Korešpondenčný seminár z programovania
XXVI. ročník, 2008/09
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 1. kola letnej časti

Milé naše riešiteľky, milí naši riešitelia,

dostávajú sa Vám do rúk ďalšie vzorové riešenia. Veríme, že sa Vám budú opäť páčiť a že ich čítaním strávite príjemné chvíle. A nezabudnite odovzdať aj ďalšie kolo, aby ste sa dostali na sústredenie.

... - -.– .-. .. -.. ... .- - — . -.. . –

KSPáci

1. Zabudnutý PIN

opravoval Dano
(max. 10 bodov)

V tomto príklade bolo vašou úlohou vypočítať posledných päť cifier čísla $N!$. Preto mi je cťou prezentovať riešenie ako fiktívny dialóg medzi tebou a Christianom Krampom – autorom zápisu $N!$.

Ty: Ahoj Christian, potreboval by som pomôcť s jednou úlohou z KSP. Pre dané číslo N, mám vypočítať posledných päť cifier $N!$.

Kramp: Ahoj. To je predsa ľahké, nie? Jednoducho v jednom *for*-cykle vynásobiš všetky čísla od 1 po N a potom vypíšes iba posledných päť cifier.

Ty: Nad tým som rozmýšľal, ale v zadaní píšu, že riešenie musí fungovať pre ľubovoľné veľké N, aké sa zmestí do premennej. A tvoje riešenie už preteká pre $N = 21$.

Kramp: To je pravda. Ale všimni si, že posledných päť cifier je to isté ako zvyšok po delení[1] 10^5. Takže nám stačí rátať iba tento zvyšok.

Ty: Súhlasím, ale ako to vyrieši problém s veľkými číslami.

Kramp: Pozri sa, ak číslo x dáva zvyšok b a číslo y dáva zvyšok d po delení 10^5, potom $x = 10^5a + b$ a $y = 10^5c + d$ pre nejaké c, d. Takže, $xy = 10^5(ad + bc + 10^5ac) + bd$, a teda zvyšok xy po delení 10^5 je ten istý ako zvyšok bd po delení 10^5.[2] Takže namiesto pamätania si celých veľkých čísel ti stačí si pamätať iba ich päť ciferné zvyšky po delení, ktoré sa ti už určite do premenných zmestia.

Ty: To vyzerá ako riešenie. Počkaj, skúsim to naprogramovať.

(O pár minút neskôr.)

Ty: To je zaujímavé, pre všetky veľké hodnoty N mi program vypíše „00000“. Nevieš prečo to tak je, alebo mám snáď dáku chybu v programe?

Kramp: Počkaj, zamyslím sa..... Už to mám! Zvyšok nula po delení znamená, že číslo $N!$ je deliteľné 10^5, čo znamená, že $N!$ je „deliteľné desiatkou aspoň päť krát“. A ak si rozpíšeš 10 na súčin prvočísel, dostaneš $10 = 2 \times 5$, čo znamená, že číslo je deliteľné 10^5 ak je deliteľné 2^5 a 5^5, čo nastane práve vtedy, ak aj dvojka aj päťka sa nachádzajú aspoň päť krát v prvočíselnom rozpise $N!$. No a 25! obsahuje ako činiteľe čísla, $2, 4, 5, 10, 15, 20, 25$ a teda 25! a následne aj každý väčší faktoriál je deliteľný 10^5.

[1] Vo väčšine programovacích jazykov sa používa *mod* alebo *%*.

[2] Pozorný čitateľ si určite všimol, že hodnota 10^5 nie je ničím špeciálna a podobný argument funguje pre ľubovoľné iné číslo.

Ty: Takže ak to zhrniem, riešenie je nasledovné. Ak $N \geq 25$, potom je odpoveď „00000“ a ak $N \leq 24$ potom v jednom cykle prenásobím čísla od 1 po N s tým, že si budem pamätať len zvyšky po delení 10^5. Kedže si musím pamätať iba zopár konštánt, pamäťová zložitosť je $O(1)$ a keďže 25 je len konštanta, nikdy nespravím viac ako konštantne veľa operácií a teda aj časová zložitosť je konštantná – $O(1)$, čo už sa nedá nijako vylepšiť a teda určite dostanem plný počet bodov!

Kramp: No nie tak úplne presne. Nezabudni, že KSPáci si potrpia na popis myšlienky, dôkaz správnosti a odhad časovej a pamäťovej zložitosti! No a takisto si všimni, ako faktoriál výrazne znížil počet Etomových možných PINov.[3]

Na záver už len niečo k bodovaniu. Za optimálne riešenie som dával plný počet bodov. Za riešenie bežiace v čase $O(N)$ som dával bodov 7. A ako už Christian Kramp naznačil, po bodíku som strhával za chýbajúci popis, dôkaz správnosti, odhady zložitosti a takisto ak ste predpokladali, že ľubovoľný faktoriál sa vám zmestí do premennej.

Listing programu:

```
var n,f,i,a: longint;
begin
   readln(n);
   if (n >= 25) then writeln('0000')
   else
      begin
         f := 1;
         for i:=1 to n do f := (f * i) mod 100000;
         a := 10000;
         for i:=1 to 5 do begin write( (f div a) mod 10); a := a div 10; end;
         writeln();
      end;
end.
```

2. Zapichnutá hrana

opravoval Maják
(max. 10 bodov)

Táto úloha mala priamočiare riešenie, na ktoré drvivá väčšina z vás aj prišla. Stačilo zistiť ktorá osoba sa nachádzala najviac v prvom a druhom „stĺpci“ vstupu, pretože to je náš hľadaný ježko, resp. ihelnička.

Najjednoduchšie ako to spraviť je pamätať si, koľko krát každá postava niekoho chcela a koľko krát bola chcená. To vieme spraviť primo pri načítavaní vstupu. Budeme mať dve polia, do ktorých si pre každú osobu budeme udržiavať počet jej výskytov na jednej a druhej strane vzťahu. Popri tejto činnosti si môžme zároveň pamätať, ktorá osoba bola zatial najviac chcená (resp. najviac chcela), a pri načítaní nového vzťahu si tento údaj aktualizujeme.

Po prečítaní celého vstupu teda budeme vedieť kto je ježko, a kto ihelnička, a tieto údaje už len vypíšeme.

V programe musím vynulovať polia, v ktorých si pamätám informácie pre všetky osoby, ktorých je N. Potom načítavam zadané vzťahy, ktorých je M. Keďže medzi N a M nemáme daný žiadny vzťah, tak nevieme povedať ktoré z nich je väčšie. Časová zložitosť bude preto $O(N + M)$.

Na výpočet si potrebujem pamätať len zopár premenných, a dve polia o dĺžke N. Pamäťová zložitosť je teda $O(N)$.

[3] Múdro a všeobecne: funkcie entropiu nezvyšujú!

Listing programu:

```
var N, M, i, ai, bi, jezko, ihelnicka :integer;
    a, b:array[1..100] of integer;
begin
  readln(N, M);
  for i := 1 to N do begin
    a[i] := 0;
    b[i] := 0;
  end;
  jezko := 1;
  ihelnicka := 1;
  for i := 1 to M do begin
    readln(ai, bi);
    inc(a[ai]);
    inc(b[bi]);
    if a[ai]>a[jezko] then jezko := ai;
    if b[bi]>b[ihelnicka] then ihelnicka := bi;
  end;
  writeln('Ihelnicka je: ', ihelnicka, ', stupen ', b[ihelnicka]);
  writeln('Jezko je: ', jezko, ', stupen ', a[jezko]);
end.
```

3. Zoltánove televízory

opravovala Katka
(max. 10 bodov)

Naši milovaní, určite si všetci uvedomujeme, ako sú televízory dôležité, preto budeme venovať problematike rozpoznávania televízorov patričnú pozornosť. Tak poďme na to.

Budeme používať označenie vrchol namiesto názvu spojovník a hrana namiesto slova tyč. Tiež si zavedieme pojem stupeň vrchola, čo je počet hrán vedúcich do vrchola. Keď sa už vieme vyjadrovať, tak sa poďme pozrieť, aké stupne môžu mať vrcholy televízora.

Televízor sa skladá z dvoch častí: bedne a anténiek. Začnime so samotnou bedňou. Tá sa podľa zadania skladá aspoň z troch vrcholov. Pričom každý vrchol, ktorý ju tvorí je stupňa dva. Vieme, že k bedni sa pridávajú ešte dve anténky. Taká anténka pozostáva aspoň z dvoch vrcholov. „Krajné“ vrcholy anténky sú stupňa jedna. Medzi týmito „krajnými“ vrcholmi sa môže nachádzať ľubovoľne veľa ďalších vrcholov (to znamená aj žiadne). Tieto akoby „vnútorné“ vrcholy sú stupňa dva. Teraz nám zostáva už iba spojiť bedňu s anténkami. Anténka sa pripája „krajným“ vrcholom so stupňom jedna. Obe anténky pripevníme na jeden spoločný vrchol v bedni. Čo sa zmenilo? Jedine to, že hrana v anténke, ktorá predtým viedla do „krajného“ vrchola teraz vedie do vrchola v bedni, kde je pripevnená. Preto sa stupeň tohto „bedňového“ vrchola zmení na štyri.

Podľa nášho predchádzajúceho rozboru teda vidíme, že televízor obsahuje dva vrcholy stupňa jedna (konce anténiek), aspoň dva vrcholy stupňa dva (vrcholy bedni a vnútorné vrcholy anténiek) a jeden vrchol stupňa štyri (vrchol, kde sa spája bedňa s anténkami). A to je všetko. Preto ľubovoľný vstup, ktorý nespĺňa tieto podmienky určite nie je televízor. Zostáva nám už iba ukázať, že vstup, ktorý spĺňa naše podmienky televízorom naozaj je. Podľa zadania predpokladáme, že vstupný útvar drží pokope. Vezmime si vrchol so stupňom štyri. Vedú z neho štyri hrany. Rozmyslite si, že vďaka skutočnosti, že iba dva vrcholy sú stupňa jedna a ostatné sú stupňa dva, vzniknú dve anténky a bedňa. Teda naozaj nám stačí overiť, či dva vrcholy majú stupeň jedna, aspoň dva vrcholy stupeň dva a jeden vrchol stupeň štyri. Tiež si skúste premyslieť, že nemusíme overovať, či aspoň dva vrcholy majú stupeň dva. Vyplýva to už z podmienok na vrcholy ostatných stupňov a z toho, že medzi dvomi vrcholmi môže viesť nanajvýš jedna hrana.

Naozaj sme väčšinu vstupu nepotrebovali, čím ste sa našťastie nedali zmiasť. Časová a pamäťová zložitosť sú konštantné, teda $O(1)$.

Listing programu:

```
var N,M,i:integer;
    a:array[1..5] of integer;
begin
readln(N,M);
for i:=1 to 5 do
read(a[i]);
if (a[1] = 2) and (a[3] = 0) and (a[4] = 1) and (a[5] = 0) then
writeln('Vstup je televizor.')
else
writeln('Vstup nie je televizor.');
end.
```

4. Obrazovka je na eskykráliky malá

opravovala Elφnka
(max. 10 bodov)

Predstavme si najprv, že sa naše králiky budú iba rozmnožovať, ale nie navzájom žrať. Pozrime sa, ako bude vyzerať prvých pár sekúnd ich života:

```
        1
      1 0 1
    1 0 2 0 1
  1 0 3 0 3 0 1
1 0 4 0 6 0 4 0 1
```

Všimnite si, že tam, kde v predchádzajúcom kroku boli králiky v ďalšom kroku králiky nie sú a naopak. Zoberme si nejaké prázdne miesto. V ďalšom kroku na dané miesto pribehne banda králikov zo severu a banda králikov z juhu. Ak máme teda situáciu, že na i-tom políčku je A králikov, na $i+1$-vom políčku je 0 králikov a na $i+2$-hom políčku je B králikov, tak v ďalšom kole bude na políčkach i a $i+2$ nula králikov a na políčku $i+1$ bude $A+B$ králikov.

Tí z vás, ktorí poznajú kombinatoriku si určite všimli, že sa to nejako nápadne podobá na Pascalov trojuholník a kombinačné čísla. Čo sú to? Kombinačné číslo je číslo, ktoré sa zapisuje ako $\binom{N}{k}$ a jeho hodnota je

$$\binom{N}{k} = \frac{N!}{k!(N-k)!}$$

pričom ! nie je výkričník, ale faktoriál. Faktoriál sa dá vypočítať nasledovne:

$$N! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot N$$

Zároveň sa dohodnime[4], že $0! = 1$ (čítame[5] nula faktoriál sa rovná 1).

Kombinačné čísla vieme napísať do Pascalovho trojuholníka tak, že horné číslo bude predstavovať riadok a spodné číslo bude predstavovať stĺpec. Daný trojuholník bude vyzerať nasledovne:

```
1                          1
1 1                      1 1
```

[4] Aby nám vzorce sedeli.

[5] Niektorí to prečítajú ako nula sa nerovná jedna :-D

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

Ako vidíte, Pascalov trojuholník sa bude nápadne podobať na zajačiky. A nielen sa podobá, ale je aj rovnaký.

Ak by sa zajačiky navzájom nežrali, tak by sme mali o všetko postarané, iba by sme vypočítali jeden riadok Pascalovho trojuholníka a doplnili ho nulami. Ale tie beštie zajačiky sa navzájom žerú...

Čo by sa stalo, keby sa králiky takto pokojne rozmnožovali, a na konci, v N-tej sekunde, by sa každá dvojica navzájom zožrala? Na mieste, na ktorom by stál párny počet králikov by neostal žiaden, a na mieste, kde by bolo nepárne veľa králikov by jeden ostal. Králiky, čo nám ostanú po takomto žraní budú rozmiestnené tak isto, ako keby sa žrali už od začiatku - ak by sa mala niekde stretnúť dvojica zajačikov, ktoré by sa zožrali, ďalej už budú ich potomkovia tiež skákať ako dvojica, ktorá sa nakoniec zožerie. Z toho teda vidíme, že nám stačí vypočítať jeden riadok Pascalovho trojuholníka, zistiť zvyšky po delení dvoma a doplniť medzi ne nuly.

Ako to teda naprogramujeme? Budeme postupne počítať jednotlivé prvky riadku Pascalovho trojuholníka a vypisovať zvyšky po delení dvoma. Ak budeme rátať každý člen Pascalovho trojuholníka „nanovo", tak dostaneme zlú časovú zložitosť a to $O(N^2)$. My ale využijeme to, že každý prvok riadku (okrem prvého, ktorý je 1) sa dá vypočítať z predchádzajúceho prvku a to nasledovným vzorcom:

$$\binom{N}{k} = \frac{N-k+1}{k}\binom{N}{k-1}$$

Prečo daný vzorec platí si skúste ukázať sami, nech sa v tom pocvičíte. Týmto spôsobom sme sa dostali k riešeniu, ktoré používa konštantný počet premenných a jeho časová zložitosť je lineárna ($O(N)$).

Za vzorové riešenie bolo samozrejme 15 bodov, za kvadratické s konštantnou pamäťou 11 bodov, a za simulovanie stavu králikov od 0. sekundy najviac 7 bodov.

Listing programu:

```
var N,i,a:integer;

begin
   readln(N);
   a:=1;
   for i:=1 to N-1 do
      begin
         write(a mod 2,'0');
         a:=(a*(N-i)) div (i);
      end;
   writeln(a mod 2);
end.
```

5. Zlý príjem

Opravoval MMx
(max. 15 bodov)

Vyriešme najprv jednoduchšiu úlohu: pre dané i budeme chcieť zistiť, koľko zrnitých čísel má presne i číslic. Na pomoc si zoberieme dynamické programovanie. Budeme vypĺňať po riadkoch tabuľku A, ktorej prvok $A[i][j]$ nám hovorí koľko čísel dĺžky $i+1$ začína číslicou $j+1$. Jednociferné číslo začínajúce konkrétnou číslicou je vždy len jedno, takže v riadku

$A[0][j]$ budú samé jednotky. Číslo dĺžky i začínajúce j môžeme vyrobiť tak, že pridáme číslicu j na začiatok nejakého čísla dĺžky $i-1$, ktoré nezačína väčšou číslicou (aby nové číslo bolo stále zrnité). Teda platí $A[i][j] = \sum_{k=0}^{j} A[i-1][k]$. To znamená, že v každom $A[i][j]$ máme sumu prvých j čísel predchádzajúceho riadku, takže $A[i][j]$ môžeme počítať ako $A[i][j-1] + A[i-1][j]$.

Nájsť m-te číslo v postupnosti vlastne znamená preskočiť prvých $m-1$ a pozrieť sa na nasledujúce. My už máme spočítané, koľko je jednociferných čísel (v $A[1][8]$), teda vieme, či ich môžeme preskočiť všetky naraz alebo je hľadané číslo niekde medzi nimi. Ak medzi nimi nie je, znížime m o ich počet a pokračujeme s číslami o 1 dlhšími, až kým nenájdeme počet cifier hľadaného čísla.

Teraz ostáva vyriešiť posledný problém: ako vyzerá m-té číslo medzi c-ciernými? Jeho prvú číslicu vieme nájsť podobne ako sme hľadali počet jeho cifier, teda preskakujeme tie začínajúce 1, 2, …, $j-1$, až kým nájdeme také, ktorých je príliš veľa (a popri tom si v m pamätáme koľko ešte ostáva preskočiť). To čo v našom čísle nasleduje za číslicou j je vlastne zrnité číslo dĺžky $c-1$, a to presne m-té také. Takže tento postup môžeme opakovať kým neurčíme všetky číslice.

Všimnite si, že pri tomto algoritme si vystačíme s obyčajnými celočíselnými premennými a nemusíme si programovať vlastné veľké čísla (ani používať programovací jazyk, ktorý ich má implementované).

Listing programu:

```
#include <cstdio>
#define CISLIC 9
#define MAXCIF 300
#define MAXK 10000000000000000LL
typedef long long int lint;

lint pocty[MAXCIF][CISLIC];
char vysl[MAXCIF+1];

int main() {
  for (int i=0; i<CISLIC; i++)
    pocty[0][i] = 1;

  for (int i=1; i<MAXCIF; i++) {
    pocty[i][0] = 1;
    for (int j=1; j<CISLIC; j++)
      pocty[i][j] = pocty[i-1][j] + pocty[i][j-1];
  }

  for (lint cislo=100; cislo<=MAXK; cislo*=10) {
    int i;
    lint ostava = cislo;

    for (i=1; ostava - pocty[i][CISLIC-1] > 0; i++)
      ostava -= pocty[i][CISLIC-1];
    vysl[i] = '\0';
    i--;

    for (int pozicia=0; i>=0; i--) {
      int j;
      for (j=0; ostava - pocty[i][j] > 0; j++)
        ostava -= pocty[i][j];
      vysl[pozicia++] = '1' + j;
    }
    printf("%s\n", vysl);
```

```
  }

  return 0;
}
```

opravoval Laci
(max. 20 bodov)

6. Obchod plný televízorov

Na úvod si rýchlo pripomeňme podstatu úlohy. Treba nájsť také usporiadanie televízorov, v ktorom nie je menší televízor pred väčším. Takémuto usporiadaniu sa hovorí topologické usporiadanie/utriedenie. Teraz sa pozrime na to, ako sa také topologické triedenie dá spraviť.

Vstup si vieme predstaviť ako orientovaný (hrany sú jednosmerné) graf v ktorom vedie hrana z **menšieho** televízora do **väčšieho**, pričom neobsahuje cyklus (ináč topologické usporiadanie neexistuje, ale také niečo by náš hrdina zo zadania namerať nemohol). Zoberme si teraz ľubovoľný vrchol v. V každom topologickom usporiadaní je vrchol v za vrcholmi, do ktorých sa dá dostať z vrchola[6] v (lebo tie majú menšiu uhlopriečku ako vrchol v). Zároveň, vrchol v musí byť v topologickom usporiadaní len pred vrcholmi, z ktorých sa dá do neho dostať. Ak teda existuje vrchol u, z ktorého sa nedá dostať do v a do ktorého sa nedá dostať do v, tak ten môže byť aj pred, aj za vrcholom v. Preto pred vypísaním vrcholu v musíme vypísať všetky vrcholy, do ktorých sa vieme z v dostať.

Toto nám dáva návod, ako nájsť topologické usporiadanie grafu. Bude to fungovať ako rekurzívna procedúra, ktorá dostane ako parameter vrchol v a ona vypíše topologické usporiadanie vrcholu v a vrcholov, do ktorých sa dá dostať z vrcholu v a potom tie vrcholy z grafu odstráni. Ak zavoláme túto procedúru na každý vrchol grafu, tak dostaneme topologické usporiadanie. Prečo? Ak ju zavoláme na prvý vrchol v_1, tak dostaneme topologické usporiadanie niektorých vrcholov. Do ostatných vrcholov sa z v_1 nevieme dostať (ináč by boli v usporiadaní) a preto s kľudom môžu[7] byť v usporiadaní za vrcholom v_1. Teraz zoberme ďalší vrchol v_2. Ak bol z grafu odstránený, tak sa nestane nič. Ak nebol, tak sa naše topologické usporiadanie predĺži a opäť vrcholy ktoré ostali môžu byť v topologickom usporiadaní neskôr. Ak budeme takto pokračovať, naše topologické usporiadanie sa bude predlžovať a časom nám už neostanú žiadne vrcholy a vtedy máme výsledok.

Ako takú procedúru naprogramovať? Namiesto odstraňovania vrcholov z grafu si budeme iba poznačovať, či sme v danom vrchole boli. Ak sa zavoláme na vrchol v, tak ak sme v ňom boli, tak procedúra skončí. V ostatných prípadoch si procedúra najskôr zapíše, že vo vrchole v už bola, rekurzívne sa zavolá na susedov vrchola v, potom vypíše vrchol v. Rozmyslite si, že takáto procedúra má požadované vlastnosti.

Hore popísanému postupu sa zvykne hovoriť prehľadávanie do hĺbky. A spôsobu vypisovania vrcholov post-order. Najzložitejšou časťou algoritmu je DFS (Depth First Search – prehľadávanie do hĺbky), ktoré má časovú zložitosť $O(N+M)$ (pretože každý vrchol a každú hranu prejde práve raz). Preto je aj časová zložitosť celého algoritmu $O(N+M)$.

Pamätať si potrebujeme všetky hrany a o vrcholoch či sme ich už spracovali, preto je pamäťová zložitosť algoritmu $O(N+M)$.

Posledné riadky patria komentáru k bodovaniu. Za optimálne riešenie, veď ako ináč, bol plný počet bodov. Ostatné riešenia boli odstupňované podľa časovej zložitosti (pamäťovú ste mali všetci rovnakú) po dvoch až troch bodoch. Za chýbajúci alebo nedostatčný popis, dôkaz správnosti a odhady zložitosti ste mohli stratiť jeden až dva body.

[6] Použitím aj viacerých hrán

[7] Niektoré ba priam musia!

Listing programu:

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

int n,m;
vector< vector<int> > graf;
vector<bool> navstivene;
vector<int> vysledok;

void dfs(int vrchol){  //prehladavanie do hlbky
   if (navstivene[vrchol]) return;
   navstivene[vrchol]=true;

   //vypis vrcholy ktore maju byt v topologickom utriedeni skor
   for(int i=0;i<graf[vrchol].size();i++) dfs(graf[vrchol][i]);

   vysledok.push_back(vrchol);
}

int main(){
   cin>>n>>m;

   navstivene.resize(n+1);
   for(int i=1;i<=n;i++) navstivene[i]=false;

   //nacitame hrany grafu
   graf.resize(n+1);
   for(int i=0;i<m;i++){
      int a, b;
      cin>>a>>b;
      graf[b].push_back(a);
   }

   //z kazdeho vrcholu spustime dfs
   for(int i=1;i<=n;i++) dfs(i);

   //vypis vysledku
   for(int i=0;i<n;i++) cout<<vysledok[i]<<endl;
   return 0;
}
```

7. Obchod s televízormi

opravoval Ivan
(max. 20 bodov)

Túto úlohu s rozumnou zložitosťou riešili traja riešitelia, čo nebol očakávaný výsledok s ohľadom na jej obtiažnosť. Žiaľ, väčšina riešení mala zložitosť $O(N^2)$.

Ak využijeme aj ohraničenie, že ceny sú celé čísla menšie alebo rovné N, existuje viacero rozdielnych algoritmov na riešenie úlohy so zložitosťou najviac $O(N \log N)$.

Najjednoduchšie riešenie s ľubovoľnými cenami je použitím haldy, do ktorej budeme priebežne ukladať dodávky a od najmenšej vyberať, keď prídu požiadavky. Tento algoritmus by mal v najhoršom prípade $O(\log N)$ časovú zložitosť pre dodávku ale pri jednotlivých zákazníkoch môže dosiahnuť až $O(N \log N)$, keď by na splnenie jednej požiadavky potreboval použiť $O(N)$ dodávok. Avšak celkovú zložitosť $O(N \log N)$ to nezmení, lebo každá dodávka

sa môže najviac raz pridať a odobrať z haldy, za predpokladu, že budeme dopredu odmietať nesplniteľné požiadavky, na čo nám stačí priebežne si rátať počet televízorov na sklade.

Trochu zložitejší prístup je použitím binárneho stromu dodávok, kde vrcholy sú usporiadané podľa ceny a v každom vrchole si pamätáme celkový počet a cenu všetkých televízorov pod ním. Pridávanie je asi zrejmé a požiadavky vieme tiež rýchlo splniť, lebo prechádzajúc od koreňa stromu, najviac jednu vetvu potrebujeme iba čiastočne vykúpiť a do tej druhej nemusíme zachádzať. Ako odľahčenie tu príde vhod ohraničenie cien na celé čísla do N, čo umožní použiť intervalový strom – (zľava) úplný binárny strom všetkých cien, kde sa potom nemusíme starať o vyváženie a môže sa aj vhodne implementovať v poli. Takéto riešenie trochu viac zodpovedá úlohy zadania, lebo na každú dodávku a každú požiadavku potrebujeme $O(\log N)$ čas, s celkovou zložitosťou $O(N \log N)$.

Mierne lepšiu zložitosť $O(N \log \log N)$ je možné dosiahnuť zmenou prvého algoritmu, aby používal všemocnú van Emde Boasovu prioritnú frontu[8], ktorá ale má hnusnú implementáciu.

Najrýchejšie riešenie, ktoré sme boli schopní vymyslieť, má zložitosť $O(N \log^\star N)$. Tá síce nie je lineárna, ale má k tomu veľmi blízko[9]. Zložitosť tohto algoritmu je dokonca ešte trochu menšia ako $O(N \log^\star N)$. Tento algoritmus ale má tú nevýhodu, že nemôže počítať riešenie priebežne (tzv. online algoritmus), ale musí spracovať celý vstup (tzv. offline), aby mohol dať hoc aj časť výsledku. Algoritmus prebieha v dvoch fázach. Najprv sa dodávky utriedia podľa cien a potom sa televízory z nich rozdeľujú požiadavkám.

Prvá časť je jednoduchá – bucket sortom utriedime dodávky v $O(N)$, pamätajúc si, pred ktorou požiadavkou sa vyskytli. Požiadavky si umiestnime do poľa v pôvodnom poradí, poznačujúc si nesplniteľné požiadavky – im netreba dávať televízory.

Druhá časť je zaujímavejšia. Pokiaľ väčšina riešení postupne k požiadavkám určuje dodávky, naše riešenie funguje opačne. Televízory z každej dodávky začne deliť medzi požiadavky, pričom začneme prvou, ktorá bola po tej dodávke a pokým je televízorov dosť, tak postupujeme k neskorším. Musíme sa ale vyhnúť opätovnému prechádzaniu cez už splnené požiadavky, aby sme nedosiahli populárnu zložitosť $O(N^2)$. Na to použijeme algoritmus union find, ktorý si najprv vysvetlíme.

Union find je algoritmus pracujúci s množinami. Podporuje dve operácie:

`find(x)` – nájde reprezentanta množiny `x`

`union(x,y)` – zjednotí množiny `x` a `y`

Algoritmus používa dátovú štruktúru prevráteného stromu – každý vrchol si pamätá svojho rodiča, ale nie deti. Množiny sú stromy reprezentované koreňom. Z každého vrcholu sa môžeme dostať ku koreňu postupným volaním sa na otca a zjednotiť množiny môžeme podvesením jedného stromu pod druhý – jednoduchým nastavením otca jedného na druhého. Na zrýchlenie týchto operácií sa používajú dve finty: vešanie menšieho stromu pod väčší a kompresia cesty. Prvá je implementovaná pamätaním si hornej hranice výšky vrchola, ktorá je na začiatku 0. Pri zjednocovaní vždy zavesíme koreň menšieho stromu pod koreň väčšieho, alebo ak to nie je možné, jednému najprv zväčšíme výšku. Kompresia cesty je ešte jednoduchšia. Vo `find` pri každom prechode z vrchola ku koreňu stromu všetky vrcholy cez ktoré sme prešli priamo zavesíme pod koreň. Každá z týchto optimalizácii použitá osamote prináša zložitosť $O(N \log N)$ pre N operácii a použité spolu zredukujú čas na menej ako $O(N \log^\star N)$. Dôkaz týchto zložitostí je príliš komplikovaný a veríme, že sa nenahneváte, ak ho tu neuvedieme.

Takže ako bude vyzerať náš algoritmus? Predpokladajme, že máme požiadavky prefiltrované – vieme, ktoré splniť nemôžeme. Čo platí pre najlacnejšiu dodávku? Televízory z nej sa zjavne použijú v najbližšej požiadavke. Ak je týchto televízorov veľa, potom sa tie zvyšné použijú ešte v tej nasledujúcej, a tak ďalej. Treba si uvedomiť, že nízka cena dodávky robí z

[8] Keby ste náhodou nevedeli, čo to je, tak si nájdite na wikipédii. V tomto vzoráku sa tomu nebudeme venovať.

[9] funkcia $log^\star$ je pre každé rozumné číslo, ktoré je menšie ako počet atómov v známom vesmíre, najviac 5.

televízorov prioritné kusy na odber a ak máme aj na sklade nejaké drahšie, tak tie nás budú zaujímať až potom, ako tie lacnejšie minieme.

Náš algoritmus teda bude postupovať po dodávkach, ktoré sme utriedili. Z požiadaviek budeme vytvárať množiny, pričom na začiatku je každá požiadavka samostatná množina. Keď nejakú požiadavku splníme, potom ju zjednotíme s nasledujúcou. V každej množine takto bude práve jedna požiadavka, ktorá je ešte nesplnená. Ak nejaká požiadavka p je splnená a patrí do množiny, kde je nesplnená požiadavka q, potom to nehovorí nič iné ako to, že každý televízor, ktorý by mal byť odteraz použitý na splnenie požiadavky p, bude použitý na splnenie požiadavky q, pretože požiadavka q je od danej dodávky najbližšia nesplnená.

Predstavme si, že máme za úlohu rozdeliť televízory z nejakej stredne-drahej dodávky. Vieme, ktorá požiadavka je tá časovo najbližšia po doručení tejto dodávky. Avšak táto požiadavka už dávno mohla byť splnená inými, lacnejšími televízormi. Vykonaním operácie `find` zistíme reprezentanta množiny požiadaviek, do ktorej patrí táto vyčerpaná, a zistíme, do ktorej požiadavky pôjdu televízory z tejto dodávky – to je tá jedna nedokončená v tejto množine, označme p. Ak je televízorov málo na uzavretie tejto požiadavky p, sme hotoví a len si príslušný fakt zapamätáme. Avšak, ak je televízorov dosť, tak vieme, že sa použijú na dokončenie tejto požiadavky p. Zistíme si množinu, do ktorej patrí požiadavka q, ktorá je najbližšia nasledujúca po p, a túto množinu zjednotíme operáciou `union` s tou, kde sme práve dokončili požiadavku – odteraz všetky televízory zo všetkých dokončených požiadaviek z týchto množín budú použité na dokončenie tejto najbližšej nedokončenej požiadavky q. V každom momente nám teda množiny požiadaviek poslúžia na to, aby sme vedeli povedať o každej požiadavke, ktorá je nasledujúca nedokončená a aby sme vedeli naraz veľa požiadavkám zmeniť nasledujúcu nedokončenú.

Celkovo voláme pre každú požiadavku najviac jednu operáciu `union`. Koľko je operácií `find`? No pre každú dodávku najprv voláme raz, aby sme vedeli, kam máme tieto televízory nasmerovať. Okrem toho ju už voláme len pri operácii `union`. Takže celkovo máme pre každú dodávku alebo požiadavku len konštantný počet týchto volaní a celkový čas je teda $O(N \log^\star N)$.

Listing programu:

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

using namespace std;

struct req {
    int need;
    int sum;
    req(int n, int s): need(n), sum(s) {}
};

struct delivery {
    int count;
    int time;
    delivery(int c, int t): count(c), time(t) {}
};

struct set {
    set * up;
    int r;
    int max;
    set() {}
    set(int m): up(0), r(0), max(m) {}
```

```
};

set * find(set * e)
{
    set * top = e;
    while (top->up)
        top = top->up;
    while (e != top) {
        set * t = e;
        e = e->up;
        t->up = top;
    }
    return top;
}

void set_union(set * a, set * b) {
    a = find(a);
    b = find(b);
    if (a->r < b->r)
        swap(a, b);
    b->up = a;
    a->r += (a->r == b->r);
    a->max = max(a->max, b->max);
}

int main()
{
    int n;
    scanf("%d", &n);
    vector<vector<delivery> > delivs(n);
    vector<req> reqs;

    // nacitanie poziadaviek a dodavok
    int have = 0;
    for (int i = 0; i < n; ++i) {
        int t, a, b;
        scanf("%d %d", &t, &a);
        if (t == 1) {
            scanf("%d", &b);
            delivs[b].push_back(delivery(a, reqs.size()));
            have += a;
        } else {
            if (a <= have) {
                reqs.push_back(req(a, 0));
                have -= a;
            } else
                reqs.push_back(req(0, -1));
        }
    }

    // strom pre union find
    set * sets = new set[reqs.size()];
    for (int i = 0; i < reqs.size(); ++i)
        sets[i] = set(i);

    for (int price = 0; price < delivs.size(); ++price)
        for (vector<delivery>::iterator j = delivs[price].begin();
                j != delivs[price].end(); ++j) {
            int have = j->count;
            int prev = -1;
```

```
            int t = j->time;
            while (have && t < reqs.size()) {
                if (prev >= 0)
                    set_union(sets + prev, sets + t);
                t = find(sets + t)->max;
                int d = min(reqs[t].need, have);
                reqs[t].need -= d;
                reqs[t].sum += d * price;
                have -= d;
                ++t;
            }
        }

    for (int i = 0; i < reqs.size(); ++i) {
        if (reqs[i].sum >= 0)
            printf("%d\n", reqs[i].sum);
        else
            printf("Nedostatok televizorov\n");
    }
    return 0;
}
```

opravoval Mic
(max. 25 bodov)

8. Ovplyvnenie predaja

Tento príklad nebol až tak veľmi zložitý, napriek tomu mal málo riešení. A úplne optimálne nebolo žiadne:-(

Poďme najskôr k označeniu. Zo zadania vieme, že N označuje počet televízorov a p_i označuje pravdepodobnosť i-teho televízora. Najpriamočiarejšie riešenie je rozdeliť si interval $(0, 1)$ na N intervalov $I_1, \ldots, I_N$ tak, že veľkosť i-teho intervalu sa rovná pravdepodobnosti p_i (i-ty interval zodpovedá i-temu televízoru).

Jednoduché riešenie je vygenerovať náhodné číslo a pozrieť sa, do ktorého intervalu padlo. Takýto prístup vedie k lineárnemu predspracovaniu a lineárnemu času vygenerovania výstupu. Ak by sme interval vyhľadávali napríklad binárnym vyhľadávaním, tak by sme došli ku generovaniu v čase $O(\log(N))$. Ale ako bolo v zadaní napísané, za takéto riešenia veľa bodov nie je. To bol mimochodom hint, že sa to dá spraviť rýchlejšie.

Rozdeľme si interval $(0, 1)$ na N rovnako veľkých častí. Ak si pre každú časť zapamätáme, ktoré intervaly do nej patria, tak keď vyberieme náhodné číslo, v konštantnom čase zistíme do ktorej časti patrí a potom pozrieme všetky intervaly v danej časti, aby sme zistili, ktorý interval sme vybrali. Ak sa nám nestane, že v jednej časti je príliš veľa intervalov, tak by čas nájdenia intervalu bol konštantný. Vo všeobecnom prípade toto nevieme zaručiť a preto je toto riešenie pomalé. Dá sa však vylepšiť. Ukážeme teraz riešenie, v ktorom nasekáme intervaly tak, že v každej časti budú práve dva intervaly.

Veľkosť jednej časti je $1/N$. Na to, aby bolo v jednej časti veľa intervalov, tak všetky musia byť malé (menšie ako $1/N$). To ale znamená, že existuje aspoň jeden interval, ktorý má veľkosť aspoň $1/N$. Túto vlastnosť využijeme v optimálnom riešení.

Rozdeľme si intervaly na malé a veľké. Malé intervaly sú tie, ktoré sú menšie ako $1/N$ a veľké intervaly sú tie ostatné. Navyše si ku každému intervalu pamätajme, ku ktorému televízoru patrí. Zoberme teraz nejaký malý interval I_m, ktorého veľkosť je p_m a jeden veľký interval I_v, ktorého veľkosť je p_v. Rozrežme veľký interval na dve časti I'_v a I''_v, pričom I'_v má veľkosť $1/N - p_m$ a I''_v má veľkosť $p_v + p_m - 1/N$. Do prvej časti dajme intervaly I_m a I'_v. Intervaly I_m a I_v odstránime zo zoznamov veľkých aj malých intervalov a I''_v pridáme do toho zoznamu, do ktorého patrí. Prvá časť je celá pokrytá a obsahuje práve dva intervaly.

Ostalo nám $N-1$ intervalov. Ak je nejaký z nich malý, tak určite existuje nejaký veľký[10] a rovnakým spôsobom vieme naplniť druhú časť. Takto budeme postupne pokračovať, až kým sa nám neminú všetky malé intervaly. Vtedy nám ostane K veľkých intervalov, ktoré chceme rozdeliť to K častí. Všimnite si, že vtedy musí mať každý interval veľkosť $1/N$ a preto každý z nich rozdelím na dva intervaly veľkosti $1/2N$ a vložím ich do patričných častí.

Ako budeme generovať našu postupnosť? Náhodne si vyberieme jednu časť, v nej máme dva intervaly a tak vygenerujeme náhodné číslo z danej časti a pozrieme sa do ktorého intervalu padlo. Toto celé trvá konštantný čas.

Aká je časová zložitosť predspracovania? Rozdeliť intervaly na veľké a malé zvládame v lineárnom čase. Na zaplnenie každej časti nám stačí spraviť konštantný počet operácii a keďže častí máme N, tak celková časová zložitosť predspracovania bude $O(N)$. Pamäťová náročnosť je tiež $O(N)$, lebo si musíme pamätať všetky intervaly.

A teraz sa môžete vrhnúť do čítania kódu.

Listing programu:

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

struct Item{
    double P;
    int index;
    Item(double p,int i){
        P=p;
        index=i;
    }
    Item(){};
};

double rnd(){
    return (double)rand()/(double)RAND_MAX;
}

int main(){
    srand(time(0));  // lepsiu nahodu nemame...
    int N;
    cin>>N;
    double tr=1.0/N;
    vector<Item> small,big;

    for(int i=0;i<N;i++){
        double a;
        cin>>a;
        if(a>=tr)
            big.push_back(Item(a,i));
        else
            small.push_back(Item(a,i));
    }

    vector<Item> intervals;
    while(small.size()!=0||big.size()!=0){
        Item sm,bg;
        bg=big.back();      // big bude vzdy neprazdne
```

[10] Kto nevie prečo, nech sa na mieste zastaví a dokáže si to!

```
        big.pop_back();
        if(small.size()==0){
            sm=Item(tr/2,bg.index);
            bg.P-=tr/2;
        }else{
            sm=small.back();
            small.pop_back();
        }
        intervals.push_back(sm);
        intervals.push_back(Item(tr-sm.P,bg.index));
        bg.P-=tr-sm.P;
        if(bg.P>=tr)
            big.push_back(bg);
        else if(bg.P>=1e-9)     // praca s floating-point premennymi nie je presna
            small.push_back(bg);
    }

    while (true) {
        unsigned int bucket=floor(rnd()*N);
        if (rnd()*tr<=intervals[bucket*2].P)
            cout << intervals[bucket*2].index+1<<endl;
        else
            cout << intervals[bucket*2+1].index+1<<endl;
    }
    return 0;
}
```

Opravoval Lukáš
(max. 25 bodov)

9. Tatranské panorámy

Tento príklad sa dal riešiť dvoma spôsobmi. Najprv si popíšeme ten spôsob, ktorý sa jednoduchšie programuje, ale vzorovú zložitosť má len v priemernom prípade.

Prvé riešenie

Budeme používať techniku *hešovania*. Namiesto toho, aby sme porovnávali celé úseky riadkov, budeme porovnávať len akési čísla, ktoré ich reprezentujú. Predstavme si, že máme nejakú funkciu, ktorá priradí každému reťazcu číslo od 0 po $P-1$. Ak sú reťazce zhodné, musia byť aj im prislúchajúce hodnoty rovnaké. Takže ak majú dva reťazce prislúchajúce hodnoty funkcie rôzne, tieto dva reťazce sú tiež rôzne. Naopak to však nemusí byť pravda, lebo dva rôzne reťazce môžu mať rovnaké číslo z rozsahu 0 až $P-1$.

Konkrétne v našom prípade budeme hešovať reťazce nasledovne: kedže reťazec je postupnosť bitov $s_0 s_1 \ldots s_{k-1}$, môžeme si ho predstaviť ako k-bitové celé číslo. Toto číslo vydelíme číslom P a zapíšeme si zvyšok, ktorý bude *heš* tohto reťazca. Je jasné, že čísel dávajúci rovnaký zvyšok po delení P je veľmi veľa. Preto platí, že 2 rovnaké zvyšky ešte neznamenajú rovnaké čísla. Rôzne zvyšky však znamenajú rôzne čísla.

Náš algoritmus bude fungovať nasledovne. Budeme skúšať prekryv veľkosti $1, 2, \ldots, M$. Keď skúšame prekryv veľkosti k, porovnáme heše posledných k písmen prvej panorámy a prvých k písmen druhej panorámy. Ak sa heše všetkých riadkov rovnajú, vyskúšame, či sa aj naozaj panorámy na týchto k stĺpcoch prekrývajú. Heše musíme vedieť vyrátať veľmi rýchlo, lebo inak bude program pomalý. Pre $k = 1$ vyrátame heš ľahko. Prekryv v i-tom riadku je $a_{i,M}$ a $b_{i,1}$, lebo z prvého obrázku berieme posledné písmeno a z druhého prvé. Obidve písmená si predstavíme ako číslo od 0 po 255, lebo ich reprezentujeme jedným bajtom. Teda heš z prvého obrázku je $a_{i,M} \bmod P$ a z druhého obrázku $b_{i,1} \bmod P$. Teraz predpokladajme, že máme vyrátaný heš pre prekryv veľkosti k a chceme ho vyrátať pre prekryv veľkosti

$k+1$. V prvom prípade je to $A = (a_{i,M-k+1} + a_{i,M-k+2}256 + \cdots + a_{i,M}256^{k-1}) \bmod P$ a chceme vyrátať hodnotu $(a_{i,M-k} + a_{i,M-k+1}256 + \cdots + a_{i,M}256^{k}) \bmod P$, čo je $(A256 + a_{i,M-k}) \bmod P$. Túto hodnotu vieme vyrátať z A v konštantnom čase. Podobne pre druhý obrázok je pre k hodnota hešu $B = (b_{i,1} + b_{i,2}256 + \cdots + b_{i,k}256^{k-1}) \bmod P$ a pre $k+1$ je to $(b_{i,1} + b_{i,2}256 + \cdots + b_{i,k}256^{k-1} + b_{i,k+1}256^{k}) \bmod P$, teda $(B + b_{i,k+1}256^{k}) \bmod P$. Za predpokladu, že hodnotu $256^k \bmod P$ máme už predrátanú, vieme $(B + b_{i,M-k}256^{k}) \bmod P$ vyrátať v konštantom čase.

Heše teda vieme pre každý riadok aktualizovať v konštantnom čase. Teraz môžete namietať, že v prípade rovnosti hešov musíme aj tak skontrolovať, či sa reťazce zhodujú. Toto je pravda, lenže udalosť, že sa heše pre jeden riadok zhodujú nastane s pravdepodobnosťou $1/P$. Dĺžka riadku je M, takže v priemere nastane takáto udalosť M/P-krát. Každá kontrola riadku zaberie $O(M)$ času, teda spolu bude zložitosť v riadku v priemere $O(M^2/P)$. A keďže chceme, aby táto zložitosť bola spolu lineárna od M, tak P musíme zvoliť aspoň také veľké, ako M. Spolu máme N riadkov, takže v priemernom prípade je časová zložitosť $O(NM)$ a rovnaká je aj pamäťová zložitosť, keďže si budeme obidva obrázky pamätať celé, aby sme ich na konci mohli vypísať. Náš zdroják je implementáciou tohto riešenia.

Druhé riešenie

Druhé riešenie je založené na Knuth-Morris-Prattovom (KMP) algoritme. Predpokladajme, že máme slová $u = u_1u_2 \ldots u_U$ a $v = v_1v_2 \ldots v_V$. Tento algoritmus nám v čase $O(U+V)$ vie pre každú pozíciu v u povedať, aký najväčší začiatočný úsek z v sa začína od pozície u_i. Teda v našom prípade pre j-tu pozíciu v i-tom riadku prvého obrázku budeme chcieť vedieť, aký najväčší začiatočný úsek i-teho riadku druhého obrázku sa na tejto pozícii začína. Ak je tento najväčší úsek až po koniec riadku, tak je prekryv úplný.

Popis KMP algoritmu nájdete v ďalšom odstavci. Náš algoritmus spustí KMP algoritmus pre každý riadok z prvého a druhého obrázku. Potom nájde najmenšie k, pre ktoré je prekryv v každom riadku úplný. Časová zložitosť je $O(NM)$ lebo pre každý riadok pustíme KMP algoritmus v čase $O(M+M) = O(M)$ a nakoniec ešte skontrolujeme prekryvy. Pamäťová zložitosť je tiež $O(NM)$.

Knuth-Morris-Prattov algoritmus

Na vstupe máme text a slovo, ktoré v ňom chceme nájsť. Toto slovo budeme volať vzorka. Predpodkadajme, že dĺžka vzorky je M a dĺžka textu je N. Jednoduché ale pomalé riešenie spočíva v tom, že vyskúšame všetky možné pozície vzorky v texte. Tých pozícii je zjavne $N - M + 1$ a pre každú pozíciu musíme v najhoršom prípade porovnať všetky písmená vo vzorke, ktorých je M, teda výsledná časová zložitosť je $O(NM-M^2)$, čo pre relatívne malé M je $O(NM)$. Toto sa dá ale ešte zlepšiť. Existuje viacero lineárnych algoritmov, ktorých časová zložitosť je $O(N+M)$. My si z nich vyberieme Knuth-Morris-Prattov (KMP) algoritmus.

Všimnime si podrobnejšie horeuvedený algoritmus. Pri skúmaní, či sa na i-tom mieste nachádza vzorka, sa môže stať, že prvých j písmeniek vo vzorke sa zhoduje s textom, ale $j+1$ písmenko sa už nezhoduje. Teraz by sme sa vrátili a skúšali, či sa vzorka nenachádza na $i+1$ písmenku. Predstavme si však, že vzorka je napríklad „baaaa“ a zistíme, že prvých j písmenok sa zhoduje s textom a $j+1$-vé už nie. V tomto špeciálnom prípade sa teraz nemusíme vracať až na $i+1$ pozíciu, pretože žiadne z predchádzajúcich $j-1$ písmen nemôže byť začiatočné písmeno vzorky, a teda môžme ďalej pokračovať až na $i+j$-tom písmenku textu. Toto ale nemôžeme urobiť pre iné vzorky. KMP algoritmus je vlastne zovšeobecnením tejto myšlienky pre ľubovoľné vzorky.

Napríklad ak by sme hľadali v texte „bababaabbb“ vzorku „babaabbb“, tak zistíme na piatom mieste, že sa písmenká nezhodujú, ale musíme znovu začať hľadať vzorku od tretieho miesta. Teda stačí nám pre každú pozíciu j v vzorke vedieť miesto `next[j]`, kde musíme pokračovať hľadanie, ak sa na nej nachádza prvá nezhoda písmeniek vzorky a textu. Toto ale závisí len od vzorky, ktoré hľadáme a možeme si to predpočítať ešte pred samotným hľadaním

vzorky v texte. Všimnime si, že teraz už nepotrebujeme znovu porovnávať písmenká na menších pozíciach vo vzorke ako `next[j]`, lebo tie písmenká sa už určite s textom zhodujú.

V príklade vyššie: Prečítali sme prvé 4 písmená textu, zhodujú sa s prvými štyrmi písmenami vzorky. Piate písmeno textu a vzorky sú rôzne. Ale z informácie, že sme mali prečítané 4 písmená kúzla (bez toho, aby sme sa na text museli opäť pozerať) vieme, že predchádzajúce dve písmená boli „ba“, preto môžeme teraz mať prečítané prvé 3 písmená vzorky.

Ak sa nezhoduje j-te písmenko vzorky s textom a prvých $j-1$ písmeniek sa zhoduje, tak si v poli `next[j]` zapamätáme, od ktorého písmenka vo vzorke musíme znovu začať porovnávať vzorku s textom. Hľadajme teda `next[j]`. Zoberme si kópiu prvých $j-1$ písmeniek a položme ich pod vzorku tak, že prvé písmenko kópie leží pod druhým písmenkom vo vzorke. Posúvajme tú kópiu doprava pokiaľ sa už žiadne písmenká neprekrývajú alebo pokiaľ prekrývajúce sa písmenká sú navzájom rovnaké. Prekrývajúce sa písmenka určujú pozíciu, kde musíme znovu začať hľadať vzorku ak sa písmenka nezhodujú na j-tom mieste – `next[j]` vypočítame ako počet prekrývajúcich sa písmenok plus jedna, pretože ich počet určuje počet zhodujúcich sa písmenok textu a vzorky ak začneme na tejto pozícii a nám stačí začať porovnávať písmenká textu a vzorky až za nimi.

Presnejšie: pre $j > 1$ zoberme reťazec R tvorený prvými $j-1$ písmenami vzorky. Potom `next[j]` je rovné najväčšiemu takému $k < j$, že prefix R dĺžky $k-1$ je aj sufixom R (t.j. prvých $k-1$ písmen R je rovnakých ako posledných $k-1$).

Takže samotný algoritmus hľadania vzorky v texte bude vyzerať nasledovne: Porovnávajme i-te písmenko textu s j-tym písmenkom vzorky. Ak sa zhodujú, tak aj i aj j zvýšime o jedna a teda sa posunieme na ďalšie písmenko v texte aj vo vzorke. Ak sa nezhodujú, tak ďalšia možná pozícia v texte, od ktorej možeme začneme hľadať vzorku je `i-next[j]+1`, ale prvých `next[j]-1` písmeniek vo vzorke sa zhoduje s prvými `next[j]-1` písmenkami v texte, a teda i nemusíme zmeniť a stačí nám zmeniť j na `next[j]`. A aká je časová zložitosť? Pri každom porovnaní písmeniek zvýšime buď naraz i aj j, alebo znížime j. Teda tých porovnaní, kde zvýšime j je N a tých porovnaní, kde ho znížime môže byť znovu len N, pretože j nie je nikdy záporné. Teda dokopy máme iba $2N$ porovnaní písmeniek. Takže časová zložitosť tejto časti programu je $O(N)$. Všimnime si, že pre ľubovoľnú vzorku a text časová zložitosť tejto časti nezávisí vôbec od dĺžky vzorky.

Teraz nám už stačí vypočítať iba pole `next`. To môžeme urobiť nasledovnou fintou: `next[1]` môžeme dať definitoricky 0, čo nám vyhovuje pri implementácii. Všimnime si, že `next[2]` je vždy 1. Hľadajme vyššie uvedeným algoritmom vzorku v sebe samom, ale začnime hľadať až od druhého písmenka. Pozrime sa lepšie na moment po inkrementovaní i aj j, teda po posunutí o jedno písmenko v texte doprava. V tomto okamihu sa prvých $j-1$ písmeniek vzorky zhoduje s písmenkami na pozíciach $i-j-1, i-j, \ldots, i-1$, čo je posledných $j-1$ písmeniek z prvých $i-1$ písmeniek vzorky. A toto je zjavne najväčšie j s touto vlastnosťou. Takže môžeme do `next[i]` priradiť j. Tento algoritmus by mohol zhavarovať, ak by sme v ňom potrebovali vedieť `next[j]` skôr než ho vypočítame. Toto sa ale nemôže stať, lebo to by znamenalo, že $i-1$ písmeniek sa zhoduje s $j-1$ písmenkami, teda by sa i rovnalo j, teda by sme museli začať hľadať vzorku od prvého písmenka, čo je v spore s tým, že hľadáme od druhého písmenka. Takže táto časť má zložitosť $O(M)$, lebo robíme presne to isté ako v normálnom vyhľadávaní a navyše vypočítame pole `next`, ktorého veľkosť je M. Takže výsledná časová zložitosť je $O(N+M)$.

Listing programu:

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

int mod = 45435991;
int main()
```

```
{
    int n, m; scanf("%d %d", &n, &m);
    char a[100][100], b[100][100];
    for (int i = 0; i < n; i++)
        scanf("%s", a[i]);
    for (int i = 0; i < n; i++)
        scanf("%s", b[i]);

    vector<int> ha(n, 0), hb(n, 0);
    long long pow256 = 1;
    for (int k = 1; k <= m; k++)
    {
        bool ok = true;
        for (int i = 0; i < n; i++)
        {
            ha[i] = (ha[i] + pow256 * a[i][m - k]) % mod;
            hb[i] = (hb[i] * 256 + b[i][k - 1]) % mod;
            if (ha[i] != hb[i])
                ok = false;
        }

        //skontrolujeme, ci sa naozaj rovnaju
        for (int i = 0; i < n && ok; i++)
            for (int o = 0; o < k && ok; o++)
                if (a[i][m - k + o] != b[i][o])
                        ok = false;
        if (ok)
        {
            for (int i = 0; i < n; i++)
            {
                printf("%s", a[i]);
                printf("%s\n", b[i] + k);
            }
            break;
        }
        pow256 = (pow256 * 256) % mod;
    }
}
```

Opravoval Zemčo
(max. 25 bodov)

10. Televízni lenivci

Tento príklad nebol taký obtiažny na pomery úlohy číslo 10. Tri riešenia prišli vzorové, aj keď jedno bez kódu. Tie zvyšné dve boli bohužiaľ nekorektné, pričom bližšie zdôvodnenie je možné nájsť si v komentári na riešení. Troška ma prekvapilo, že korektné riešenia sa zhodli v myšlienke, pretože prístupov, ako riešiť túto úlohu, je celkom dosť. Hoci sú niektoré pomalšie a majú iné nedostatky, uvedieme si ich aspoň stručne (domyslieť detaily si môžete sami), pretože sú zaujímavé a v budúcnosti sa môže oplatiť ich vedieť. Počet robotníkov budeme, podobne ako v zadaní, označovať N.

Binárne vyhľadávanie

Jedným z prístupov je skúsiť hľadanú hodnotu veľkosti strany kamery binárne vyhľadať. Platí, že ak nám na zachytenie všetkých robotníkov stačí televízor veľkosti d, potom nám postačí aj hocijaká väčšia hodnota – vieme, že môžeme všetkých robotníkov zachytiť v tom istom momente ako pri použití kamery veľkosti d. Takže funkcia *dostatočnosti veľkosti strany*

televízora je monotónna a môžeme hodnotu, kedy sa mení z *nestačí* na *stačí* binárne vyhľadať. Dolný odhad potrebnej veľkosti je 0, horný je vzdialenosť robotníkov pri počiatočnom rozostavení. Ostáva vymyslieť, ako overiť, či nám nejaká uvažovaná hodnota h stačí. Treba si uvedomiť, že moment, kedy sa oplatí zachytiť všetkých robotníkov kamerou so stranou h je moment, keď sú nejakí dva robotníci vzdialení od seba presne h (premyslite si sami, prečo je to tak). Pre zvolenú dvojicu robotníkov ľahko nájdeme ten moment a potom už aj určíme oblasť staveniska, ktorú bude kamera zaberať. Následne stačí len pre všetkých ostatných overiť, či sa v nej budú nachádzať. Takže celé overenie nás bude stáť čas $O(N^3)$ (dá sa to zlepšiť na $O(N^2)$). Počet iterácií binárneho vyhľadávania je približne binárny logaritmus opísaneho štvorca pre počiatočné rozostavenie, ale závisí samozrejme aj od toho, ako veľmi presné riešenie chceme.

Ternárne vyhľadávanie

Ďalším smerom, kadiaľ sa mohlo vaše uvažovanie uberať a dospieť k úspešnému koncu, bolo všimnúť si, že ak sa nejakí dvaja robotníci vzdialia, potom sa už budú od seba vzďaľovať donekonečna (pre jednoduchosť si odmyslime Kolumba a Gagarina a považujme Zem za nekonečnú rovinu). Ďalej je zjavné, že dvaja robotníci sa síce môžu k sebe spočiatku približovať, ale určite nastane moment, kedy sa k sebe približovať prestanú. Posledné pozorovanie je, že ak sa vzdialenosť dvoch robotníkov nemení, potom sa nemení od začiatku pohybu až do konca sveta.

Skúsme teraz uvažovať funkciu $f(t)$, ktorá nám pre daný moment vráti veľkosť kamery potrebnej na zachytenie všetkých robotníkov. Inými slovami – štvorec opísaný aktuálnym pozíciam robotníkov (rovnobežný so súradnicovými osami). Ak by sme sa pokúsili ju nakresliť, mohla by vyzerať takto

- Najprv bude klesať a od istého momentu začne rásť.
- Chvíľu bude klesať, potom stagnovať a po chvíli začne rásť.
- Na začiatku bude stagnovať a po chvíli začne rásť.
- Od začiatku stagnovať až do nekonečna.

No a čuduj sa svete, toto sú jediné 4 možnosti. Prípad, kedy funkcia stagnuje až do nekonečna, sa ošetrí ľahko zvlášť, a na tie ostatné sa dá napasovať ternárne vyhľadávanie bodu, v ktorom funkcia začne rásť do nekonečna. V tomto bode je (možno jeden z mnohých) moment, kedy sa nám oplatí robotníkov zachytiť. Zistenie funkčnej hodnoty pre daný bod (okamih) nám trvá lineárny čas.

Pre úplnosť ešte treba povedať, čo to je ternárne vyhľadávanie. Nie je to žiadna mágia – predstavme si, že máme funkciu f, ktorá na intervale $\langle l, h\rangle$ chvíľu klesá a potom rastie. Chceme nájsť bod, v ktorom dosahuje minimum. Rozsekneme si interval na tretiny a pozrieme sa na dva body vnútri – $B_1 < B_2$ a funkčnú hodnotu v nich. Čo môžeme povedať o výskyte maxima, ak $f(B_1) < f(B_2)$? Na intervale $\langle B_2, h\rangle$ funkcia rastie[11] a preto hľadané minimum nemôže byť na intervale $\langle B_2, h\rangle$. Túto tretinu môžeme zahodiť a pokračovať v ďalšom pýtaní sa – máme nové hranice $\langle l, B_2\rangle$ (obdobne ošetríme aj prípad $f(B_1) > f(B_2)$). Skončíme, keď už sú hranice dosť blízko a prehlásime, že tu niekde to minimum je. Všeobecne sa pri funkciách vyžaduje striktný rast a striktné klesanie (bez striktnosti by bolo toto vyhľadávanie nepoužiteľné, porozmýšľajte prečo), v našom prípade ale môže nastať aj rovnosť. Tá nám ale vôbec nevadí, ba naopak. Keď sa pozrieme na 4 prípady vyššie, vidíme, že úsek stagnácie je zároveň úsek hľadaného minima a ak $f(B_1) = f(B_2)$, potom môžeme rovno skončiť, lebo výsledok sme našli (prísne vzaté, ešte sa môže stať, že natrafíme na rovnosť a nebude to úsek stagnácie. Vo všeobecnosti sa tento prípad neošetruje, lebo nastáva s minimálnou pravdepodobnosťou. V našom špecifickom prípade sa to ošetrí napríklad jednou pomocnou otázkou na stred medzi bodmi B_1 a B_2).

Vzorové riešenie

[11] Prečo? A prečo nemusí na celom intervale $\langle B_1, h\rangle$?

Uvedené dve riešenia spájala myšlienka, že sme nejaký bod vyhľadávali v priestore reálnych čísel a približovali sa k nemu[12]. Pri popisovaní vzorového riešenia zvolíme inú filozofiu a budeme problém riešiť priamym vypočítaním. Hoci okrem pomalého času patrí medzi negatíva uvedených riešení aj väčšia citlivosť na chyby pri práci s reálnymi číslami, ich pozitívum je v jednoduchosti implementácie. Súčasťou nášho vzorového postupu bude algoritmus, ktorý bude pripomínať riešenie úlohy Osakský Gusto - tento (26.) ročník, 1. kolo, úloha 8. Ak si na tento algoritmus spomínate, potom sa vám nasledujúce riešenie bude chápať oveľa jednoduchšie. V opačnom prípade čítajte pozorne a žujte pomaly, aby vám nezabehlo...

Na úvod si úlohu rozdelíme na x-ovú a y-ovú časť a na pobehujúcich robotníkov sa začneme dívať ako na lineárne funkcie. Ako sa nám to rozdelenie podarí a na čo to bude dobré? No každého robotníka ktorého popisuje jeho začiatočná pozícia $[p_x, p_y]$ a vektor pohybu (v_x, v_y), môžeme tiež popísať dvoma funkciami jednej premennej $f_x(t)$ a $f_y(t)$. Funkcia f_x nám vráti x-ovú pozíciu robotníka v čase t a funkcia f_y jeho y-ovú pozíciu. Funkcie sa ľahko definujú predpisom $f_x(t) = v_x t + p_x$, $f_y(t) = v_y t + p_y$.

Uvažujme teraz len jednu súradnicu – x. Predstavme si, že chceme nájsť funkciu, kde je premenná čas a ktorá nám povie, ktorý robotník má v danom čase najvyššiu súradnicu x (na rovine by bol najviac na východ). Zjavne, na riešenie tejto úlohy nepotrebujeme informácie o pohybe po y-ovej súradnici a preto nám to rozdielnie vec zjednodušilo a pomohlo. Tí, ktorí poznajú riešenie úlohy o Osakskom Gustovi, môžu zvyšok odstavca a ďalšie dva preskočiť. My ostatní si najprv uvedomme, že v čase 0 budú robotníci rozostavení podľa ich začiatočných bodov. Neskôr sa začnú hýbať a tí, ktorí sa hýbu rýchlejšie začnú predbiehať tých, ktorí sa hýbu pomalšie. Inými slovami, rozhodujúca je hodnota v_x a ak sú dvaja robotníci, ktorí majú x-ovú zložku vektora posunu $v_{1_x} < v_{2_x}$, tak potom alebo bude druhý robotník celý čas východnejšie ako prvý, alebo možno na začiatku nebude, ale určite nastane moment, kedy toho pomalšieho predbehne a potom až do nekonečna bude východnejšie on. Z tejto myšlienky teraz spravíme algoritmus.

Výsledkom nášho algoritmu bude zoznam dvojíc (čas, robotník), ktorá bude hovoriť, že od nejakého času bude najvýchodnejší daný robotník. Zoznam bude utriedený rastúco podľa času a robotník bude vždy najvýchodnejší, až kým nenastane čas ďalšej položky, kedy začne byť najvýchodnejší iný robotník. Posledný robotník bude najvýchodnejší až donekončna. Majme utriedených robotníkov rastúco podľa rýchlosti pohybu a v prípade rovnosti podľa bodu začiatku. Začneme tak, že si zoberieme prvého robotníka (s číslom 0) a prehlásime, že on je najvyšší po celý čas – vložíme do zoznamu prvú položku $(0.0, 0)$. Postupujeme ďalším robotníkom – on určite tohto jedného niekedy pretne (alebo nepretne, ale bude celý čas nad ním). Toto ľahko vyrátame – pre robotníkov $R_0 = k_0 t + q_0$ a $R_1 = k_1 t + q_1$ ten čas nájdeme zo vzťahu $t = \frac{q_1 - q_0}{k_0 - k_1}$. Ak je tento čas menej ako 0, potom to znamená, že nový robotník bude východnejšie celý čas – v takom prípade zo zoznamu odstránime prvok $(0.0, 0)$ a vložíme $(0.0, 1)$. Ak sa stretnú v čase $t > 0$, potom do zoznamu na koniec len pridáme položku $(t, 1)$, takže zoznam bude vyzerať $(0.0, 0), (t, 1)$. Všimnite si, že platí, že zoznam je utriedený rastúco podľa času.

Takto budeme postupovať všeobecne – máme zoznam, v ktorom už niektoré položky sú a navyše sú utriedené podľa času. Zoznam bude fungovať ako stack. Ak budeme uvažovať nového robotníka R_i, ten raz predbehne všetkých doteraz uvažovaných (alebo nepredbehne, ale bude východnejšie). Ak predbehne, otázka znie – kedy? Postupujeme teda tak, že si vezmeme posledného robotníka v zozname a zistíme prienik s R_i. Ak R_i predbehne robotníka v zozname skôr ako sa ten stane najvyšším, potom tento starý nebude najvýchodnejší nikdy a zo zoznamu ho vyhodíme. Pokračujeme robotníkom, ktorý je na vrchu zoznamu po vyhodení a takto až dovtedy, kým nebude zoznam prázdny (čo znamená, že R_i bude spomedzi všetkých doteraz uvažovaných robotníkov celý čas najvýchodnejší), alebo sa R_i nestretne s robotníkom

[12] Učenými slovami – konvergovali k nemu.

na vrchu zoznamu počas toho, ako bude robotník v zozname najvýchodnejší – vtedy len pridáme položku (R_i, t), kde t príslušný čas predbehnutia na vrch zoznamu. Po spracovaní všetkých robotníkov budeme mať presne zoznam aký chceme – ktoré úseky je ktorý robotník najvýchodnejšie.

Keď už teraz vieme, ako zistiť, kto bude kedy najvýchodnejšie, ľahko sa to upraví na zisťovanie najjužnejšieho, najzápadnejšieho a najsevernejšieho. Pre najzápadnejšieho stačí len upraviť porovnávanie tak, aby bolo treba mať čo najnižšiu hodnotu posunu a nie najvyššiu. Keď máme celý čas sveta rozdelený takýmto spôsobom na úseky, potom už len treba nejak vedieť nájsť výsledok. Dalo by sa skonštruovať funkcie rozdielov v zvislom a horizontálnom smere – čiary, ktoré by nám hovorili pre každý moment, ako ďaleko sú od seba najvzdialenejší robotníci v jednom či druhom smere. Potom by bol výsledok alebo moment stretnutia týchto čiar, alebo maximum z miním. Tento prístup volil Tobiáš, my ale pôjdeme jednoduchšou cestou.

Ak skúsime spojiť všetky štyri rozsekania času do jedného, dostaneme hustejšie rozsekaný čas na menšie úseky. Čo ale platí? Počas každého takéhoto malého úseku sú stále tí istí robotníci najsevernejší, najjužnejší, najzápadnejší a najvýchodnejší. Práve zmena niektorého z krajných robotníkov nás zakaždým privedie do iného úseku. No ale keď sú na okraji tí istí robotníci, to znamená, že aj rozmery štvorca kamery potrebnej na ich zachytenie sa počas úseku nemôžu meniť len tak hocijako. Presnejšie, potrebný x-ový rozmer sa dá popísať lineárnou funkciu, ktorá je rozdielom pohybu najvýchodnejšieho a najzápadnejšieho robotníka, obdobne y-ový rozmer. No a keď máme úsek a dve lineárne funkcie, tak potom ľahko nájdeme miesto, kedy budú robotníci na tomto úseku najbližšie a bude stačiť najmenšia kamera – môže to byť na začiatku, na konci, ale teoreticky aj niekde v strede, ale to ľahko dorátame. V strede to totiž bude vtedy a len vtedy, keď sa tieto obe lineárne funkcie rozdielov pretnú v danom časovom úseku. Optimálna hodnota bude potom tento ich prienik. Koniec úseku v našej implementácii nevyrátavame, pretože je to rovné začiatku ďalšieho úseku. Preto ľahko zistíme hodnotu na začiatku úseku a prípadnú najvýhodnejšiu hodnotu niekde v strede úseku. Keď takto prejdeme všetky úseky, nájdeme najvýhodnejší moment a najmenšiu potrebnú veľkosť kamery.

Skúsme popísať časovú zložitosť. Na začiatku si vytvárame $4N$ funkcií, potom robíme 4 triedenia v celkovom čase $O(N \log N)$. Následne štyrikrát lineárne rozsekávame čas na úseky. Lineárne preto, lebo do zoznamu každú funkciu práve raz pridáme a najviac raz odstránime a iné operácie nerobíme. Potom už len prechádzame úseky a každý riešime v konštantnom čase, pretože o každom vieme v konštantnom čase povedať, ktorí robotníci a zodpovedajúce funkcie pohybu musíme uvažovať. Takže celkový čas nášho komplikovaného riešenia je $O(N \log N)$. Pamäťová náročnosť je zjavne $O(N)$.

Listing programu:

```
#include <cstdio>
#include <vector>
#include <deque>
#include <iostream>
#include <cmath>
#define INF 10e12
#define EPS 10e-8
using namespace std;
//typ pre linearnu funkciu - first=k, second = q, STL nam da zadarmo comparator aky nam treba
typedef pair<int,int> funkcia;
//zaznam v utriedenom zozname funkcii, okrem funkcie si pamatame aj povodny index
typedef pair<funkcia,int > funkcia_zaznam;
double mx(double a,double b){return a < b ? b : a;}
double mn(double a,double b){return a > b ? b : a;}
```

```
double fh(funkcia f, double t){ return ((double)f.first)*t+(double)f.second; }
//metoda vrati zoznam (okamih, index funkcie), ze kedy bola ktora funkcia najvyssia
vector< pair<double,int> > zorad_funkcie( vector<funkcia_zaznam> f ){
   //pouzijeme deque, aby sme mohli pracovat s jednym koncom a napokon z druheho
skopirovat do vectora
   deque< pair <double,int> > vratim; vratim.clear();
   //na zaciatku je najpomalsie rastuca funkcia najvyssia az do nekonecna
   vratim.push_front(make_pair(0.0,0));
   int size=1;
   for (int i=1;i<(int)f.size();i++){
      int w = size-1;
      //vo while cykle postupne vyhadzujeme polozky zo zoznamu
      while (true){
         pair<double,int> p = vratim.front();
         double d = -1.0;
         if (f[i].first.first != f[p.second].first.first){
             d = ( (double)(f[i].first.second -
f[p.second].first.second)/(double)(f[p.second].first.first - f[i].first.first) );
         }
         if (d <= p.first){
            vratim.pop_front();
            w--;
            if (w < 0){
               vratim.push_front(make_pair(0.0,i));
               w=0;
               break;
            }
         }
         else{
            vratim.push_front(make_pair(d,i));
            w++;
            break;
         }
      }
      size = w+1;
   }
   vector< pair <double,int > > v(0);
   while (vratim.size()>0){ v.push_back( vratim.back() ); vratim.pop_back(); }
   return v;
}

int main(){
   int N;
   scanf("%d ",&N);
   //zaznamy pre funkcie kvoli ich zoradeniu v prislusnych smeroch
   vector<funkcia_zaznam> fxmax(0),fxmin(0),fymax(0),fymin(0);
   vector<funkcia> vstupx(0),vstupy(0);;
   for (int i=0;i<N;i++){
      int px,py,vx,vy;
      scanf("%d %d %d %d ",&px,&py,&vx,&vy);
      vstupx.push_back(make_pair(vx,px));
      vstupy.push_back(make_pair(vy,py));
      //aby sme mohli pouzit jednu zoradovaciu metodu pre vsetky
      //4 smery, tak si funkcie trikovo upravime
      fxmax.push_back(make_pair( make_pair(vx,px),i ));
      fxmin.push_back(make_pair( make_pair(-vx,-px),i ));
      fymax.push_back(make_pair( make_pair(vy,py),i ));
      fymin.push_back(make_pair( make_pair(-vy,-py),i ));
   }
   sort(fxmax.begin(),fxmax.end());
```

```
    sort(fxmin.begin(),fxmin.end());
    sort(fymax.begin(),fymax.end());
    sort(fymin.begin(),fymin.end());
    vector<pair<double,int> > xmax = zorad_funkcie(fxmax);
    vector<pair<double,int> > xmin = zorad_funkcie(fxmin);
    vector<pair<double,int> > ymax = zorad_funkcie(fymax);
    vector<pair<double,int> > ymin = zorad_funkcie(fymin);
    //ako mantinel si vsade dame na koniec nekonecno, kvoli lahsiemu spracovavaniu
    pair<double,int> p = make_pair(INF,-1);
    xmax.push_back(p); xmin.push_back(p); ymax.push_back(p); ymin.push_back(p);
    int usekov = xmax.size()+xmin.size()+ymax.size()+ymin.size();
    //indexy, kde sme v zoznamoch udalosti, kedy je ktora funkcia naj...
    int xmaxi=0,xmini=0,ymaxi=0,ymini=0;
    double best = INF; double cas = 0.0;
    for (int i=0;i<usekov;i++){
        //vyratame funkciu rozdielu x a y suradnice na tomto useku
        funkcia xrozdiel = make_pair(vstupx[ fxmax[xmax[xmaxi].second].second  ].first
- vstupx[ fxmin[xmin[xmini].second].second ].first, vstupx[
fxmax[xmax[xmaxi].second].second ].second - vstupx[
fxmin[xmin[xmini].second].second ].second);
        funkcia yrozdiel = make_pair(vstupy[ fymax[ymax[ymaxi].second].second  ].first
- vstupy[ fymin[ymin[ymini].second].second ].first, vstupy[
fymax[ymax[ymaxi].second].second ].second - vstupy[
fymin[ymin[ymini].second].second ].second);
        //zistime, kedy sa usek konci ( = zacina najblizsi)
        double najblizsi=INF; int posun=-1;
        if (xmax[xmaxi+1].first < najblizsi){najblizsi = xmax[xmaxi+1].first; posun=0;}
        if (xmin[xmini+1].first < najblizsi){najblizsi = xmin[xmini+1].first; posun=1;}
        if (ymax[ymaxi+1].first < najblizsi){najblizsi = ymax[ymaxi+1].first; posun=2;}
        if (ymin[ymini+1].first < najblizsi){najblizsi = ymin[ymini+1].first; posun=3;}
        //aku velku klietku nam treba na zaciatku?
        double d = mx( fh(xrozdiel,cas),fh(yrozdiel,cas));
        //zistime, ci nam nestaci niekedy v prostred useku mensia klietka
        double doba_stretu = -1.0;
        if (xrozdiel.first != yrozdiel.first){
              doba_stretu = ( (double)(xrozdiel.second -
yrozdiel.second)/(double)(yrozdiel.first - xrozdiel.first) );
        }
        if ( (fabs(najblizsi-INF) > EPS) && (cas <= doba_stretu && doba_stretu <=
najblizsi)){
            d = mn( d, mx( fh(xrozdiel,doba_stretu),fh(yrozdiel,doba_stretu))  );
        }
        best = mn(d,best);
        cas = najblizsi;
        if (posun==0){ xmaxi++; }
        if (posun==1){ xmini++; }
        if (posun==2){ ymaxi++; }
        if (posun==3){ ymini++; }
        if (fabs(INF-cas) < EPS) i = usekov;
    }
    printf("%.10lf\n",best);
    return 0;
}
```

Výsledková listina po 1. kole kategórie KSP-Z

Chýba súbor Z!!!

Výsledková listina po 1. kole kategórie KSP-O

Chýba súbor O!!!

Výsledková listina po 1. kole kategórie KSP-T

Chýba súbor T!!!