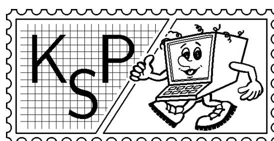


Korešpondenčný seminár z programovania
XXV. ročník, 2007/08
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

KSP finančne podporujú: MICROSTEP-MIS spol. s r.o.
Whitestein Technologies spol. s r.o.

Vzorové riešenia 2. kola zimnej časti

Milé naše riešiteľky, milí naši riešitelia, milé naše riešiteľatá,

zase raz sa dala mohutná mašinéria zvaná „banda lenivých vedúcich“ do pohybu a priniesla vám úplne nové vzoráky. Sú to dobré vzoráky... plné emócií, nádejí, správnych riešení, tragédií i splnených želaní. Veríme, že Vás potešia aspoň tak, ako nás ich úspešná kolaudácia.

Nečítajte túto vetu![1]

KSPáci

1. Zradca?

opravoval Maty
(max. 15 bodov)

Toto bol jeden z najľahších geometrických príkladov, ale mnohým sa v ňom podarilo narobiť chyby. Za lineárne riešenie s konštantnou pamäťou ste mohli dostať 15 bodov. Za lineárnu pamäť, chýbajúce odhady zložitosti, prípadne neošetrenie niektorých prípadov som vám strhával 1–4 body.

Sever–juh: A teraz k vzoráku. Prvá vec, ktorú si môžeme všimnúť je, že bojová línia je priamka, teda graf lineárnej funkcie, ktorej všeobecný predpis je $f(x) = k \cdot x + q$. Vieme, že táto priamka prechádza cez body $(0, 0)$ a (a, b), takže $q = 0$ a dosadením do funkcie dostávame, že $f(a) = k \cdot a = b$, teda $k = b/a$ a $f(x) = \frac{b}{a}x$. S týmto už možno ľahko zistiť, či je i-ty vojak so súradnicami (x_i, y_i) na severe, alebo na juhu. Stačí sa pozrieť, akú ma y-ovú súradnicu bojová línia v bode x_i. Vojak patrí severu, ak platí $y_i > f(x_i)$ a juhu, ak $y_i < f(x_i)$. Rovnosť nastať nemôže, pretože žiadny vojak sa nenachádza priamo na bojovej línii. Tu by som chcel upozorniť tých z vás ktorý testovali podobnú nerovnosť a narobili v nej chyby. Aby nemuseli deliť, prenásobili nerovnosť a-čkom, a testovali, či $a \cdot y_i > b \cdot x_i$. Problém je, že tieto nerovnosti nie sú ekvivalentné – ak je a záporné, treba otočiť znamienko nerovnosti.

Vľavo-vpravo: Ani táto nerovnica však nie je na zahodenie. Čo vlastne vyjadruje $a \cdot y_i - b \cdot x_i > 0$? Ak $a > 0$, nerovnicu spĺňajú všetky body severu a pre $a < 0$ zase body juhu. Všimnite si, že keď sa postavíme do bodu $(0, 0)$ a pozrieme sa smerom na bod (a, b), našu nerovnosť spĺňajú *práve* tie body, ktoré sú *naľavo*. (Akurát pre $a > 0$ „naľavo“ znamená sever a pre $a < 0$ je naľavo juh.) Šípku z bodu $(0, 0)$ do (a, b) voláme *vektor* a hovoríme, že body spĺňajúce $a \cdot y_i - b \cdot x_i$ sú naľavo od vektora (a, b).

Najjednoduchšie je, samozrejme, riešiť úlohu tak, ako sme to popísali vyššie. Tento vzorák pokračuje iba kvôli tomu, že v skutočnosti je táto nerovnica v geometrií (a geometrických príkladoch) oveľa dôležitejšia. Všimnite si, že naša nová nerovnosť funguje aj pre zvislý vektor[2], lebo ak plati $b \cdot x_i < 0$ (čo je naša nerovnosť po dosadení $a = 0$), tak je bod x_i naľavo (pre kladné b je naľavo $x_i < 0$, pre záporné b je naľavo $x_i > 0$).

[1] Kto sa pozrel aj sem je ale úplná lama.

[2] kde nemá zmysel hovoriť o severe či juhu, ale o pravej a ľavej strane áno

Rozmyslite si, ako sa pomocou takejto (ne)rovnice dá zistiť, či daný bod leží na priamke, vľavo alebo vpravo. Tiež skúste vymyslieť, ako sa dá (jednoducho) zistiť, ktorá z dvoch priamok zviera väčší uhol s x-ovou osou.

Nakoniec, pomocou tejto nerovnice vieme riešiť aj danú úlohu – stačí zistiť, kde leží bod $(0,1)$ a tým určiť sever. Malá výhoda takéhoto riešenia je, že nepoužívame delenie, takže sa vyhneme práci s reálnymi číslami.

Listing programu:

```
program zradca;
var a,b,n,x,y,i,r,l :longint;
Begin
  r:=0;l:=0;{right-vpravo,left-vlavo}
  readln(a,b);
  readln(n);
  for i:=1 to n do begin
    readln(x,y);
    if (b*x-a*y)>0 then inc(r) else inc(l);
  end;
  if (b*0-a*1)>0 then {ci je bod (0,1) vpravo alebo vlavo}
    if r>l then writeln('pridaj sa k severu') else writeln('pridaj sa k juhu')
  else
    if r>l then writeln('pridaj sa k juhu') else writeln('pridaj sa k severu');
end.
```

2. Zaspievajme si s vĺčkom

opravoval Hermi
(max. 15 bodov)

Toto bol pomerne ľahký príklad, aj napriek tomu, že málo riešení vytrímalo moj nekompromisný a prísny (a spravodlivý)[3] pohľad bez ujmy na bodoch. Ale bez ďalšieho otáľania sa vrhnime na vzorové riešenie.

Skoro všetkých z vás ihneď napadlo s malými obmenami toto priamočiare riešenie: načítame pesničku, vypisujeme, ak narazíme na znak [spozornieme, pozrieme, aký akord sa nachádza za zátvorkou. Prevedieme ho na číslo, posunieme jeho hodnotu a vypíšeme miesto neho akord už transponovaný. Je jasné, že na každý znak pesničky sa budeme musieť pozrieť, preto čas tohto triviálneho riešenia už je optimálny. To, čo sa ešte budeme snažiť znížiť, je pamäťová náročnosť riešenia. Veľa z vás si neuvedomilo, že takýto postup znamená to, že v niektorom momente držíme v pamäti celú pesničku, čo môže byť náročné na pamäť (pesnička môže byť veľmi dlhá). Ak sa zamyslíme, uvedomíme si, že akonáhle znak, ktorý nie je akord načítame, môžeme ho rovno vypísať. Neskôr ho už nevyužijeme, a preto je zbytočné ukladať ho do pamäte. Po krátkej úvahe dospejeme aj k tomu, že vysporiadať sa s akordami vieme podobne. Niektorí z vás, ktorí robili načítanie po znakoch, akonáhle narazili na zátvorky, načítali celý obsah zátvorky a potom akord transponovali a vypísali. Toto už zaberie menej pamäte ako mať načítanú celú pesničku, ale akordy vedia byť veľmi zložito zapísané (napríklad `A#sus2add13/5-/C`[4]), preto by sme chceli pamäť ešte zjednodušiť. Stačí nám uvedomiť si, ktorá časť akordu sa mení pri transponovaní. Sú to najviac prvé 2 znaky. Čiže nám stačí najprv načítať prvé 2 znaky, zistiť o aký akord sa jedná, transponovať, a môžeme ďalej postupovať naším osvedčeným spôsobom načítavať po znakoch a priamo vypisovať až kým sa nevyskytne ďalšia zátvorka.

[3] óóó veľký Hermi, prísny a spravodlivý

[4] kto nájde najrýchlejšie najkratší možný zápis akordu a ozve sa mi, získa sladkú odmenu

Pozrime sa teraz ešte bližšie na posun akordu. Najefektívnejším riešením bolo očíslovať si postupne tóny a vložiť ich do poľa 0..11 (hneď sa dozvieme prečo od 0). Teraz budeme prevádzať tak, že prejdením poľa zistíme, aké číslo má akord, ktorý sme načítali. K tomuto číslu pripočítame N. Čo sa nám však mohlo stať? Áno, tento súčet už nemusí byť z intervalu 0..11, ba čo viac, môže byť dokonca záporný. Čo môžeme v takomto prípade robiť? Keďže pole je cyklické, môžeme k výsledku bez následkov pričítavať aj odčítavať 12. Vlastne nás teda zaujíma, aké číslo nám ostane keď odčítame všetky dvanástky. Čo je presne zvyšok po delení[5] 12. Použijeme teda modulo. Tu však treba spomenúť to, na čo veľa z vás doplatilo. *Ako funguje modulo v programovacích jazykoch:* modulo v takmer všetkých programovacích jazykoch sa správa inak ako matematické modulo. Totiž *ak je číslo záporné*, aj po modulovaní *záporné ostane*. Výpočet sa spraví na absolútnej hodnote a znamienko sa opíše. Preto bolo treba v programe vyriešiť to, ak bolo číslo po modulovaní z intervalu $-11..-1$. Toto sa dá ľahko vyriešiť napríklad tým, že k tomuto číslu pričítame 12. Vo vzoráku je toto vyriešené tak, že na začiatku N-ko zmodulujeme a pričítame 12, týmto sme si zaručili, že N je určite kladné, potom pri posúvaní stačí pričítať k indexu akordu a opäť spraviť modulo. Tu si dovolím ešte poznámku: veľa z vás tento postup nahradilo obyčajnou podmienkou, kde pričítali/odčítali 12 ak bolo číslo pod/nad rozsah intervalu. Bohužiaľ, aj keď vy ste si správne uvedomili, že nemá cenu zadávať N väčšie ako 11 a menšie ako -11, nebolo nikde spomenuté, že N v tomto intervale bude. Aj do budúcnosti treba pamätať, že je chyba predpokladať o vstupe niečo, čo nie je explicitne uvedené v zadaní.

Listing programu:

```
program Zpievajuci_vlcok; {takto sa mala úloha volať, kým ju Mišof nescenzuroval}
var
  stupnica: array[0..11] of string[2] =
('C','C#','D','D#','E','F','F#','G','G#','A','B','H');
  c: char;
  n,i: integer;

begin
  readln(n);
  n:=(n mod 12)+12;                          {týmto zaručíme, že N je kladné}
  while not EOF do begin
    read(c);
    write(c);
    if c='[' then begin
      read(c);
      i:=0;
      while(stupnica[i]<>c) do inc(i);    {zistíme ktorý tón v poradí to je}
      read(c);
      if c='#' then begin                 {ak sa za ním nachádza #, zvýšime o 1}
        inc(i);
        write(stupnica[(n+i) mod 12]);    {vypíšeme transponovaný akord}
      end else
        write(stupnica[(n+i) mod 12],c); {ak 2. načítaný znak nebol #, musíme ho
vypísať}
    end;
  end;
end.
```

Posledná vec, ktorú by som vám chcel spomenúť je *používanie knižničných funkcií*. Viacerým z vás bohužiaľ to, že ste použili funkcie *insert* a *delete* zhoršilo zložitosť. Ak sa rozhodnete

[5] odborne nazývaný aj modulo

používať funkcie, ktoré už sú v jazyku implementované, treba sa vždy poriadne zamyslieť, či vám to nezhorší časovú zložitosť. Ak máte pochybnosti, pokúste sa odhadnúť, v akom čase by mohli funkcie pracovať. Napríklad, ak chcete insertom vložiť znak doprostred reťazca, znamená to, že musí funkcia posunúť všetky znaky od miesta, kam vkladáme do konca aby vytvorila miesto pre znak, ktorý chceme vložiť. Toto je približne úmerné dĺžke reťazca. K podobnému záveru dospejeme aj s funkciou delete. Ďalší prípad je C-čková funkcia *strlen*, ktorá vždy prechádza string až kým nenájde na ukončovací znak, preto ak toto použijeme v podmienke cyklu, pri každom overovaní podmienky sa prechádza celý reťazec. Riešenie je napríklad si výsledok funkcie *strlen* proste zapamätať do premennej (treba ale dbať na to, aby sme dĺžku stringu nemenili, inak je treba ju opäť zrátať nanovo).

3. Zase ten set

opravovala Elfinka
(max. 15 bodov)

Set sú nejaké 3 karty, ktoré spĺňajú nejaké vlastnosti – nájdeme teda všetky možné trojice kariet, a overíme, či nie je set. Takto by sa dala popísať väčšina riešení, ktoré prišli. Takýto prístup vedie k riešeniu v čase $O(kn^3)$. Prvá vec, ktorá sa dala o setoch zistiť je, že ak máme ľubovoľné dve karty, existuje k nim práve jedna tretia s ktorou tvoria set. Môžeme teda v čase $O(n^2)$ vybrať všetky dvojice, k nim dopočítať tretiu a potom zistiť, či sa nachádza na stole. Nájdenie kartičky nám však zatiaľ trvá $O(kn)$, a teda sa čas nezlepšil. Pomohlo by však, ak by sme si karty na stole lexikograficky utriedili.

Čo to znamená lexikograficky utriediť? Napríklad slová v slovníku sú lexikograficky utriedené. Ako teda dve karty lexikograficky porovnáme? Najskôr ich porovnáme podľa prvej vlastnosti. Ak sú prvé vlastnosti rovnaké, tak ich porovnáme podľa druhej vlastnosti. Takto pokračujeme, až kým sa nám neminú všetky vlastnosti. Keďže takéto porovnanie trvá $O(k)$, tak preto sú skoro všetky časové zložitosti v tomto vzoráku vynásobené k.

Triediť môžeme napríklad Quicksortom – vlastnosti si v poradí ako boli na vstupe zapamätáme do stringu (môžeme využiť predpoklad, že $k \leq 7$). Stringy potom už vieme jednoducho porovnávať. Ako taký quicksort funguje? Z postupnosti, ktorú chceme utriediť, si vyberieme jeden prvok (ktorý si vybrať, popíšem neskôr) a nazveme ho pivot. Potom našu postupnosť preusporiadame tak, že najskôr budú prvky menšie alebo rovné ako pivot (túto časť nazvem ľavá časť) a potom budú prvky väčšie alebo rovné ako pivot (pravá časť). Ak by sa nám podarilo ľavú aj pravú čast utriediť, tak dostaneme utriedenú postupnosť. Na to ale použijeme rovnaký postup (optimálne je použiť v tomto prípade rekurziu). Postupnosti dĺžky 1 už nebudeme triediť, lebo sú utriedené.

Ako dlho to bude trvať? V najhoršom prípade to bude trvať v čase $O(kn^2)$ a to vtedy, ak by sme napríklad dostali utriedenú postupnosť, a ako pivota by sme vyberali prvý prvok z postupnosti. Keby sa nám darilo vyberať pivota tak, že si postupnosť rozdelíme na dve rovnako veľké časti, tak by časová zložitosť bola $O(kn\log(n))$. Môžeme sa o tom presvedčiť takto – každé porovnanie trvá $O(k)$ a porovnaní spravíme $O(n)$ za rozdelenie celého poľa $+2\cdot O(n/2)$ za rozdelenie polovíc $+4\cdot O(n/4)$ za štvrtiny $+\ldots+2^{\log(n)}O(1)$ čo je $O(kn\log n)$. Ako teda vyberať pivota? Jeden možný spôsob je vyberať pivota zo stredu (ako v našom kóde). Ale aj na to existuje škaredý vstup, na ktorom to bude trvať $O(kn^2)$. Avšak na priemernom vstupe to trvá $O(kn\log(n))$. Preto sa zvykne používať taká finta, že ako pivota vyberieme náhodný prvok, a teda žiaden vstup nebude pre náš algoritmus najhorší.

Medzi takto utriedenými kartičkami už vieme binárne vyhľadať prvok v čase $O(k\log n)$. Pozrieme sa na kartičku v strede poľa. Ak to nie je tá, ktorú hľadáme, tak sa pozrieme, či je táto lexikograficky väčšia alebo menšia od hľadanej karty. Ak je väčšia, tak s istotou vieme, že karta, ktorú hľadáme, je v prvej polovici nášho zoznamu, pretože je abecedne utriedený. Ak je menšia, tak vieme, že karta, ktorú hľadáme, je v druhej polovici. Teraz opakujeme to isté so zostávajúcou polovicou. Pozrieme sa na stredný prvok polovice, porovnáme a vyberieme vhodnú štvrtinu atď. Takto pokračujeme v delení a porovnávaní, až kým nenastane jedna z

dvoch možností. Buď nájdeme našu kartu a pridáme k počtu setov jeden alebo nám ostane časť poľa dĺžky 2. Ak nastane tá druhá možnosť, tak buď je naša karta jeden z krajných bodov časti poľa alebo karta v zozname nie je.

Výsledná časová zložitosť takéhoto riešenia bude $O(kn^2 \log n)$. Čo sa týka pamäti, pamätáme si celý stôl – čiže n kariet s k vlastnosťami. Zložitosť teda bude $O(nk)$.

Listing programu:

```
var i,j,K,l,N:integer;
    stol:array[1..50] of string;
    hladane:string;
    setov:integer;
    c,d:char;


procedure qsort(a,b:integer);
var i,j:integer;
    pivot, druhe:string;
begin
  pivot:=stol[(i+j)div 2];
  i:=a;
  j:=b;
  while i<j do begin
    while (stol[i]<pivot) and (i<j) do inc(i);
    while (stol[j]>=pivot) and (i<j)  do dec(j);
    if i<j then begin
      druhe:=stol[i];
      stol[i]:=stol[j];
      stol[j]:=druhe;
    end;
  end;
  qsort(a,i);
  qsort(j,b);
end;

procedure najdi(a,b:integer);
begin
  if a=b then begin
    if stol[a]=hladane then inc(setov);
  end else begin
    if stol[(a+b)div 2]>hladane then najdi((a+b) div 2, b)
    else najdi(a,(a+b) div 2 - 1);
  end;
end;

begin
  readln(N,K);
  setov:=0;
  for i:=1 to N do
    for j:=1 to K do begin
      read(c);
      stol[i]:=stol[i]+c;
    end;
  qsort(1,N);
  for i:=1 to N-1 do
    for j:=1 to N do begin
      for l:=1 to K do begin
        hladane:=stol[i];
        if (stol[i][l]<>stol[j][l]) then begin
```

```
          if (stol[i][l]<stol[j][l]) then begin
            c:=stol[i][l];
            d:=stol[j][l];
          end else begin
            c:=stol[j][l];
            d:=stol[i][l];
          end;
          if c='2' then hladane[l]:='1'
          else begin
            if d='2' then hladane[l]:='3'
            else hladane[l]:='2';
          end;
        end;
      end;
      najdi(j,N);
    end;
end.
```

opravoval Luki
(max. 15 bodov)

4. Zhodnotenie vedátorov

Tento príklad ma prekvapil svojim veľkým počtom riešení. Opravoval som, opravoval a ono stále neubúdalo :). Presne takto to má byť a dúfam, že nabudúce dostane toľkoto riešení aj 10. príklad.

Začnime prvou podúlohou, s ktorou to bolo veselé[6]. Mali ste ukázať, že h-index je definovaný jednoznačne. Pri definícii, ktorú ste dostali do rúk, to však nebola pravda. Ukážeme si prečo: uvažujme ľubovoľného vedátora, pre ktorého existuje číslo $h > 1$ také, že h je jeho h-index. Teraz uvažujme ľubovoľné menšie číslo, napríklad $h - 1$. Prvá podmienka zjavne platí, pretože platí pre nejaké vyššie číslo. Avšak platí aj druhá podmienka, pretože v prvej podmienke zahrnieme všetky články, ktoré majú počet citácii aspoň $h-1$, a preto sa nemáme ako dostať do sporu. Ak uvedenú úvahu rozvinieme, uvidíme, že ak je nejaké prirodzené číslo h h-index nejakého vedátora, potom sú aj všetky menšie čísla jeho h-indexy.

Náš úmysel nebol zmiasť vás a za chybu sa ospravedlňujeme. Každý, kto si uvedený nedostatok všimol, získal bonusový bod.

Pôvodná definícia sa od našej líši v prvej podmienke. Tá znie originálne takto:
– napísal h článkov, z ktorých každý má aspoň h citácií.

Vypadlo nám prvé slovo aspoň a preto nám zlyháva náš kontrapríklad. A naozaj, s takouto definíciou už môžeme dokazovať jednoznačnosť, a to sporom: nech má nejaký vedátor dva rôzne h-indexy h_1 a h_2 také, že $h_1 > h_2$. Potom ale existuje h_1 článkov s počtom citácii aspoň h_1 a zvyšných $K - h_1$ článkov má počet citácii najnajvýš h_1. Podmienka „napísal h_2 článkov, z ktorých každý má aspoň h_2 citácií“ tiež platí (existuje dokonca h_1 takých článkov). Podmienka „každý z jeho ostatných článkov má najviac h_2 citácií“ už však neplatí, pretože existuje aspoň $h_1 - h_2$ článkov (keďže $h_1 > h_2$, tak aspoň jeden), ktoré majú aspoň h_1 citácii. Spor.

Riešenie 1.: V *citacii*$[i]$ si uložíme počet citácií pre i-ty článok vedátora. Potom toto pole usporiadame od najväčšieho počtu citácii až po najmenší a hľadáme prvý taký index i, že *citacii*$[i] < i$. Vtedy vieme, že na prvej až $(i-1)$-vej pozícii majú články $\geq$ počet citácii ako $i-1$. Vyplýva to z usporiadania poľa citacii. a od i-tej po k-tu pozíciu sú články s $\leq$ počtom citácií ako $i-1$. Teda $i-1$ je h-index.

[6] Aj ked by sa dalo doplniť možno vhodnejšie prídavné meno.

Vzorové riešenie: Toto riešenie používa algoritmus countsort, ktorý nám umožňuje robiť zdanlivo nemožné – triediť v lineárnom čase. V čom je háčik? Za takúto rýchlosť platíme vysokou možnou pamäťovou zložitosťou. Celý trik sa spolieha na to, že máme konečne veľa prvkov – vezmime si čísla z rozsahu od 1 po N, ktoré sú na vstupe.

Jediné, čo budeme potrebovať, je N-prvkové pole, ktoré bude na začiatku naplnené nulami. Teraz nám stačí postupne čítať postupnosť, ktorú chceme triediť a pre každé prečítané číslo i pripočítať na i-tu pozíciu v poli jednotku. Keď takto budeme mať prečítané všetko, čo chceme triediť, pozrieme sa na naše pole. Na prvom mieste vidíme počet jednotiek, na druhom dvojok, ... Ak chceme zotriedenú postupnosť, stačí, ak postupujúc od najmenšieho, vypíšeme i-ty prvok toľkokrát, aké je číslo na i-tom mieste v poli.

Do poľa *cit* si pre každé i uložíme počet citácií pre i-ty článok vedátora. Uvedomme si, že h-index môže byť najviac K, lebo máme iba K článkov. Countsortom v poli *count* spočítame, koľko existuje pre index i článkov s počtom citácií práve i. A z toho už nie je problém určiť h-index. Prechádzame pole *count* od K smerom k nižším hodnotám a pri tom spočítavame počet článkov v premennej *pocet*. Hľadáme prvý index i, pre ktorý platí, že $(pocet + count[i]) \geq i$. Teda existuje i článkov s počtom citácií aspoň i a taktiež zvyšné články majú najviac i citácií.

Ako sa hodnotilo: Riešenia boli rôzneho druhu, niektoré nefunkčné (max. 3 body). Niektoré s časovou zložitosťou $O(M \cdot K)$ a pamäťovou zložitosťou $O(M)$ (max. 8 bodov), iné v čase $O(M+K^2)$ a pamäti $O(K)$ (max. 10 bodov). Riešenie, ktoré využívalo quicksort a pracovalo v čase $O(M + K \log K)$ a pamäti $O(K)$ dostalo max. 12 bodov a vzorové riešenie bežiace v čase $O(M + K)$ a pamäti $O(K)$ dostalo 15 bodov. Za chýbajúci popis -2 body a zvyšok podľa uváženia :).

Listing programu:

```
const MAX=100;
type POLE = array[1..MAX] of integer;
var pocet,i,j,kto,koho,n,m,k:integer;
    citacii,count: POLE;

begin
    readln(n,k,m);
    for i:=1 to k do
    begin
        citacii[i] := 0;
        count[i] := 0;
    end;
    for i:=1 to m do
    begin
        readln(kto,koho);
        if (kto > k)and (koho <= k) then inc(citacii[koho]);
    end;

    for i:=1 to k do
    begin
        if (citacii[i] > k) then inc(count[k])
        else inc(count[citacii[i]]);
     end;

    i:=k;
    pocet:=0;
    while ((pocet+count[i]) < i) do
    begin
        pocet := pocet + count[i];
        dec(i);
```

```
  end;
  writeln(i);
end.
```

opravoval Mišo
(max. 15 bodov)

5. Odhodlaná Anička

Troška teórie: Rodostrom predstavoval binárny strom, ktorý bol zadaný inorderom. Binárny strom je štruktúra pozostávajúca z vrcholov (uzlov), ktoré majú najviac dvoch synov a jedného rodiča. Rodostrom pozostáva z ľudí, ktorí majú najviac 2 známych predkov a jedného potomka. Takže istá paralela by mala byť viditeľná. Na odlíšenie budeme používať predok a potomok pre rodostrom a rodič a syn pre binárny strom. Inorder je usporiadanie vrcholov do postupnosti také, že v usporiadaní je ľavý syn pred svojim otcom a pravý syn za svojím otcom, pričom nemusí byť hneď za ním, pokojne aj obďaleč.
Najprv sa pozrime na rekurziu v mojom kóde. Rekurzívna procedúra spracúvavajúca podstrom má na vstupe dve čísla. Tieto dve čísla určujú začiatok a koniec inorder zápisu podstromu v zápise celého stromu. Potrebujeme nájsť vrchol, ktorý už nemá rodiča v danom podstrome[7]. Pre koreň platí, že počet predkov z otcovej strany (veľkosť podstromu tvoreného ľavým synom) sa rovná poradovému číslu koreňa v inorder usporiadaní tohto podstromu.

Toto však neplatí len pre koreň – platí to aj pre jeho pravého syna, pre pravého syna tohto pravého syna, atď. Ale platí, že koreň je spomedzi takýchto vrcholov v poradí najneskôr. Jedným prechodom celej postupnosti sa preto dá nájsť vrchol, ktorého index je rovný počtu predkov zo strany otca a je najďalej. Počet predkov zo strany matky je rozdiel veľkosti podstromu a indexu koreňa. Následne sa rekurzívne zavoláme na ľavý a pravý podstrom. V priemernom prípade[8] zložitosť síce bude $O(N \log N)$, ale v najhoršom prípade (napr. samé matky) $O(N^2)$. Pre každé vnorenie vykonáme lineárne veľa operácií podľa veľkosti podstromu. Za toto riešenie bolo 10 bodov, častá chyba bola nesprávny odhad zložitosti ako $O(N \log N)$, za čo som strhol bod.

Chvíľka zamyslenia: V spomenutom rekurzívnom riešení sme veľakrát prechádzali tie isté vrcholy. Ako by sa to dalo lineárne? Pri spracovávaní vrchola chceme vedieť jeho počet predkov z matkinej strany. Takže by sme mali vrcholy spracovávať od konca. Posledný vrchol je určite človek, ktorá nemá matku. Preto má 0 predkov zo strany matky. Ak má predkov zo strany otca (vieme zo zadania počet), tak sa rekurzívne zavoláme na všetkých týchto predkov – x-tý vrchol v inorder usporiadaní má predkov zo strany otca na pozíciách $x - o[x]$ až $x-1$, kde $o[x]$ je počet predkov z otcovej strany. Keď opúšťame vrchol x, vieme počet jeho predkov zo strany matky $m[x]$ a počet predkov zo strany otca $o[x]$. Následne sa posunieme, kedže sme spracovali všetkých predkov aktuálneho vrcholu x, na jeho potomka, ktorý má index v usporiadaní menší o $o[x]+1$. Toto opakujeme, kým sa máme kam posunúť, teda len po ľavý okraj intervalu. Časová zložitosť je $O(N)$, pretože každý prvok riešime len pomocou konštantného počtu operácii. Za takéto riešenie bolo 15 bodov. Bod som strhol za zlý odhad zložitosti.

Listing programu:

```
var m,o:array[1..20]of longint;
    meno:array[1..20]of string;
    i,n:longint;

procedure vnor(zac, kon:longint);
```

[7] Takému vrcholu hovoríme obyčajne koreň.

[8] Výpočet priemerného prípadu nebudeme robiť, musíte nám jednoducho veriť

```
var i:longint;
begin
    i:=kon;
    while i>zac do begin
        vnor(i-o[i],i-1);
        m[i]:=kon-i;
        i:=i-o[i]-1;
    end;
end;

begin
readln(n);
writeln(n);
for i:=1 to n do begin
    readln(meno[i]);
    readln(o[i]);
end;

vnor(1,n);

for i:=1 to n do
    writeln(meno[i],' ',m[i]);
end.
```

Opravoval Zemčo
(max. 15 bodov)

6. Overená klebeta

Tohto príkladu ste sa, zdá sa, zľakli, pretože prišlo pomerne málo riešení v porovnaní s príkladmi porovnateľných náročností v tejto sérii. Ďalej musím smutne konštatovať, že viacero vašich riešení nebolo korektných. Nabudúce bude užitočné po tom, ako niečo vymyslíte, venovať chvíľu aj úvahe, či to vôbec dáva na každý vstup správny výsledok. Najlepšie je si túto skutočnosť formálne dokázať. Keďže korektné riešenie v horšom čase bolo skoro tak ťažké vymyslieť ako vzorové, poďme rovno na to.

Sieť, po ktorej sa šíri klebeta si budeme reprezentovať ako graf, o ktorom predpokladáme, že je súvislý. Vieme, že medzi každými dvoma vrcholmi existuje práve jedna cesta, takže to bude strom.

Prvotné zhodnotenie skúseného programátora možno tiplo lineárny vzorák, ako sa nám to už pri stromoch často stáva. Tentoraz to ale pôjde prinajrýchlejšom pomalšie, a to v $O(N \log N)$ čase[9]. Avšak pri stromoch bežne používaný postup modifikovaného prehľadávania do hĺbky a zisťovania si informácii o podstromoch sa vyskytne.

Veroniku budem vo zvyšku tohto textu hanlivo nazývať koreňom. Ak sa má klebeta rozšíriť všade, musí sa najprv rozšíriť k bezprostredným susedom koreňa. Bezprostrední susedia koreňa sú v podobnej situácii ako koreň – okrem hrany, po ktorej prišla klebeta majú (možno) aj nejaké hrany, po ktorých musia klebetu poslať ďalej. Jadrom programu bude teda rekurzívna funkcia, ktorá začne v koreni a postupne sa zavolá do všetkých vetiev, po ktorých treba klebetu posunúť ďalej. Funkciu zavoláme do vrcholu v a tá nám vráti čas, koľko najmenej trvá od dozvedenia klebety vrcholom v po rozšírenie do všetkých vrcholov, do ktorých má klebeta putovať cez v. Inými slovami, z vrcholu v sa nám vráti čas potrebný na to, aby sa klebetu dozvedel celý podstrom pod v po tom, ako sa ju dozvie v. V prípade, že dievča bude vedieť, koľko trvá rozšírenie klebety do všetkých vetiev (samozrejme okrem

[9]Platí tvrdenie, že každé lineárne riešenie, ktoré prišlo, bolo nekorektné. Ostatne, bola by sranda, keby také tvrdenie neplatilo.

tej, po ktorej prišla klebeta), bude vedieť vypočítať aj celkový čas, ktorý má vrátiť tomu vrcholu, od ktorého klebeta prišla.

Najjednoduchšia situácia nastáva, keď klebeta príde k niekomu, kto ju už nemá kam ďalej posunúť. Vtedy jednoducho vrátime nulu. Keď vrchol nemá žiadnych potomkov, nepotrebuje žiadny čas, aby k nim klebetu rozšíril. Teraz uvažujme situáciu, v ktorej má vrchol v dva kontakty, označme v_1 a v_2. Od v_1 prišla klebeta a k v_2 ju treba ešte poslať. Rekurzívne sa zavoláme do v_2 a keď sa vráti očakávaná hodnota, nebude ťažké vypočítať, čo máme vrátiť vrcholu v_1. Bude to hodnota, ktorú sme obdržali od v_2 zvýšená o hodnotu hrany medzi nami a vrcholom v_2. Práve toto bude hľadaný čas, pretože po obdržaní klebety vrcholom v treba zavolať do v_2 a tam to bude trvať už známy čas.

Kľúčová otázka zostala, čo robiť, ak vrchol musí klebetu poslať ďalej po viac ako jednej hrane. Označme tento vrchol u. Nech vrchol u má k susedov, $k \geq 2$. O vetve so susedom v_i označíme t_i čas potrebný na rozšírenie klebety v tejto vetve a w_i cenu hrany medzi u a v_i. Hrany budú obvolávané postupne a jeho cieľom je, aby najvyšší čas, po ktorom sa to dozvie posledný vrchol v celom podstrome bol čo najmenší. Dokážeme si tvrdenie, že optimálne poradie je obvolávať vetvy podľa dĺžky šírenia správy v príslušnom podstrome. Kde to bude trvať najdlhšie, tam zavoláme najskôr. V ďalšom budem nazývať vetvu s vysokým časom šššírenia klebety t_i niekedy aj veľká, prípadne drahá.

Vrchol v už všetky hodnoty t_i pozná a má určiť hodnotu, koľko bude trvať rozšírenie klebety do všetkých vetiev. Uvažujme poradie, v ktorom vetvy neboli obvolávané v nerastúcom poradí podľa t_i. Potom existuje vetva, ktorá bola zavolaná beprostredne pred nejakou väčšou. Označme väčšiu vetvu v_i a vetvu pred ňou v_j. Platí $t_i > t_j$. Ukážeme, že najdlhší čas sa určite nezvýši, ak poradie týchto dvoch vetiev vymeníme. Formálne zapísané:

$w_i + w_j + t_i = \max\{w_i + w_j + t_i, w_j + t_j\} \geq \max\{w_i + t_i, w_i + w_j + t_j\}$

Prvý člen označuje čas potrebný na rozšírenie klebety do druhého z uvažovaných podstromov od momentu, kedy sme začali telefonovať do prvého. Musíme telefonovať do vetvy j, potom do vetvy i a potom čakať čas t_i, než sa klebeta rozšíri tam. Prvá rovnosť hovorí, že tento čas určite nie je menší ako čas, ktorý musíme čakať na rozšírenie klebety do prvej vetvy, pretože pri prvej vetve nemusíme čakať na hranu w_i a navyše $t_i > t_j$. Nerovnosť hovorí, že keď vymeníme poradie telefonovania, maximum určite nevzrastie. Čas rozšírenia klebety vo vetve i bude teraz $w_i + t_i$, a to je určite menej ako pôvodný čas rozšírenia v tejto vetve, pretože nečakáme hranu w_j. Čas rozšírenia vo vetve j bude teraz $w_i + w_j + t_j$ a to je určite menej ako $w_i + w_j + t_i$, pretože $t_i > t_j$. Preto takou výmenou určite nič nepokazíme. Ľahko sa ale dá nájsť príklad, kedy ňou niečo získame. Na usilovného čitateľa nechám aj dôkaz, že na poradí rovnako drahých vetiev nezáleží.

Výsledok je teda taký, že všetky hodnoty t_i, ktoré prišli z podstromov si zotriedime podľa veľkosti a vyrátame, aké bude maximum pri poradí obvolávania od najväčšej po najmenšiu vetvu.

Takýmto spôsobom dostaneme korektné riešenie. Upravené prehľadávanie do hĺbky nás vo svojej holej podstate stojí čas $O(N)$, pretože sme v strome. Je tam ale ešte triedenie. Ak pripustíme pre jednoduchosť, že budeme triediť aj jednoprvkové postupnosti, potom každá z $N - 1$ hrán predstavuje prvok, ktorý treba práve raz v nejakom vrchole utriediť. Môže sa nám stať, že všetky prvky budeme triediť v jednom vrchole[10]. Potom by riešenie trvalo čas $O(N \log N)$ za použitia efektívneho algoritmu. Vo vzoráku nájdete Quicksort. Ak sa nám hrany, ktoré treba triediť, rozdelili do viacerých vrcholov, bude to ešte lepšie, pretože pre každé prirodzené $R, L, K, R = K + L, K \leq L$, platí: $K \log K + L \log L \leq K \log L + L \log L = (K + L) \log L = R \log L \leq R \log R$. Celková časová zložitosť algoritmu bude teda $O(N + \sum_{v \in V}(deg(v) - 1) \log(deg(v) - 1) = O(N \log N)$. Pamäťové nároky sú lineárne. Bonusová

[10] To by bol koreň. Myslím že na nejakých cvičeniach nazval nejaký cvičiaci takýto typ grafu *ježko*. Neviem však, nakoľko sa toto pomenovanie zhoduje so svetovou odbornou literatúrou.

úloha na záver: skúste pouvažovať a nebodaj aj formálne dokázať, že to vo všeobecnosti[11] rýchlejšie nepôjde. Alebo toto tvrdenie vyvráťťe tým, že nájdete rýchlejší algoritmus ako náš vzorák :).

Bodovanie bolo pri tomto príklade veľmi rôznorodé. Keďže s výnimkou vzorákov neprišli žiadne dve riešenia, ktoré by sa zhodovali v korektnosti, úplnosti prevedenia a v efektivite zároveň, dá sa povedať, že som hodnotil individuálne. Tam chýbal kód, tam zase popis, hento bolo nekorektné, tamto bolo pomalé a hento bolo síce korektné, ale bez dôkazu. Vo všeobecnosti som za nekorektné riešenie dával najviac 1 bod a za absenciu kódu som stŕhal polovicu zisku. Najviac mi chýbali absencie zdôvodnení korektnosti. Nesmie to fungovať tak, že ak je vaše riešenie nekorektné, tak vám to mám ja ukázať, ale tak, že ak je vaše riešenie korektné, tak ma o tom máte presvedčiť vy! Jediná istota v tomto príklade bola teda 15 bodov za optimálne riešenie, zvyšok som sa snažil ohodnotiť spravodlivo vzhľadom k nedostatkom.

Listing programu:

```
program klebety;
const MAXN = 1000;
type hrana = record ciel,hodnota: integer; end;
     data = record hodnota,poradie: integer; end;
     pole = array[1..MAXN]of data;

var graf: array[1..MAXN,1..MAXN] of hrana;
    bol: array[1..MAXN] of boolean;
    deg: array[1..MAXN] of integer;
    x,y,z,n,i: integer;

procedure swap(var A: pole; x,y:integer);
var t: data;
begin t := A[x]; A[x] := A[y];A[y] := t; end;

procedure qsort(var A: pole; l,r: integer);
var pivot,i,j:integer;
begin
   if (r - l <= 0)then exit;
   pivot := random(r-l+1);
   pivot := A[l+pivot].hodnota;
   i := l;
   j := r;
   repeat
      while (A[i].hodnota < pivot)do inc(i);
      while (A[j].hodnota > pivot)do dec(j);
      if (i <= j)then
      begin
         swap(A,i,j);
         inc(i);
         dec(j);
      end;
   until i>j;
   if i < r then qsort(A,i,r);
   if j > l then qsort(A,l,j);
end;

function zavolaj(v: integer):integer;
var A: pole;
    synov,i,max,nadbytok: integer;
```

[11] Tým máme na mysli, že nič navyše o grafe nebudeme predpokladať a postup bude fungovať úplne univerzálne na každom grafe.

```
begin
   {nastavime si ze sme prisli a vynulujeme vsetky pomocne premenne}
   bol[v] := true;
   max := 0; nadbytok := 0; synov := 0;
   for i:=1 to deg[v] do if bol[graf[v][i].ciel] = false then
   begin
         {nasli sme syna, a preto sa zavolame dalej}
 inc(synov);
         A[synov].hodnota := zavolaj(graf[v][i].ciel);
         A[synov].poradie := i;
   end;
   {utriedime casy, ktore nam vratili synovia}
   qsort(A,1,synov);
   {a zistime maximum}
   for i:=synov downto 1 do
   begin
      if max < nadbytok+A[i].hodnota+graf[v][A[i].poradie].hodnota then
max:=nadbytok+A[i].hodnota+graf[v][A[i].poradie].hodnota;
      nadbytok := nadbytok+graf[v][A[i].poradie].hodnota;
   end;
  zavolaj := max;
end;

begin
   readln(n);
   for i:=1 to n do begin deg[i] := 0; bol[i] := false; end;
   for i:=1 to n-1 do
   begin
      readln(x,y,z);
      inc(deg[x]);
      graf[x][deg[x]].hodnota := z;
      graf[x][deg[x]].ciel := y;
      inc(deg[y]);
      graf[y][deg[y]].hodnota := z;
      graf[y][deg[y]].ciel := x;
   end;
   writeln(zavolaj(1));
end.
```

7. Otáčanie zátvoriek

opravoval Laci
(max. 15 bodov)

Kategoricky sa vyskytli dva druhy riešení a to riešenia pracujúce v čase $O(MN)$ a vzorové v $O(N + M \log N)$. V skratke si ukážeme to jednoduchšie v $O(MN)$ a potom sa budeme venovať vzorovému riešeniu, ktoré z neho vychádza.

Korektne uzátvorkovaný reťazec zátvoriek môžeme v skratke definovať touto podmienkou: Ku každej ľavej zátvorke prislúcha práve jedna pravá zátvorka, pričom ľavá sa nachádza pred pravou. Ak nahradíme ľavé zátvorky číslom $+1$ a pravé zátvorky číslom -1 a je splnená podmienka, že ku každej ľavej zátvorke prislúcha práve jedna pravá zátvorka, je súčet všetkých zátvoriek rovný 0. Ak platí dodatok podmienky, čiže žiadna zátvorka nebola ukončená skôr, ako sa začala, znamená to, že žiaden zo súčtov od 1 po $k, 0 < k \leq N$ nie je záporný. Na základe tohto máme postup na overenie korektnosti uzátvorkovania v čase $O(N)$: Budeme prechádzať zátvorky zľava doprava, za každú ľavú zátvorku si počítadlo (inicializované na 0) zvýšime o jedna a za každú pravú znížime o jedna. Ak počítadlo nenadobudlo zápornú

hodnotu a na konci je rovné 0, je to korektne uzátvorkovaný reťazec. Takže pre každý dotaz ohľadom korektnosti uzátvorkovania (je ich rádovo $O(M)$) odpovedáme v čase $O(N)$, ak máme obrátiť dáku zátvorku, jednoducho ju obrátime. Časová zložitosť tohto riešenia je teda $O(MN)$. Vystačíme si s jedným poľom pre uloženie zátvoriek, preto pamäťová zložitosť je $O(N)$.

Vždy zisťujeme celkový súčet a či každý čiastkový súčet je nezáporný. Túto úlohu však vieme riešiť aj rýchlejšie ako v lineárnom čase. Použijeme intervalové stromy, pre každú vyššie uvedenú úlohu jeden. Čo je to intervalový strom? Je to binárny strom, v ktorom pre každý uzol platí, že uchováva informáciu o intervale. Jeho ľavý syn uchováva informáciu o ľavej polovici intervalu, pravý syn o pravej polovici intervalu. V prvom prípade si budeme pre každý interval pamätať súčet hodnôt v danom intervale. V koreni teda bude súčet celého intervalu $1, \dots, N$. V uzloch druhého stromu si budeme pamätať najmenší súčet nejakého podintervalu, ktorého začiatok je zároveň začiatkom celého intervalu. Formálnejšie, ak vrcholu prislúcha interval $[a, b]$, tak si danom vrchole pamätáme $\min\{A(a, k) | a \leq k \leq b\}$, kde $A(a, k)$ je súčet hodnôt na intervale $[a, k]$. Ak je v koreni oboch stromov nula, je to korektne uzátvorkovaný reťazec. Zmenu zátvorky na mieste a riešime zmenou hodnoty v uzle pre interval $[a, a+1]$. Takýto špeciálny uzol (špeciálny preto, že nemá synov) nazveme list. Takisto musíme zmeniť hodnoty v uzloch na ceste z listu do koreňa.

Aby sme vedeli takýto úplný binárny strom postaviť, nájdeme najbližšiu väčšiu/rovnú mocninu dvoch od N. Toľko listov bude mať náš strom (asymptotickú zložitosť nám to neovplyvní, listov je stále $O(N)$), pričom do listov ktoré sme museli pridať, dáme nuly a to nám súčty neovplyvní. Strom vieme postaviť v čase $O(N)$ tak, že začneme od spodu, najprv dátami naplníme listy (najnižšiu úroveň) potom vypočítame hodnoty uzlov v druhej najnižšej úrovni a takto až po koreň. Vnútorných uzlov (to sú také čo nie sú listy, čiže majú potomka) je v úplnom binárnom strome s N listami presne $N - 1$. Každý uzol spracujeme v čase $O(1)$, preto postavenie stromu má časovú zložitosť $O(N)$.

Teraz vieme na otázky ohľadom korektnosti uzátvorkovania odpovedať v konštantnom čase. Zmeny zátvoriek riešime v čase závislom od hĺbky stromu (dĺžky cesty od listu do koreňa), čo je $O(\log N)$. Keď to sčítame: postavenie stromu, $O(M)$ otázok v $O(\log N)$ čase, máme časovú zložitosť $O(N + M \log N)$. Čo sa týka pamäte, máme dva stromy ktoré zaberajú $O(N)$ miesta a pár premenných, takže pamäťová zložitosť je $O(N)$.

Za riešenia v $O(N + M \log N)$ čase bolo 15 bodov, za $O(MN)$ sa dalo získať maximálne 10 bodov. Za neadekvátny popis, chýbajúci alebo zlý odhad časovej a pamäťovej zložitosti som odčítaval po jednom bode. Navyše bolo aj pár prípadov, ktoré nezapadajú do uvedených kategórií, tie boli ohodnotené individuálne.

Listing programu:

```
#include <iostream>
#include <vector>
#define se second
#define fi first
using namespace std;

int N,M,mocnina=1;
vector< pair<int,int> > strom;  //first= súčet, second= minimálny súčet
int tmp,dotaz; //pomocné premenné
char a;

void update(int uzol){ //obnoví požadované vlastnosti stromu pre daný vnútorný uzol
    strom[uzol].fi= strom[2*uzol].fi + strom[2*uzol+1].fi;
    strom[uzol].se= min(strom[2*uzol].se, strom[2*uzol].fi + strom[2*uzol+1].se);
    }
```

```
int main(){
   cin>>N>>M;
   while (mocnina<N) mocnina <<= 1; //bitový posun do ľava = vynásobenie dvoma
   strom.resize(mocnina << 1); //veľkosť pola kde bude uložený strom bude 2*mocnina
   for(int i=mocnina; i<mocnina+N; i++){ //načítame listy stromu (čiže zátvorky)
           cin>>a;
           strom[i].fi=strom[i].se= (a=='(')? 1 : -1;
           }

   for(int i=mocnina-1;i>0;i--) update(i); //postavenie intervalového stromu v O(N)

   while(M--){ //ideme spracovať dotazy
     cin>>dotaz;
     if(dotaz==0) cout<<( (abs(strom[1].fi)+abs(strom[1].se)==0)?"ANO":"NIE")<<endl;
      else {
           tmp=mocnina+dotaz-1;
           strom[tmp]=make_pair(-strom[tmp].fi,-strom[tmp].se); //výmena zátvoriek
           while(tmp>>=1) update(tmp); //update cesty od zmeneného listu po koreň
           }
     }
   return 0;
}
```

Opravoval Mic
(max. 15 bodov)

8. Okšoramove trampoty

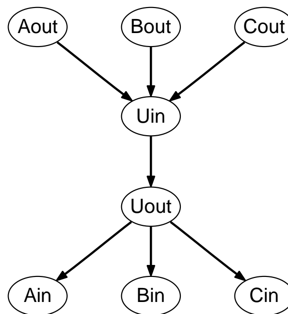
Dnes sa dozvieme, ako sa dá pomôcť Okšoramovi pri hľadaní. Ale najskôr poviem niečo k bodovaniu.

1. Celých 15 bodov mohol dostať ten, komu sa podarilo nájsť algoritmus s časovou zložitosťou $O(N + M)$.
2. Maximálne 8 bodov dostali tí, ktorí napísali riešenie s exponenciálnou časovou zložitosťou.
3. Nefunkčné riešenie mohlo dostať najviac 3 body.
4. Bonusy -1 bod som rozdával za chyby v programoch a za zlé odhady zložitosti.

Poďme si našu úlohu preformulovať do reči teórie grafov. Máme graf G a chceme nájsť takú cestu z vrcholu A do vrcholu C, na ktorej leží vrchol B. Alebo, preformulované trochu inak: Máme zadaný graf a našou úlohou je nájsť dve vrcholovo disjunktné[12] cesty začínajúce vo vrchole B a končiace vo vrcholoch A, resp. C. Ak také dve cesty nájdeme, ich spojením dostaneme hľadanú cestu.

Možností, ako optimálne riešiť túto úlohu, bolo viacero, tu odznie to jednoduchšie. Začneme tým, že si úlohu (na prvý pohľad možno nie, ale naozaj) zjednodušíme. Nebudeme hľadať dve *vrcholovo* disjunktné cesty, ale dve *hranovo* disjunktné cesty v trochu pozmenenom orientovanom grafe G': Každý vrchol v z grafu G rozdelíme na dva vrcholy v_{in} a v_{out} spojené hranou $v_{in} \to v_{out}$. Každú neorientovanú hranu u–v grafu G nahradíme dvoma orientovanými hranami $u_{out} \to v_{in}$ a $v_{out} \to u_{in}$ (pozri obr.). Všimnime si, že ľubovoľná cesta z u_{in} do w_{out} v pozmenenom grafe G' má tvar $u_{in} = v^0_{in} \to v^0_{out} \to v^1_{in} \to v^1_{out} \to \cdots \to v^n_{in} \to v^n_{out} = w_{out}$. Teda striedajú sa vstupné (v_{in}) a výstupné (v_{out}) vrcholy a po v_{in} vždy nasleduje v_{out} (tieto vrcholy sú spojené jedinou hranou). Takáto cesta zodpovedá ceste $u = v^0, v^1, \ldots, v^n = w$ v pôvodnom grafe. Presvedčte sa, že dvom hranovo disjunktným cestám v G' prislúchajú dve vrcholovo disjunktné cesty v G.

[12] okrem vrcholu B nemajú spoločný vrchol



Na obrázku vidíte, ako skončí vrchol U, ktorý má 3 susedov: A, B, C.

Ako nájdeme dve hranovo disjunktné cesty v orientovanom grafe? Najskôr nájdeme nejakú cestu z B do A[13], napríklad prehľadávaním do hĺbky, pričom každú hranu na ceste si označíme. Potom začneme znovu prehľadávať graf z vrcholu B, ale s tým rozdielom, že po neoznačených hranách budeme chodiť v smere orientácie hrany a po označených hranách budeme chodiť v protismere orientácie hrany. Ak týmto prehľadávaním nájdeme cestu do vrcholu C, tak všetky označené hrany na tejto ceste odznačíme a všetky neoznačené hrany na ceste označíme. Ako zázrakom budú označené cesty tvoriť cestu z A do C cez B. (Vyskúšajte si to na nejakom príklade.)

Prečo to funguje? Táto úloha súvisí s tokmi v grafe. Určite si prečítajte vzorové riešenie príkladu 10, kde je vysvetlené, čo sú toky, ako sa hľadajú a čo je to zlepšujúca cesta. Náš algoritmus vlastne v grafe G' (s jednotkovými kapacitami, zdrojom B_{out} a odtokmi A_{out}, C_{out}) hľadá tok veľkosti 2 a na to stačí nájsť dve zlepšujúce cesty.

Celé toto sa dá naprogramovať v lineárnom čase a v lineárnej pamäťovej zložitosti, rozumej $O(M+N)$. Kto chce, môže sa nechať inšpirovať vzorákom.

Listing programu:

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

struct edge{
    int from,to;
    int used;
    edge(int f,int t,int u) { from=f; to=t; used=u; }
};

int N, O, P, A, M, dest;

vector<edge> E; //zoznam hran
vector<vector<int> > G, GB; //G-cisla vystupnych hran, GB-cisla vstupnych hran
vector<bool>was;

int dfs(int v){
    if(v==dest)return 1;
    if(was[v])return 0;else was[v]=1;
    for(unsigned int i=0;i<G[v].size();i++){
        if(E[G[v][i]].used==0){
            if(dfs(E[G[v][i]].to)){
                E[G[v][i]].used=1;
                return 1;
            }
        }
```

[13] presnejšie napríklad z B_{out} do A_{out}

```
    }
    for(unsigned int i=0;i<GB[v].size();i++){
        if(E[GB[v][i]].used==1){
            if(dfs(E[GB[v][i]].from)){
                E[GB[v][i]].used=0;
                return 1;
            }
        }
    }
    return 0;
}

void write(){
    int v=O,nv;
    while(v!=P+N){//idem po spatnych hranach
        if(v<N)cout<<v+1<<" ";
        for(unsigned int i=0;i<GB[v].size();i++){
            if(E[GB[v][i]].used==1)nv=E[GB[v][i]].from;
        }
        v=nv;
    }
    cout<<P+1<<" ";
    while(v!=A){//idem po normalnych hranach
        if(v<N)cout<<v+1<<" ";
        for(unsigned int i=0;i<G[v].size();i++){
            if(E[G[v][i]].used==1)nv=E[G[v][i]].to;
        }
        v=nv;
    }
    cout<<A+1<<endl;
}

int main(){
    cin>>N>>O>>P>>A>>M;
    O--;P--;A--;
    G.resize(2*N);GB.resize(2*N);

    for(int i=0;i<N;i++){//vytvorim zdvojenie vrcholov
        E.push_back(edge(i,N+i,0));
        G[i].push_back(i);
        GB[N+i].push_back(i);
    }

    for(int i=0;i<M;i++){
        int a,b;
        cin>>a>>b;
        a--;b--;
        E.push_back(edge(a+N,b,0));
        E.push_back(edge(b+N,a,0));
        G[a+N].push_back(E.size()-2);
        GB[b].push_back(E.size()-2);
        G[b+N].push_back(E.size()-1);
        GB[a].push_back(E.size()-1);
    }
    was.resize(N*2,false);
    dest=O;
    int len=dfs(P+N);
    was.resize(0);
    was.resize(N*2,false);
    dest=A;
```

```
    len+=dfs(P+N);
    if(len<2)cout<<"Taka cesta neexistuje:-(\n"; else write();
    return 0;
}
```

9. Ťažká matematika

opravoval Lukáš
(max. 15 bodov)

Vyskytli sa len dva typy správnych riešení. Jeden typ boli exponenciálne so zložitosťou $O(2^k)$ prípadne $O(\varphi^k)$, kde $\varphi \approx 1.618$. Tieto exponenciálne riešenia skúšali všetky možnosti resp. v druhom prípade skoro všetky možnosti rozostavenia váh a dalo sa za ne získať okolo 6 bodov. Druhý typ riešení boli riešenia v čase $O(k)$. A práve toto riešenie si popíšeme.

Najprv zavedieme dôležité označenia. $a \equiv b \pmod{c}$ znamená, že a dáva po delení číslom c rovnaký zvyšok ako dáva b po delení c. Teda a sa dá zapísať ako $b+k\cdot c$ pre nejaké celé číslo k. Platí napríklad $-3 \equiv 5 \pmod{2}$, keďže obidve čísla sú nepárne. Predpokladajme, že sme vyvážili knihu – na jednu stranu váh sme položili knihu hmotnosti M a závažia dokopy hmotnosti x a na druhú stranu závažia hmotnosti spolu y. Potom platí $M + x = y$ a zároveň v zápise čísel x a y v dvojkovej sústave neexistuje žiadny taký bit, kde by mali obidve čísla jednotku. Znamenalo by to totiž, že závažie hmotnosti 2^i je na oboch stranách váhy.

Keďže platí $M = y - x$, platí aj $M \equiv y - x \pmod{2^n}$. Znamená to, že posledných n bitov čísla $y - x$ sa musí zhodovať s poslednými n bitmi čísla M. A posledných n bitov čísla $y - x$ ovplyvňujú len závažia veľkosti nanajvýš 2^{n-1}. Nech teraz $\hat{x}_n$ je číslo vytvorené z posledných n bitov čísla x a $\hat{y}_n$ vytvorené z posledných n bitov čísla y. Teda $\hat{x}_n \equiv x \pmod{2^n}, \hat{y}_n \equiv y \pmod{2^n}$ a $0 \le \hat{x}_n, \hat{y}_n < 2^n$. Potom platí $-2^n + 1 \le \hat{y}_n - \hat{x}_n \le 2^n - 1$. Zároveň však musí platiť $M \equiv \hat{y}_n - \hat{x}_n \pmod{2^n}$. Z tohto vyplýva, že pre hodnotu $\hat{y}_n - \hat{x}_n$ sú najviac dve možnosti – jedna možnosť je celé číslo a, pričom $a \ge 0$ a druhá možnosť je $a - 2^n$ (ak $a = 0$, tak táto možnosť nie je platná, lebo $a - 2^n = -2^n < -2^n + 1$). Všimnime si, že hodnoty $a, a - 2^n$ dávajú rovnaký zvyšok po delení 2^n.

Zo závaží nanajvýš veľkosti 2^{n-1} teda vychádzajú dve možnosti $\hat{y}_n - \hat{x}_n \in \{a, a - 2^n\}$, pričom $a \ge 0$. Závažie hmotnosti 2^n môžeme položiť na jednu z dvoch strán váh alebo ho nepoložíme na váhy vôbec. Tým dostaneme 4 možnosti $\hat{y}_{n+1} - \hat{x}_{n+1} \in \{a + 2^n, a, a - 2^n, a - 2^{n+1}\}$. Ako už vieme, z týchto možností môžu nastať len dvojice $(a + 2^n, a - 2^n)$ a $(a, a - 2^{n+1})$, lebo obidve možnosti musia dávať rovnaký zvyšok po delení číslom 2^{n+1}. Ak je $(n + 1)$-ty bit čísla M jedna (tento bit reprezentuje hodnotu 2^n), tak do úvahy prichádza len možnosť $(a + 2^n, a - 2^n)$. Naopak, ak je $(n + 1)$-vý bit nula, tak do úvahy prichádza len možnosť $(a, a - 2^{n+1})$. Ak by sme totiž zvolili v oboch prípadoch opačnú možnosť, neplatilo by $M \equiv \hat{y}_{n+1} - \hat{x}_{n+1} \pmod{2^{n+1}}$.

Zaveďme teraz hodnoty $f(n, 0)$ a $f(n, 1)$. Prvá z nich hovorí, koľkými rôznymi spôsobmi vieme dať na váhy závažia nanajvýš váhy 2^{n-1} a pritom hodnota $\hat{y}_n - \hat{x}_n$ bude nezáporná, teda $\hat{y}_n - \hat{x}_n = a \ge 0$. Hodnota $f(n, 1)$ označuje počet možností, keď $\hat{y}_n - \hat{x}_n$ je záporné, teda $\hat{y}_n - \hat{x}_n = a - 2^n$. Zaujíma nás, ako vyrátať $f(n + 1, 0)$ a $f(n + 1, 1)$ z hodnôt $f(n, 0)$ a $f(n, 1)$. Ak je $(n+1)$-vý bit nula, potrebujeme z dvoch možností $(a, a-2^n)$ vyrobiť možnosti $(a, a-2^{n+1})$ pomocou závažia 2^n. Dá sa to iba takto: $a = a+0, a = (a-2^n)+2^n, a-2^{n+1} = (a - 2^n) - 2^n$. Podobne v druhom prípade z možností $(a, a - 2^n)$ chceme vyrobiť možnosti $(a + 2^n, a - 2^n)$. Dá sa to iba takto: $a + 2^n = a + 2^n, a - 2^n = a - 2^n, a - 2^n = (a - 2^n) + 0$. Pre počet možností v jednotlivých prípadoch preto dostávame:

$$\text{ak je bit } 0: f(n + 1, 0) = f(n, 0) + f(n, 1), \quad f(n + 1, 1) = f(n, 1)$$

$$\text{ak je bit } 1: f(n + 1, 0) = f(n, 0), \quad f(n + 1, 1) = f(n, 0) + f(n, 1)$$

Napríklad rovnosť $f(n+1,1) = f(n,0) + f(n,1)$ vyplýva z toho, že $a - 2^n$ vieme vyrobiť z hodnôt a aj $a - 2^n$. Ešte dodáme, že hodnota $f(0,0)$ je 1 a $f(0,1)$ je 0. Totiž ak sme ešte nepoužili žiadne závažie, vieme dosiahnuť len súčet 0.

Teraz už vieme všetko potrebné na vyrátanie výsledku. Zaujíma nás hodnota $f(k,0)$ – chceme použiť závažia nanajvýš hmotnosti 2^{k-1} a hodnota $\hat{y}_k - \hat{x}_k$ musí byť nezáporná, keďže aj hmotnosť M je nezáporná. Hodnotu $f(k,0)$ vieme ľahko vyrátať v čase $O(k)$. Čo sa pamäťovej zložitosti týka, hodnota $f(k,0)$ závisí od k exponenciálne[14]. Preto na zápis výsledku potrebujeme rádovo $O(k)$ bitov.

Listing programu:

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

int main() {
    int M, k;
    scanf("%d %d", &M, &k);

    int f[32][2];
    f[0][0] = 1;
    f[0][1] = 0;
    for (int i = 0; i < k; i++)
    {
        bool bit = M % 2;
        M /= 2;

        f[i+1][0] = f[i][0];
        f[i+1][1] = f[i][1];
        f[i+1][bit] += f[i][!bit];
    }

    //skontrolujeme, či sa vôbec M dá vyvážiť
    if (M > 0) printf("0\n");
    else printf("%d\n", f[k][0]);
}
```

10. Tentokrát o grafe

opravoval Mišof
(max. 15 bodov)

V mnohých prípadoch, keď sa stretneme s operáciou xor, sa dá využiť jej dôležitá vlastnosť: xor je (na rozdiel od napríklad takého sčítania) bitová operácia. Teda pre každé x platí, že x-tá cifra výsledku závisí len od x-tých cifier xorovaných čísel. Preto ak sa v úlohe vyskytne operácia xor, oplatí sa položiť si otázku: „Nedá sa tá úloha riešiť postupne, zvlášť pre každú cifru?“

Pozrime sa teda, ako to bude vyzerať v našej úlohe. Predstavme si, že sme už doplnili čísla na prázdne papieriky a ideme vyhodnocovať divnosť grafu. Tú dostaneme tak, že postupne spracujeme všetky hrany. A pre každú hranu máme zarátať xor ohodnotení jej vrcholov. Inými slovami, postupne sa pozeráme na jednotlivé (binárne) cifry čísel v dvojici vrcholov, a vždy, keď sa nám líšia, dostaneme za to pokutu. (Tým väčšiu, čím vyšší je rád, kde je chyba, ale na tom teraz nezáleží.)

A sme tam, kde sme chceli byť. Na to, aby sme optimálne zvolili posledné cifry neznámych čísel, nám stačia posledné cifry čísel zadaných. A keď už tieto cifry určíme, nijako nám

[14] Skúste nájsť napríklad vstup, pre ktorú existuje $2^{k/2}$ možností

neovplyvnia zvyšok úlohy. Takto môžeme pokračovať ďalej a ďalej, až kým neurčíme neznáme čísla celé.

Presnejšie, budeme teda 32× riešiť nasledovnú úlohu: Daný je graf, v niektorých vrcholoch sú jednotky, v niektorých nuly, a zvyšné vrcholy sú prázdne. Chceme doplniť do prázdnych vrcholov jednotky a nuly tak, aby divnosť výsledného grafu bola minimálna.

Ak ti doterajšia časť vzorového riešenia povedala niečo nové, tu je dobré miesto prestať ho čítať a skúsiť si riešenie jednoduchšej verzie vymyslieť.

A čože je to v našej ľahšej verzii tá divnosť grafu? Nuž, pre nuly a jednotky sa xor správa jednoducho: xor rovnakých číslic je 0, rôznych 1. Inými slovami, divnosť nášho grafu je jednoducho počet hrán, ktorých konce sú rôzneho typu. Chceme teda vrcholy rozdeliť do dvoch množín (nuly a jednotky) tak, aby počet hrán, ktoré vedú medzi množinami, bol minimálny.

A tu prichádzajú k slovu, prekvapivo, toky v grafoch. Ale poďme pekne po poriadku.

Rozdelenie vrcholov grafu na dve časti voláme *(vrcholový) rez*. Veľkosť rezu je počet hrán, ktoré ho tvoria. Našou úlohou je teda nájsť v zadanom grafe rez, ktorý má dve vlastnosti. Prvá: Všetky zadané jednotky sú v jednej časti a všetky zadané nuly sú v druhej časti. Druhá: Zo všetkých takýchto rezov má ten náš byť najmenší.

Odbočka do sveta tokov

Predstavme si teraz náš graf ako sieť potrubí. Pre jednoduchosť nech môže každou hranou za sekundu pretiecť (hociktorým smerom) liter vody. Samozrejme, keď na dvore len tak pohodíme potrubie, nebude ním tiecť nič. Musíme niekam vodu priviesť. A takisto ju potom musíme z iného miesta potrubia pustiť von.

Vrcholy, kde voda priteká, budeme volať *zdroje*, a vrcholy, kde môže z potrubia vytiecť, *odtoky*.

Všimnime si, že v každom vrchole, ktorý nie je ani zdroj, ani odtok, musí platiť analógia Kirchhoffovho zákona – koľko vody do vrchola za sekundu pritečie, toľko z neho musí aj odtiecť. Ale pozor. Ani tieto podmienky dokopy s rozmiestnením zdrojov a odtokov ešte neurčujú jednoznačne, ako bude tok v danom potrubí vyzerať.

Na oboch obrázkoch je tá istá sieť potrubí. Čierne vrcholy sú zdroje, sivé sú odtoky. Obrázky ukazujú dva rôzne veľké toky. Cez čiarkované hrany voda netečie, cez plné tečie v smere šípky. Vľavo je tok veľkosti 2, vpravo tok veľkosti 3.

Nás bude samozrejme zaujímať, ako má tok vyzerať, aby sme každú sekundu pretlačili cez potrubie vody čo najviac. Takýto tok voláme *maximálny tok*. (Všimnite si, že pre sieť z obrázkov je tok vpravo zjavne maximálny, keďže všetky hrany vchádzajúce do odtokov sú už „plné“.)

Fordov-Fulkersonov algoritmus

Maximálny tok vieme nájsť napríklad takto. Začneme s prázdnou sieťou, kde nikde nič netečie. Postupne budeme hľadať cesty, po ktorých sieťou „poslať“ ďalší a ďalší liter vody. (Tieto cesty budeme volať *zlepšujúce*.) A keď sa nám už ďalšiu cestu nájsť nepodarí, tak máme maximálny tok a vyhrali sme.

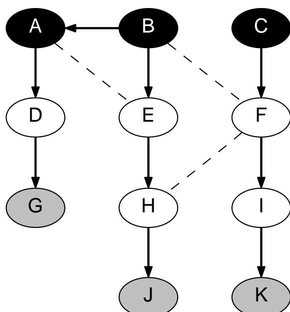
POZOR! Tu si treba uvedomiť, že toto tvrdenie **nie je** samozrejmé. Neprajník Móricko by mohol namietať: „Hm, ale čo ak sme tie cesty hľadali zle, a tým sme si niečo zablokovali? Napríklad pozri na obrázok vľavo, tam bol tok veľkosti 2, a už sa zlepšiť nedá.“ Móricka sklameme – ukážeme, že tok z obrázku sa zlepšiť dá, a že žiaden iný spôsob hľadania nám veľkosť výsledného toku naozaj nezmení.

Trik bude v tom, že v našom toku z obrázka vľavo vieme poslať viac vody nielen po čiarkovaných hranách, ale aj po plných hranách v protismere.

Predstavme si, že by sme v toku z obrázka vľavo zastavili vodu, ktorá tečie z H do F. Čo by sme tým dostali? V H by sme mali liter vody navyše, a v F by nám jeden liter chýbal – presne rovnaký efekt, ako keby sme poslali liter vody z F do H. Inými slovami, rovnaký efekt ako „tečie liter z F do H, aj liter z H do F“ vieme prípustným spôsobom dosiahnuť jednoducho tak, že po hrane FH nepotečie nič.

Keď teda budeme hľadať zlepšujúcu cestu, po ktorej vieme poslať ďalšiu vodu, môžeme používať nie len voľné hrany, ale aj plné hrany v protismere.

V prípade toku z obrázku vľavo príkladom zlepšujúcej cesty je CFHJ. Jej použitím dostaneme tok z nasledujúceho obrázka. (Rôznych zlepšujúcich ciest môže byť viac, iná zlepšujúca cesta v našej situácii by mohla byť napr. AEBFHJ.)



Zlepšujúcu cestu ľahko nájdeme napr. prehľadávaním do šírky, ktoré začína vo všetkých zdrojoch.

Čo sa ale stane, keď už nevieme nájsť ďalšiu zlepšujúcu cestu? Pozrime sa na to, ako dopadlo posledné jej hľadanie. Rozdelíme si vrcholy do dvoch množín: v množine A budú tie, kam sa ešte zlepšujúcou cestou dostať vieme, no a v množine B budú tie, kam sa už dostať nedá.

Všimnime si, že všetky zdroje sú v množine A, a všetky odtoky sú v množine B. Inými slovami, našli sme v našom grafe rez, ktorý oddeľuje zdroje od odtokov.

Aká je veľkosť nami nájdeného rezu? To je ľahké: Spomeňte si, že veľkosť rezu je počet hrán medzi danými množinami vrcholov. Máme teda nejaké hrany, ktoré vedú medzi A a B. A teraz si stačí uvedomiť, že každou z nich už musí tiecť voda z A do B (inak by sme sa do B vedeli dostať zlepšujúcou cestou).

Inými slovami: *Veľkosť práve nájdeného rezu je rovná veľkosti práve nájdeného toku.*

Súvis tokov a rezov

Pozrime sa teraz na situáciu z opačnej strany. Máme graf, v ňom nejaké zdroje a nejaké odtoky. Zvoľme si ľubovoľný rez, ktorý oddeľuje zdroje od odtokov. Nech je veľkosť tohoto rezu k. Potom nech robíme, čo chceme, viac ako k litrov vody za sekundu z prvej množiny do druhej nedostaneme. Preto platí: *Veľkosť ľubovoľného toku je nanajvýš rovná veľkosti každého z rezov (ktoré oddeľujú zdroje od odtokov).*

A z týchto dvoch tvrdení už vyplýva správnosť nášho algoritmu. Keď sa nám minuli zlepšujúce cesty, našli sme rez a tok, ktoré majú rovnakú veľkosť. Žiaden tok už nemôže byť väčší, lebo ho nami nájdený rez nepustí. A naopak, žiaden rez nemôže byť menší, lebo keby existoval menší rez, nepretlačili by sme cez neho takto veľký tok.

Preto náš algoritmus naraz nájde obe veci – aj maximálny tok v zadanom grafe, aj minimálny rez, ktorý oddelí zdroje od odtokov.

A vyriešime si našu úlohu

Chceme teda nájsť minimálny rez, ktorý oddelí zadané jednotky od zadaných núl. Nájdeme ho tak, že zadané jednotky prehlásime za zdroje, zadané nuly za odtoky, a nájdeme vyššie popísaným algoritmom maximálny tok.

V grafe s N vrcholmi a M hranami má zjavne maximálny tok veľkosť nanajvýš rovnú M, lebo po každej hrane môže tiecť najviac jeden liter. Každú iteráciu (nájdenie zlepšujúcej cesty a úprava toku na nej) vieme spraviť prehľadávaním v $O(M+N)$, celkovú časovú zložitosť teda vieme odhadnúť ako $O(M(M+N))$.

Existujú aj lepšie algoritmy na nájdenie toku. V našej situácii (jednotkové kapacity hrán a žiadne násobné hrany) je vhodné použiť napr. Dinicov algoritmus. Ten sa od Fordovho-Fulkersonovho algoritmu líši nasledovným trikom: Keď prebehne prehľadávanie do šírky, môže sa stať, že z jeho výsledkov vieme poskladať aj viac hranovo disjunktných zlepšujúcich ciest, nie len jednu. V každej iterácii preto pažravo nájdeme nejakú maximálnu[15] množinu takýchto zlepšujúcich ciest. Podrobnou analýzou sa dá ukázať, že v uvedených podmienkach má Dinicov algoritmus časovú zložitosť $O(MN^{2/3})$. Podrobnejšie vysvetlenie nájdete v skriptách od Martina Mareše, online na http://mj.ucw.cz/vyuka/ga/.

Program implementuje lenivú ale v praxi rýchlu variáciu na túto tému, niečo na pol ceste medzi Dinicom a Fordom-Fulkersonom.

Listing programu:

```
// another fine solution by misof
#include <iostream>
#include <queue>
using namespace std;
#define REP(i,n) for(int i=0;i<(int)(n);++i)

int N, M, K;  // pocet vrcholov, hran, zadanych hodnot
int cost[600], viem[600];  // aka je hodnota vo vrchole a ci je dana vstupom
int Q[600][600],Qdeg[600];  // povodny graf ako zoznamy okoli
int G[600][600],Gdeg[600];  // kopia grafu upravena na hladanie toku
int C[600][600];  // aktualne volne miesto na hranach
int cut[600],cut_size;  // pole na ulozenie vysledku

void get_flow(int source, int sink) {
  int flow_size = 0;
  int from[600];
  while (1) {
    // prehladavanim do sirky najdeme najkratsiu zlepsujucu cestu
    memset(from,-1,sizeof(from));
    queue<int> Q;
    Q.push(source);
    from[source]=-2;
    while (!Q.empty()) {
      int where = Q.front(); Q.pop();
      for (int i=0; i<Gdeg[where]; i++) {
        int dest = G[where][i];
        if (from[dest] != -1) continue; // tu sme uz boli
        if (C[where][dest] == 0) continue; // nevieme dotlacit vodu
        from[dest] = where; Q.push(dest);
        if (dest == sink) break;
      }
```

[15] V zmysle „nezväčšiteľnú“, nie v zmysle „najväčšiu zo všetkých možných“

```
      if (from[sink] >= 0) break;
    }
    // ak sme ziadnu nenasli, sme hotovi, do pola vyplnime najdeny rez
    if (from[sink]==-1) {
      cut_size=0;
      REP(i,N) if (from[i]!=-1) cut[cut_size++]=i;
      return;
    }
    // zostrojime podla from[] niekolko zlepsujucich ciest
    for (int i=0; i<Gdeg[sink]; i++) {
      int where = G[sink][i];
      if (from[where]==-1) continue; // sem ziadna nevedie
      if (C[where][sink] == 0) continue; // odtialto nevieme ist do odtoku
      // zisti kolko vieme po ceste pretlacit (mozno aj 0)
      int can_push = C[where][sink];
      int curr = where;
      while (1) {
        if (from[curr]==-2) break;
        can_push = min(can_push,C[ from[curr] ][curr]);
        curr=from[curr];
      }
      // znova prejdi po ceste a uprav volne kapacity hran
      flow_size += can_push;
      C[where][sink] -= can_push;
      C[sink][where] += can_push;
      curr = where;
      while (1) {
        if (from[curr]==-2) break;
        C[ from[curr] ][ curr ] -= can_push;
        C[ curr ][ from[curr] ] += can_push;
        curr = from[curr];
      }
    }
  }
}


int main() {
  int x,y;
  cin >> N >> M;
  while (M--) {
    cin >> x >> y;     x--; y--;
    Q[x][y]=Q[y][x]=1;
    Q[x][Qdeg[x]++] = y;  Q[y][Qdeg[y]++] = x;
  }
  cin >> K;
  while (K--) {
    cin >> x >> y;     x--;
    cost[x]=y; viem[x]=1;
  }
  N += 2; // pridame dva vrcholy: fiktivny zdroj a fiktivny odtok
  REP(order,31) {
    // skopirujeme graf do pola G
    REP(i,N) REP(j,Qdeg[i]) {
      G[i][j]=Q[i][j];
      int x = G[i][j];
      C[i][x]=C[x][i]=1;
    }
    REP(i,N) Gdeg[i]=Qdeg[i];
    // pridame do G hrany z fiktivneho zdroja do jednotiek a detto pre odtoky
```

```
    REP(i,N-2) if (viem[i]) {
      if (cost[i] & (1<<order)) {
        C[i][N-1] = C[N-1][i] = 63000;
        G[i][Gdeg[i]++] = N-1;
        G[N-1][Gdeg[N-1]++] = i;
      } else {
        C[i][N-2] = C[N-2][i] = 63000;
        G[i][Gdeg[i]++] = N-2;
        G[N-2][Gdeg[N-2]++] = i;
      }
    }
    // najdeme maximalny tok a minimalny rez
    get_flow(N-1,N-2);
    // neznamym vrcholom nastavime podla najdeneho rezu dalsi bit vystupu
    REP(i,cut_size) if (cut[i]<N-2) if (!viem[cut[i]]) cost[cut[i]] |= 1<<order;
  }
  REP(i,N-2) printf("%d\n",cost[i]);
}
```

Výsledková listina po 2. kole kategórie KSP-Z

Chýba súbor Z!!!

Výsledková listina po 2. kole kategórie KSP-O

Chýba súbor O!!!

Výsledková listina po 2. kole kategórie KSP-T

Chýba súbor T!!!