Korešpondenčný seminár z programovania
XXIII. ročník, 2005/06
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

KSP finančne podporujú: aSc – Applied Software Consultants spol. s r.o.
Gnome spol. s r.o.
MICROSTEP-MIS spol. s r.o.

Vzorové riešenia 2. kola zimnej časti

Milé naše riešiteľky!

Opäť máme pol roka KSP úspešne za sebou, a opäť raz milá povinnosť písať tento úvod pripadla mne. (Hádajte, kto som? ;-)) Nuž, ale mne sa do toho moc nechce, tak budem stručný. (Alebo stručná?) Gratulujeme Pershingáčovi k suverénnej výhre, na najlepšie z vás sa tešíme v dňoch 13. až 19. marca na sústredení, a to už je na dnes naozaj všetko.

Dobrú noc

KSPáci

1. O nemocnici

opravoval Kubo
(max. 15 bodov)

Zdravím. Volám sa Kubo a poviem vám dačo o vzorovom riešení.

Jednoduché riešenie bolo udržiavať si postupnosť čísel zo vstupu utriedenú a v každom kroku len načítať nové číslo čase $O(n)$ ho vložiť na správne miesto (tak ako pri insert-sorte). Vypísať medián, ak máme vstup už utriedený, by potom nemal byť problém ;).

Iné riešenie: To, čo nás v tomto riešení spomaľuje, je utriedené pole, do ktorého „vkladáme“ v $O(n)$. Budeme sa teda musieť poobhliadnuť po dákej lepšej dátovej štruktúre. Čo nejde silou, ide brutálnou silou. Vytiahneme na to kanón – vyvážené stromčeky. V strome si okrem vstupných čísel budeme pre každý vrchol pamätať počet menších a počet väčších čísel (t.j. veľkosť ľavého a pravého podstromu). Túto pomocnú informáciu si vieme (pri vkladaní a pri rotáciách) udržiavať. No a vďaka nej vieme nájsť medián v logaritmickom čase (v čase úmernom hĺbke stromu): postavíme sa do koreňa a ak je menších čísel oveľa viac ako väčších, ideme doľava (medián je menší), naopak, ak je väčších oveľa viac, ideme doprava.

Toto, samozrejme nie je vzorové riešenie. Veď komu sa chce písať vyvažované stromčeky.[1] Vzorové riešenie je oveľa jednoduchšie.

Vzorové riešenie: Pozrime sa na to takto: v každom okamihu si budeme pamätať mediána a zvlášť čísla menšie (v krabičke vľavo) a čísla väčšie ako medián (v krabičke vpravo). Zjavne, aby to bol medián, čísel vľavo musí byť zhruba rovnako ako čísel vpravo[2]. Čo sa stane, keď nám príde nové číslo? Nuž, ak je väčšie ako medián, ide do pravej krabičky, ak je menšie, ide doľava. Problém, ktorý môže nastať, je, že jedna krabička sa nám tak nafúkne, že ich už nebude „zhruba“ rovnako. Teda náš medián už nebude medián.

Otázka na mieste je takáto: Medián našej postupnosti sa z času na čas mení. Ako? Keď sa nad tým na chvíľu zamyslíte, sami prídete na to, že to nemôže byť úplne ľubovoľný prvok. Novým mediánom sa stane buď mediánov predchodca, t.j. najväčšie číslo z čísel menších ako medián (a to v tom prípade, keď je menších čísel príliš veľa), alebo to bude jeho nasledovník (najmenšie väčšie číslo; ak je veľkých veľa). Každopádne, keď si predstavíme

[1] mne

[2] pre detailistov: „zhruba rovnako“ znamená rovnako, alebo tých väčších je o 1 viac

utriedenú postupnosť a pridáme do nej nové číslo, medián sa pohne najviac o 1. To, čo teda potrebujeme, je vedieť vybrať z ľavej krabičky (z menších čísel) to najväčšie číslo a z pravej krabičky (z väčších čísel) to najmenšie číslo.

Takže ešte raz, algoritmus bude vyzerať nejak takto: Na začiatku načítame prvé číslo – to je medián jednoprvkovej postupnosti. Vľavo od neho nie je nič, vpravo tiež nič (vpredu, vzadu tiež nič). Keď príde nové číslo, vložíme ho do príslušnej krabičky: ak je nové číslo menšie ako medián, pôjde vľavo, inak vpravo. Ak je menších čísel zhruba rovnako ako veľkých čísel, medián sa nemení a veselo pokračujeme ďalej. Ak sa nám ale táto krehká rovnosť naruší, musíme „vyvažovať“. Ak je teda menších čísel veľa, vhodíme mediána do pravej krabičky (medzi väčšie čísla) a nový medián bude najväčšie číslo z ľavej krabičky. Naopak, ak je veľkých čísel veľa, vhodíme medián do ľavej krabičky a nový medián je ten najmenší z veľkých.

Čo sa týka implementácie, krabičky vieme implementovať ako haldy. Tým pádom máme vkladanie aj vyberanie minima resp. maxima v čase $O(\log n)$. Celkovo teda dostávame peknú zložitosť $O(n \log n)$.

Lepšiu zložitosť sa nám docieliť ani nepodarí. V opačnom prípade by sme vedeli triediť n čísel asi takto: zadáme n-krát $-\infty$ (dostatočne malé číslo). Potom dodáme n čísel, ktoré chceme triediť (najmenšie číslo z nich je teraz skoro medián). Ďalej pridáme $(2n-1)$-krát $+\infty$ – postupne nám ako výsledok budú vypadávať čísla v utriedenom poradí (každé 2 nekonečná jedno číslo). Keby sme teda spáchali toto lepšie ako $O(n \log n)$, vedeli by sme triediť lepšie ako v $O(n \log n)$ – a to nejde (iba ak v špeciálnych prípadoch, keď čo-to predpokladáme o vstupe, napr. že máme iba celé čísla a pod.).

Hodnotenie: Hodnotenie bolo priaznivé. Za optimálne $O(n \log n)$ riešenie bol plný počet. Veľmi ma potešil dôkaz optimálnosti a schytával ešte bod navyše. Za kvadratické riešenie bolo 7 bodov a za horšie najviac 3. Za zlý popis bolo zlé hodnotenie.

Listing programu:

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

int main() {
  int N, x, m;
  priority_queue<int, vector<int>, greater<int> > vacsie;
  priority_queue<int, vector<int>, less<int> > mensie;
  cin >> N;
  if (N--) { cin >> m; cout << m << ' '; }
  while (N--) {
    cin >> x;
    if (x < m) mensie.push (x);
         else vacsie.push (x);
    if (mensie.size() > vacsie.size()) {
      vacsie.push(m);
      m = mensie.top(); mensie.pop();
    }
    if (vacsie.size() > mensie.size()+1) {
      mensie.push(m);
      m = vacsie.top(); vacsie.pop();
    }
    cout << m << ' ';
  }
  return 0;
}
```

2. O problémoch s prehadzovaním

opravoval Tom
(max. 15 bodov)

Tento príklad sa mi opravoval dosť dobre, keďže prišlo len (asi) 17 riešení. Navyše len približne 7 z nich bolo funkčných. Vám sa tento príklad nepáčil?

Bodovanie bolo jednoduché. Počet bodov za vaše riešenie sa dal vyjadriť v tvare $b(\alpha+\beta)$, kde b je ohodnotenie hlavnej myšlienky vášho riešenia (od 0 do 15), α je za popis (od 0 do 0.6) a β je za implementáciu (od 0 do 0.4).

Za riešenie, ktoré prehľadávalo všetky možnosti bolo $b = 7$, s trochou optimalizácie mohlo byť $b = 8$. Pre funkčné riešenia, ktorých zložitosť nápadne pripomínala zložitosť vzorového riešenia, platí $14 \leq b \leq 15$. No a nefunkčné riešenia mali len $b < 3$.

Pozrime sa teda na vzorové riešenie. Povieme si najprv niečo o permutáciách. Ak máme konečnú množinu X, tak jej permutácia ja každé zobrazenie $\varphi : X \to X$ také, že každý prvok má vzor (teda $\forall y \in X : \exists x \in X : \varphi(x) = y$), alebo inými slovami, žiadne dva prvky sa nezobrazia na ten istý (teda $\forall x, y \in X : x \neq y \Rightarrow \varphi(x) \neq \varphi(y)$).

Keď máme dve permutácie φ, ψ tej istej množiny, môžeme spraviť ich kompozíciu (budeme vravieť aj súčin), teda permutáciu $\varphi\psi$ takú, že $\varphi\psi(x) = \varphi(\psi(x))$. k-násobnú kompozíciu permutácie φ samej so sebou budeme značiť φ^k a volať k-ta mocnina permutácie φ. Špeciálny typ permutácií sú cyklické permutácie. Permutáciu φ nazveme cyklická, ak existuje postupnosť $a_1, a_2, \ldots, a_n$, že $\varphi(a_i) = a_{i+1}$ pre $i = 1, 2, \ldots, n-1$ a $\varphi(a_n) = a_1$ a pre ostatné prvky platí, že $\varphi(x) = x$. No a ešte dve permutácie φ, ψ nazveme disjunktné, ak $\forall x \in X : \varphi(x) = x \vee \psi(x) = x$. Za povšimnutie stojí, že ak sú permutácie φ a ψ disjunktné tak sú komutatívne: $\varphi\psi = \psi\varphi$.

Toľko teoretická príprava, pusťme sa do vzorového riešenia. Všetci, čo ste riešili tento príklad, ste si všimli, že program pre poprehadzovávač je vlastne permutácia miest (preto sme si o nich čo-to povedali). Úlohou je teda k danej permutácii φ a číslu k nájsť takú permutáciu ψ, že $\psi^k = \varphi$, teda nájst k-tu odmocninu z danej permutácie.

Ďalšia vec, ktorú ste si skoro všetci všimli je, že každá permutácia sa dá rozložiť na súčin disjunktných cyklov. (Môžete si premyslieť, prečo je to tak[3]). Niektorí z vás sa snažili odmocniť jednotlivé cykly permutácie na vstupe, ale táto cesta neviedla k správnemu riešeniu. Ako neskôr uvidíme, bolo treba aj niektoré cykly pospájať.

Uvažujme teda permutáciu ψ (tú, ktorú treba nájsť) a jej rozklad na disjunktné cykly. Teda $\psi = c_1 c_2 \ldots c_m$ a $\psi^k = (c_1 c_2 \ldots c_m)^k = c_1^k c_2^k \ldots c_m^k = \varphi$ (lebo c_i sú disjunktné a teda komutujú). Zrejme aj permutácie c_i^k sú navzájom disjunktné. Úlohu budeme riešiť tak, že identifikujeme jednolivé c_i^k a zrekonštruujeme z nich zodpovedajúce c_i. Pozrime sa teda, ako vyzerá také c_i^k. Po nie až tak ťažkej úvahe[4] prídeme k záveru, že c_i^k sa skladá z $\mathrm{nsd}(k, d)$ cyklov dĺžky $d/\mathrm{nsd}(k, d)$, kde d je dĺžka cyklu c_i.

Pozrime sa na to teraz z druhej strany. Máme permutáciu φ, rozložíme ju na disjunktné cykly a tie potrebujeme pospájať do skupiniek zodpovedajúcich jednotlivým c_i^k. Podľa predchádzajúceho v skupinke budú len cykly rovnakej dĺžky. Vezmime si teda cykly dĺžky napríklad l. Hlavnou otázkou je, koľko takých cyklov dať do jednej skupinky. Skúsme napríklad g. To by znamenalo, že týchto g cyklov dĺžky l vzniklo umocnením jedného cyklu dĺžky gl. Ale pritom podľa predchádzajúceho muselo platiť $\mathrm{nsd}(gl, k) = g$. Alebo ekvivalentne $\mathrm{nsd}(l, k/g) = 1$. Veľkosti skupiniek teda nemôžeme voliť ľubovoľne.

Vezmime si teraz nejaký prvočíselný deliteľ p čísla l. Nech aj $p^e \mid k$ (pre vhodné e). Zrejme musí byť aj $p^e \mid g$, lebo inak by bolo $p \mid \mathrm{nsd}(l, k/g)$ a teda $\mathrm{nsd}(l, k/g) \neq 1$. Použitím tejto úvahy na všetky prvočíselné delitele čísla l a maximálne prípustné e dostaneme nasledujúci výsledok: Nech $l = p_0^{e0} p_1^{e1} \ldots p_t^{et}$ a $k = p_0^{f0} p_1^{f1} \ldots p_t^{ft} q$, kde $p_i \not| q$, tak potom ak $\mathrm{nsd}(l, k/g) =$

[3] Stačí si prvky z X nakresliť ako guličky a permutáciu ako šípky medzi nimi a všimnúť si, že z každej guličky vychádza jediná šípka a do každej jediná vchádza.

[4] Opäť si stačí nakresliť guličky a šípky.

1 tak $G = p_0^{f_0} p_1^{f_1} \dots p_t^{f_t} \mid g$. Zároveň ale zrejme $\text{nsd}(l, k/G) = 1$ a teda $\text{nsd}(Gl, k) = G$, teda G je dobrá veľkosť skupinky.

To je už ale návod na vzorové riešenie: Vezmeme si vstupnú permutáciu φ a rozložíme ju na súčin disjunktných cyklov. Potom pre každé prípustné l spracujeme cykly dĺžky l. No a to tak, že preň nájdeme číslo G (ktoré sme vyššie popísali) a overíme, či počet cyklov dĺžky l je deliteľný číslom G. Ak nie, tak požadovaná permutácia ψ neexistuje (to by bolo treba poskupinkovať tie cykly do skupiniek, ktorých veľkosť je násobkom G, ale to nejde). Ak áno, tak tie cykly pospájame do skupiniek po G cyklov a z každej zostavíme jeden cyklus dĺžky Gl.

Takéto riešenie bude mať časovú zložitosť $O(N \log K)$, lebo nájdenie čísla G nám trvá $O(\log K)$ a týchto operácií robíme najviac N. Ostatné veci (skladanie a rozkladanie tých permutácií) robíme v čase $O(N)$.

Listing programu:

```
#include <stdio.h>

#define MAXN 4717

int phi[MAXN];          /*vstupna permutacia*/
int psi[MAXN];          /*vystupna permutacia*/
int cyk[MAXN][2];       /*cykly, [0] je dlzka, [1] je niektory jeho prvok*/
int cyks[MAXN][2];      /*cykly utriedene podla dlzky*/
int len[MAXN];          /*[i] je pocet cyklov dlzky i*/

int N,K,pc;

void nacitaj()  /*nacita vstup*/
{
  int i;
  scanf("%d %d ",&N,&K);  /*nacita N, K*/
  for(i=0;i<N;i++)
  {
    int a,b;
    scanf("%d %d ",&a,&b);  /*a program*/
    phi[a-1]=b-1;
  }
}

void rozloz() /*rozlozi phi na disjunktne cykly*/
{
  int vis[MAXN];
  int i,j;
  for(i=0;i<N;i++){ vis[i]=0; len[i]=0; }
  pc=0;
  for(i=0;i<N;i++)if(!vis[i])
  {
    cyk[pc][0]=1; cyk[pc][1]=i;
    vis[i]=1;
    j=phi[i];
    while(!vis[j]){ cyk[pc][0]++; vis[j]=1; j=phi[j]; }
    len[cyk[pc][0]]++;
    pc++;
  }
}

void utried() /*utriedi cykly podla dlzky*/
{
```

```
  int pos[MAXN];
  int i;
  pos[0]=0;
  for(i=1;i<N;i++)pos[i]=pos[i-1]+len[i-1];
  for(i=0;i<pc;i++)
  {
    cyks[pos[cyk[i][0]]][0]=cyk[i][0];
    cyks[pos[cyk[i][0]]][1]=cyk[i][1];
    pos[cyk[i][0]]++;
  }
}

int G(int l)  /*zrata G*/
{
  int i,kk=K;
  for(i=2;i<=l;i++)if(l%i==0)while(kk%i==0)kk/=i;
  return K/kk;
}

int zostav(int s,int g,int l)  /*zostavi velky cylkus poskladanim g cylkov z pola cyks
pocnuc tym s-tym*/
{
  int tmp[MAXN];
  int i,j,k;
  int Gl=g*l;

  for(i=0;i<g;i++)
  {
    k=cyks[s+i][1];
    for(j=0;j<l;j++){ tmp[(i+j*K)%Gl]=k; k=phi[k]; }
  }
  for(i=0;i<Gl;i++)psi[tmp[i]]=tmp[(i+1)%Gl];
}

int vypis()
{
  int i;
  for(i=0;i<N;i++)printf("%d -> %d\n",i+1,psi[i]+1);
}

int main()
{
  int i,j;
  nacitaj(); rozloz(); utried();
  i=0;
  while(i<pc)  /*ideme po cykloch*/
  {
    int l=cyks[i][0];  /*vezmeme si cykly dlzky l*/
    int g=G(l);        /*zratme si g*/
    if(len[l]%g!=0)    /*ak je zle tak koncime*/
    {
      printf("neda sa !!\n");
      return 7;
    }
    for(j=0;j<len[l]/g;j++)zostav(i+j*g,g,l); /*pospajame*/
    i+=len[l];
  }
  vypis();
  return 0;
}
```

3. O Rômeovi a Juliäi

opravoval Lukáš
(max. 15 bodov)

Musím sa priznať, že som čakal viac správnych riešení. Nebol to totiž až taký ťažký príklad. Za riešenia bežiace v čase $O(N^2)$ alebo $O((N+M)\log N)$ som dával 15 bodov. Za riešenia bežace v čase $O(N!)$ ste mohli získať 8 bodov a za nefunkčné najviac 3 body.

Aby sme vedeli určiť, po ktorých hranách grafu môžu Rômeo a Juliä ísť spolu, musíme poznať najkratšie vzdialenosti od ich domov do zvyšku grafu. Na to môžeme použiť napríklad Dijkstrov algoritmus. Ten vieme rozumne implementovať[5] v čase $O(N^2)$ alebo $O((N+M)\log N)$ s haldou. Za obidve implemetácie som dával rovnako veľa bodov.

Teraz si popíšeme Dijkstrov algoritmus. Ak ho poznáte, môžete preskočiť pár odstavcov. Nech s je vrchol, z ktorého chceme nájsť najkratšie cesty do zvyšku grafu. V každom vrchole v si budeme pamätať dĺžku doteraz nájdenej najkratšej cestu zo štartu s. Vrcholy budeme mať rozdelené do dvoch skupín: V prvej skupine budú vrcholy (pracovne si ich označme ako ofarbené), pre ktoré už vieme skutočnú najkratšiu cestu od štartu. V druhej skupine budú všetky ostatné vrcholy (tie nebudú zatiaľ ofarbené), pre ktoré budeme poznať dĺžku najkratšej cesty, ktorá vedie len cez ofarbené vrcholy, prípadne u nich budeme mať zaznačené, že taká cesta neexistuje.

Náš algoritmus bude fungovať tak, že si vždy vyberieme nejaký neofarbený vrchol v, ten ofarbíme a aktualizujeme údaje o najkratších cestách ostatným vrcholom. Keď ofarbíme aj cieľ, tak budeme pre neho vedieť najkratšiu cestu od štartu. Pozrime sa lepšie na taký neofarbený vrchol v, ktorý je najbližšie k štartu. Vieme z neho najkratšiu takú cestu o_v zo štartu, ktorá ide len cez ofarbené vrcholy. Vtip je v tom, že cesta, ktorá ide aj cez neofarbené vrcholy určite nie je kratšia. Ako vyzerá najkratšia cesta d_v, ktorá môže viesť cez ľubovolné vrcholy? Tvrdíme, že dĺžka o_v je rovnaká ako dĺžka d_v. Rozoberme dva prípady:

- d_v prechádza len cez ofarbené vrcholy. Najkratšia taká cesta je o_v a d_v od nej nemôže byť dlhšia.
- d_v prechádza aj cez nejaký neofarbený vrchol u, tzn. d_v má tvar: $s, \ldots, u, \ldots v$. To ale znamená, že neofarbený vrchol u je bližšie[6] k štartu ako v, čo je v spore s tým, že v je najbližší taký vrchol, teda tento prípad nemôže nastať.

Zistili sme, že už vlastne poznáme najkratšiu cestu od štartu po v, teda môžeme v pokojne ofarbiť. Teraz musíme ešte aktualizovať údaje o najkratších cestách nejakým iným neofarbeným vrcholom, pretože sme zmenili množinu ofarbených vrcholov. Všimnime si, že najkratšia cesta od štartu idúca len po označených vrcholoch sa mohla zmeniť len susedom vrchola v. Takže ich musíme všetkých prejsť a pozrieť sa, či cesta prechádzajúca cez vrchol v nie je kratšia ako doteraz nájdená.

Predpokladajme, že sme Dijkstrovym algoritmom našli najkratšie cesty z Rômeovho domu a z Juliäinho domu. Potom vieme zostrojiť nový graf, po ktorom môžu ísť spolu Rômeo a Juliä. Vieme totiž pre každú hranu v pôvodnom grafe povedať, či sa po nej obidvaja priblížia k svojim domom. Zamyslime sa trochu lepšie, ako tento graf bude vyzerať. Každá hrana bude orientovaná, lebo len jedným smerom sa môžeme priblížiť k obidvom domom. Navyše, graf bude acyklický. Toto tvrdenie dokážeme sporom. Nech v grafe je cyklus $v_1, v_2, \ldots, v_n, v_1$. Nech $r(v_i)$ je vzdialenosť i-teho vrcholu od domu Rômea, $j(v_i)$ je vzdialenosť od domu Juliäi. Potom $r(v_1) > r(v_2) > \ldots > r(v_n) > r(v_1)$ a $j(v_1) > j(v_2) > \ldots > j(v_n) > j(v_1)$. Vyšlo nám, že $r(v_1) > r(v_1)$ a $j(v_1) > j(v_1)$, čo je spor.

[5] Fibonacciho haldu za rozumnú implementáciu nepovažujem :-)

[6] toto platí za predpokladu, že hrany v grafe sú nezáporné

Keďže graf je acyklický, najdlhšiu cestu v ňom vieme nájsť veľmi rýchlo. Konkrétne v čase $O(N+M)$ pomocou prehľadávania do hĺbky. Nech $s_1, \ldots, s_k$ sú synovia vrcholu v. Uvedomme si, že najdlhšie cesty začínajúce v synoch vrchola v nebudú prechádzať cez vrchol v a ani cez žiadneho predka vrchola v. Toto nám zaručuje acyklickosť grafu. Spravíme preto nasledovnú vec: najprv nájdeme najdlhšie cesty, ktoré sa začínajú vo vrcholoch $s_1, \ldots, s_k$ a pomocou nich nájdeme dĺžku najdlhšej cesty pre vrchol v. Označme $l(s_i)$ dĺžku najdlhšej cesty pre syna s_i. Potom dĺžka najdlhšej cesty $l(v)$ pre vrchol v bude $l(v) := \max\{d(v, s_i) + l(s_i) \mid 1 \leq i \leq k\}$, kde $d(v, s_i)$ je dĺžka hrany medzi vrcholmi v a s_i.

Najdlhšia cesta sa dá hľadať aj tak, že si acyklický graf najprv topologicky utriedime a potom dynamickým programovaním nájdeme najdlhšiu cestu. Riešenie pomocou prehľadávanie do hĺbky je však kratšie a jednoduchšie.

Najpomalšia časť nášho algoritmu je hľadanie najkratších vzdialeností v grafe, preto celkovo pracuje v čase $O(N^2)$. Pamäťová zložitosť je vďaka spájanému zoznamu $O(N + M)$.

Listing programu:

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

typedef pair<int, double> hrana;
vector<double> vz[2]; // vz[0] sú najkratšie vzdialenosti pre Rômea, vz[1] pre Juliäu
vector<double> najdlh;
list<hrana> l[100];

double najdlhsia(int vrchol) {
    if (najdlh[vrchol]!=-1) return najdlh[vrchol];
    //už sme tu raz boli, preto len vrátime zapamätanú hodnotu

    double pom=0;
    for (list<hrana>::iterator g=l[vrchol].begin(); g!=l[vrchol].end(); g++)
        if (vz[0][vrchol]>vz[0][g->first] && vz[1][vrchol]>vz[1][g->first])
        //skontrolujeme, či sa obidvaja priblížia k svojim domom
            pom=max(pom, najdlhsia(g->first)+g->second);
    return najdlh[vrchol]=pom;
}

int main() {
    int n, m; cin>>n>>m;
    int a, b, c; cin>>a>>b>>c; a--; b--; c--;

    for (int i=0; i<m; i++) {
        int x, y; double z;
        cin>>x>>y>>z;
        x--; y--;
        l[x].push_back(hrana(y, z));
        l[y].push_back(hrana(x, z));
    }

    for (int k=0; k<2; k++) {
        //Dijkstrov algoritmus pre Rômea a Juliäi
        vz[k].assign(n, 10000000);
        if (k==0) vz[k][a]=0;//Rômeo
        else vz[k][b]=0;//Juliä
        vector<bool> bol(n, false);
```

```
        for (int j=0; j<n; j++) {
            int naj=-1;
            for (int i=0; i<n; i++) if (!bol[i])
                if (naj==-1 || vz[k][naj]>vz[k][i]) naj=i;
            bol[naj]=true;

            for (list<hrana>::iterator g=l[naj].begin(); g!=l[naj].end(); g++)
                vz[k][g->first]=min(g->second+vz[k][naj], vz[k][g->first]);
        }
    }

    najdlh.assign(n, -1);
    cout<<najdlhsia(c)<<endl;//už len vypíšeme najdlhšiu cestu z vrcholu c
}
```

4. O klinci

opravoval WSX
(max. 15 bodov)

Táto úloha patrila medzi tie ľahšie. Pozostávala z dvoch podúloh:

- Nájdenie ťažiska mnohouholníka: za optimálne riešenie s časovou zložitosťou $O(N)$ bolo možné získať maximálne 10 bodov, za riešenie pracujúce v čase $O(N \log N)$ 7 bodov a za ľubovoľné pomalšie fungujúce riešenie 5 bodov. Body sa dali stratiť za nedostatočný popis (až 5 bodov) a za ďalšie chyby v riešení (podľa závažnosti).
- Overenie, či sa nájdené ťažisko nachádza vo vnútri mnohouholníka (5 bodov): za funkčné riešenie (v čase $O(N)$) ste mohli získať maximálne 5 bodov. Body ste takisto ako v prvej podúlohe mohli stratiť za nedostatočné vysvetlenie vášho riešenia (až 3 body) a za ďalšie chyby – najmä riešenie okrajových prípadov.

A teraz k samotnému riešeniu úlohy: Predstavme si, že daný mnohouholník nám niekto rozrezal na trojuholníky. Keďže pliešok je homogénny a rovnako hrubý, hmotnosť každého trojuholníka je priamo úmerná k jeho obsahu – ten ľahko vypočítame pomocou vektorového súčinu. Reprezentujme si každý trojuholník hmotným bodom ležiacim v ťažisku trojuholníka a s hmotnosťou trojuholníka. Potom ťažisko celého pliešku sa nachádza v ťažisku týchto hmotných bodov. Ľahko sa dá ukázať, že ťažisko trojuholníka leží v aritmetickom priemere jeho vrcholov (nezávisle po súradniciach). Z fyziky ďalej vieme, že spoločné ťažisko dvoch hmotných bodov leží s nimi na jednej priamke a z momentovej vety zase vieme, že pomer vzdialeností hmotných bodov od ťažiska je rovnaký ako pomer ich hmotností. Analytickou geometriou sa dá ľahko ukázať, že pre polohu ťažiska T dvoch hmotných bodov A, B o hmotnostiach m_A, m_B platí vzťah:

$$T_x = \frac{m_A A_x + m_B B_x}{m_A + m_B}$$

$$T_y = \frac{m_A A_y + m_B B_y}{m_A + m_B}$$

Vektorovo:

$$T = \frac{m_A A + m_B B}{m_A + m_B}$$

Teda hmotné body A, B o hmotnostiach m_A, m_B vieme reprezentovať jedným hmotným bodom T o hmotnosti $m_A + m_B$. Matematickou indukciou ľahko dostaneme všeobecný vzťah pre polohu ťažiska n hmotných bodov $B_1, B_2, \ldots, B_n$:

$$T = \frac{\sum_{i=1}^{n} m_{B_i} B_i}{\sum_{i=1}^{n} m_{B_i}}$$

Ale keďže nám nikto nepovedal, ako rozrezať náš mnohouholník (tvorený vrcholmi V_1, V_2, …, V_n) na trojuholníky, musíme si pomôcť sami. Ak by bol tento mnohouholník konvexný, je to jednoduché – stačí si zvoliť ľubovoľný bod B ležiaci vnútri mnohouholníka a zobrať do úvahy trojuholníky tvorené trojicami vrcholov (B, V_1, V_2), (B, V_2, V_3), …, (B, V_{n-1}, V_n), (B, V_n, V_1). Uvažujme, čo by sa ale stalo, ak by bod B ležal mimo mnohouholníka. Plocha patriaca mnohouholníku by bola v týchto trojuholníkoch započítaná raz, avšak plochu medzi bodom B a mnohouholníkom by sme započítali dvakrát. Ľahko sa však dá všimnúť, že túto plochu raz započítavame v smere pohybu hodinových ručičiek a druhý krát v opačnom smere. Stačilo by nám teda pravotočivé trojuholníky pripočítavať a ľavotočivé odpočítavať. Presne to nám však poskytuje vzťah na výpočet obsahu trojuholníka pomocou vektorového súčinu:

$$S_{ABC} = \frac{(B_x - A_x)(C_y - A_y) - (C_x - A_x)(B_y - A_y)}{2}$$

V tejto chvíli môže mať pozorný čitateľ morálny problém s odčítavaním trojuholníkov. Keďže však momentová veta platí aj pre záporné hmotnosti (pôsobenie sily nahor), stačí matematicky overiť, či horeuvedené vzťahy fungujú aj pre odčítavanie.

Tento princíp pričítavania a odčítavania trojuholníkov teraz využijeme pri hľadaní ťažiska nekonvexného mnohouholníka. Ukážeme, že pre ľubovoľný bod B trojuholníky vyššie popísaného tvaru pokrývajú celý mnohouholník podobne ako pri konxexných mnohouholníkoch v absolútnej hodnote práve raz (ak by boli body mnohouholníka zadané v smere hodinových ručičiek, dostali by sme výslednú plochu zápornú). Bez ujmy na všeobecnosti, nech vrcholy mnohouholníka na vstupe sú zadané v pravotočivom poradí. Uvažujme ľubovoľný uhol v bode B pretínajúci mnohouholník a zároveň neobsahujúci žiadny jeho vrchol. Označme strany mnohouholníka, s ktorými má prienik, v poradí smerom k bodu B, $s_0, s_1, \ldots, s_{n-1}$, teda časti medzi stranami s_{2k} a s_{2k+1} vymedzené uhlom patria mnohouholníku. Ľahko sa dá vidieť, že úsek vymedzený stranami s_k a s_{k+1} bol započítaný $k+1$ krát, striedavo pripočítavaný a odpočítavaný, čo znamená, že úsek medzi s_k a s_{k+1} bol započítaný práve $(k+1) \bmod 2$ krát, teda úseky započítané v ťažisku sú len tie, kde k je párne, teda patriace mnohouholníku.

Pre jednoduchú implementáciu popísaného algoritmu položíme $B = [0, 0]$, čím sa počítanie obsahu trojuholníka (B, V_i, V_{i+1}) zjednoduší na polovicu vektorového súčinu bodov V_i a V_{i+1} považovaných za vektory. Keďže trojuholníkov je N a pre každý trojuholník algoritmus vykoná konštantne veľa práce, časová zložitosť je $O(N)$. Za predpokladu, že vrcholy mnohouholníka už máme uložené v pamäti, je zvyšná pamäťová zložitosť $O(1)$. Aj napriek tomu, že vrcholy mnohouholníka vieme spracovávať postupne, v pamäti ich potrebujeme pre druhú časť úlohy – zisťovanie, či sa nájdené ťažisko nachádza vnútri mnohouholníka.

Túto podúlohu je možné riešiť viacerými spôsobmi. Asi najelegantnejší je nasledovný: Predpokladajme, že máme daný bod B (polohu ťažiska z prvej podúlohy) a vrcholy V_1, V_2, …, V_n mnohouholníka. Zadefinujme U_{ABC} ako uhol zvieraný medzi polpriamkami BA a BC, pričom ak bod C je naľavo od polpriamky BA, veľkosť uhlu je záporná (interval $(-\pi, 0)$), v opačnom prípade je kladná (interval $\langle 0, \pi \rangle$). Inými slovami, ak prechádzame k bodu proti smeru hodinových ručičiek, uhol sa bude zmenšovať, v smere hodinových ručičiek sa bude zväčšovať.

Pre konvexný mnohouholník platí nasledovné tvrdenie: Ak sa bod B nachádza vnútri mnohouholníka, platí, že suma uhlov $U_{V_i B V_{i+1}}$ je 2π v prípade pravotočivého mnohouholníka a -2π v prípade ľavotočivého (začneme vrcholom V_1 a postupne obídeme celých 2π jedným zo smerov kým prídeme naspäť do vrcholu V_1). Ak sa B nachádza mimo mnohouholníka, táto suma bude nulová (je možné nájsť taký vrchol, ktorým keď začneme, pôjdeme najskôr v

smere hodinových ručičiek (suma sa bude zväčšovať) a po určitom čase zmeníme smer (suma sa bude zmenšovať až sa dostaneme do vrcholu v ktorom sme začali).

Toto tvrdenie však platí aj pre nekonvexný mnohouholník, čo dokážeme. V každom nekonvexnom mnohouholníku existuje rez (úsečka medzi dvomi jeho vrcholmi nepretínajúca žiadnu stranu mnohouholníka), ktorý mnohouholník rozdelí na dva mnohouholníky. Opakovaním konečného počtu týchto rezov dostaneme konečný počet konvexných mnohouholníkov.

Tento fakt využijeme pri dôkaze našej vety matematickou indukciou vzhľadom na štruktúru mnohouholníka. Pre konvexné mnohouholníky sme už vetu dokázali. Majme nekonvexný mnohouholník M, ktorý úsečkou V_iV_j rozdelíme na mnohouholníky M_1 a M_2. Bez ujmy na všeobecnosti, nech M_1 obsahuje vrcholy $V_1, V_2, \ldots, V_i, V_j, V_{j+1}, \ldots, V_n$ a M_2 vrcholy $V_i, V_{i+1}, \ldots, V_j$. Z indukčného predpokladu veta platí pre mnohouholníky M_1 a M_2. Zrejme platí, že suma uhlov pre mnohouholník M je:

$$S_M = (S_{M_1} - U_{V_iBV_j}) + (S_{M_2} - U_{V_jBV_i}) = S_{M_1} + S_{M_2} - (U_{V_iBV_j} + U_{V_jBV_i}) = S_{M_1} + S_{M_2}$$

Keďže bod B sa nemôže súčasne nachádzať v oboch mnohouholníkoch M_1 aj M_2, S_M bude nadobúdať hodnoty $\pm 2\pi$ práve vtedy, keď sa bude nachádzať vo vnútri mnohouholníka M. Tým je veta dokázaná.

Časová zložitosť algoritmu je $O(N)$, keďže pre každú stranu mnohouholníka potrebujeme konštatne veľa výpočtových operácií.

Listing programu:

```
#include <math.h>
#include <vector>

using namespace std;

#define EPS 1E-8

class Vrchol {
 public:
  double x, y;

  /* Konstruktor vektoru */
  Vrchol (double x, double y) { this->x = x; this->y = y; }

  /* Dlzka vektoru */
  double dlzka () { return hypot (this->x, this->y); }

  /* Uhol vektoru */
  double uhol () { return atan2 (this->y, this->x); }

  /* Scitanie dvoch vektorov */
  Vrchol operator+ (Vrchol v) { return Vrchol (this->x + v.x, this->y + v.y); }

  /* Odcitanie dvoch vektorov */
  Vrchol operator- (Vrchol v) { return Vrchol (this->x - v.x, this->y - v.y); }

  /* Vynasobenie vektora konstantou */
  Vrchol operator* (double r) { return Vrchol (this->x * r, this->y * r); }

  /* Skalarny sucin vektorov */
  double operator* (Vrchol v) { return this->x * v.y - this->y * v.x; }
};
```

```
int main() {
  int n;
  vector<Vrchol> v;

  scanf ("%d ", &n);

  for (int i = 0; i < n; i++) {
    double x, y;

    scanf ("%lf %lf ", &x, &y);
    v.push_back (Vrchol (x, y));
  }

  v.push_back (v[0]); /* v[n] = v[0] */

  double obsah = 0;
  Vrchol tazisko (0, 0);

  for (int i = 0; i < n; i++) {
    double tobsah = (v[i] * v[i + 1]) / 2;

    obsah += tobsah;
    tazisko = tazisko + (v[i] + v[i + 1]) * (tobsah / 3);
  }

  tazisko = tazisko * (1 / obsah);

  printf ("Tazisko sa nachadza na [%.2f, %.2f], ", tazisko.x, tazisko.y);

  for (int i = 0; i < n; i++)
    if ((v[i] - tazisko).dlzka () < EPS) {
      printf ("lezi v bode mnohouholnika [%.2f, %.2f].\n", v[i].x, v[i].y);
      return 0;
    }

  double uhol = 0;

  for (int i = 0; i < n; i++) {
    /* Orientovany uhol medzi v[i], tazisko, v[i + 1] */
    double uuhol = (v[i + 1] - tazisko).uhol () - (v[i] - tazisko).uhol ();

    /* Normalizacia na interval <-PI, PI> */
    if (uuhol < M_PI)
      uuhol += 2 * M_PI;
    if (uuhol > M_PI)
      uuhol -= 2 * M_PI;

    /* Ak je uhol M_PI, tazisko lezi na hrane */
    if (M_PI - fabs (uuhol) < EPS) {
      printf("lezi na hrane [%.2f, %.2f]-[%.2f, %.2f]\n", v[i].x, v[i].y, v[i+1].x, v[i+1].y);
      return 0;
    }

    uhol += uuhol;
  }

  if (fabs (uhol) < EPS)
    printf ("lezi mimo mnohouholnika.\n");
   else
    printf ("lezi vnutri mnohouholnika.\n");
```

```
  return 0;
}
```

5. Opäť dešifrovať?!

opravoval MišoF.
(max. 12 bodov)

Úspešne šifru vyriešili len traja riešitelia. Trochu ma to sklamalo, lebo v podstate nešlo o nič ťažké. Celý potrebný postup bol v zadaní... a vo vzorovom riešení príkladu 5 z minulej série.

Ako na vec?

Začneme tým, že si napíšeme funkciu, ktorá bude hodnotiť, nakoľko nejaký text „znie dobre“ v danom jazyku. Budeme to robiť presne podľa popisu v zadaní – spravíme si štatistiku tetragramov z nejakého (z webu poťahaného, dostatočne veľkého) textu v danom jazyku, a potom pre daný text jednoducho spočítame v zadaní definované „prekvapenie“.

Majúc túto funkciu, už stačí len nájsť správny kľúč. Tu príde k slovu *hill climbing*. Začneme z nejakého náhodného kľúča, teda náhodnej permutácie písmen. Vždy, keď máme nejaký kľúč, vyskúšame v ňom spraviť nejaké drobné zmeny (vhodnou zmenou je napríklad vymeniť dvojicu písmen), a pre každú pozrieme, či dešifrovaný text vyzerá lepšie. Ak sa nám podarilo nájsť lepší kľúč, pokračujeme ďalej s ním. Inak znovu začneme hľadať z nejakej inej náhodnej permutácie.

Prax ukáže, že tento postup, nakŕmený správnou štatistikou tetragramov, dokáže ľubovoľnú rozumne dlhú (150+ znakov) substitučnú šifru rozbiť v priebehu sekúnd.

Prečo je to tak? Všimnime si náhodný kľúč. Ak sme trafili trebárs len 2–3 častejšie písmená, na miestach, kde sú pri sebe, začínajú vznikať zrozumiteľné tetragramy. Postup hill climbingu potom postupne správne doplní písmená, ktoré sa v šifrovom texte nachádzajú v ich blízkosti, a tak ďalej.

Už zostávalo len správne si tipnúť jazyk (bola to nemčina, uhádnuť sa to dalo z neuveriteľne veľkej frekvencie najčastejšieho písmena) a bolo hotovo. Ešte dodám, že výsledný text začínal slovenskou vetou, aby to nebolo také ľahké.

Jedzte mrkvu, je zdrava.

Die Forscher glauben, dass der Effekt ein bereits bekanntes Phanomen widerspiegelt: Je schwieriger eine Aufgabe ist, desto starker verzerrt sich die Wahrnehmung. Dem Wanderer etwa erscheint ein Ziel weiter entfernt oder ein Hugel sehr viel steiler, wenn er einen schweren Rucksack schleppen muss. Fur Reich und Arm gilt dasselbe: Eine Munze erscheint einem Reichen sehr viel kleiner als einem Armen.

Fur Baseballspieler gilt also: Bei einer Pechstrahne steigert sich das Gefuhl, sich mehr anstrengen zu mussen, weshalb der Ball schrumpft. Die Frage, die alle Neider umtreibt, bleibt allerdings offen: Was war zuerst da? Der grosse Ball oder der Erfolg? Denn ob die Treffsicheren so gut sind, weil der Ball fur sie schon immer grosser war, oder erst der Erfolg den Ball wachsen lasst, wissen auch die Psychologen nicht.

A tak to je aj v súťažiach v programovaní. Dôležitým krokom k úspechu je veriť si. A mrkva prospieva očiam :-)

Listing programu:

```
// another fine solution by misof
#include <iostream>
#include <cstdio>
#include <cmath>
#include <cctype>
using namespace std;
```

```
#define REP(i,n) for(__typeof(n) i=0;i<(n);++i)
#define CHARS 26

int tetra[32][32][32][32]; // statistika tetragramov
char CT[10000]; // sifrovy text
int CL; // dlzka sifroveho textu
char key[32] = "abcdefghijklmnopqrstuvwxyz"; // aktualny kluc

void loadTetragrams() // nacita statistiku tetragramov {{{
{
  FILE *ftetra = fopen("tetra.txt","r");
  if (!ftetra) { printf("Can't find tetragram stats -- tetra.txt\n"); exit(1); }
  while (!feof(ftetra)) {
    char buf[16]; int x; int A[4];
    fscanf(ftetra,"%s %d ",buf,&x);
    REP(i,4) A[i] = buf[i]-'a';
    tetra[A[0]][A[1]][A[2]][A[3]] += x;
  }
  fclose(ftetra);
} // }}}

void loadCiphertext() /// nacita sifrovy text {{{
{
  string res; char ch;
  CL=0;
  while (1) {
    if (scanf("%c",&ch) != 1) break;
    if (!isalpha(ch)) continue;
    ch = tolower(ch);
    CT[CL++] = ch;
  }
  CT[CL]=0;
} // }}}

double score(char *key) // zhodnoti ako dobre znie plaintext pre dany kluc {{{
{
  double res = 0.0;
  if (CL<4) return res;
  REP(i,CL-3) {
    int A[4];
    REP(j,4) A[j]=key[ int(CT[i+j]-'a') ]-'a';
    res += log( tetra[A[0]][A[1]][A[2]][A[3]] + 1.0 );
  }
  return res;
} // }}}

string decode(const string &CT, const string &key) // dekoduje podla kluca {{{
{
  string res;
  REP(i,CT.size()) res += key[ int(CT[i]-'a') ];
  return res;
} // }}}

string inverseKey(const string &key) // spocita inverznu permutaciu kluca {{{
{
  string res(CHARS,' ');
  REP(i,CHARS) res[ int(key[i]-'a') ] = 'a'+i;
  return res;
} // }}}
```

```
int main() {
  srand(time(NULL));
  loadTetragrams();
  loadCiphertext();
  double best = -1.0;
  random_shuffle( key, key+CHARS );

  while (1) {
    double localbest = -1.0;
    double current = score(key);
    int bi,bj;

    // try all swaps
    REP(i,CHARS) REP(j,i) {
      swap(key[i],key[j]);
      double toto = score(key);
      if (toto > localbest) {
        localbest = toto;
        bi = i; bj = j;
      }
      swap(key[i],key[j]);
    }

    // check for local improvement
    if (localbest > current) {
      swap(key[bi],key[bj]);

      // check for global improvement
      if (localbest > best) {
        best = localbest;

        printf("\nnew best solution with value: %10.3f\n",best);
        string S1=key, S2=inverseKey(key), S3=decode(CT,key);
        printf("key: '%s'\ninverse key: '%s'\n\nSOLUTION: '%s'\n",
          S1.c_str(),S2.c_str(),S3.c_str());
      }
    } else {
      // do some changes
      if (rand()&15 <= 7) {
        random_shuffle( key, key+CHARS );
      } else {
        REP(i,5) {
          int x=0,y=0;
          while (x==y) { x = rand()%CHARS; y = rand()%CHARS; }
          swap(key[x],key[y]);
        }
      }
    }
  }
  return 0;
}
// vim: fdm=marker:commentstring=\ \"\ %s:nowrap:autoread
```

Výsledková listina po 2. kole kategórie KSP

	Meno a priezvisko	Škola	Trieda		21	22	23	24	25	Σ
1	Perešíni Peter	Gym. Tajovského B. Bystrica	4	72	16	15	15	15	11	144
2	Danilák Michal	Gym. Hubeného BA	3	59	15	2	15	11	12	114
3	Bílka Ondřej	Gym. Zlín	4	65	10	10	12	10		107
4	Imriška Jakub	Gym. Jura Hronca BA	4	57	15	14	3	14		103
5	Fedák Matúš	Gym. Stará Ľubovňa	4	58	15	7	15	3	3	101
5	Herman Peter	Gym. Jura Hronca BA	3	44	15	7	15	8	12	101
7	Bundala Daniel	Gym. Jura Hronca BA	4	57	10	8	15	6		96
8	Mikuláš Ján	Gym. Haličská Lučenec	4	40	15	1	15	15		86
9	Rampášek Ladislav	Gym. Jura Hronca BA	3	39	15		3	12	3	72
10	Králik Martin	Gym. Grösslingová BA	4	45	7		6	1	6	65
11	Petruchová Zuzana	Gym. Grösslingová BA	4	33	15	2	8			58
12	Kováč Michal	Gym. Grösslingová BA	4	26	8	2	8	5	5	54
12	Mikuláš Ondrej	Gym. Haličská Lučenec	3	35	7	2		10		54
14	Jerguš Ján	Gym. Alejová Košice	3	31	7	7	8			53
15	Brezáni Samuel	Gym. Rajec	3	30	7	2	1	10		50
16	Ďuďák Juraj	Gym. Golianova Nitra	4	35	6		2	5		48
17	Windisch Martin	Gym. Vrútky	4	45						45
18	Tomcsányi György	Gym. H. Selyeho Komárno	3	25	6	2		8		41
19	Tříska Martin	Gym. P. de Coubertina Piešťany	3	20	7	2	3	5	3	40
20	Pavlíček Tomáš	SPŠE Piešťany	3	37						37
21	Pančík Andrej	Gym. Tajovského B. Bystrica	3	23	5		8			36
22	Okruhlica Adam	Gym. Jura Hronca BA	3	0	15	4	3	8	3	33
23	Danko Juraj	Gym. Jura Hronca BA	3	28						28
24	Halamíček Radovan	Gym. Jura Hronca BA	4	21						21
25	Nagy Anton	Gym. maďarské BA	3	17	3					20
26	Sudolský Michal	Gym. Tajovského B. Bystrica	3	12						12
27	Bašista Peter	Gym. P. Horova Michalovce	4	8						8
28	Košdy Martin	Gym. Jura Hronca BA	1	0	7					7
28	Sucha Martin	Gym. Jura Hronca BA	2	0	7					7
30	Hegedušová Monika	Gym. P. Horova Michalovce	4	0	3			2	0	5