo majú tieto prefixy spoločné? No predsa rozdiel počtu $a$-čok a $b$-čok, ktoré obsahujú. A práve tento rozdiel bude pre nás podstatný. Je totiž veľmi ľahké zistiť, kedy sa počet $a$ a $b$ rovná. Ak totiž $a = b$ tak $a - b = 0$.

Riešenie je teraz už jednoduché. Postupne prechádzam reťazec. Udržiavam si premennú `rozdiel` a keď narazím na $a$ zväčším ju o 1, pri $b$ ju o 1 zmenším. Pre každý prefix si poznačím do vhodnej dátovej štruktúry (mapa, set, poprípade aj obyčajné pole, lebo hodnoty sú od $-n$ do $n$), daný rozdiel a pozíciu kde som ho dosiahol. Pre každú hodnotu premennej `rozdiel` si naviac pamätám len jej prvý výskyt. A vždy, keď vypočítam novú hodnotu premennej `rozdiel`, tak sa pozriem, či som už rovnakú hodnotu nedosiahol niekedy predtým. Ak áno, viem, že úsek medzi prvým a týmto dosiahnutím tvorí vhodný vyvážený reťazec, takže si vypočítam veľkosť a pozíciu tohoto úseku a porovnám ju s doteraz najlepším riešením.

Týmto riešením viem vyriešiť aj ďalšiu sadu, ktorá síce obsahuje trojpísmenovú abecedu, ale vieme, že vo výslednom reťazci sú najviac dve z nich. Existujú len tri možnosti, ktoré dve písmená to budú a teda môžeme vyskúšať všetky tri – (postupne počítam $a - b$, $b - c$ a $a - c$). Uvedomte si ešte, že vždy keď nájdete tretie písmeno (to ktoré ste si nezvolili, že bude vo výsledku) treba zmazať obsah našej štruktúry a začať za ním úplne odznova.

**Viacznakové abecedy:**

Posledným krokom ku vzorovému riešeniu tejto úlohy je zistiť, ako riešiť inštancie s abecedou, ktorá obsahuje viac ako 2 znaky. Začnime zľahka s abecedou $\{a, b, c\}$ a predpokladajme, že optimálne riešenie naozaj obsahuje všetky tri znaky. Ako teda pridáme informáciu o písmene $c$?

Keď sme používali len dve písmená, pamätali sme si rozdiel $a - b$ a vedeli sme, že keď sa tento rozdiel rovná 0, tak máme rovnako veľa písmen $a$ ako písmen $b$. Rozumné by teda bolo si pamätať aj rozdiel $a - c$. Ak by sa obe tieto hodnoty rovnali 0, znamenalo by to, že $a = b$ a $a = c$, teda aj $b = c$. A to je presne to čo vyžadujeme. Pre tri písmená si teda budeme pamätať tieto dve hodnoty ako dvojicu čísiel pre každý prefix. Opäť môžeme vidieť, že ak nájdeme rovnakú dvojicu pre iný prefix, tak úsek medzi nimi je vyvážený, lebo rozdiel $a - b = 0$ a $a - c = 0$.

Toto nás už navádza na optimálne riešenie. Vidíme totiž, že ak máme trojpísmenkovú abecedu musíme vyskúšať všetky možnosti, ktoré písmená budú vo výslednom reťazci – 3 pre dvojice a 1 pre štvoricu.

Tento prístup by sa teda mal dať zovšeobecniť aj na viacznakové abecedy. Majme abecedu obsahujúcu $k$ rôznych znakov označených $\{x_1, x_2, \ldots x_k\}$. Nezostáva nám nič iné ako skúsiť každú možnosť toho, ktoré znaky budú vo výslednom riešení. Ak si však už vyberieme nejakú podmnožinu týchto znakov (týchto podmnožín je $2^k$) tak už viem použiť vyššie popísaný algoritmus, ktorý sa postupne pozerá na všetky prefixy daného reťazca. Ak však vyberieme podmnožinu, ktorá obsahuje $l$ prvkov, tak si musíme pamätať $(l-1)$-ticu čísiel – rozdiely $(x_1 - x_2, x_1 - x_3 \ldots x_1 - x_l)$. Uvedomme si, že keď sa táto $(l-1)$-tica rovná $(0, 0, \ldots 0)$ tak je reťazec, ktorý predstavuje vyvážený, lebo počty všetkých písmen sa rovnajú počťu písmen $x_1$.

Ostáva nám odpovedať na otázku, aká je časová zložitosť takéhoto riešenia. Naše prefixové riešenie malo zložitosť $O(n \log n)$, lebo pre každý prefix sme sa pozerali do mapy (teraz je už potrebná zložitejšia dátová štruktúra ako pole, lebo si ukladáme celé $k$-tice čísiel). Naviac však pracujeme s $k$-ticami, takže práca s mapou sa $k$ krát spomalí. Dostávame $O(nk \log n)$. Toto riešenie však musíme spustiť pre každú možnú podmnožinu našich $k$ písmen. Teda celková časová zložitosť bude $O(2^k kn \log n)$.

## Listing programu (C++)

```
#include <cstdio>
#include <map>
#include <set>
#include <string>
#include <vector>
using namespace std;

#define For(i,n) for(int i=0; i<(n); i++)

int main() {
    char C[1000047];
    scanf("%s",C);
    string s=C;
    int n=s.length();
    set<char> Pom;
    For(i,n) Pom.insert(s[i]);
    vector<char> znaky(Pom.begin(),Pom.end());
    int k=znaky.size();
    map<vector<int>,int> M;
    vector<char> z;
    vector<int> nula;
    vector<int> stav;
    int zac=1,kon=0;
    for(int i=1; i<1<<k; i++) {
        Pom.clear();
        z.clear();
```

```
        For(j,k)
            if(i&(1<<j)) z.push_back(znaky[j]);
        M.clear();
        For(j,z.size()-1) nula.push_back(0);
        M[nula]=-1;
        stav=nula;
        For(j,n) {
            int pom=-1;
            For(k1,z.size()) if(z[k1]==s[j]) pom=k1;
            if(pom==-1) {
                M.clear();
                M[nula]=j;
                stav=nula;
                continue;
            }
            if(pom==0) For(k1,z.size()-1) stav[k1]++;
            else stav[pom-1]--;
            if(M.find(stav)==M.end()) M[stav]=j;
            if(kon-zac<j-M[stav]-1) {zac=M[stav]+1; kon=j;}
            else if(kon-zac==j-M[stav]-1 && M[stav]+1<zac) {zac=M[stav]+1; kon=j;}
        }
    }
    printf("%d %d\n",zac,kon);
}
```

**Lepšie riešenia:**

Úloha má aj lepšie riešenia, tie ale nebolo na dosiahnutie plného počtu bodov treba. V predchádzajúcom riešení sa napr. vieme faktoru $k$ v zložitosti zbaviť tak, že si budeme pamätať počty $a-b$, $b-c$, $c-d$, atď. a namiesto ukladania si celej $(k-1)$-tice použijeme vhodné hešovanie.

Existujú dokonca aj riešenia, ktorých časová zložitosť závisí od $n$ približne lineárne (možno s faktorom $\log n$ navyše) a aj od $k$ polynomiálne. Napovieme, že jedna možnosť, ako takéto riešenie zostrojiť, je, že začneme tým, že si zvolíme počet $x$ rôznych písmen ktoré má hľadaný reťazec obsahovať. Následne vo vstupnom reťazci nájdeme všetky maximálne úseky obsahujúce práve $x$ rôznych písmen a každý z nich spracujeme samostatne.

## A-III-6 ACGT

Najprv vyriešme jednoduchší problém: Pre zadané reťazce $R$, $S$ chceme zistiť minimálny počet zmien potrebných na prerobenie $R$ na $S$.

Toto sa dá vyriešiť dynamickým programovaním, ktoré sa veľmi podobá na hľadanie najdlhšej spoločnej podpostupnosti. Budeme si vypĺňať maticu $A$, kde $A[i][j]$ nám hovorí minimálny počet zmien nutných na prerobenie prvých $i$ znakov z $R$ na prvých $j$ znakov z $S$.

Pokiaľ $i = 0$ tak je počet zmien rovný $j$; detto pre $j = 0$. Inak sa pozrime

na znaky $R[i-1]$ a $S[j-1]$. Ak sú rovnaké, je zjavne optimálne ich ponechať. V takomto prípade je optimálnym riešením riešenie pre $i-1$ znakov $R$ a $j-1$ znakov $S$. A ak sa líšia, musíme to nejak napraviť. Tu sú naše možnosti: môžeme zmazať znak $R[i-1]$, môžeme vložiť do $R$ za pozíciu $i-1$ potrebný znak $S[j-1]$, alebo môžeme znak $R[i-1]$ zmeniť na znak $S[j-1]$. Toto vedie k nasledovným vzťahom:
Ak $R[i-1] = S[j-1]$, tak $A[i][j] = A[i-1][j-1]$. A ak $R[i-1] \neq S[j-1]$, tak $A[i][j] = \min(A[i-1][j]+1, A[i][j-1]+1, A[i-1][j-1]+1)$.
Výpočet poľa $A$ nám zaberie čas priamo úmerný jeho veľkosti, teda $O(|R| \cdot |S|)$.

Toto sa dá zlepšiť, ak máme limit $d$ na počet zmien ktoré smieme spraviť. Všimnime si, že všetky políčka, kde $A[i][j] > d$, počítame zbytočne, môžeme ich výpočet teda vynechať. Dá sa ukázať, že potrebných políčok bude $O((|R| + |S|) \cdot d)$. Návod: ak $|i-j| > d$, tak nutne $A[i][j] > d$.

Na práve použitý algoritmus sa dá dívať aj z iného uhla. Vyrobíme si graf, ktorého vrcholmi sú dvojice indexov: vrchol $(i, j)$ predstavuje stav, v ktorom sme prerobili prvých $i$ písmen reťazca $R$ na prvých $j$ písmen reťazca $S$.

V tomto grafe budú vrcholy spojené orientovanou hranou dĺžky 0, ak sa medzi nimi dá prejsť bez zmeny reťazcov (prípad 1 vyššie) alebo orientovanou hranou dĺžky 1 ak to vieme spraviť jednou zmenou (prípad 2 vyššie). No a v našom grafe hľadáme najkratšiu cestu z vrcholu $(0, 0)$ do vrcholu $(|R|, |S|)$. Keďže hrany majú ceny len 0 a 1, môžeme použiť tzv. *0-1 prehľadávanie do šírky*, čo je v podstate obyčajné prehľadávanie do šírky, ale s tým, že pokiaľ vzdialenosť do vrcholu $v$ zlepšíme, pričom sme doň prišli hranou dĺžky 0, vložíme $v$ nie na koniec, ale na začiatok fronty. Tento spôsob bude mať pri dobrej implementácii opäť časovú zložitosť $O((|R| + |S|) \cdot d)$.

Teraz sa už môžeme pustiť do riešenia samotnej úlohy. Za 4 body stačí na vstupoch, kde $n = 2$ a $m = 1$, správne implementovať vyššie uvedené postupy a overiť tak, že jediné prípustné riešenie skutočne má dostatočne málo chýb. Piaty bod vieme ľahko pridať ošetrením špeciálneho prípadu: ak $d = 0$, môžeme správnu postupnosť zostrojiť pažravo – stačí využiť to, že vždy, keď treba prejsť na nový fragment, vieme podľa prvého písmena, na ktorý.

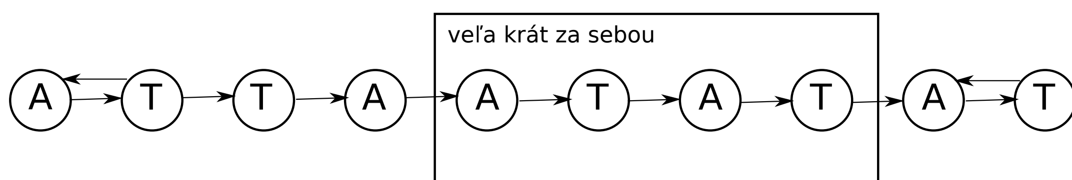
Teraz sa pozrime na riešenie všeobecného prípadu. Potrebujeme teda rozšíriť vyššie popísaný postup na netriviálny graf – teda situáciu, kedy máme viac ako 2 fragmenty a veľa rôznych dvojíc fragmentov čo po sebe môžu nasledovať. Do stavu prehľadávania nám pri tomto rozšírení pribudne fragment, v ktorom sa práve nachádzame.

Presnejšie, vrcholy nášho grafu budú trojice $(i, f, j)$ prirodzených čísel. Trojica $(i, f, j)$ znamená, že momentálne sme na fragmente $f$, spracovali sme už $j$ prvých písmen tohoto fragmentu, a spolu so skôr spracovanými fragmentmi sme vyrobili prvých $i$ znakov reťazca $R$. Hrany v rámci fragmentu vyzerajú rovnako ako vo vyššie popísanom riešení pre dva reťazce. Navyše budeme mať hrany dĺžky 0 zodpovedajúce tomu, že za jeden fragment priložíme ďalší, ktorý zaň priložiť smieme.

V takomto grafe zostrojíme pomocou 0-1 prehľadávania do šírky množinu všetkých jeho vrcholov, ktoré sú vo vzdialenosti najviac $d$ od začiatočného vrcholu $(0, u, 0)$. Na záver sa len pozrieme, či je medzi dosiahnuteľnými vrcholmi aj cieľový vrchol $(|R|, v, |F_v|)$, a ak áno, zostrojíme cestu, ktorou sme ho dosiahli. Z tej ľahko prečítame použitú postupnosť fragmentov.

Na záver už len odhadneme časovú zložitosť tohto algoritmu. Označme $r$ dĺžku reťazca $R$, ktorý zostrojujeme, a $f$ súčet dĺžok fragmentov na vstupe. Ľahký horný odhad je $O(rf)$: pre každú pozíciu v reťazci $R$ môžeme zároveň byť na hociktorej pozícii hociktorého fragmentu. Už vďaka tomuto hornému odhadu vieme, že toto riešenie bude dostatočne efektívne pre väčšinu sád testovacích vstupov.

Bohužiaľ, na rozdiel od prípadu dvojice reťazcov sa pre tento algoritmus nedá dokázať tesnejší odhad. Predstavme si totiž napríklad fragmenty ako na obrázku (každý krúžok je jeden fragment):




Pokiaľ by sme hľadali reťazec, ktorý by bol zložený z veľa `AT` po sebe, začínali fragmentom úplne vľavo, končili fragmentom úplne vpravo a povolili by sme 2 zmeny, tak by sme navštívili skoro každý stav.

V prípade „slušných" vstupov sa ukazuje, že až tak veľa stavov nenavštívime, keďže v prípadoch viacznakových fragmentov potrebujeme, aby sa celý fragment takmer zhodoval s nejakou časťou reťazca $R$, a toto nastane len na málo miestach.

## Listing programu (C++)

```
#include <cstdio>
```

```
#include <vector>
#include <string>
#include <deque>
#include <unordered_map>
#include <algorithm>
using namespace std;
char buf[1000000];

struct Pos {
  int tp;
  int nod;
  int sp;
  Pos() {}
  Pos(int a, int b, int c) : tp(a), nod(b), sp(c) {}

  bool operator==(const Pos &b) const {
    return tp == b.tp && nod == b.nod && sp == b.sp;
  }
};

namespace std {
template<>
struct hash<Pos> {
  size_t operator()(const Pos& a) const {
    return hash<int>()(a.tp) ^ (hash<int>()(a.nod) << 3) ^ (hash<int>()(a.sp) << 5)
      ^ (hash<int>()(a.tp) >> 2) ^ (hash<int>()(a.nod) >> 1) ^ hash<int>()(a.sp);
  }
};
}

int main() {
  int n, m; scanf("%d␣%d␣", &n, &m);
  vector<string> nodes;
  for (int i = 0; i < n; i++) {
    scanf("%s", buf);
    nodes.push_back(string(buf));
  }

  vector<vector<int>> g(n);
  for (int i = 0; i < m; i++) {
    int a, b; scanf("%d␣%d␣", &a, &b);
    a--; b--;
    g[a].push_back(b);
  }

  int d, start, end;
  scanf("%d␣%d␣%d␣", &d, &start, &end);
  start--; end--;
  scanf("%s", buf);
  string target(buf);

  deque<pair<Pos,int>> fr;
  unordered_map<Pos, Pos> back;
  back.reserve(target.size()*5*d);
  fr.push_back(make_pair(Pos(0, start, 0), 0));
  unordered_map<Pos, int> dists;
  dists.reserve(target.size()*5*d);

  while (!fr.empty()) {
    Pos x = fr.front().first;
    int dd = fr.front().second; fr.pop_front();
    if (dd > d) continue;
    if (dd > dists[x]) continue;
    if (x.tp == target.size() && x.nod == end && x.sp == nodes[x.nod].size()) {
```

```
      vector<int> path;
      Pos cp = x;
      path.push_back(cp.nod);
      while (back.count(cp)) {
        Pos prev = back[cp];
        if (prev.nod != cp.nod) {
          path.push_back(prev.nod);
        }
        cp = prev;
      }
      reverse(path.begin(), path.end());
      for (int i = 0; i < path.size(); i++) {
        printf("%d%c", path[i] + 1, i + 1 == path.size() ? '\n' : '␣');
      }
      return 0;
    }
    if (x.sp < nodes[x.nod].size()) {
      if (x.tp < target.size()) {
        if (nodes[x.nod][x.sp] == target[x.tp]) {
          Pos nx(x.tp+1, x.nod, x.sp+1);
          if (dists.count(nx) == 0 || dists[nx] > dd) {
            back[nx] = x;
            fr.push_front(make_pair(nx, dd));
            dists[nx] = dd;
          }
        } else {
          Pos nx(x.tp+1, x.nod, x.sp+1);
          if (dists.count(nx) == 0) {
            back[nx] = x;
            fr.push_back(make_pair(nx, dd+1));
            dists[nx] = dd+1;
          }
        }
      }
      Pos nx(x.tp, x.nod, x.sp+1);
      if (dists.count(nx) == 0) {
        back[nx] = x;
        fr.push_back(make_pair(nx, dd+1));
        dists[nx] = dd+1;
      }
    } else {
      for (int i = 0; i < g[x.nod].size(); i++) {
        int nnod = g[x.nod][i];
        Pos nx(x.tp, nnod, 0);
        if (dists.count(nx) == 0 || dists[nx] > dd) {
          back[nx] = x;
          fr.push_front(make_pair(nx, dd));
          dists[nx] = dd;
        }
      }
    }
    if (x.tp < target.size()) {
      Pos nx(x.tp+1, x.nod, x.sp);
      if (dists.count(nx) == 0) {
        back[nx] = x;
        fr.push_back(make_pair(nx, dd+1));
        dists[nx] = dd+1;
      }
    }
  }
  printf("-1\n");
}
```

# 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. Bial Bence | 2 Gym. Fiľakovo | 8 | 10 | 2 | 7 | 27 |
| 2. Csernok Ladislav | 2 Gym. Fiľakovo | 2 | 2 | 0 | 5 | 9 |

**Bratislavský kraj:**

| Meno | Ročník, škola | 1. | 2. | 3. | 4. | Σ |
|---|---|---|---|---|---|---|
| 1. Veselá Simona | 2 Gym. Jura Hronca BA | 2 | 6 | 4 | 7 | 19 |
| 2. Bánhegyi Tomáš | 2 Gym. Jura Hronca BA | 4 | 5 | 3 | 5 | 17 |
| 3. Kúšik Lukáš | 2 Gym. Jura Hronca BA | | 2 | 2 | 3 | 7 |

**Košický kraj:**

| Meno | Ročník, škola | 1. | 2. | 3. | 4. | Σ |
|---|---|---|---|---|---|---|
| 1. Micheľová Henrieta | 2 Gym. Alejová Košice | 2 | 8 | 2 | 7 | 19 |

**Nitriansky kraj:**

| Meno | Ročník, škola | 1. | 2. | 3. | 4. | Σ |
|---|---|---|---|---|---|---|
| 1. Porubský Michal | 2 SKŠ Nitra, Gym. sv. Cyrila a Metoda | 7 | 9 | 0 | 7 | 23 |
| 2. Varga Adam | 2 SPŠE Nové Zámky | 3 | 3 | 5 | 4 | 15 |
| 3. Daniš Tomáš | 2 SPŠE Nové Zámky | 0 | 2 | | 4 | 6 |

**Prešovský kraj:**

| Meno | Ročník, škola | 1. | 2. | 3. | 4. | Σ |
|---|---|---|---|---|---|---|
| 1. Jurečko Dominik | 2 SPŠE Prešov | 5 | 5 | 2 | 7 | 19 |
| 2. Jurašek Patrik | 2 SPŠE Prešov | 1 | 4 | 6 | 0 | 11 |
| 3. Dujava Jozef | 2 Gym. Konštantínova Prešov | 1 | 2 | 6 | 0 | 9 |

**Trenčiansky kraj:**

| | Meno | Ročník, škola | 1. | 2. | 3. | 4. | Σ |
|---|---|---|---|---|---|---|---|
| 1. | Hlaváč Adam | 2 Gym. Nedožerského Prievidza | 3 | 9 | 5 | 7 | 24 |
| 2. | Baláž Lukáš | -1 Gym. Bánovce nad Bebravou | 5 | 5 | 6 | 7 | 23 |
| 3. | Staník Michal | -1 ZŠ Kubranská Trenčín | 4 | 5 | 0 | 7 | 16 |
| 4. | Kňazovický Marek | 2 Gym. Ľudovíta Štúra Trenčín | 1 | | 6 | 7 | 14 |
| 5. | Kotlaba Lukáš | 2 Gym. Ľudovíta Štúra Trenčín | 5 | 3 | | 2 | 10 |
| 6. | Arbet Jakub | -1 ZŠ Kubranská Trenčín | 3 | | 0 | 6 | 9 |
| | Liška Jakub | 2 Gym. Ľudovíta Štúra Trenčín | 0 | 2 | 0 | 7 | 9 |
| | Neuschl Tomáš | 2 Gym. Ľudovíta Štúra Trenčín | 3 | 2 | 1 | 3 | 9 |
| 9. | Zongor Milan | 1 Gym. Ľudovíta Štúra Trenčín | 5 | 2 | | | 7 |
| 10. | Tupý Martin | 2 Gym. Ľudovíta Štúra Trenčín | 4 | 0 | | | 4 |
| 11. | Fabo Samuel | 2 Gym. Ľudovíta Štúra Trenčín | 0 | 2 | 1 | | 3 |

**Trnavský kraj:**

| | Meno | Ročník, škola | 1. | 2. | 3. | 4. | Σ |
|---|---|---|---|---|---|---|---|
| 1. | Sobkuliak Roman | 2 Gym. Hlohovec | 7 | 6 | 6 | 7 | 26 |
| 2. | Korman Andrej | 1 Gym. Hlohovec | 8 | 4 | 6 | 5 | 23 |

**Žilinský kraj:**

| | Meno | Ročník, škola | 1. | 2. | 3. | 4. | Σ |
|---|---|---|---|---|---|---|---|
| 1. | Majchrák Pavol | 2 SPŠ Kysucké Nové Mesto | 3 | 2 | 3 | 6 | 14 |
| 2. | Bobenič Ján | 2 SPŠ Kysucké Nové Mesto | 4 | 3 | 5 | 0 | 12 |

## Výsledky celoštátneho kola kategórie A

Celoštátne kolo kategórie A sa v 29. ročníku Olympiády v informatike uskutočnilo v dňoch 26. až 29. marca v Bratislave, na pôde Fakulty matematiky, fyziky a informatiky Univerzity Komenského. Najlepších šestnásť riešiteľov bolo vyhlásených za úspešných a z nich najlepších sedem za víťazov celoštátneho kola.

Prvých 13 riešiteľov bolo pozvaných na výberové sústredenie, kde dostali príležitosť zabojovať o účasť na medzinárodných súťažiach.

| | Meno | Ročník, škola | 1. | 2. | 3. | 4. | 5. | 6. | Σ |
|---|---|---|---|---|---|---|---|---|---|
| 1. | Batmendijn Eduard | 3. Cirk. gym. Stará Ľubovňa | 8 | 10 | 10 | 10 | 10 | 10 | 58 |
| 2. | Lipovský Mário | 4. Gym. Jura Hronca BA | 8 | 10 | 8 | 10 | 4 | 5 | 45 |
| 3. | Madaj Pavel | 3. Gym. Nedožerského Prievidza | 8 | 6 | 4 | 10 | 10 | 1 | 39 |
| 4. | Ralbovský Peter | 2. Škola pre mim. nad. deti BA | 7 | 6 | 10 | 10 | 4 | 0 | 37 |
| 5. | Ivan Lukáš | 4. Gym. Jura Hronca BA | 7 | 9 | 5 | 10 | 4 | 0 | 35 |
| | Truc Lam Bui | 3. Gym. Grösslingová BA | 1 | 9 | 10 | 10 | 5 | 0 | 35 |
| 7. | Marko Alan | 0. Gym. Nové Zámky | 8 | 9 | 10 | 3 | 4 | 0 | 34 |
| 8. | Kováčová Barbora | 4. Škola pre mim. nad. deti BA | 8 | 6 | 8 | 6 | 4 | 0 | 32 |
| | Sládeček Michal | 2. Gym. Varšavská Žilina | 7 | 9 | 4 | 8 | 4 | 0 | 32 |
| 10. | Pokrývka Filip | 4. Gym. Bánovce nad Bebravou | 8 | 6 | 3 | 10 | 4 | 0 | 31 |
| 11. | Korbela Michal | 4. Gym. Bánovce nad Bebravou | 6 | 6 | 4 | 10 | 4 | 0 | 30 |
| 12. | Havelka Jakub | 4. Gym. Jura Hronca BA | 8 | 4 | 8 | 3 | 4 | 0 | 27 |
| | Sládek Samuel | 2. Gym. Námestovo | 8 | 0 | 5 | 10 | 4 | 0 | 27 |
| 14. | Eckhaus Róbert | 4. Gym. Konštantínova Prešov | 8 | 6 | 5 | 3 | 4 | 0 | 26 |
| | Páll Juraj Eduard | 4. Gym. Konštantínova Prešov | 6 | 6 | 7 | 3 | 4 | 0 | 26 |
| 16. | Vozárová Viktória | 4. Gym. Jura Hronca BA | 8 | 6 | 1 | 6 | 4 | 0 | 25 |
| 17. | Matušák Filip | 4. Gym. Námestovo | 2 | 7 | 6 | 4 | 5 | 0 | 24 |
| 18. | Slivka Norbert | 4. Gym. Tajovského B. Bystrica | 5 | 6 | 4 | 4 | 4 | 0 | 23 |
| 19. | Králik Matej | 3. Gym. Jura Hronca BA | 6 | 1 | 8 | 3 | 4 | 0 | 22 |
| | Tkáčik Maroš | 3. Gym. Michalovce | 7 | 5 | 0 | 10 | 0 | 0 | 22 |
| 21. | Bátoryová Jana | 4. Gym. Alejová Košice | 6 | 3 | 4 | 4 | 4 | 0 | 21 |
| 22. | Liu Zhen Ning | 3. Gym. Jura Hronca BA | 8 | 3 | 5 | 4 | 0 | 0 | 20 |
| | Tesař Emanuel | 2. Gym. B. S. Timravy Lučenec | 5 | 6 | 2 | 3 | 4 | 0 | 20 |
| 24. | Bali Michal | 3. Gym. Párovská Nitra | 6 | 3 | 3 | 2 | 4 | 0 | 18 |
| 25. | Longa Marián | 3. Gym. Jura Hronca BA | 6 | 2 | 4 | 3 | 2 | 0 | 17 |
| 26. | Schönfeld Róbert | 3. Gym. Poštová Košice | 6 | 3 | 0 | 3 | 4 | 0 | 16 |

| | Meno | Ročník, škola | 1. | 2. | 3. | 4. | 5. | 6. | Σ |
|---|---|---|---|---|---|---|---|---|---|
| 27. | Pančík Juraj | 4. Gym. Tajovského B. Bystrica | 5 | 0 | 0 | 6 | 4 | 0 | 15 |
| | Sivoň Michal | 3. Gym. Paulinyho-Tótha Martin | 8 | 3 | 0 | 4 | 0 | 0 | 15 |
| 29. | Dargaj Jakub | 4. Gym. Poštová Košice | 3 | 5 | 0 | 3 | 0 | 0 | 11 |
| 30. | Bohdal Ondrej | 3. Gym. Jura Hronca BA | 2 | 0 | 4 | 2 | 2 | 0 | 10 |
| | Šandalová Michaela | 4. Škola pre mim. nad. deti BA | 3 | 0 | 0 | 3 | 4 | 0 | 10 |
| 32. | Sadlek Lukáš | 3. Gym. Čadca | 6 | 0 | 0 | 2 | 0 | 0 | 8 |

# Výsledky výberového sústredenia

V dňoch 1. až 8. mája 2014 sa v Bratislave konalo výberové sústredenie. O účasť na medzinárodných akciách na ňom bojovalo najlepších 13 riešiteľov a riešiteliek celoštátneho kola OI-A. Ako každý rok, najlepší štyria slovenskí riešitelia sa kvalifikovali na Medzinárodnú olympiádu v informatike. Na základe výberového sústredenia taktiež SK OI vybrala reprezentačný tím pre Stredoeurópsku olympiádu v informatike a pre prípravné Česko-poľsko-slovenské stretnutie.

V nasledujúcej tabuľke sú uvedené výsledky výberového sústredenia. Stĺpec „štart“ obsahuje body, s ktorými súťažiaci začínali (t.j. súčet bodov za celoštátne kolo OI a domáce úlohy).

| | Meno | $\sum$ | štart | št | pi | so | ne | po | ut | st | št |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. | Eduard Batmendijn | 1026.0 | 99.0 | 61.0 | 140.0 | 150.0 | 58.0 | 151.0 | 147.0 | 150.0 | 70.0 |
| 2. | Bui Truc Lam | 540.7 | 72.2 | 50.0 | 66.0 | 80.5 | 44.5 | 82.5 | 5.5 | 70.5 | 69.0 |
| 3. | Mário Lipovský | 491.5 | 68.0 | 34.0 | 76.0 | 66.5 | 35.5 | 56.5 | 77.0 | 52.0 | 26.0 |
| 4. | Michal Korbela | 490.5 | 63.0 | 46.0 | 80.0 | 66.0 | 43.5 | 75.0 | 46.0 | 60.0 | 11.0 |
| 5. | Lukáš Ivan | 439.7 | 46.2 | 30.0 | 61.0 | 73.0 | 36.5 | 44.0 | 71.0 | 68.0 | 10.0 |
| 6. | Pavel Madaj | 423.5 | 43.0 | 18.0 | 91.5 | 46.5 | 32.0 | 58.5 | 42.5 | 61.5 | 30.0 |
| 7. | Peter Ralbovský | 420.6 | 63.6 | 36.0 | 57.5 | 57.0 | 50.0 | 70.5 | 8.0 | 60.5 | 17.5 |
| 8. | Barbora Kováčová | 417.0 | 55.0 | 13.5 | 75.0 | 62.5 | 43.5 | 41.5 | 52.0 | 44.5 | 29.5 |
| 9. | Jakub Havelka | 346.0 | 36.0 | 34.0 | 56.5 | 25.0 | 36.5 | 25.5 | 47.0 | 33.5 | 52.0 |
| 10. | Michal Sládeček | 333.7 | 52.2 | 25.0 | 57.5 | 46.0 | 45.5 | 39.0 | 5.5 | 48.5 | 14.5 |
| 11. | Samuel Sládek | 271.0 | 41.0 | 11.0 | 54.0 | 36.5 | 23.5 | 44.5 | 14.0 | 41.5 | 5.0 |
| 12. | Filip Pokrývka | 270.3 | 46.8 | 31.0 | 29.0 | 44.5 | 26.5 | 41.5 | 18.0 | 25.0 | 8.0 |
| 13. | Alan Marko | 255.5 | 53.0 | 14.0 | 44.0 | 36.0 | 41.5 | 31.5 | 6.5 | 22.0 | 7.0 |

Za vyzdvihnutie stojí výkon Eda Batmendijna, ktorý získal vyše 96% možných bodov a zanechal zvyšok štartového poľa ďaleko za sebou.

## Medzinárodné prípravné sústredenie v Davose

Vďaka blízkej spolupráci medzi národnými olympiádami v informatike na Slovensku a vo Švajčiarsku mali aj tento rok vybraní riešitelia KSP a OI možnosť zúčastniť sa prípravného sústredenia vo švajčiarskom Davose, a to v dňoch 10. až 14. februára 2014. Sústredenía sa tiež zúčastnili delegácie z Izraelu a Ruska. V prvej tabuľke uvádzame výsledky dlhodobej súťaže, ktorá prebiehala počas celého sústredenia.

| | Meno | kraj. | day1 | day2 | day3 | day4 | $\sum$ |
|---|---|---|---|---|---|---|---|
| 1. | Tom Kalvari | ISR | 440 | 450 | 500 | 385 | 1775 |
| 2. | Petr Smirnov | RUS | 380 | 400 | 425 | 299 | 1504 |
| 3. | Ohad Klein | ISR | 290 | 370 | 400 | 324 | 1384 |
| 4. | Mário Lipovský | SVK | 240 | 380 | 306 | 303 | 1229 |
| 5. | Michal Korbela | SVK | 340 | 240 | 300 | 237 | 1117 |
| 6. | Ron Ryvchin | ISR | 240 | 226 | 346 | 265 | 1077 |
| 7. | Anna Nikiforovskaya | RUS | 300 | 202 | 300 | 249 | 1051 |
| 8. | Bui Truc Lam | SVK | 160 | 280 | 346 | 238 | 1024 |
| 9. | Emanuel Tesař | SVK | 220 | 220 | 240 | 243 | 923 |
| 10. | Benjamin Schmid | SUI | 240 | 160 | 260 | 234 | 894 |
| 11. | Mikhail Anoprenko | RUS | 220 | 160 | 240 | 245 | 865 |
| 12. | Timon Stampfli | SUI | 200 | 156 | 280 | 200 | 836 |
| 13. | Ian Boschung | SUI | 155 | 120 | 220 | 226 | 721 |
| 14. | Alexander Morozov | RUS | 140 | 210 | 140 | 226 | 716 |
| 15. | Fabian Lyck | SUI | 130 | 148 | 140 | 236 | 654 |
| 16. | Kevin De Keyser | SUI | 120 | 120 | 200 | 160 | 600 |
| 17. | Raphael Fischer | SUI | 75 | 92 | 198 | 208 | 573 |
| 18. | Lorenz Widmer | SUI | 200 | 52 | 98 | 214 | 564 |
| 19. | Ramy Massalha | ISR | 75 | 52 | 198 | 200 | 525 |
| 20. | Kasimir Tanner | SUI | 120 | 64 | 178 | 108 | 470 |
| 21. | Joël Mathys | SUI | 160 | 58 | 100 | 128 | 446 |
| 22. | Zheng Chen Man | SUI | 50 | 24 | 98 | 138 | 310 |
| 23. | Mugeeb Al-Rah Hassan | SUI | 0 | 16 | 156 | 100 | 272 |
| 24. | Elias Boschung | SUI | 20 | 34 | 88 | 128 | 270 |

V druhej tabuľke uvádzame výsledky jednokolovej súťaže „Davos Cup“.

| | Meno | kraj. | úl.1 | úl.2 | úl.3 | úl.4 | $\sum$ |
|---|---|---|---|---|---|---|---|
| 1. | Tom Kalvari | ISR | 100 | 100 | 100 | 100 | 400 |
| | Petr Smirnov | RUS | 100 | 100 | 100 | 100 | 400 |
| 3. | Ohad Klein | ISR | 100 | 100 | 60 | 100 | 360 |
| 4. | Ron Ryvchin | ISR | 100 | 100 | 60 | 40 | 300 |
| 5. | Mário Lipovský | SVK | 100 | 100 | 40 | 40 | 280 |
| 6. | Mikhail Anoprenko | RUS | 100 | 100 | 20 | 20 | 240 |
| | Michal Korbela | SVK | 100 | 100 | 20 | 20 | 240 |
| | Alexander Morozov | RUS | 100 | 100 | 20 | 20 | 240 |
| | Emanuel Tesař | SVK | 100 | 100 | 0 | 40 | 240 |
| 10. | Benjamin Schmid | SUI | 100 | 70 | 20 | 40 | 230 |
| 11. | Fabian Lyck | SUI | 100 | 60 | 20 | 40 | 220 |
| | Anna Nikiforovskaya | RUS | 100 | 100 | 20 | - | 220 |
| 13. | Lorenz Widmer | SUI | 100 | 90 | - | 20 | 210 |
| 14. | Bui Truc Lam | SVK | 100 | 100 | 0 | - | 200 |
| | Kevin De Keyser | SUI | 100 | 100 | 0 | - | 200 |
| | Raphael Fischer | SUI | 100 | 100 | 0 | - | 200 |
| | Ramy Massalha | ISR | 100 | 100 | - | - | 200 |
| | Timon Stampfli | SUI | 100 | 100 | - | 0 | 200 |
| 19. | Ian Boschung | SUI | 100 | 70 | - | - | 170 |
| 20. | Kasimir Tanner | SUI | 100 | 20 | 0 | 0 | 120 |
| 21. | Joël Mathys | SUI | 20 | 70 | - | 0 | 90 |
| 22. | Elias Boschung | SUI | 60 | 0 | 0 | 20 | 80 |
| 23. | Mugeeb Al-Rah Hassan | SUI | - | - | - | 20 | 20 |
| 24. | Zheng Chen Man | SUI | 0 | - | - | 0 | 0 |

# 1. vyšehradské prípravné sústredenie

Po pätnástich rokoch česko-poľsko-slovenských prípravných sústredení sa hlavne zásluhou slovenských organizátorov táto akcia opäť rozrástla. V dňoch 28. 6. až 6. 7. sme v Danišovciach privítali delegácie z Čiech, Poľska a po prvýkrát aj z Maďarska. Počas prípravného sústredenia absolvovali súťažiaci až šesť súťažných dní. Celkovým víťazom prípravného sústredenia sa stal Slovák: Eduard „Baklažán“ Batmendijn. Podrobné výsledky uvádzame v tabuľke.

| | meno | kraj. | d.1 | d.2 | d.3 | d.4 | d.5 | d.6 | $\sum$ |
|---|---|---|---|---|---|---|---|---|---|
| 1. | Eduard Batmendijn | SVK | 280 | 259.50 | 340 | 300.0 | 300.0 | 230 | 1709.50 |
| 2. | Jarosław Kwiecień | POL | 260 | 196.37 | 300 | 200.0 | 200.0 | 300 | 1456.37 |
| 3. | Stanisław Barzowski | POL | 200 | 184.21 | 240 | 300.0 | 180.0 | 233 | 1337.21 |
| 4. | Michał Glapa | POL | 200 | 194.98 | 200 | 200.0 | 200.0 | 290 | 1284.98 |
| 5. | Maciej Holubowicz | POL | 280 | 164.52 | 200 | 140.0 | 167.5 | 200 | 1152.02 |
| 6. | Jan-Sebastian Fabík | CZE | 200 | 164.14 | 150 | 156.5 | 180.0 | 163 | 1013.64 |
| 7. | Martin Raszyk | CZE | 240 | 120.00 | 43 | 173.0 | 237.5 | 163 | 976.50 |
| 8. | Jan Tabaszewski | POL | 0 | 153.59 | 173 | 200.0 | 212.5 | 230 | 969.09 |
| 9. | Albert Citko | POL | 40 | 151.70 | 200 | 200.0 | 237.5 | 110 | 939.20 |
| 10. | Michal Korbela | SVK | 140 | 148.26 | 100 | 240.0 | 142.5 | 163 | 933.76 |
| 11. | Weisz Ambrus | HUN | 160 | 152.59 | 83 | 249.5 | 142.5 | 130 | 917.59 |
| 12. | Erdős Márton | HUN | 60 | 129.54 | 133 | 200.0 | 82.5 | 110 | 715.04 |
| 13. | Somogyvári Kristóf | HUN | 140 | 146.90 | 175 | 40.0 | 82.5 | 100 | 684.40 |
| 14. | Pavel Madaj | SVK | 140 | 134.46 | 140 | 20.0 | 87.5 | 63 | 584.96 |
| 15. | Ondřej Hübsch | CZE | 180 | 143.19 | 33 | 56.5 | 100.0 | 63 | 575.69 |
| 16. | Michal Sládeček | SVK | 120 | 46.99 | 63 | 73.0 | 102.5 | 163 | 568.49 |
| 17. | Székely Szilveszter | HUN | 80 | 80.30 | 50 | 143.0 | 112.5 | 100 | 565.80 |
| 18. | Mernyei Péter | HUN | 100 | 81.74 | 50 | 153.0 | 52.5 | 120 | 557.24 |
| 19. | Zarándy Álmos | HUN | 140 | 46.99 | 50 | 120.0 | 142.5 | 50 | 549.49 |
| 20. | Peter Ralbovský | SVK | 100 | 102.57 | 73 | 0.0 | 65.0 | 163 | 503.57 |
| 21. | Václav Rozhoň | CZE | 140 | 126.75 | 33 | 40.0 | 110.0 | 30 | 479.75 |
| 22. | Matěj Konečný | CZE | 60 | 120.00 | 0 | 173.0 | 20.0 | 63 | 436.00 |
| 23. | Dominik Smrž | CZE | 140 | 20.00 | 10 | 120.0 | 112.5 | 0 | 402.50 |
| 24. | Alan Marko | SVK | 0 | 106.90 | 33 | 0.0 | 70.0 | 130 | 339.90 |

## Stredoeurópska olympiáda v informatike

V roku 2014 sa Stredoeurópska olympiáda v informatike (CEOI) konala v nemeckom meste Jena v dňoch 18. až 24. júna. Okrem členských krajín CEOI (Česka, Slovenska, Poľska, Maďarska, Nemecka, Rumunska a Chorvátska) sa súťaže zúčastnil aj pozvaný tím zo Švajčiarska a tím reprezentujúci „domácu“ nemeckú spolkovú republiku Durínsko (Thüringen).

Slovensko na CEOI 2014 reprezentovali štyria stredoškoláci: Michal Bui Truc Lam (Gym. Grösslingová Bratislava), Lukáš Ivan (Gym. Jura Hronca Bratislava), Michal Korbela (Gym. Bánovce nad Bebravou) a Pavel Madaj (Gym. Nedožerského Prievidza).

Našu delegáciu na tejto súťaži viedli doc. RNDr. Gabriela Andrejková, CSc. (PF UPJŠ v Košiciach), RNDr. Rastislav Krivoš-Belluš, PhD. (PF UPJŠ v Košiciach) a ako pozorovateľ sa zúčastnil Mgr. Vladimír Boža (FMFI UK v Bratislave).

Všetci štyria naši súťažiaci sa domov vrátili s medailami. Ich podrobné výsledky uvádzame v tabuľke.

| poradie a meno | 1. deň | | | 2. deň | | | $\sum$ | medaila |
|---|---|---|---|---|---|---|---|---|
| 9. Lukáš Ivan | 100 | 100 | 27 | 100 | 0 | 0 | 327 | striebro |
| 17. Pavol Madaj | 20 | 100 | 37 | 0 | 100 | 0 | 257 | bronz |
| 19. Michal Korbela | 100 | 0 | 0 | 100 | 35 | 0 | 235 | bronz |
| 21. Michal Bui Truc Lam | 100 | 0 | 100 | 30 | 0 | 0 | 230 | bronz |

# Medzinárodná olympiáda v informatike

Už dvadsiaty šiesty ročník Medzinárodnej olympiády v informatike uvítal Taipei, hlavné mesto Taiwanu. Štvorica slovenských súťažiacich síce tento rok na zlato nedosiahla, napriek tomu však podala kvalitné výkony a ani tento rok sa v medzinárodnej konkurencii nestratila. Išlo o doteraz najväčšiu Medzinárodnú olympiádu v informatike: zúčastnilo sa jej 311 súťažiacich z 81 krajín

Slovensko tento rok ako súťažiaci reprezentovali Eduard Batmendijn z Cirkevného gymnázia sv. Mikuláša v Starej Ľubovni, Bui Truc Lam z Gymnázia na Grösslingovej ulici v Bratislave, Michal Korbela z Gymnázia v Bánovciach nad Bebravou a Mário Lipovský z Gymnázia Jura Hronca v Bratislave.

Ako vedúci sa výpravy zúčastnili RNDr. Andrej Blaho, PhD. a RNDr. Michal Forišek, PhD. (obaja FMFI UK). V hlasovaní na zasadnutí lídrov krajín bol Michal Forišek zvolený na nasledujúce tri roky do medzinárodnej odbornej komisie. Táto komisia má na starosti odbornú stránku súťaže vrátane výberu súťažných úloh.

Kompletné výsledky našich súťažiacich uvádzame v nasledujúcej tabuľke.

| Meno | 1. deň | | | 2. deň | | | $\sum$ | medaila |
|---|---|---|---|---|---|---|---|---|
| Eduard Batmendijn | 30 | 100 | 100 | 100 | 100 | 7 | 437 | 34. miesto, striebro |
| Bui Truc Lam | 56 | 32 | 100 | 100 | 46 | 47 | 381 | 57. miesto, striebro |
| Mário Lipovský | 30 | 100 | 100 | 55 | 61 | 23 | 369 | 63. miesto, striebro |
| Michal Korbela | 56 | 32 | 42 | 75 | 35 | 47 | 287 | 111. miesto, bronz |

**IUVENTA – Slovenský inštitút mládeže** je príspevková organizácia priamo riadená Ministerstvom školstva, vedy, výskumu a športu SR. Pripravuje a riadi množstvo zaujímavých programov a projektov pre mladých ľudí, pracovníkov s mládežou a ľudí zodpovedných za mládežnícku politiku. Snaží sa o to, aby mladí ľudia poznali svoje možnosti, boli aktívni a pracovali na sebe tak, aby boli raz úspešní a uplatnili sa na trhu práce. IUVENTA vychováva mládež k ľudským právam, podporuje rozvoj dobrovoľníctva, vzdelávacie programy a tiež mladé talenty.

IUVENTA je realizátorom národných projektov v oblasti práce s mládežou KomPrax – Kompetencie pre prax a Praktik – Praktické zručnosti cez neformálne vzdelávanie v práci s mládežou, ktoré sú podporené z Európskeho sociálneho fondu. Zároveň administruje Programy finančnej podpory aktivít detí a mládeže MŠVVaŠ SR ADAM, je národnou agentúrou programu EÚ Mládež v akcii a národným partnerom európskej informačnej siete pre mládež Eurodesk.

Kontakt: IUVENTA – Slovenský inštitút mládeže
Búdková 2
SK-811 04 Bratislava 1
tel.: (421-2) 59 29 61 12
fax: (421-2) 59 29 61 23
e-mail: `iuventa@iuventa.sk`
web: `www.iuventa.sk`, `www.facebook.com/iuventa`

---

RNDr. Michal Forišek PhD., Slovenská komisia OI
Dvadsiaty deviaty ročník Olympiády v informatike
Vydavateľ: IUVENTA – Slovenský inštitút mládeže
Rok vydania: 2014
Rozsah: 149 strán
Náklad: vydané len v elektronickej podobe
ISBN: 978-80-8072-156-5
Neprešlo jazykovou úpravou

9 788080 721565