**Korešpondenčný seminár z programovania**
**XXVII. ročník, 2009/10**
**Katedra základov a vyučovania informatiky FMFI UK,**
**Mlynská Dolina, 842 48 Bratislava**

*KSP finančne podporujú: MICROSTEP-MIS spol. s r.o.*
*Agentúra na podporu výskumu a vývoja*

# Vzorové riešenia 2. kola letnej časti

opravoval Feki

## 1. Zuckermannova torta

(max. 10 bodov)

Ako bolo vidno na množstve riešení, nejeden riešiteľ by si rád vymenil úlohu s Miňom, motivovaný sladkými cukríkmi na tortách.

Bežnému človeku a o to viac programátorovi, ktorý veľmi nerád trávi čas a pamäť nad problémami bežného života, určite napadne nasledovný postup hľadania maxima z radu čísel: Najprv si povieme, že prvá hodnota je (zatiaľ) najväčšia. Budeme ju porovnávať s nasledujúcimi hodnotami a keď nájdeme väčšiu, tak si ju zapamätáme – teda sa z nej stane nové maximum.

To znamená, že hodnota, ktorú sme si zapamätali, je určite väčšia ako všetky predchádzajúce. Keď teda dôjdeme k poslednej hodnote, máme určite zapamätané globálne maximum.

Použijeme túto metódu na každý zo stolov a postupne sčítavame nájdené maximálne počty cukríkov.

Pár slov k zložitostiam: Skoro všetci mali správne riešenie, až na maličké chybičky. Podľa mňa nebolo kde pokaziť časovú zložitosť, ktorá bola $O(M \cdot N)$. Išlo skôr o formálnu stránku, teda či viete správne určiť zložitosti.

Pamäťové zložitosti sa ale rôznili. Často sa vyskytovali riešenia, ktoré si ukladali celý riadok vstupu, prípadne celý vstup. Vzorové riešenie využíva konštantné množstvo pamäte, teda zložitosť je $O(1)$.

Bodovanie:

- 1 bod som strhol za každú nesprávnu alebo neuvedenú zložitosť,
- 1 bod som strhával za $O(N)$ pamäte,
- 2 body za pamäťovú zložitosť $O(M \cdot N)$,
- 1 bod stratilo riešenie, ktoré nenulovalo niektoré premenné,
- 2 body stratili riešenia bez popisu.

Našťastie ste mali riešenia poväčšine správne, takže sa nikomu nepodarilo stratiť všetky body na maličkostiach.

### Listing programu:

```
var max, n, m, i, j, x, vys: longint;
begin
  readln(m, n);
  vys := 0;
  for i := 1 to m do begin
    max := 0;
    for j := 1 to n do begin
      read(x);
      if x > max then max := x;
    end;
```



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

```
    readln;
    vys := vys + max;
  end;
  writeln(vys);
end.
```

opravoval Maják
(max. 10 bodov)

## 2. Zákerné kartičky

**Dolný odhad zložitosti**

Aby sme vedeli zistiť, či si dve kartičky navzájom *zodpovedajú*, tak sa musíme na oboch z nich pozrieť na všetky čísla.[1]

Toto tvrdenie si dokážeme sporom. Nech existuje algoritmus, ktorý sa nepozerá na všetky čísla na oboch kartičkách a vždy správne rozhodne, či si dve kartičky *zodpovedajú*.

Zoberme si dve kartičky, ktoré si naozaj *zodpovedajú* a navyše v žiadnej z nich nie sú dve rovnaké čísla. Bez ujmy na všeobecnosti, nech sa náš algorimus nepozrie na aspoň jedno číslo na druhej karte. Vyberieme si jedno z nich a nahradíme ho za iné. Tým dostaneme tretiu kartu, pričom platí, že prvá a tretia karta si určite *nezodpovedajú*.

Čo sa stane, keď náš algoritmus spustíme na prvej a tretej karte? Jeho výpočet bude rovnaký, ako by bol na prvej a druhej karte, pretože druhá sa od tretej líši len v jednej pozícii, a na tú sa nepozrie. Preň sú to teda rovnaké vstupy a jeho odpoveď bude, že si karty *zodpovedajú*. To je ale spor s tým, že tento algoritmus správne rozhoduje, a preto neplatí predpoklad – takýto algoritmus neexistuje.

Ukázali sme si, že na každej karte sa musíme pozrieť na každé číslo, čo je $2 \cdot N^2$ operácií. Z toho máme dolný odhad zložitosti, rýchlejšie ako $O(N^2)$ to nepôjde.

**Triviálny algoritmus**

Najjednoduchšie, ako zistiť, či si dve karty *zodpovedajú*, je pre každý možný spôsob „otočenia“ postupne po dvojiciach porovnávať zodpovedajúce čísla z oboch kariet. Teda napríklad pre otočenie o 90 stupňov budem porovnávať číslo na pozícií $(i, j)$ s číslom na pozícii $(N - j + 1, i)$.[2]

Ak aspoň pre jeden z možných spôsobov sú porovnávané čísla stále rovnaké, tak si karty *zodpovedajú*, a obrátene. Preto je tento postup korektný.

**Časová zložitosť**

Možných „otočení“ je podľa zadania päť:

- rotácia o 90 stupňov,
- rotácia o 180 stupňov,
- rotácia o 270 stupňov,
- preklopenie okolo vodorovnej osi,
- preklopenie okolo zvislej osi.

Pri rotácii nie je špecifikovaný smer, ale v tomto prípade to nevadí.

Preto stačí päťkrát zopakovať toto priame porovnávanie, čím dostaneme algoritmus so zložitosťou $O(N^2)$. Ako sme si už vyššie ukázali, rýchlejší ani neexistuje, a preto ten náš je optimálny.

**Pamäťová zložitosť**

[1] Pričom ešte nevravíme, čo s nimi spravíme.

[2] Kde N je veľkosť karty; pozície číslujem od 1.



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

Náš algoritmus potrebuje mať v pamäti aspoň celú prvú kartu, aby sa do nej mohol pozerať na porovnávanie s číslom z druhej. Preto pamäťová zložitosť tohto algoritmu je $O(N^2)$.

Poznámka na okraj: Neukázali sme, že neexistuje iný algoritmus, ktorý by pracoval na inej báze ako porovnávaní jednotlivých prvkov a teda by si nemusel pamätať až $N^2$ čísiel. Otázku existencie riešenia s menšou pamäťou ponechávam ako úlohu na zamyslenie.

**Zákernosti zadania a bodovanie**

Povšimnutiahodná je skutočnosť, že dve identické karty si *nezodpovedajú*, pretože v zadaní sa o tom nepíše. Ak ste ich za zhodné považovali, tak ste prišli o jeden bod.

Na našom fóre sa objavila aj alternatívna interpretácia zadania, ktorá povoľovala skladať uvedené zobrazenia. Preto sme uznávali aj takéto riešenia, ale v tom prípade zas museli obsahovať všetky možné zobrazenia, inak ste stratili bod.

Za neodhadnuté alebo nesprávne odhadnuté zložitosti ste mohli stratiť do dvoch bodov. Zvyšné body ste mohli stratiť za nedostatočný či chýbajúci popis a za nekorektnosť riešenia.

## Listing programu:

```
var
  N, i, j, cislo: integer;
  karta: array[1..1000,1..1000] of integer;
  o90, o180, o270, zvislo, vodorovne: boolean;


begin
  o90 := true;
  o180 := true;
  o270 := true;
  zvislo := true;
  vodorovne := true;

  readln(N);
  for i:=1 to N do begin
    for j:=1 to N do begin
      read(karta[i,j]);
    end;
    readln();
  end;

  for i:=1 to N do begin
    for j:=1 to N do begin
      read(cislo);
      if cislo<>karta[N-j+1, i] then o90 := false;
      if cislo<>karta[N-i+1, N-j+1] then o180 := false;
      if cislo<>karta[j, N-i+1] then o270 := false;
      if cislo<>karta[N-i+1, j] then zvislo := false;
      if cislo<>karta[i, N-j+1] then vodorovne := false;
    end;
    readln();
  end;

  if (o90 or o180 or o270 or zvislo or vodorovne) then writeln('ano')
  else writeln('nie');
end.
```



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

# 3. Záhada obuvníkovho sna

opravoval Maty
(max. 10 bodov)

Väčšina riešiteľov si s týmto príkladom hravo poradila. Menej bodov ste mohli získať za neoptimálne triedenie, prípadne kvadratickú zložitosť pri hľadaní správnych topánok. Body ste tiež mnohí stratili za chýbajúce alebo zlé odhady časovej a pamäťovej zložitosti.

A teraz už k riešeniu. To čo spravíme prvé je, že utriedime pole nôh aj topánok. Nohy budeme obúvať v poradí od najmenšej po najväčšiu. Každej nohe sa budeme snažiť nájsť čo najtesnejšiu topánku, ktorá sa ňu ešte zmestí.

Je jasné, že ak existuje riešenie, tento postup ho určite nájde. Zišlo by sa ale dokázať, že tento postup nájde aj optimálne riešenie. Ukážeme, ako prerobiť optimálne riešenie na naše, pritom maximálny rozdiel sa nám nezhorší.

Vezmime najmenšiu nohu $i$, v ktorej by sa naše a optimálne riešenie líšilo. V našom riešení sme $i$ obuli najtesnejšiu topánku $t_i$ a preto musí mať v optimálnom riešení väčšiu topánku. Ak sme v optimálnom riešení topánku $t_i$ nepoužili, tak ľahko upravíme optimálne riešenie do nášho tvaru, pričom sa rozdiel nezhorší. Ak sme ju použili, tak sme topánku $t_i$ obuli nejakej väčšej nohe $j$, keďže vo všetkých menších sa naše a optimálne riešenie zhodujú. Teraz ale môžeme v optimálnom riešení prehodiť topánku, ktorú má noha $i$, s topánkou, ktorú má noha $j$, pričom maximálny rozdiel sa znova nezhorší.

Týmto postupom vieme postupne prerobiť optimálne riešenie na naše riešenie. A keďže sa nám maximálny rozdiel nezhoršuje, aj naše riešenie je optimálne.

Ako teda efektívne hľadať ku každej nohe najtesnejšiu topánku? Veľmi jednoducho. Pre najmenšiu nohu začneme prechádzať pole topánok, až kým nenarazíme na topánku, ktorú môžeme obuť. Keď chceme nájsť topánku pre druhú najmenšiu nohu, stačí nám pokračovať v hľadaní tam, kde sme s predošlou nohou skončili, pretože topánky, ktoré sme neobuli na predošlú nohu, neobujeme ani na túto. Toto niektorým ľuďom nenapadlo a hľadať najtesnejšiu topánku vždy začali od začiatku.

Časová zložitosť bude zložitosť triedenia plus prejdenie obidvoch polí teda $O(N \log N + K \log K + N + K)$. Ak máme $K < N$, tak program hneď skončí a jeho zložitosť je $O(1)$. Zaujímavý je teda prípad $K \geq N$ a preňho môžeme čas zjednodušiť na $O(K \log K)$.[3] Pamäťová zložitosť je lineárna, teda $O(N + K)$.

## Listing programu:

```
program priklad3;
type pole = array[1..1000] of longint;
var n, k: longint;
    noh, top: pole;
    p1, p2, max, i: longint;

procedure quicksort(var A: pole; l, r: integer);
var i, j, pivot, pom: integer;
begin
  i := l; j := r;
  pivot := A[(l + r) div 2];
  repeat
    while A[i] < pivot do inc(i);
    while pivot < A[j] do dec(j);
    if i <= j then begin
      pom := A[i];
      A[i] := A[j];
      A[j] := pom;
      inc(i); dec(j);
    end;
```

[3] Toto ohraničenie platí pre všetky prípady, nie len $K \geq N$, keďže $O(1) \subseteq O(K \log K)$.



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

```
  until i > j;
  if j > l then quicksort(A, l, j);
  if i < r then quicksort(A, i, r);
end;


begin
  readln(n, k);
  for i := 1 to n do read(noh[i]);
  for i := 1 to k do read(top[i]);
  quicksort(noh, 1, n);
  quicksort(top, 1, k);
  p1 := 1; p2 := 1; {p1 pre pole noh, p2 pre pole topanok}
  max := 0;
  while (p1 <= n) and (p2 <= k) do begin
    if noh[p1] <= top[p2] then begin
      if (max < top[p2] - noh[p1]) then max := top[p2] - noh[p1];
      inc(p1);
      inc(p2);
    end else inc(p2); {posuniem sa iba v poli topanok}
  end;
  if p1 = n + 1 then
     writeln(max)
  else
     writeln('Nejde to');
end.
```

## 4. Zapeklitý Top Ológ

opravovala Halucinka
(max. 15 bodov)

Táto úloha je „šitá“ na algoritmus zvaný topologické triedenie, ktorý vytvorí topologické usporiadanie. Najprv si povedzme, čo to je:

Majme orientovaný graf. Topologické usporiadanie vrcholov tohto grafu je lineárne usporiadanie všetkých vrcholov také, že ak graf obsahuje hranu $(a, b)$, potom sa $a$ nachádza v tomto usporiadaní skôr ako $b$. Topologické usporiadanie vrcholov sa dá nájsť práve vtedy, keď je graf acyklický. Prečo?

Ak by sme mali cyklický graf, potom existuje cyklus medzi nejakými vrcholmi:

$$v_1 \to v_2 \to \cdots \to v_n \to v_1$$

Potom z týchto vrcholov nevieme vybrať jeden, ktorý bude v usporiadaní pred ostatnými. Teda riešenie, ako zapísať tieto vrcholy do usporiadania, neexistuje.

Ak je náš graf acyklický, potom vždy nájdeme topologické usporiadanie. Algoritmus ako ho nájdeme teraz opíšeme:

Pre každý vrchol si pamätáme, koľko hrán ide do neho, a pamätáme si, do ktorých vrcholov vedú hrany z tohto vrcholu. Prvé v usporiadaní budú tie vrcholy, do ktorých nejdú žiadne hrany. Tieto môžeme rovno zapísať do výsledného usporiadania. Nech je takýmto vrcholom aj vrchol V. Teraz už vrchol $V$ máme usporiadaný – už o ňom neuvažujeme.

Pozrieme sa na hrany, ktoré idú z vrcholu $V$. Tieto hrany zrušíme, teda pre každý vrchol, do ktorého išla hrana z $V$, sa zníži počet hrán, ktoré idú o neho o 1. Do niektorých vrcholov teraz už nemusí ísť žiadna hrana. Tie pridáme do usporiadania a rovnako ich odstránime z grafu spolu s hranami, ktoré vedú z nich do iných vrcholov. Takto postupujeme, až kým nemáme v usporiadaní všetky vrcholy.



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

Ak už neexistuje vrchol, do ktorého nevedie žiadna hrana, a napriek tomu v usporiadaní nie sú všetky vrcholy, tak nutne nejaké z týchto vrcholov musia tvoriť cyklus a neexistuje takéto usporiadanie.

Teraz sa vráťme k zadaniu: Uvedomme si, že my chceme riešiť rovnakú úlohu. Máme $M$ dvojíc Ológov. Ak máme dvojicu $(a, b)$, potom sa $a$ musí v našom usporiadaní nachádzať skôr ako $b$. Vytvoríme si teda orientovaný graf tak, že Ológovia budú vrcholy a hrany vytvoríme tak, že pre každú dvojicu $(a, b)$ budeme mať orientovanú hranu z $a$ do $b$. Znamená to, že $a$ musí byť v usporiadaní skôr ako $b$. Teda hľadáme topologické usporiadanie. Ako to teda bude vyzerať pre túto úlohu?

Pre každéhoOlóga si pamätáme nad kým vyhral a taktiež s koľkýmiOlógmi prehral. Na začiatku nájdeme Ológov, ktorí s nikým neprehrali a tých dáme do výsledného poradia (je jedno v akom poradí – vraví zadanie).

Teraz postupne ideme po Ológoch, ktorých sme dali do výsledného poradia a tým sme ich už vyradili z grafu. Zoberieme siOlóga z výsledného poradia a všetkým tým, ktorí s ním prehrali, znížime počet prehier o 1 (odoberáme hrany grafu, ktoré vedú od vyradenéhoOlóga). Kontrolujeme, či sa týmto neznížil niekomu počet prehier na 0. Ak áno, pridáme ho na koniec poradia.

Takto spracujeme postupne všetkých Ológov vo výslednom poradí a zároveň na koniec tohto poradia dávame ďalších Ológov. Teda po dokončení budeme mať vo výslednom poradí všetkých Ológov. Ak nie, potom musel vzniknúť cyklus a neexistuje požadované poradie.

**Časová zložitosť**

Na začiatku prebehneme celé pole a hľadáme Ológov, ktorí majú 0 prehier. Potom už len znižujeme počty prehier Ológov tak, že na konci budú mať všetci nulu. Spolu je všetkých prehier toľko, koľko hrán, teda $M$. Čiže časová zložitosť bude $O(M + N)$.

**Pamäťová zložitosť**

Pamätáme si všetky hrany a pre každý vrchol si pamätáme dve čísla: počet prehier a počet výhier. Teda pamäťová zložitosť bude $O(M + N)$.

**Bodovanie**

Riešenia s časovou zložitosťou $O(N^2)$ dostávali 12-13 bodov. Rovnako veľa bodov dostávali aj riešenia, ktoré využívali nejaké triedenie. Horšia časová zložitosť – väčšinou to bola $O(N^3)$ – dostávala 8 až 10 bodov. Za zabudnuté napísanie zložitostí šiel dole bodík za každú. Objavilo sa pár riešení, ktoré nefungovali, ale idea bola veľmi dobrá, tým sa ušlo okolo 7 bodíkov. Vcelku ma prekvapilo, koľko relatívne dobrých a hlavne funkčných riešení prišlo. Ste super :-).

## Listing programu:

```
const maxN = 100;
var vyhra_nad: array [1..maxN, 1..maxN] of longint;
    pocet_prehier: array [1..maxN] of longint;
    pocet_vyhier: array [1..maxN] of longint;
    vysledne_poradie: array [1..maxN] of longint;
    k, N, M, i, l, Olog, a, b, prehral_s_Ologom: longint;

begin
  readln(N, M);
  for i := 1 to N do begin
    pocet_vyhier[i] := 0;
    pocet_prehier[i] := 0;
  end;

  for i := 1 to M do begin
    readln(a, b);
```



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

```
    {ulozi si s kym vsetkym vyhral a pocita kolko ich bolo}
    inc(pocet_vyhier[a]);
    inc(pocet_prehier[b]);
    vyhra_nad[a, pocet_vyhier[a]] := b;
  end;

  k := 1;
  for i := 1 to N do begin  {hladame topologov, co maju nula prehier}
    if pocet_prehier[i] = 0 then begin
      {ak ma uz nula prehier, tak teraz je ten spravny cas zaradit ho do vysledneho
poradia}
      vysledne_poradie[k] := i;
      inc(k);
      pocet_prehier[i] := -1; {-1 znaci, ze uz je spracovany tento vrchol grafu}
    end;
  end;

  {prechadzame vysledne poradie a znizujeme pocet_prehier este nespracovanych
vrcholov, pokial prehrali s vrcholom, ktory uz je vo vyslednom poradi}
  for i := 1 to N do begin
    if k > i then begin
      {je to Olog, ktory uz je spracovany, teda tym, co s nim prehrali, decrementneme
pocet prehier}
      Olog := vysledne_poradie[i];
      for l := 1 to pocet_vyhier[Olog] do begin
        {kazdemu, nad kym vyhral Olog, zmensime pocet prehier o jeden, lebo Ologa
sme uz vyradili}
        prehral_s_Ologom := vyhra_nad[Olog, l];
        dec(pocet_prehier[prehral_s_Ologom]);
        if pocet_prehier[prehral_s_Ologom] = 0 then begin
          {ak sa takto nejakemu Ologovi zmensi pocet prehier na 0, mozeme ho
spracovat}
          vysledne_poradie[k] := prehral_s_Ologom;
          inc(k);
          pocet_prehier[prehral_s_Ologom] := -1;
        end;
      end;
    end;
  end;

  if k = N + 1 then begin
    for i := 1 to N do
      write(vysledne_poradie[i], ' ');
    writeln;
  end
  else writeln('Poradie neexistuje');
end.
```

## 5. Ozajstná lotéria!

vzorák písal mišof
(max. 17 bodov)

Toto vzorové riešenie sa skladá z dvoch častí. Prvá je písaná z pohľadu riešiteľa a vysvetľuje, ako sa dalo s minimálnou námahou k bodom dopracovať. Na to stačilo odhaliť, **čo** použitý program robí.

No a druhá je písaná z pohľadu autora a vysvetľuje aj to, **ako** to použitý program dosiahne a **prečo** vlastne funguje.



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

**Prvý pokus**

Spustím program, zadám vstup, počkám kým dobehne. Ajhľa, po chvíli dobehol a mám výstup. Spustím program znovu, zadám iný vstup, počkám kým dobehne. Ajhľa, výstup. Ak som nemal zrovna smolu, dostal som presne ten istý výstup ako pred chvíľou. Skúsime ešte tretíkrát. Ajhľa, zase je tu to isté. Žeby všetky tie reči o náhodnej lotérii boli zase len kecami? No neviem, nechám si to na zajtra.

**Druhý pokus**

Spustím program, zadám ten istý vstup ako včera, počkám kým dobehne. Ajhľa, výstup. Iný ako včera.

Čo sa zmenilo? Nuž, napríklad čas. Pohľad do programu ukazuje, že sa tam nejaké to volanie `time` veru nachádza. Asi na ňom bude záležať.

**Experiment**

Nielen v našej lotérii, ale napríklad aj pri riešení matematických úloh často platí: keď nevieš pochopiť, ako niečo funguje, začni si to skúšať na konkrétnych príkladoch. Možno odpozoruješ nejaké závislosti, ktoré ti pomôžu.

V našom prípade sa oplatí začať „najvnútornejšou“ funkciou `magia`. Niečo sa tam sčituje, niečo sa tam mieša... to sa nám predsa nechce analyzovať. Skúsme sa na ňu pozrieť ako na čiernu krabičku. Upravíme náš program tak, že keď ho spustíme, vypíše nám najskôr postupnosť čísel, ktorú do funkcie `magia` vkladá, a následne hodnotu, ktorá z nej vypadne.

Dostaneme tak napríklad nasledujúce pozorovania:

| **vstup** | **výstup** |
|---|---|
| (0,29,47,109,1441,2025,4747,...,3995702,3995766,3999950,1274573512) | 530 |
| (0,29,47,109,1441,2025,4747,...,3995702,3995766,3999950,1274573522) | 540 |
| (0,29,47,109,1441,2025,4747,...,3995702,3995766,3999950,1274573526) | 544 |
| (0,1,2,3,4,5,6,7,8,9,10,...,3995702,3995766,3999950,1274573993) | 64 |

V prvých troch prípadoch uvedených v tabuľke sme na tiket nenapísali nič, v poslednom prípade sme na vstup dali postupnosť čísel od 1 po 10.

Môžeme si všimnúť množstvo zaujímavých skutočností, napríklad:

- Vyzerá to, že prvá je vždy nula.
- Vyzerá to, že posledné číslo je rádovo väčšie ako všetky ostatné.
- Väčšina čísel je stále rovnaká – napr. 3999950 sme sa ani zmenou vstupu nezbavili.

V prvých troch prípadoch dokonca jediné, čo sa zmenilo, je posledná hodnota. Čo je zač? Pri pohľade na hlavný program je odpoveď jasná: ide o výstup funkcie `time(0)`, teda unixový timestamp zodpovedajúci aktuálnemu dátumu a času. (Ide o počet sekúnd od polnoci dňa 1. 1. 1970.)

Ako sa mení výstup funkcie `magia` v závislosti od času? Čas narástol o 10 sekúnd, výstup sa zväčšil o 10. Čas narástol o ďalšie 4, výstup sa zväčšil o 4. O nejakú chvíľu neskôr je však výstup zase malý. Z toho už ľahko sformulujeme hypotézu:

*Výstup funkcie* `magia` *je rovný aktuálnemu timestampu modulo 947.*

**Rozpracovanie hypotézy**

Kto sa dopracoval až sem, mal už takmer vyhrané.

Upravme teda program tak, že `magia` bude rovno vracať poslednú hodnotu poľa `data`, modulo 947. Čo sa teraz s touto funkciou robí?

Riadok `for (int i=1; i<=P; ++i) data[i] -= magia(data);` je jasný – túto hodnotu odčítame od každého prvku v poli. (Všimnite si, že odčítavame stále tú istú hodnotu, lebo poslednú hodnotu v poli `data` zmenšíme až na koniec.)

Po vykonaní tohto riadku sa teda hodnoty v poli `data` o trochu zmenšia. A špeciálne sa stane to, že timestamp, ktorý je v poli `data` na poslednej pozícii, sa zaokrúhli dodola na najbližší násobok 947.



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

Keď sme my starí bývali mladí, chodila v telke relácia „900 sekúnd o knihách“ a trvala presne štvrť hodiny. Odtiaľ mám užitočný rýchly odhad, koľko je tisíc sekúnd.

Čo to znamená v našom prípade? Ak je naozaj pravda, že výstup lotérie závisí len od času, tak sa tento výstup bude meniť zhruba raz za štvrť hodiny.

**Overenie hypotéz**

Najpriamočiarejšie, čo sa v tejto chvíli sa dalo spraviť, bolo odskúšať všetky možnosti. Začneme s aktuálnym timestampom a s krokom 947 postupne vyskúšame všetky až po deadline na odovzdanie riešení. Zakaždým spustíme program, aby nám povedal, koľko bodov sa počas dotyčnej štvrťhodiny bude rozdávať.

Potom si overíme, že najbližších niekoľko výstupov sa skutočne zhoduje s výstupmi pôvodného programu. A keď všetko sedí, uveríme, že už to máme.

Zaujímavé bodové zisky sú v nasledujúcej tabuľke.

| **timestamp** | **čas** | **body** |
|---|---|---|
| 1272358301 | utorok 27. 4. 2010 o 10:51:41 | 17 |
| 1272929342 | utorok 4. 5. 2010 o 01:29:02 | 16 |
| 1273920851 | sobota 15. 5. 2010 o 12:54:11 | 15 |
| 1274104569 | pondelok 17. 5. 2010 o 15:56:09 | 12 |

(Ku každému bodovému zisku uvádzame posledný interval, kedy sa uvedené body dali získať. Konkrétny timestamp leží niekde v strede daného intervalu.)

No a už len stačilo odovzdať v správnom čase a overiť si, že naozaj na konte nabehol správny počet bodov `:-)`.

**Sľubovaná druhá časť vzorového riešenia**

Ou kej, body máme, ale ako sa to riešenie zbaví všetkého okrem timestampu? Ako to, že výstup nezávisí ani od vstupu, ani od toho, aké náhodné čísla generátor vygeneruje? A ako dosiahne funkcia `magia`, že jej výstup od skoro ničoho nezávisí?

**Funkcia magia**

Všetky sčítania uvedené v tejto časti treba automaticky chápať modulo $P = 947$, lebo sa mi to nechce všade rozpisovať a bude to takto prehľadnejšie.

Odsimulujme si beh funkcie `magia` na menšom vstupe, napríklad $(a, b, c, d, e)$. V každej iterácii sa po sebe idúce dvojice čísel sčítajú. Teda po prvej budeme mať $(a+b, b+c, c+d, d+e)$, po druhej $(a + b + b + c, b + c + c + d, c + d + d + e) = (a + 2b + c, b + 2c + d, c + 2d + e)$, po tretej to bude $(a + 3b + 3c + d, b + 3c + 3d + e)$ a po poslednej, štvrtej iterácii dostávame jediný prvok s hodnotou $a + 4b + 6c + 4d + e$.

Tieto koeficienty by už mali byť každému, kto sa trochu obtrel o matematiku, povedomé – sú to kombinačné čísla. A skutočne, ľahko dokážeme matematickou indukciou podľa $N$, že ak začneme funkciu `magia` s poľom $(a_0, a_1, a_2, \ldots, a_N)$, tak ju skončíme s číslom

$$a_0 \binom{N}{0} + a_1 \binom{N}{1} + \cdots + a_N \binom{N}{N}$$

V našom prípade nás zaujíma $N = P = 947$. Lenže $P$ je prvočíslo, a teda každé z kombinačných čísel $\binom{P}{1}$ až $\binom{P}{P-1}$ je deliteľné $P$. Preto hodnota, ktorú `magia` vráti, vôbec nezávisí od hodnôt v poli `data` okrem tých na indexoch 0 a $P$. Presnejšie, `magia` vráti hodnotu (`data`$[0]$ + `data`$[P]$) mod $P$. No a keďže sme šikovne zabezpečili, že na pozícii 0 v poli `data` je nula (lebo sme si ju tam vložili – vidíte kde? – a nič menšie tam nikdy nedáme), tak `magia` musí vrátiť hodnotu `data`$[P]$ mod $P$.

**Funkcia zazrak**

Už sme si vysvetlili, že ako prvé odpočítame od každej z hodnôt v poli `data` tú istú hodnotu.

V nasledujúcom riadku programu postupne každú z týchto hodnôt použijeme na inicializáciu generátora náhodných čísel. Toto je obyčajná kamufláž. Keď totiž najskôr zavoláme



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

`srand(X)` a potom `srand(Y)`, tak dostaneme generátor náhodných čísel inicializovaný hodnotou `Y`. Dôležité je teda len úplne posledné volanie `srand`, a jeho parametrom je, ako ináč, náš starý známy nadol zaokrúhlený timestamp.

Od tohto okamihu sú všetky hodnoty, ktoré budú vracať volania `rand`, jednoznačne určené.

Jediným problémom je, že v poli `data` ešte stále máme nejaké údaje, ktoré zadal používateľ. Ako sa ich zbavíme?

Vytlačíme ich von. Posledný trik je v tom, že číslo 42 je párne. Ak teda prenásobíme konkrétnu hodnotu `data[j]` číslom 42, na poslednom mieste v binárnom zápise novej hodnoty bude určite 0. Potom pripočítame nejakú náhodnú hodnotu – tá je ale jednoznačne určená timestampom. V tomto okamihu teda posledný bit `data[j]` závisí len od timestampu a od ničoho iného.

Keď sa to isté zopakuje znova, budú timestampom jednoznačne určené posledné dva bity `data[j]`. A tak ďalej. Keďže $P + 97$ je oveľa viac ako 32, po $P + 97$ opakovaniach úprav poľa `data[j]` už nebude po pôvodných hodnotách ani chýru, ani slychu.

**Nepovinná domáca úloha**

Vyrobiť zadanie tejto úlohy bolo pravdepodobne náročnejšie ako ju vyriešiť – bolo totiž potrebné nájsť také konštanty a takú presnú formu zadania, aby časy, kedy sa bude dať „vyhrať“ veľa bodov, vyšli rozumne – teda napr. 15 bodov každému, kto sa do riešenia pustí skôr ako cez posledný víkend a 12 každému, kto tak spraví aspoň cez ten víkend. Skúste sa zamyslieť, ako by ste spravili zadanie podobnej úlohy, ak by termín odovzdania riešení bol inokedy.

## 6. Orientálny trh

opravoval Mio
(max. 20 bodov)

Najprv by som vyjadril svoje sklamanie, že som nikomu nemohol dať plný počet bodov. Prišiel totiž len jeden vzorák, aj tomu chýbal zdrojový kód.

**Bodovanie:**

- Vzorák (čas $O(k)$, pamäť $O(k)$) by dostal 20 bodov,
- program, ktorý bežal v čase $O(k \log k)$ dostal max. 15b,
- program, ktorý bežal v horšom čase ($k^2$, $n \cdot k$) dostal max. 10b,
- za chýbajúci zdroják bola odmena -5 bodov,
- za chýbajúce zložitosti bolo -3b,
- za drobné chyby (lenivosť skompilovať to po sebe) ste mohli získať bonus: odpočítanie jedného bodu.

Predtým, ako sa dostaneme k samotnému riešeniu, spravíme si ešte jazykové okienko. Viacerí z vás napísali, že rátali prefixové sumy. Toto nie je celkom pravda. Prefixové sumy sú od slova prefix = predpona. To znamená, že suma ide od začiatku postupnosti, po nejaký prvok. V tomto prípade ste ale počítali od konca. Keď to už chceme nazvať nejakým cudzím slovom, tak tieto sumy nazveme suffixové.

**A teraz konečne ten vzorák:**

Stručné zhrnutie algoritmu:

1. Vypočítame si výsledné ceny za jednotlivé dámy v stánkoch.
2. V lineárnom čase si nájdeme medián (stredný prvok) výsledných cien.
3. Spočítame si počet dám v prvej polovici poľa.

Tu nám môžu nastať 3 možnosti – buď ich je viac ako $N$, alebo po pripočítaní niekoľkých dám z mediánu ich je presne $N$, alebo je ich po pripočítani všetkých dám z mediánu menej ako $N$. V prvom prípade skočíme na krok 2, ale budeme počítať len s prvou polovicou



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

poľa, v druhom prípade už máme výsledok a v treťom prípade skočíme na krok 2 pre druhú polovicu poľa, ale už potrebujeme len $N-$ (počet dám v prvej polovici poľa) dám. Toto celé zbehne v lineárnom čase.

**Príprava**

Prvou vecou, čo treba spraviť, je vypočítať si výslednú cenu dámy v stánku. Vypočítame si teda, koľko dáma z tohto stánku nechá na druhej strane uličky. Spravíme si suffixové sumy – odzadu vypĺňame pole tak, že $i$-ty prvok dostaneme súčtom $(i+1)$-vého prvku a sumy, ktorú nechá v $i$-tom stánku. Tieto ceny potom pripočítame k cenám dám v stánkoch. Táto časť má časovú zložitosť $O(k)$.

**Hľadanie mediánu**

Hľadanie mediánu je najzložitejšou časťou algoritmu. Medián totiž potrebujeme nájsť v lineárnom čase. Algoritmus pracuje takto:

Rozdelí prvky na pätice a tieto utriedi. Ak nie je počet prvkov deliteľný 5, tak posledná skupinka má menej ako 5 prvkov. Triediť 5 prvkov vieme v konštantnom čase, a teda utriedenie všetkých pätíc trvá lineárny čas.

Z každej z týchto skupín potom vyberieme medián (v konštantnom čase, čiže spolu máme zasa lineárny čas). Ak máme v poslednej skupinke párny počet prvkov, vyberieme väčšieho z prostredných dvoch.

Rekurzívne nájdeme medián týchto vybraných mediánov, označme ho $M$. Rozdelíme si pole na dve časti: do prvej dáme prvky menšie ako $M$ a do druhej väčšie ako $M$. Prvky rovné $M$ rozmiestnime rovnomerne. Medián sa nachádza vo väčšej z týchto dvoch častí, preto ho ďalej rekurzívne hľadáme tam.

Aspoň polovica z 5-členných skupín má väčší alebo rovnaký medián ako $M$; v takýchto skupinách sú aspoň tri prvky väčšie alebo rovnaké ako $M$ (zanedbávame skupinu s menej ako 5 prvkami). Z toho vyplýva, že aspoň rádovo $3n/10$ prvkov je väčších alebo rovnakých ako $M$; podobne $3n/10$ prvkov je menších alebo rovnakých ako $M$. V najhoršom prípade sa preto rekurzívne zavoláme na $7n/10$ prvkov.

V jednom rekurzívnom volaní potrebujeme lineárny čas na triedenie pätíc a vyberanie mediánov, ďalej sa rekurzívne zavoláme na najviac $n/5$ prvkov a potom na najviac $7n/10$ prvkov. Časovú zložitosť môžeme preto vypočítať takto:

$$T(n) = T(n/5) + T(7n/10) + O(n)$$

Z tejto rovnosti pomocou trochy matematiky vyplýva $T(n) = O(n)$, takže celé hľadanie mediánu beží v lineárnom čase.

**Samotné riešenie**

V podstate hľadáme najdrahší stánok, z ktorého musíme niečo kúpiť. Dá sa to prirovnať k binárnemu vyhľadávaniu. Postavíme sa do stredu a pozrieme sa, či máme dosť. Ak nie, hľadáme v pravej polovici poľa; ak máme akurát, tak sme skončili; a ak máme málo, tak hľadáme v ľavej polovici. Problém je, že pole nie je utriedené, takže stredný prvok nájdeme pomocou algoritmu na nájdenie mediánu. A to, či máme dosť, zistíme len tak, že to po jednom spočítame.

Nájdeme teda medián a spočítame počet dám v ľavej polovici. Keďže algoritmus nám upravil pole tak, aby pred mediánom boli menšie prvky ako on a po ňom väčšie, tak sa nám viac oplatí brať z ľavej polovice. Pokiaľ je ich viac ako $N$, tak z pravej polovice určite nikoho nechceme, lebo by nás to stálo viac peňazí. Ale tiež nechceme všetky dámy z ľavej polovice, preto opakujeme algoritmus na tejto polovici. Pokiaľ ich počet po pripočítaní niekoľkých (od 0 po počet dám v stánku, ktorý je medián) dám z mediánu je $N$, tak máme výsledok; už stačí len zrátať výslednú cenu. Pokiaľ ich je málo, tak kúpime všetky dámy v ľavej polovici



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

a opakujeme algoritmus v pravej polovici, pričom už chceme dokúpiť len $N-$(počet dám naľavo).

Toto celé funguje v lineárnom čase. Dôkaz: Medián nájdeme v lineárnom čase, počet dám a ich výslednú cenu tiež. Keďže sa algoritmus cyklí a v prvom kroku vykoná $k$ operácií, v druhom $k/2$, ... čiže dostávame sumu $k + k/2 + k/4 + \ldots < 2k$.

Čiže dostávame časovú zložitosť $O(k)$ a pamäťovú $O(k)$, lebo si pamätáme len konštantne veľa informácií ku každému stánku.

Poznámka k zdrojáku: na väčšinu vecí sa dá použiť nejaký algoritmus z STL (ktorý takmer zaručene funguje lepšie ako moje implementácie), ja som ich tu ale pre názornosť rozpísal.

## Listing programu:

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

typedef pair<int, int> Stanok; // (cena, pocet dam)

//obycajny selection sort, kedze bezi pre najviac 5 prvkov, tak bezi v konstantnom case
//triedi interval <begin, end)
void sort(int begin, int end, vector<Stanok> &p){
    for(int i = begin; i < end; i++)
        for(int j = i + 1; j < end; j++)
            if(p[i] > p[j])
                swap(p[i], p[j]);
}

//rozdeli interval <begin, end) na 2 casti podla pivota a vrati pocet prvkov lavej casti
int partition(int begin, int end, vector<Stanok> &p, Stanok pivot){
    vector<Stanok> l, r;
    bool nalavo = false;
    for(int i = begin; i < end; i++){
        //prvky mensie ako pivot idu do laveho pola, vacsie do praveho,
        //rovne striedavo
        if(p[i] < pivot || p[i] == pivot && nalavo)
            l.push_back(p[i]);
        else
            r.push_back(p[i]);

        if(p[i] == pivot)
            nalavo = !nalavo;
    }

    for(int i = 0; i < end - begin; i++) //prepiseme originalne pole
        if(i < (int)l.size())
            p[begin + i] = l[i];
        else
            p[begin + i] = r[i - l.size()];

    //vratime velkost lavej casti
    return l.size();
}

//hladanie medianu v intervale <begin,end)
Stanok median(int begin, int end, vector<Stanok> &p, int w){
    int k = end - begin;
```



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

```
    //utriedime patice
    for(int i = begin; i + 5 <= end; i += 5)
        sort(i, i + 5, p);
    sort(end - k % 5, end, p);

    //ak mame male pole, hned vratime median
    if(k <= 5)
        return p[w];

    //zistime mediany patic
    vector<Stanok> mediany5;
    for(int i = begin; i + 5 <= end; i += 5)
        mediany5.push_back(p[i + 2]);
    if(k % 5 > 0)
        mediany5.push_back(p[end - (k % 5 + 1) / 2]);

    //najdeme median medianov patic
    Stanok median_medianov = median(0, mediany5.size(), mediany5, mediany5.size()
/ 2);

    //rozdelime na 2 casti
    int l = begin + partition(begin, end, p, median_medianov);

    //bud sme nasli median a hned ho vratime
    if(l == w)
        return median_medianov;

    //alebo si vyberieme cast v ktorej je median(pochopitelne - vacsia z 2 casti)
    if(l > w)
        return median(begin, l, p, w);
    else
        return median(l, end, p, w);
}

//spocita damy v intervale <begin,end)
int zrataj_damy(int begin, int end, vector<Stanok> &p){
    int sum = 0;
    for(int i = begin; i < end; i++)
        sum += p[i].second;
    return sum;
}

//spocita cenu vsetkych dam v intervale <begin,end)
int zrataj_cenu(int begin, int end, vector<Stanok> &p){
    int sum = 0;
    for(int i = begin; i < end; i++)
        sum += p[i].first * p[i].second;
    return sum;
}

//samotne riesenie ulohy
int zrataj(int k, int n, vector<Stanok> &p){
    int begin = 0, end = k, cena = 0;
    while(true){
        int m = (begin + end) / 2;
        median(begin, end, p, m);

        int pocet_l = zrataj_damy(begin, m, p);
        int pocet_m = p[m].second;
```



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

```
        if(pocet_l <= n && n <= pocet_l + pocet_m){
            cena += zrataj_cenu(begin, m, p);
            cena += (n - pocet_l) * p[m].first;
            return cena;
        }else if(n < pocet_l){
            end = m;
        }else{
            cena += zrataj_cenu(begin, m, p);
            n -= pocet_l;
            begin = m;
        }
    }
}


int main(){
    int k, n;
    cin >> n >> k;
    vector<Stanok> pole;
    vector<int> sperky;
    pole.resize(k);
    sperky.resize(k);
    //nacitanie vstupu
    for(int i = 0; i < k; i++)
        cin >> pole[i].first;
    for(int i = 0; i < k; i++)
        cin >> pole[i].second;
    for(int i = 0; i < k; i++)
        cin >> sperky[i];

    //zratanie suffixovych suctov
    for(int i = k - 2; i >= 0; i--)
        sperky[i] += sperky[i + 1];

    //zratanie vyslednej sumy za damu v i-tom obchode
    for(int i = 0; i < k; i++)
        pole[i].first += sperky[i];

    cout << zrataj(k, n, pole) << endl;
}
```

## 7. Oranžové stužky

opravoval Bob
(max. 20 bodov)

Ach jaj, prišli iba štyri riešenia. O čo ľahšie sa to bude opravovať, o to ťažšie sa bude vymýšľať vzorák.

Fajn, posťažoval som sa, prekonal som vrodenú lenivosť aj letné horúčavy a teraz hurá na riešenie: Rozhodcov plán, podľa ktorého pospája stužkami dievčatá a chlapcov z jedného tímu, je vlastne graf. Vrcholy tohto grafu reprezentujú deti a hrany zase stužky.

Počet hrán, ktoré vychádzajú[4] z vrcholu $v$, označujeme ako *stupeň* $v$. Úlohou je dosiahnuť, aby mal každý vrchol takú paritu svojho stupňa, akú požaduje vstup. Jediné, čo môžeme urobiť, je odstrániť z grafu niektoré hrany.

[4] Alebo vchádzajú? Máme neorientovaný graf, tak neviem.



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

Ak graf nie je súvislý, stačí sa zaoberať každým komponentom osobitne – odstránením nejakej hrany totiž zmeníme stupne len dvom vrcholom z toho istého komponentu. Ďalej preto budeme predpokladať, že graf je súvislý.

**Kedy treba kričať, že sa to nedá?**

Keď sčítame stupne všetkých vrcholov grafu, dostaneme dvojnásobok počtu hrán. Prečo? Každá hrana má dva konce, preto ju započítame do stupňa práve dvom vrcholom. Z tohto tvrdenia vyplýva užitočný poznatok: súčet stupňov všetkých vrcholov grafu je párny.

Ak je v grafe nepárny počet chlapcov, potom by mal byť vo výslednom grafe nepárny počet vrcholov s nepárnym stupňom (uf). To ale znamená, že aj súčet stupňov všetkých vrcholov by mal byť nepárny, čo nie je možné.

Máme teda nutnú podmienku pre existenciu riešenia: počet chlapcov v grafe musí byť párny.[5]

**Ako nájsť riešenie**

Predpokladajme, že chlapcov je párny počet. Najprv odstráňme z grafu všetky hrany, potom budeme možno niektoré z nich pridávať naspäť.

Každý vrchol má teraz stupeň 0. Podmienka pre dievčatá je splnená – všetky vrcholy majú párny stupeň (áno, 0 je párne číslo). Horšie je to s chlapcami. Ak však v grafe žiadni nie sú, potom sme vyhrali a máme riešenie.

Ak tam predsa len nejakí chlapci sú, tak sú tam aspoň dvaja. Vyberme ľubovoľných dvoch, označme ich $u$ (Urban) a $v$ (Vlado). Medzi $u$ a $v$ existovala v pôvodnom grafe cesta (bol predsa súvislý). Vezmime všetky hrany na nejakej $uv$ ceste a zmeňme ich „stav“: ak sa hrana v aktuálnom riešení nenachádza, pridáme ju; inak ju vyhodíme.

Týmto sme pomenili stupne vrcholom na $uv$ ceste. Vo všeobecnosti nevieme povedať, ako sa zmenili jednotlivé čísla – našťastie to ani nepotrebujeme. Dôležitá je iba parita stupňa a tá zostala pre každý vnútorný vrchol cesty[6] zachovaná. Overiť sa to dá rozobratím všetkých prípadov alebo počítaním modulo 2: do vnútorného vrcholu cesty vedú dve hrany, ktorým bol zmenený stav; každá zmena stavu (pridanie alebo vyhodenie hrany) zmení paritu stupňa, preto sa dve zmeny na parite neprejavia.

Vrcholy $u$ a $v$ sú ale krajné, do každého vedie len jedna hrana z cesty. Parita ich stupňa sa preto určite zmení – čo je dobré, keďže pôvodne mali párny stupeň. Takto máme o dvoch nespokojných chlapcov menej.

Ak ešte zostali nejakí chlapci s párnym stupňom, celý postup pre nich zopakujeme. Všimnite si, že ostatným vrcholom paritu nemeníme. V každom kroku znížime počet nespokojných chlapcov o 2, takže nakoniec sa dopracujeme k riešeniu (pri nepárnom počte chlapcov by to zjavne nefungovalo).

**Vzorové riešenie**

Priamočiara implementácia riešenia z predchádzajúcej časti by mala časovú zložitosť $O(N \cdot (N + M))$ – chlapcov môže byť $O(N)$ a pre jednu dvojicu pri hľadaní cesty prejdeme celý graf.

Všimnime si, že vysoký počet hrán v grafe nie je veľmi užitočný, skôr naopak. Nás postup totiž vyžaduje iba to, aby bol graf súvislý, teda by fungoval aj na strome. Na druhej strane, veľa hrán predlžuje hľadanie ciest.

Tu by sme mohli na ceste k vzorovému riešeniu odbočiť a zrýchliť algoritmus, ktorý už máme: stačí ho spustiť iba na kostre[7] daného grafu. Časová zložitosť sa tak zlepší na

[5] V prípade nesúvislého grafu to znamená, že aj v každom komponente musí byť počet chlapcov párny.

[6] Okrem krajných vrcholov $u$ a $v$ sú všetky vrcholy cesty vnútorné.

[7] Kostra grafu je najmenší súvislý podgraf obsahujúci všetky vrcholy – vždy je to strom.



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

$O(M + N^2)$, čo možno niekoho poteší, ale nám to stačiť nebude. Zaujímavou časťou je však nájdenie nejakej kostry grafu.

Pretože nám stačí skutočne ľubovoľná kostra, riešenie je jednoduché: použijeme prehľadávanie do hĺbky. Do kostry budú patriť práve tie hrany, ktorými sme sa presúvali do nenavštíveného vrcholu.

Takže sme sa už zbavili zbytočných hrán a pôvodnú úlohu máme splniť na strome. Označme jeden z jeho vrcholov ako koreň a spustime z neho ďalšie prehľadávanie do hĺbky, ktoré bude vracať *true*, ak je v prehľadanom podstrome nepárny počet chlapcov, inak vráti *false*.

Okrem toho budeme počas prehľadávania vyberať hrany do výsledného grafu tak, že keď prehľadáme nejaký podstrom, všetky vrcholy v ňom okrem jeho koreňa budú mať správnu paritu stupňa. Ak je v podstrome párny počet chlapcov, aj koreň bude mať správnu paritu stupňa, v opačnom prípade nie (vtedy to ani nie je možné).

Ak sme v liste, vrátime *false* alebo *true* podľa typu vrcholu (dievča alebo chlapec). Ak sme vo vrchole $v$, ktorý má nejakých potomkov $u_1, \ldots, u_k$, spúšťame z nich rekurzívne prehľadávanie.

Rozoberme teraz dva prípady: Ak prehľadávanie z potomka $u$ vráti *false*, celý jeho podstrom vrátane $u$ spĺňa požadované parity stupňov vrcholov. Ak vráti *true*, potom pridáme do výsledného grafu hranu $vu$ a už aj vrchol $u$ bude mať správnu paritu stupňa.

Keď toto vykonáme pre všetkých potomkov vrcholu $v$, celý podstrom okrem $v$ už bude mať správne parity stupňov. Aby sme vedeli, akú paritu stupňa má teraz vrchol $v$, musíme zistiť, koľko hrán vchádzajúcich do $v$ sme pridali do výsledného grafu. Toto číslo bude mať určite rovnakú paritu ako počet chlapcov v podstromoch $u_1, \ldots, u_k$ (lebo pridáme jednu hranu za každý podstrom s nepárnym počtom chlapcov).

Zostáva doriešiť, čo má prehľadávanie z vrcholu $v$ vrátiť. Sú štyri možnosti:

| vrchol $v$ | počet chlapcov v podstromoch $u_1, \ldots, u_k$ | návratová hodnota |
|---|---|---|
| dievča | párny | *false* |
| chlapec | nepárny | *false* |
| dievča | nepárny | *true* |
| chlapec | párny | *true* |

Všimnite si, *true* vrátime práve vtedy, keď je počet chlapcov v podstrome $v$ nepárny, a tiež práve vtedy, keď vrchol $v$ nemá správnu paritu stupňa.

Vo vzorovom programe sa všetky fázy (overenie nutnej podmienky, hľadanie kostry aj generovanie riešenia) vykonávajú počas jediného prehľadávania do hĺbky. Časová aj pamäťová zložitosť je $O(N + M)$ – stačí nám pamätať si graf a raz ho prejsť.

Tak, sme na konci. Užite si vzorový kód a nezabúdajte na správny pitný režim.

## Listing programu:

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

typedef pair<int, int> PI; // (druhý koniec, číslo) hrany

vector<vector<PI> > G; // daný graf
vector<bool> V; // bol vrchol navštívený?
vector<bool> E; // je hrana v riešení?
string P; // požadovaná parita vrcholov
```



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

```
bool DFS(int x){
    V[x] = true;
    bool res = (P[x] == 'C');

    for(vector<PI>::iterator i = G[x].begin(); i != G[x].end(); ++i)
        if(!V[i->first] && DFS(i->first)){
            E[i->second] = true;
            res = !res;
        }

    return res;
}

int main(){
    int N, M;
    cin >> N >> M;
    G.resize(N);
    for(int i = 0; i < M; ++i){
        int a, b;
        cin >> a >> b;
        --a, --b;
        G[a].push_back(PI(b, i));
        G[b].push_back(PI(a, i));
    }
    cin >> P;

    bool dasa = true;
    V.resize(N, false);
    E.resize(M, false);

    for(int i = 0; i < N; ++i)
        if(!V[i] && DFS(i))
            dasa = false; // počet chlapcov v komponente má byť párny

    if(!dasa)
        cout << "nemozne" << endl;
    else{
        bool prvy = true;
        for(int i = 0; i < M; ++i)
            if(E[i]){
                if(prvy)
                    prvy = false;
                else
                    cout << ' ';
                cout << i + 1;
            }
        cout << endl;
    }
}
```

opravoval Mic
(max. 25 bodov)

# 8. Temné častice

Riešení neprišlo veľa a čo je horšie, žiadne z nich nebolo vzorové, čo znamená v čase $O(n \log n)$. Prišlo však niekoľko správnych riešení s časovou zložitosťou $O(n^2)$. Najskôr si ukážeme toto riešenie a potom si povieme, ako ho vylepšiť na vzorové riešenie.



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

Zrekapitulujme si zadanie. Úlohou bolo nájsť k $n$ priamkam najmenší taký obdĺžnik, že obsahuje všetky priesečníky všetkých priamok.

**Pomalé riešenie**

Zoberme si najvyšší priesečník. Keďže tento priesečník musí byť vo vnútri obdĺžnika, horná strana obdĺžnika musí byť nad ním alebo ním prechádzať. Ak je však nad ním, vieme spraviť menší obdĺžnik tak, že hornú stranu obdĺžnika posunieme nižšie, aby prechádzala najvyšším priesečníkom. Toto isté platí pre najnižší priesečník a spodnú hranu a podobne aj pre najľavejší a najpravejší priesečník. Preto nám tieto 4 priesečníky jednoznačne určujú hľadaný obdĺžnik.

Ako nájsť tieto priesečníky? Môžeme vyskúšať všetky dvojice priamok, nájsť ich priesečníky a vybrať z nich najvyšší, najnižší, najpravejší a najľavejší priesečník. Toto vieme spraviť napríklad dvoma vnorenými for-cyklami, čo má časovú zložitosť $O(n^2)$.

**Rýchle riešenie**

Ukážeme si, ako nájsť tie 4 dôležité priesečníky v čase $O(n \log n)$. Využijeme fakt[8], že ak si zoradíme všetky priamky podľa uhla (berieme uhly od 0 po 180 stupňov), ktorý zvierajú s osou $x$, tak najvyšší z priesečníkov je priesečníkom dvoch susedných priamok. Preto nám stačí usporiadať priamky podľa uhla, ktorý zvierajú s osou $x$, vyrobiť priesečníky zo susedných dvojíc a vybrať z nich ten najvrchnejší.

Analogický postup sa dá použiť pre ostatné priesečníky, ibaže všetko budeme robiť otočené o 90, 180, 270 stupňov. Ak sa nad tým hlbšie zamyslíme, tak si môžeme všimnúť, že štyrikrát triedime, ale poradie priamok bude vždy veľmi podobné. Ak totiž otočíme priamky o 90 stupňov a utriedime, dostaneme poradie, ktoré je len cyklicky posunuté oproti pôvodnému poradiu. Preto budeme triediť iba raz a zo zoznamu priamok si spravíme cyklický zoznam (za posledným prvkom ide prvý prvok). Potom stačí pozrieť priesečníky všetkých susedov v zozname a nájsť medzi nimi najvyšší, najnižší, ...

Utriedenie priamok vieme spraviť v čase $O(n \log n)$ a nájdenie priesečníkov vieme spraviť v čase $O(n)$, preto je časová zložitosť celého algoritmu $O(n \log n)$.

Ostáva nám vyriešiť dva detaily. Ako nájdeme priesečník dvoch priamok a ešte si musíme dokázať, že platí fakt, ktorý sme použili na konštrukciu rýchlejšieho riešenia. Začneme dôkazom.

**Dôkaz**

Chceme ukázať, že tie 4 dôležité priesečníky sú niektoré z priesečníkov priamok, ktoré sú vedľa seba po utriedení podľa uhla, ktorý zvierajú s osou $x$. Tvrdenie ukážeme len pre najvrchnejší priesečník, pre ostatné priesečníky je dôkaz analogický.[9]

Tvrdenie dokážeme sporom. Nech $A = (x, y)$ je najvrchnejší priesečník, to znamená taký, že má maximálne $y$ zo všetkých priesečníkov (ale nevylučujeme, že neexistuje iný priesečník s rovnakou „výškou"). Nech $A$ je priesečníkom priamok $p$ a $q$, pričom $p$ nech je v usporiadaní skôr ako $q$ a priesečníky všetkých priamok medzi $p$ a $q$ s priamkou $p$ sú rôzne od $A$. Nech tieto priamky nie sú susedné v usporiadaní a teda existuje priamka $r$, ktorá je v usporiadaní medzi priamkami $p$ a $q$. Pozrime sa na jej priesečník s priamkou $p$. Označme tento priesečník $B = (x', y')$. Máme tri možnosti:

1. $y' > y$: Tento prípad nemôže nastať, lebo $A$ je najvyšší priesečník.
2. $y' = y$: Toto tiež nemôže nastať, lebo potom $A = B$ a teda medzi $p$ a $q$ existuje priamka, ktorá má priesečník s $p$ v bode $A$, čo je spor s predpokladom.
3. $y' < y$: Zoberme si teraz bod $C = (x'', y'')$, ktorý tvorí priesečník priamky $r$ a $q$. Ukážeme, že $y'' > y$ čím prídeme do sporu s tým, že $A$ je najvrchnejší priesečník. Ak by

[8] ... ktorý neskôr aj dokážeme.

[9] Akurát otočený o 90, 180 alebo 270 stupňov.



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

totiž $y'' < y$, tak potom priamka $r$ musí byť v usporiadaní pred priamkou $p$. Rovnako $y'' \neq y$, lebo potom by $r$ bola totožná s priamkou $p$. To znamená, že $y'' > y$.

Ako vidíme, každá situácia nás priviedla do sporu a preto v usporiadaní medzi $p$ a $q$ nesmie byť žiadna priamka.

**Hľadanie prieniku dvoch priamok**

Ostáva nám vyriešiť problém hľadania prieniku dvoch priamok. Každú priamku zo vstupu budeme reprezentovať rovnicou $ax + by = c$. Priamku budú tvoriť všetky také body $(x, y)$, ktoré spĺňajú danú rovnosť.

Na vstupe dostávame priamky zadané dvojicou bodov. Našu reprezentáciu priamky vieme jednoducho vytvoriť. Majme dva body $(x_1, y_1)$, $(x_2, y_2)$, ktoré určujú priamku $ax+by = c$. Chceme určiť koeficienty $a, b, c$. Dosadíme oba body do rovnice priamky a dostaneme dve rovnice o troch neznámych: $ax_1 + by_1 = c$ a $ax_2 + by_2 = c$. Táto sústava rovníc má veľa riešení, jedno z nich je napríklad $a = y_1 - y_2$, $b = x_2 - x_1$, $c = x_2y_1 - x_1y_2$. Existuje veľa iných, toto má výhodu v tom, že neobsahuje žiadne delenie, a preto ak máme celočíselné súradnice, aj koeficienty $a, b, c$ budú celočíselné.

Na hľadanie prieniku dvoch priamok použijeme podobný postup. Nech $a_1x + b_1y = c_1$ a $a_2x + b_2y = c_2$ sú rovnice určujúce dve priamky. Ak nájdeme riešenie tejto sústavy rovníc, nájdeme spoločný bod týchto priamok. Keďže máme zaručené, že žiadne dve priamky nie sú rovnobežné, táto sústava rovníc má vždy práve jedno riešenie. Toto riešenie je prienikom daných dvoch priamok.

Existuje niekoľko spôsobov, ako vyriešiť tieto nerovnice. Ak si to vyriešime symbolicky, môžeme sa dostať k nasledovnému riešeniu: $x = \frac{c_1b_2-c_2b_1}{a_1b_2-a_2b_1}$ a $y = \frac{a_1c_2-a_2c_1}{a_1b_2-a_2b_1}$. Vieme to napríklad dostať tak, že sčítame obe rovnice a vyjadríme z výslednej rovnice $y$. Potom odčítame obe pôvodné rovnice a vyjadríme z toho $y$, dáme to do rovnosti (lebo $y = y$) a vyjadríme z toho $x$. $y$ dostaneme analogicky. Všimnite si výraz v menovateli. Ten bude nulový práve vtedy, ak sú priamky rovnobežné a teda my sa delenia nulou báť nemusíme (lebo vieme, že nie sú rovnobežné).

**Bodovanie**

Za vzorové riešenie by som dal 25 bodov, keby nejaké prišlo. Za kvadratické riešenie som dával 15 bodov, ak mu nič nechýbalo. V opačnom prípade som strhal body. Za chýbajúce odhady časovej zložitosti som strhal 1 bod, za zlý popis som strhal dva body (ale zlý popis je lepší ako žiadny, tam by som delil dvoma). Za chýbajúci zdrojový kód som strhal polovicu bodov.

## Listing programu:

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

typedef pair<double, double> Bod;

struct Priamka{
    double a, b, c;
    Priamka(){};
    Priamka(Bod A, Bod B){
        //Vypocitame parametre priamky
        a = A.second - B.second;
        b = B.first - A.first;
        c = A.second * B.first - A.first * B.second;
    }
```



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

```
    bool operator < (const Priamka &p) const{
        //Porovnavaci operator, porovnavame podla "uhla"
        if(this->b == 0) return !(this->b == p.b);
        if(p.b == 0) return this->b == p.b;
        return (-this->a / this->b) < (-p.a / p.b);
    }

    Bod intersect(Priamka &p){
        //Vypocitame prienik
        return make_pair(
            (this->c * p.b - p.c * this->b) / (this->a * p.b - p.a * this->b),
            (this->a * p.c - p.a * this->c) / (this->a * p.b - p.a * this->b));
    }
};

int main(){
    vector<Priamka> A;

    int N;
    cin >> N;
    for(int i = 0; i < N; i++){
        Bod b1, b2;
        cin >> b1.first >> b1.second >> b2.first >> b2.second;
        A.push_back(Priamka(b1, b2));
    }

    sort(A.begin(), A.end());

    Bod UL = A[0].intersect(A[1]), BR = A[0].intersect(A[1]);

    //Pridame zaciatok na koniec, nech mame cyklicky zoznam
    A.push_back(A[0]);

    for(unsigned int i = 0; i < A.size() - 1; i++){
        Bod B = A[i].intersect(A[i + 1]);
        UL.first = min(UL.first, B.first);
        BR.first = max(BR.first, B.first);
        UL.second = min(UL.second, B.second);
        BR.second = max(BR.second, B.second);
    }
    cout << UL.first << " " << UL.second << " " << BR.first << " " << BR.second <<
endl;
}
```

opravoval U$Amec
(max. 25 bodov)

# 9. Tancujúce víly

Vítam vás v receptári najlepšieho programátora. Niekedy pred mesiacom mi víly poskytli krásnu príležitosť, ako sa dobre zabaviť. Aby ste si niekedy v budúcnosti mohli užiť aj vy, ukážem vám, ako na to.

**Koľko bude trvať tanec**

Máme zadanú permutáciu (to, kam sa ktorá víla presunie). Ako zadanie napovedá, permutáciu treba rozbiť na cykly. Napríklad keď víla číslo 5 prejde počas tanca políčka $3, 2, 7, 5$, tak víly $2, 3, 7$ tiež prejdú len tieto políčka (len v inom poradí) a preto $5 \to 3 \to 2 \to 7 \to 5$



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

tvorí cyklus dĺžky 4. Zároveň do tohoto cyklu sa žiadna iná víla nepripletie (lebo by musela smerovať na jedno z políčok $2, 3, 5, 7$ a tam už smeruje niekto iný).

Takto rozložíme celú permutáciu. Označme dĺžky cyklov $c_1, c_2, \ldots, c_m$. Zjavne po $c_i$ krokoch je $i$-ty cyklus na svojom pôvodnom mieste. Takže keď $k$ je počet krokov, keď je všetko na svojom mieste, tak musí byť deliteľné $c_1, c_2, \ldots, c_m$. A najmenšie také $k$ je $\mathrm{lcm}(c_1, c_2, \ldots, c_m)$ (lcm označuje najmenší spoločný násobok).

**Triviálne riešenie**

Môžeme si všimúť, že keď si definujeme dĺžky cyklov, tak vieme podľa toho zostrojiť permutáciu. Prvý cyklus spravíme medzi vílami $1, 2, \ldots, c_1$; druhý cyklus medzi $c_1 + 1, c_1 + 2, \ldots, c_1 + c_2$; ... Keďže máme zadanú dĺžku permutácie, tak chceme, aby súčet dĺžok cyklov bola dĺžka permutácie: $c_1 + c_2 + \ldots + c_m = N$. Takže môžeme vyskúšať všetky možné rozdelenia $N$ na súčet čísel a pre každe spočítať, koľko takýto tanec bude trvať. Vylepšenie je skúšať súčet rôznych čísel (zjavne sa nám dva rovnako dlhé cykly neoplatia). Toto je stále exponenciálne riešenie, keďže približný počet rozložení na rôzne časti je $O\left(\frac{e^{\pi\sqrt{n/3}}}{n^{3/4}}\right)$[10].

**Vzorové riešenie**

Poďme predchádzajúce riešenie vylepšiť o niekoľko užitočných pozorovaní.

Pozorovanie prvé: **Viac víl, viac zábavy**. V preklade: keď zvýšime počet víl z $k$ na $k + 1$, dĺžka tanca pre $k + 1$ víl bude aspoň toľko ako pre $k$. Dôkaz: Keď máme $k + 1$ víl, môžeme nechať poslednú vílu stáť na mieste a máme tanec ako pre $k$ víl.

Pozorovanie druhé: Nič nepokazíme, ak najväčší spoločný deliteľ dĺžok cyklov bude 1. Dôkaz: Nech $c_i$, $c_j$ majú najväčší spoločný deliteľ $d$ a ich najmenší spoločný násobok je $k$, potom aj $c_i$, $c_j/d$ majú najmenší spoločný násobok $k$ a zvyšné víly, čo nič nerobia, môžeme využiť iných (alebo ich nechať stáť).

Pozorovanie tretie: Nič nepokazíme, ak dĺžka cyklu bude mocnina prvočísla. Dôkaz: Nech $c_i = p_1^{a_1} p_2^{a_2} \ldots p_r^{a_r}$. Tento cyklus zmeníme na cykly s dĺžkami $p_1^{a_1}, p_2^{a_2}, \ldots, p_r^{a_r}$. Zjavne ich najmenší spoločný násobok bude $c_i$ a súčet ich dĺžok bude menší ako $c_i$.

Zhrňme si to: Optimálne riešenie sa dá vyskladať z cyklov, ktorých dĺžka je mocnicna prvočísla, a niekoľkých cyklov dĺžky 1; ich dĺžky sú pritom navzájom nesúdeliteľné. Dôkaz: Zoberieme iné optimálne riešenie a prerobíme ho na naše. Ak dĺžky niektorých cyklov nie sú mocniny prvočísel, upravíme ich podľa tretieho pozorovania. Ak nám ostali nejaké cykly, ktorých dĺžka je súdeliteľná, upravíme ich podľa druhého pozorovania. Takto máme riešenie, ktoré je rovnako dobré a spĺňa zopár podmienok navyše.

Takto sme si trochu obmedzili prehľadávaný priestor. Zvyšok je už len dynamické programovanie.

Označme $f(x, y)$ najväčší rád (počet otočení víl) permutácie takej, že má $x$ prkov a jej cykly majú dĺžky, ktoré sú mocninami prvých $y$ prvočísel. Zjavne $f(x, 0) = 1$ pre ľubovoľné $x \geq 0$. Ešte treba spôsob, ako vypočítať $f(x, y)$ pre ľubovoľné $x$, $y$.

Venujme časť našej permutácie cyklu dĺžky $p_y^i$ (kde $p_y$ je $y$-te prvočíslo). Zvyšok permutácie má dĺžku $x - p_y^i$ a už v ňom nepotrebujeme cyklus, ktorého dĺžka je mocninou $p_y$. A tam je najlepšie riešenie $f(x - p_y^i, y - 1)$. A teda náš rád bude $p_y^i \cdot f(x - p_y^i, y - 1)$ (môžeme násobiť, keďže to bude nesúdeliteľné). Takže na zistenie $f(x, y)$ stačí zistiť tento výsledok pre všetky rozumné $i$ (čiže aby $p_y^i \leq x$) a vybrať maximum.

Toto vieme robiť v časovej zložitosti $O(NQ)$, kde $Q$ je počet mocnín prvočísel menších ako $N$. Tu máme ale tichý predpoklad, že vieme násobiť ľubovoľne veľké čísla v konštantnom čase. Číslo $Q$ môžeme zhora odhadnúť ako $N$. Presnejší odhad by bol napríklad $O(N/\log N)$.

[10] http://mathworld.wolfram.com/PartitionFunctionQ.html



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

Pamäťovú zložitosť vieme zredukovať na $O(N)$, keďže si stačí pamätať len dva „riadky“, teda čísla $f(0,y)$, $f(1,y)$, ..., $f(N,y)$ a $f(0,y+1)$, $f(1,y+1)$, ..., $f(N,y+1)$; alebo dokonca ak patričný riadok vylepšujeme od konca, tak si stačí pamätať iba jeden riadok.

Na to, aby sme potom zrekonštruovali dĺžky cyklov, nám stačí už len vyfaktorizovať číslo $f(N,p_x)$.

## Listing programu:

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

vector<long long> pc;

void gen(long long N){
    vector<bool> p(N + 10, true);
    for(int i = 2; i < N + 10; i++)
        if(p[i]){
            pc.push_back(i);
            for(int j = 2 * i; j < N + 10; j += i)
                p[j] = false;
        }
}

int main(){
    long long N;
    scanf("%Ld", &N);
    gen(N);
    vector<long long> sol(N + 1, 1);
    for(int i = 0; i < pc.size(); i++)
        for(long long j = N; j >= 0; j--){
            long long moc = pc[i];
            while(moc <= j){
                sol[j] = max(sol[j], sol[j - moc] * moc);
                moc *= pc[i];
            }
        }

    long long sum = 0;
    for(int i = 0; i < pc.size(); i++){
        long long f = 1;
        while(sol[N] % pc[i] == 0){
            sol[N] /= pc[i];
            f *= pc[i];
        }
        if(f > 1){
            for(int j = 0; j < f - 1; j++)
                printf("%Ld ", sum + j + 2);
            printf("%Ld ", sum + 1);
            sum += f;
        }
    }
    while(sum < N)
        printf("%Ld ", ++sum);
    printf("\n");
}
```



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

# 10. Teroristické bunky

opravoval Zemčo
(max. 25 bodov)

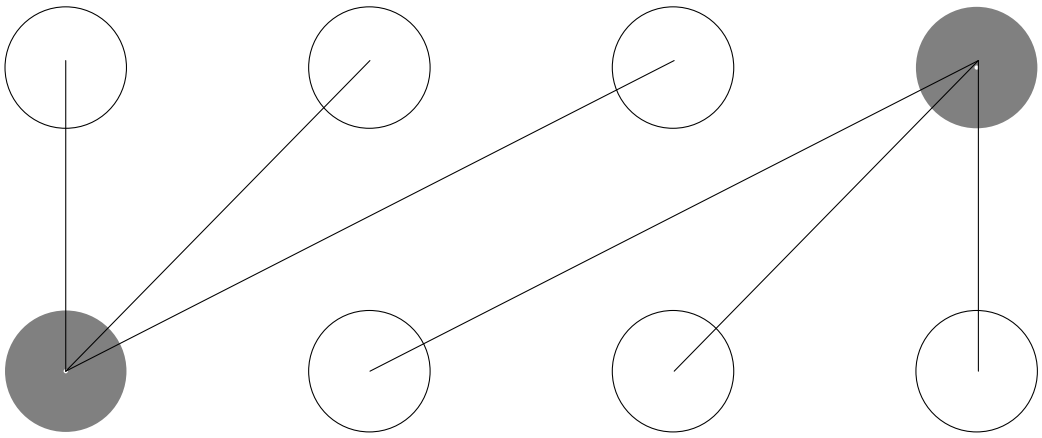
Riešenie tohto príkladu vyžaduje celkom pokročilé teoretické vedomosti a preto nebol ľahký. Napriek tomu je škoda, že ste ho nevyriešili viacerí. Prišli len dve riešenia a obe získali plný počet bodov. Každý, kto to myslí so súťažami v programovaní vážne a túto úlohu riešiť nevedel, by si mal prečítať tento vzorák, pretože podobné úlohy sa čas od času vyskytujú napríklad na TopCoderi, vysokoškolských programátorských súťažiach a všade možne inde.

Našou úlohou je teda odstrániť z grafu čo najmenej vrcholov tak, aby z $A$ do $B$ neexistovala cesta dĺžky jeden, dva ani tri. Cesta dĺžky jeden neexistuje podľa predpokladu zo zadania o nesusednosti $A$ a $B$. Cesta dĺžky dva vedie cez spoločného suseda vrcholov $A$ a $B$. Je teda zjavné, že tento vrchol musíme určite odstrániť. V ďalšom teda budeme predpokladať, že spoloční susedia sa v grafe už nenachádzajú. A ako je to s cestami dĺžky tri?

Všimnime si, že cesta dĺžky tri vychádza z vrcholu $A$, smeruje do nejakého jeho suseda, odtiaľ do nejakého suseda vrcholu $B$ a odtiaľ do $B$. Túto cestu musíme zrušiť a môžeme to spraviť len tak, že odstránime suseda vrcholu $A$ alebo suseda vrcholu $B$. Odstránenie iného vrcholu je zbytočné a preto pre nás tieto vrcholy nie sú zaujímavé.

Pripomeňme si, že graf je bipartitný, ak sa jeho vrcholy dajú ofarbiť na červené a čierne tak, aby každá hrana grafu spájala čierny a červený vrchol. Dve časti grafu sa učene volajú partície. Odmyslime si na chvíľu vrcholy, ktoré nie sú susedmi $A$ ani $B$ a odmyslime si aj vrcholy $A$ a $B$. Dostávame bipartitný graf, ktorého jedna partícia sú susedia $A$ a druhá susedia $B$. Graf je bipartitný, pretože $A$ a $B$ nemajú spoločných susedov. Chceme vyznačiť niektoré vrcholy, ktoré potom odstránime. Musíme ich vyznačiť tak, aby každá hrana mala aspoň jeden vrchol vyznačený, pretože ak by nejaká nemala, nezničili by sme cestu dĺžky tri v pôvodnom grafe, ktorá prechádza týmito dvoma vrcholmi. Okrem toho ale chceme, aby bol počet vyznačených vrcholov čo najmenší.

Tejto úlohe sa hovorí minimálne vrcholové pokrytie grafu.[11] Vyznačovaním vrcholov pokrývame hrany, až kým ich nepokryjeme všetky. Vo všeobecných grafoch je tento problém NP-úplný, čo prakticky znamená, že nepoznáme polynomiálny algoritmus na jeho riešenie. My ale pracujeme s bipartitným grafom, pre ktorý je tento problém ľahší. V odstavcoch nižšie môžete nájsť algoritmus pracujúci v čase $O(NM)$. Riešením je teda nájsť toto vrcholové pokrytie a vypísať vyznačené vrcholy. Keďže všetky ostatné fázy pohodlne stíhame v čase $O(N^2)$, potom bude celkový čas riešenia našej úlohy $O(NM)$, za zjednodušujúceho predpokladu, že hrán je približne aspoň tak veľa ako vrcholov. Analýza by sa dala spraviť aj kvalitnejšia a do odhadu zložitosti by sme dostali počty susedov $A$ a $B$. Každopádne, obeť, ktorá si bude čítať a chápať tento vzorák, má už aj tak dosť krutý osud...



[11] Po anglicky Minimum vertex cover problem.



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

Príklad bipartitného grafu s vyznačeným minimálnym vrcholovým pokrytím. Vždy môžeme dosiahnúť vrcholové pokrytie vybraním celej jednej partície. Tento príklad ukazuje, že to nie je vždy optimálne riešenie.

Poznámka na záver: s vrcholovým pokrytím úzko súvisí úloha maximálnej nezávislej množiny. Máme daný graf, v ktorom chceme vyznačiť niektoré vrcholy tak, aby pre každú hranu platilo, že je vyznačený najviac jeden jej vrchol. Chceme maximalizovať počet vyznačených vrcholov. Pre bipartitné grafy si môžeme predstaviť nasledovnú úlohu z praxe: ste učiteľ a máte na starosti stredoškolskú triedu s chlapcami a dievčatami. Chceli by ste zorganizovať stanovačku, ale neradi by ste, aby sa tam diali nekalé veci. Máte daný zoznam dvojíc, pre ktoré hrozí, že by si mohli dať pusu. Chcete zobrať na stanovačku čo najviac študentov, ale z každej rizikovej dvojice najviac jedného. Premyslite si, že ak nájdete najmenšie vrcholové pokrytie, tak ste vlastne našli aj najväčšiu nezávislú množinu.[12]

**Hľadanie najväčšieho párenia v bipartitnom grafe**

V tejto časti spomenieme na prvý pohľad nesúvisiacu úlohu o párení v bipartitnom grafe. Uvedieme si tiež jednoduchý algoritmus, ktorý tento problém rieši. Tí, ktorí túto úlohu a tento algoritmus na jej riešenie poznajú, môžu preskočiť na nasledujúcu časť. Existujú samozrejme aj lepšie algoritmy na hľadanie najväčšieho párenia, avšak tie sú príliš komplikované a algoritmus zlepšujúcich ciest, ktorý nájdete tu, prakticky na súťaže v programovaní postačuje. Ešte dodajme, že správne slovenské slovo pre túto úlohu je naozaj párenie. Párovanie je čechizmus.

Takže zavrime oči[13] a predstavme si, že sme učitelia v tanečnej škole. Na hodinu prišli nejaké dievčatá a nejakí chlapci a keďže si zaplatili, tak by sa možno chceli naučiť aj nejaký ten tanec. Každý z nich je ale fajnový a nechce tancovať len tak s hocikým. Každý chlapec chce tancovať iba s niektorými dievčatami a každé dievča je schopné tancovať iba s niektorými chlapcami. Navyše platí, že ak chlapec $CH$ chce tancovať s dievčaťom $D$, tak aj dievča $D$ chce tancovať s chlapcom $CH$.

Situáciu si môžeme reprezentovať grafom - vrcholy budú chlapci a dievčatá. Hrana medzi dvoma ľuďmi pôjde vtedy, ak chcú spolu tancovať. Keďže príslušníci rovnakého pohlavia spolu tancovať nechcú nikdy, potom každá hrana vedie medzi chlapcom a dievčaťom a preto je tento graf bipartitný. Chceme urobiť z chlapcov a dievčat páry na parkete a popri rešpektovaní ich požiadaviek chceme, aby ich tancovalo čo najviac. Každému chlapcovi chceme priradiť najviac jedno dievča a každému dievčaťu chceme priradiť najviac jedného chlapca. Inými slovami: chceme vybrať niektoré hrany grafu tak, aby mal každý vrchol spomedzi svojich hrán vybratú najviac jednu. Tejto úlohe sa hovorí hľadanie párenia v grafe.

Máme sformulovanú úlohu, poďme ju skúsiť riešiť. Najprv si treba uvedomiť, že to vôbec nie je také triviálne, ako by sa mohlo zdať a všelijaké hlúpe greedy algoritmy môžu ľahko skončiť pochované nejakým kontrapríkladom. Správny algoritmus ale napokon príliš zložitý nebude.

Základná myšlienka je nasledovná. Predstavme si, že sme už našli nejakých $k$ hrán, tieto budeme volať párovacie. Radi by sme buď našli ďalšiu, alebo vyhlásili, že sa to nedá. Na to si ale musíme zaviesť nový pojem – zlepšujúca cesta. Je to cesta, ktorá začína vo vrchole, skladá sa striedavo z nepárovacích a párovacích hrán, pričom nepárovacou hranou začína aj končí. Ľahko vidieť, že ak takúto cestu v grafe nájdeme, tak nám stačí vymeniť jej párovacie hrany za nepárovacie a naopak – a keďže nepárovacích bolo o jednu viac, tak sme práve našli párovanie veľkosti $k+1$. Premyslite si, že táto zmena je naozaj uskutočniteľná a po nej bude naďalej platiť, že každý vrchol je spárený s najviac jedným iným.

[12] Riešenie: doplnok k nezávislej množine je vrcholové pokrytie. Doplnok k najväčšej nezávislej množine je minimálne vrcholové pokrytie.

[13] a čítajme pritom ďalej vzorák!



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

Zostáva ukázať, že ak už zlepšujúcu cestu nenájdeme, je aktuálne párenie najväčšie možné. V prvom rade si rozmyslite, že toto nie je zjavné! Vieme totiž zatiaľ len to, že naším postupom toto naše párenie nevieme zlepšiť. Nevieme nič o tom, či sa iným, lepším spôsobom nedá vyrobiť lepšie párenie.

Nuž, nedá. Zoberme si dve párenia grafu, z ktorých jedno je maximálne a druhé je od neho menšie. Pozrime sa na podgraf tvorený hranami, ktoré sú v práve jednom z týchto párení. V tomto podgrafe má každý vrchol stupeň najviac dva, pretože každá hrana v tomto podgrafe je v práve jednom z dvoch párení. Takže komponenty tohto podgrafu sú len kružnice párnej dĺžky (prečo párnej?) a cesty. Fakt, že hrán z lepšieho párenia je aspoň o jednu viac, nás privádza k záveru, že aspoň jeden z komponentov tohto podgrafu zodpovedá zlepšujúcej ceste pre menšie párenie. Z toho vyplýva, že z ľubovoľného párenia viem pomocou zlepšujúcich ciest dostať maximálne – a teda ak už zlepšujúca cesta neexistuje, párenie je nutne najväčšie možné.

Poďme si teraz presnejšie popísať náš algoritmus. Použijeme upravené prehľadávanie do hĺbky, pretože pri návrate z rekurzie môžeme pohodlne prehodiť párovacie hrany za nepárovacie a naopak. Navyše budeme využívať fakt, že ak sa nám niekedy počas vykonávania algoritmu z nejakého vrcholu nepodarí nájsť zlepšujúca cesta, potom sa nám to už nemôže podariť ani nikdy potom. Okrem toho platí, že ak z nejakého vrcholu zlepšujúcu cestu nájdeme, potom bude tento vrchol od teraz až do konca behu algoritmu spárený. Kto posledným dvom vetám neverí, nech si premyslí. Obe sa dajú odôvodniť pozorovaním, čo sa vlastne pri hľadaní zlepšujúcich ciest so spárenými a nespárenými vrcholmi deje. Vďaka týmto dvom vetám stačí z každého vrcholu skúsiť hľadať zlepšujúcu cestu len raz. Naše prehľadávanie bude navyše prechádzať pre jednoduchosť implementácie len po vrcholoch jednej partície.

Keď sme v nejakom vrchole $u$, pozrieme sa na všetky vrcholy, s ktorými susedí. Ak je niektorý z nich nespárený, spárime ho s $u$ a vrátime nejakú hodnotu (v našej implementácii `true`), ktorá bude signalizovať, že sa našla zlepšujúca cesta a máme poprehadzovať hrany. Ak vidíme suseda $w$, ktorý je spárený s vrcholom $v$ ($v \neq u$), tak sa zavoláme na rekurzívne na vrchol $v$. Ak nám tento vrchol vráti, že sa našla zlepšujúca cesta, tak $w$ spárime s $u$ (všimnite si, že týmto ho odpárime od $v$, čím efektívne prehadzujeme hrany) a tiež vrátime signál o nájdení zlepšujúcej cesty. Ak nám ale všetky susedné vrcholy vrátia, že sa cesta nenašla, tak vrátime aj my, že sa nenašla. Takto teda v čase $O(N+M)$ nájdeme zlepšujúcu cestu pre jeden vrchol. Celkovo prehľadávanie voláme $N$-krát a dokopy nám to dáva $O(N(N+M))$, alebo jednoduchšie $O(NM)$.

Pre úplnosť ešte poznamenajme, že aj pre nebipartitné grafy má úloha o párení zmysel a existuje polynomiálny algoritmus na jej riešenie. Je ale o dosť komplikovanejší ako ten, ktorý sme práve prezentovali.

**Minimálne vrcholové pokrytie v bipartitnom grafe**

Ako sme vysvetlili v prvom odstavci, potrebujeme hľadať najmenšie vrcholové pokrytie v bipartitnom grafe. Aby sme si s tým poradili, musíme si uviesť rozhodujúcu vetu, ktorá sa volá Konigova a vznikla dávno predtým, ako boli vyrobené prvé moderné počítače. Táto veta hovorí, že pre bipartitný graf je počet hrán v najväčšom párení rovný počtu vrcholov v minimálnom vrcholovom pokrytí. Táto veta je jednou z viacerých min-max viet v teórii grafov.

Takže vidíme, že párenie a vrcholové pokrytie v bipartitnom grafe spolu naozaj súvisia a že časť vyššie je naozaj k veci a nie je len výplodom pubertálneho humoru autora tohto vzoráku. My ale nechceme len počet vrcholov, ktoré treba odstrániť, ale aj nejakú konkrétnu množinu vrcholov. Na to použijeme nasledovný algoritmus.

Predstavme si, že sme našli najväčšie párenie. Teda máme bipartitný graf, v ktorom sú niektoré hrany vybrané do párenia. Označme si dve partície $R$ a $L$. Uvažujme množinu vrcholov $T$. Na začiatku do $T$ dajme vrcholy z $L$, ktoré sú nespárené. Okrem toho do $T$ pridajme vrcholy, do ktorých sa dá dostať z nespárených vrcholov z $L$ po takých cestách,



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

ktoré vedú smerom z $L$ do $R$ po nespárených hranách a smerom z $R$ do $L$ po spárených hranách. Vrcholové pokrytie dostaneme, keď zoberieme vrcholy z $L$, ktoré nepatria do $T$ a vrcholy z $R$, ktoré patria do $T$. Dobre požujte.

## Listing programu:

```
#include <iostream>
#include <vector>
#include <cstring>
#define FOR(i, N) for(int i = 0; i < (N); i++)
#define MAXN 1000
using namespace std;

//SUSA,SUSB: je vrchol sused A/B?,
int G[MAXN][MAXN], SUSA[MAXN], SUSB[MAXN];
int D[MAXN], SPARENE[MAXN], T[MAXN];
//graf vo forme zoznamu susedov, kvoli rychlemu DFS
vector<vector<int> > E;
int N, M, A, B, prvy;

//metoda hlada DFSkom zlepsujucu cestu v grafe
bool zlepsi(int v){
    D[v] = 1;
    //hladame vsetkych susedov, ktori su z druhej particie a este sme v nich neboli
    FOR(i, E[v].size())
        if(D[E[v][i]] == 0 && SUSB[E[v][i]] == 1){
            //je tento sused zatial nespareny? vyborne, skoncili sme
            if(SPARENE[E[v][i]] == -1){
                SPARENE[E[v][i]] = v;
                SPARENE[v] = E[v][i];
                return true;
            }
            //dokazeme cez tohto suseda najst dalej zlepsujucu cestu?
            if(zlepsi(E[v][i])){
                SPARENE[E[v][i]] = v;
                SPARENE[v] = E[v][i];
                return true;
            }
        }
    return false;
}

//metoda ma identifikovat mnozinu T, ktoru pouzijeme na najdenie vrcholoveho
pokrytia
void hladajt(int v){
    T[v] = 1;
    FOR(i, E[v].size())
        if(SUSB[E[v][i]] == 1 && T[E[v][i]] == 0 && SPARENE[v] != E[v][i]){
            T[E[v][i]] = 1;
            if(SPARENE[E[v][i]] != -1)
                hladajt(SPARENE[E[v][i]]);
        }
}

int main(){
    cin >> N >> M;
    E.resize(N, vector<int>(0));
    FOR(i, M){
        int x, y;
        cin >> x >> y;
```



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

```
        x--; y--;
        G[x][y] = 1;
        G[y][x] = 1;
        E[x].push_back(y);
        E[y].push_back(x);
    }
    prvy = 1;
    cin >> A >> B;
    A--; B--;
    //odstranime spolocnych susedov A a B a zapiseme si, kto susedi s A a B
    FOR(i, N){
        if(G[i][A] == 1 && G[i][B] == 1){
            if(prvy == 0){ cout << " "; }
            else{ prvy = 0; }
            cout << (i + 1);
            FOR(j, N){
                G[i][j] = 0;
                G[j][i] = 0;
            }
            continue;
        }
        if(G[i][A] == 1) SUSA[i] = 1;
        if(G[i][B] == 1) SUSB[i] = 1;
    }
    //ideme hladat najvacsie parenie
    memset(SPARENE, -1, sizeof(SPARENE));
    FOR(i, N)
        if(SUSA[i] == 1){
            memset(D, 0, sizeof(D));
            zlepsi(i);
        }
    //skonstruujeme mnozinu T
    FOR(i, N)
        if(SUSA[i] == 1 && SPARENE[i] == -1 && T[i] == 0)
            hladajt(i);
    //identifikujeme najmensie vrcholove porkytie - vrcholy, ktore treba odstranit
    FOR(i, N)
        if((SUSA[i] == 1 && T[i] == 0) || (SUSB[i] == 1 && T[i] == 1)){
            if(prvy == 0){ cout << " "; }
            else{ prvy = 0; }
            cout << (i + 1);
        }
    if(prvy == 1) cout << "Ziadne vrcholy netreba odstranit!";
    cout << endl;
}
```



Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09