Korešpondenčný seminár z programovania
XXVIII. ročník, 2010/11
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

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

Vzorové riešenia 2. kola zimnej časti

1. Zmiešaná postupnosť

opravovala Dominika
(max. 10 bodov)

Niektorí riešitelia sa nechali nachytať a rátali s tým, že na vstupe dostanú usporiadanú postupnosť. To nie je pravda, a preto ich riešenie nebolo správne, dostali maximálne 2 body. Vzorové riešenie malo lineárnu časovú zložitosť $O(N)$ a konštantnú pamäťovú zložitosť $O(1)$. Body som sťahovala za horšiu časovú zložitosť, za chýbajúci popis k riešeniu a za chýbajúci alebo zlý odhad zložitosti.

Správne riešenie môže použiť rôzne prístupy. Ja som si zvolila najjednoduchší z nich. Najprv si musíme uvedomiť, že na vstupe dostaneme aspoň tri prvky a že všetky prvky v postupnosti sú rôzne. Ak sú prvé tri prvky postupnosti neusporiadané, potom je neusporiadaná celá postupnosť.

Preto nám stačí kontrolovať prvé tri prvky postupnosti. Ak sú usporiadané, tak vymeníme prvý a druhý prvok, z čoho nám vznikne neusporiadaná postupnosť. Ak sú prvé tri prvky neusporiadané, tak ich vypíšeme odzadu, aby sme dostali postupnosť rôznu od pôvodnej, pričom neporušíme podmienku neusporiadanosti. Zvyšné prvky postupnosti nám potom jednoducho stačí načítavať a vypisovať v jednom cykle a s jednou premennou.

Listing programu:

```
program Zmiesana_postupnost;
var N, a, b, c, i: longint;

begin
  ReadLn(N);
  Read(a);
  Read(b);
  Read(c);
  if ((a < b) and (b < c)) or ((a > b) and (b > c)) then
    Write(b, ' ', a, ' ', c)
  else
    Write(c, ' ', b, ' ', a);

  for i := 4 to n do begin
    Read(a);
    Write(' ', a);
  end;
  WriteLn;
end.
```


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

2. Zase tie závislosti

opravoval Hermi
(max. 10 bodov)

Úvodom musím povedať, že tento príklad bol z ľahších a väčšina riešiteľov získala plný počet. Ak niekto nejaké body stratil, mal buď kvadratickú pamäť, zlý/chýbajúci odhad, alebo vôbec nezdôvodnil, prečo by mal jeho program dávať dobrý výsledok. Poďme ale priamo na riešenie.

Balíčky a ich závislosti si reprezentujeme *grafom*. Graf tvoria *vrcholy* a *hrany*, ktoré spájajú niektoré dvojice vrcholov. V našom prípade sú vrcholmi jednotlivé balíčky, hranami sú zase závislosti medzi nimi. Pretože závislosť nie je symetrická, naše hrany budú *orientované* (kreslíme ich ako šípku z jedného vrcholu do druhého). Presnejšie, hrana povedie z A do B, ak je balíček A závislý na balíčku B. Keďže podľa zadania závislosti netvoria cyklus, náš graf je *acyklický*.

Pozrime sa teraz na nejaký balíček, označme ho B. Jednoduchou úvahou zistíme, že ak existuje balíček A, ktorý závisí na B, tak nemá zmysel B inštalovať. Ak totiž namiesto B nainštalujeme A, dostaneme všetky balíčky, ktoré by sme dostali nainštalovaním B, a aspoň jeden navyše – samotný balíček A. (Keďže v závislostiach nie je cyklus, nainštalovaním B by sme A určite nedostali.)

Ak je teda na balíčku nejaký iný závislý, nemusíme ho inštalovať. Naopak, ak na balíčku nie je nič závislé, musíme ho nainštalovať ručne, lebo by nám ho nič iné automaticky nenainštalovalo. Potrebujeme teda vyberať práve tie balíčky, na ktorých nie je nič závislé.

Samotný algoritmus je jednoduchý. Do poľa `boolean`ov si budeme počas načítavania vstupu pre každý balíček značiť, či je na ňom nejaký iný balíček závislý. Nepotrebujeme si zapamätať celý vstup, takže nám stačí $O(N)$ pamäte (závislostí môže byť teoreticky až $O(N^2)$).

Časová zložitosť tohto algoritmu je $O(N^2)$ – počet čísiel, ktoré treba načítať. Prípadne môžeme povedať, že je to $O(M)$, kde M je celkový počet závislostí.

Listing programu:

```
program Zavislosti;
var n, k, c, i, j: integer;
    videl: array [1..10000] of boolean;

begin
  ReadLn(n);
  { predpokladáme, že treba nainštalovať všetky }
  for i := 1 to n do
    videl[i] := false;
  for i := 1 to n do begin
    Read(k);
    for j := 1 to k do begin
      { postupne si označujeme balíčky, ktoré netreba }
      Read(c);
      videl[c] := true;
    end;
  end;
  { na konci vypíšeme balíčky, ktoré treba nainštalovať }
  for i := 1 to n do
    if not videl[i] then Write(i, ' ');
  WriteLn;
end.
```


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

3. Zábudlivý dôchodca

opravoval Maják
(max. 10 bodov)

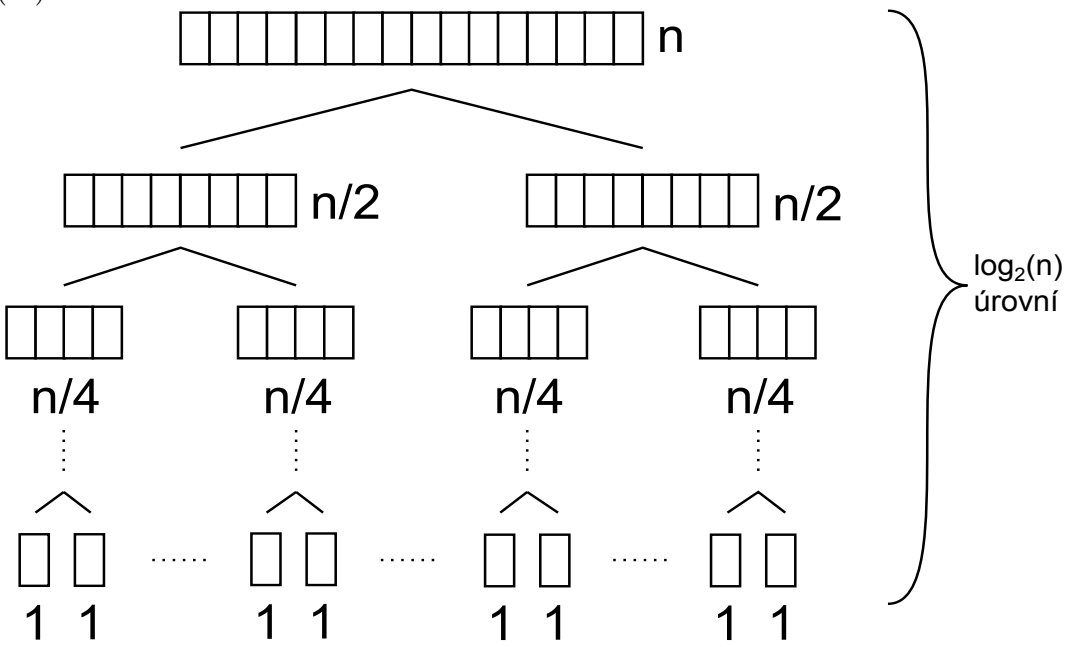
Pán Michal dokáže na niekoľko krokov vymeniť ľubovoľné dva prvky na párnych pozíciách (podobne pre nepárne), určite ale nedokáže vymeniť nijaký prvok z párnej pozície s prvkom na nepárnej. Preto môže triediť samostatne postupnosť zloženú z prvkov na párnych pozíciách a postupnosť zloženú z prvkov na nepárnych pozíciách. Tieto dve špeciálne podpostupnosti budeme ďalej nazývať len podpostupnosti.

Na zistenie toho, či dokáže utriediť celú postupnosť, sa stačí pozrieť, či bude usporiadaná po usporiadaní podpostupností. Prečo? Vždy, keď je postupnosť usporiadaná, tak sú aj jej podpostupnosti. Takže nemôže nastať taká situácia, pri ktorej by podpostupnosti neboli utriedené a samotná postupnosť by bola.

Prejdime teraz k najdôležitejšej časti riešenia, a to k triedeniu. Existuje viac spôsobov, ako triediť, my si ukážeme Merge sort[1]. Funguje na princípe „rozdeľuj a panuj“. To znamená, že si náš problém najskôr rozdelíme na menšie, ktoré vieme jednoducho vyriešiť. Následne tieto vyriešené podproblémy spojíme a dostaneme riešenie celého problému.

Postupnosť budeme rozdeľovať tak, že ju v polovici rozstrihneme a dostaneme dve menšie. To budeme opakovať s novými postupnosťami, kým sa nedostaneme k jednému alebo dvom prvkom, pretože tie vieme jednoducho hneď utriediť.

Spájanie dvoch už utriedených postupností nie je zložité. Stačí vždy porovnať najmenšie prvky z nich, a ten menší preložiť do spojenej postupnosti. Tento postup robíme dovtedy, kým sme obe postupnosti nespojili. Ak ich dĺžka je dokopy N, tak zložitosť tohto spojenia je $O(N)$.



Obrázok naznačuje, ako bude beh tohto algoritmu vyzerať. Úrovní je $O(\log N)$, pretože celá postupnosť má N prvkov a v každom rekurzívnom volaní sa rozdelí na polovicu. Na každej úrovni dokopy spájame N prvkov, čo nám zaberie $O(N)$ času. Spolu za všetky úrovne to bude $O(N \log N)$

Pamäťová zložitosť bude $O(N)$, pretože okrem vstupnej postupnosti a pomocných polí pri spájaní si nič iné nepotrebujeme pamätať.

Plný počet bodov bol za optimálne riešenie s odhadmi zložitostí a s vysvetlením, prečo je váš algoritmus správny. Za riešenie v $O(N^2)$ ste mohli dostať najviac 7 bodov. Za pomalšie

[1] triedenie spájaním

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

ešte menej. A za úplne nefunkčné riešenia ste nemohli očakávať viac ako dva body, čo bol prípad najmä tých z vás, ktorí nepochopili zadanie.

Listing programu:

```
program Dochodca;
var n, i, j, k: integer;
    ok: boolean;
    podpostupnosti: array [1..2, 1..10000] of integer;
    pomocne: array [1..10000] of integer;

procedure mergesort(zaciatok, koniec, p: integer);
var polovica: integer;
begin
  polovica := (koniec - zaciatok) div 2 + zaciatok;
  { ak mame viac prvkov ako dva, tak sa rekurzivne zavolam na mensie casti }
  if (koniec - zaciatok) >= 2 then begin
    mergesort(zaciatok, polovica, p);
    mergesort(polovica + 1, koniec, p);
  end;
  { ak nemame len jeden prvok, tak spojim dve utriedene casti }
  if zaciatok < koniec then begin
    for i := zaciatok to koniec do
      pomocne[i] := podpostupnosti[p, i];
    i := zaciatok;
    j := polovica + 1;
    for k := 0 to koniec - zaciatok do begin
      if (j > koniec) or ((i <= polovica) and (pomocne[i] <= pomocne[j])) then
begin
        podpostupnosti[p, zaciatok + k] := pomocne[i];
        Inc(i);
      end else begin
        podpostupnosti[p, zaciatok + k] := pomocne[j];
        Inc(j);
      end;
    end;
  end
end;

begin
  { nacitame data }
  ReadLn(n);
  for i := 0 to n - 1 do
    Read(podpostupnosti[(i mod 2) + 1, (i div 2) + 1]);

  { usporiadame podpostupnosti }
  mergesort(1, (n + 1) div 2, 1);
  mergesort(1, n div 2, 2);

  { overime, ci je aj cela postupnost utriedena }
  ok := true;
  for i := 0 to n - 2 do
    if podpostupnosti[(i mod 2) + 1, (i div 2) + 1] >
       podpostupnosti[((i + 1) mod 2) + 1, ((i + 1) div 2) + 1] then
      ok := false;
  if ok then
    WriteLn('Ano.')
  else
    WriteLn('Nie.');
end.
```


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

opravovalo sa samo, vzorák Mišof
(max. 15 bodov)

4. Zákerné kopce

Vždy, keď sa vás zadanie úlohy pýta na najmenší počet krokov, na ktorý sa dá nejaký cieľ dosiahnuť, mali by sa vaše myšlienky okamžite vydať nasledujúcim smerom: Podľa situácie v úlohe si zostrojíme vhodný graf a v tomto grafe nájdeme najkratšiu cestu. Graf vznikne zakaždým rovnako: jeho jednotlivé vrcholy zodpovedajú stavom, v ktorých sa môžeme nachádzať, a hrany vychádzajúce z vrcholu zodpovedajú ťahom, ktoré môžeme v danej situácii spraviť.

Ako to bude vyzerať v našom prípade? Stavy, teda vrcholy grafu, budú pochopiteľne jednotlivé pozície, na ktorých sa Artur môže nachádzať. Keďže $R, C \leq 1000$, bude náš graf teda mať nanajvýš milión vrcholov. Každý vrchol vieme jednoznačne popísať dvomi číslami: jeho súradnicami. No a z každého vrcholu povedú hrany zodpovedajúce krokom, ktoré v tej chvíli vie Artur spraviť. Keď si napríklad zoberiete príklad vstupu a výstupu v zadaní, tak z vrcholu $[2, 4]$ (políčko s výškou 8) povedú dve hrany: na políčka $[3, 2]$ (s výškou 5) a $[4, 6]$ (s výškou 6). Pre ľubovoľný vrchol bude vychádzajúcich hrán najviac 24, lebo len z toľkých okolitých políčok máme na výber.

Keď už sme zostrojili graf, zostáva v ňom nájsť najkratšiu cestu. Na to použijeme algoritmus nazvaný prehľadávanie do šírky (breadth-first search, BFS, niekedy tiež nazývané flood-fill). Toto prehľadávanie predstavuje spôsob, akým by daný graf preskúmala voda, šíriaca sa zo začiatočného vrcholu rovnomerne do všetkých smerov: najskôr spracujeme začiatočný vrchol (vzdialenosť doň je 0), potom postupne všetkých jeho susedov (do každého z nich je vzdialenosť 1), potom susedov jeho susedov (vzdialenosť 2), a tak ďalej.

Prehľadávanie do šírky ľahko implementujeme pomocou dátovej štruktúry fronta:

```
prehľadaj(vrchol v):
    označ v ako navštívený
    vyrob frontu Q obsahujúcu jediný prvok v
    kým Q nie je prázdna:
        vyber zo začiatku fronty Q prvok x
        pre každú hranu vedúcu z x:
            ak jej koncový vrchol w nie je navštívený:
                označ w ako navštívený
                vzdialenosť do w = 1 + vzdialenosť do v
                vlož w na koniec fronty Q
```

Časová aj pamäťová zložitosť tohto riešenia je lineárna od veľkosti mapy, teda $O(RC)$. Prečo to tak je? V prvom rade si uvedomme, že máme RC vrcholov a najviac $24RC$ hrán, teda dokopy máme $O(RC)$ vrcholov a hrán. Každý vrchol najviac raz vložíme do fronty (v okamihu, keď sa nám doň prvýkrát podarí dostať) a najviac raz ho z nej potom vyberieme. Každú hranu tiež spracujeme najviac raz: vtedy, keď spracúvame vrchol, z ktorého vedie. Celkový počet krokov, ktoré náš algoritmus spraví, je preto priamo úmerný počtu vrcholov a hrán v našom grafe.

V našej implementácii si nepotrebujeme samostatne značiť, ktoré vrcholy už sú navštívené: poznáme to jednoducho tak, že akonáhle má vrchol vzdialenosť inú ako nekonečno, už sme ho niekedy navštívili.

Listing programu:

```
const nekonecno = 123456789;
var R, C, K, r0, c0, rF, cF, r1, c1, r2, c2, i, j: longint;
    mapa, vzdialenost: array [1..1000, 1..1000] of longint;
    fronta: array [1..1000047, 1..2] of longint;
```


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

```
    fz, fk: longint; { premenne ukazujuce na zaciatok a *za* koniec fronty }

begin
  { nacitame vstup }
  Read(R, C, K, r0, c0, rF, cF);
  for i := 1 to R do
    for j := 1 to C do
      Read(mapa[i, j]);

  { inicializujeme premenne }
  for i := 1 to R do
    for j := 1 to C do
      vzdialenost[i, j] := nekonecno;
  vzdialenost[r0, c0] := 0;
  fz := 1;
  fk := 2;
  fronta[fz, 1] := r0;
  fronta[fz, 2] := c0;

  { prehladavame graf }
  while fk > fz do begin
    { vyberieme z fronty vrchol }
    r1 := fronta[fz, 1];
    c1 := fronta[fz, 2];
    Inc(fz);

    { najdeme vsetky vrcholy, kam sa mozeme pohnut }
    for i := -2 to 2 do
      for j := -2 to 2 do begin
        r2 := r1 + i;
        c2 := c1 + j;

        { overime, ci smieme spravit krok z [r1, c1] na [r2, c2] }
        if (r2 < 1) or (r2 > R) or (c2 < 1) or (c2 > C) then
          Continue;
        if abs(mapa[r2, c2] - mapa[r1, c1]) > K then
          Continue;

        { ak sme na [r2, c2] este neboli, zaradime ho do fronty }
        if vzdialenost[r2, c2] <> nekonecno then
          Continue;
        vzdialenost[r2, c2] := vzdialenost[r1, c1] + 1;
        fronta[fk, 1] := r2;
        fronta[fk, 2] := c2;
        Inc(fk);
      end;
  end;

  { vypiseme vystup }
  if vzdialenost[rF, cF] = nekonecno then
    WriteLn('Artur, mas smolu.')
  else
    WriteLn(vzdialenost[rF, cf]);
end.
```


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

5. Ošemetná situácia

opravoval Tomi
(max. 15 bodov)

Normálne síce zoradenie N-prvkového poľa musí trvať aspoň $O(N \log N)$ času, ale ak sa vopred vie, že v poli je len málo rôznych prvkov, ide to rôznymi trikmi zlepšiť. Tu môžu byť na jednom políčku len tri rôzne čísla a to sa dá využiť.

Riešenie s $O(N)$ výmenami, $O(N)$ časom a $O(N)$ pamäťou je napríklad takéto: V cykle prejdeme poľom zo začiatku na koniec a všetky jednotky, ktoré nájdeme, dáme na začiatok (t.j. vymeníme tú jednotku s najskorším políčkom, kde jednotka ešte nie je – to si pamätáme v pomocnej premennej). Potom rovnako pôjdeme od konca a dáme trojky na koniec. Všetky jednotky sú určite na začiatku a všetky trojky na konci, takže dvojky musia byť v strede.

Tento postup výmenami veľmi nešetrí – dokopy sa mohlo stať až $2N$ výmen a to je často zbytočne veľa. Ale pri O-notácii je jedno, či je to $N/2$ alebo $3N + 47$, pre O-notáciu je to všetko stále $O(N)$.

Lepšie ako $O(N)$ času to nepôjde, lebo to by sme ani nestihli načítať celý vstup. Lepšie ako $O(N)$ výmen to nepôjde, lebo existujú vstupy, ktoré očividne potrebujú aspoň $N/2$ výmen – napríklad vstup pozostávajúci z $N/2$ dvojok a $N/2$ jednotiek. Ide ešte vôbec niečo zlepšiť?

Ide – pamäťová zložitosť. Tento program má tú nevýhodu, že musí všetky vstupné dáta najprv naraz načítať do pamäte a až potom začne rátať výstup. Keď má vstupný súbor tri terabajty, tak to začne byť problém. Ale existuje riešenie, ktoré už za behu vypisuje výsledok a potrebuje len $O(1)$ pamäte.

Keby sme už mali načítanú nejakú časť vstupu a táto časť by náhodou bola správne zoradená, nemuseli by sme si ju celú pamätať – vieme, že tam je najprv A jednotiek, potom B dvojok a C trojok. Stačí vedieť, koľko sú A, B, C. A keď načítame ďalší prvok, môžeme to udržať zoradené – stačí poposúvať hranice medzi jednotkovou, dvojkovou a trojkovou oblasťou a na to vždy treba najviac dve výmeny.

Takže: na začiatku nemáme načítané nič, a „nič“ je zoradené. A s každým ďalším prvkom si to vieme udržať, takže aj na konci to bude celé zoradené. A stačia nám na to len tri premenné A, B, C. Máme pamäť $O(1)$ a nepokazili sme si ani čas $O(N)$, ani výmeny $O(N)$.

Optimálne riešenia mohli dostať 15 bodov. Funkčné a dobre popísané riešenia s dobrým časom a výmenami, ale $O(N)$ pamäťou, dostali väčšinou 12 bodov. Pomalšie riešenia, napríklad $O(N^2)$, dostávali okolo piatich bodov. Body sa strhávali hlavne za chybnú implementáciu (t.j. myšlienka je dobre, ale program nejaké vstupy nesprávne rieši) a za chýbajúci alebo nesprávny odhad zložitosti.

Listing programu:

```
var N, I, A, B, C, p: integer;
begin
  Read(N);
  A := 0;
  B := 0;
  C := 0;
  for I := 1 to N do begin
    { teraz je to A jednotiek, B dvojok, C trojok... a nieco nacitam }
    Read(p);
    if p = 1 then begin
      { teraz je to A jednotiek, B dvojok, C trojok, 1 jednotka }
      if C <> 0 then WriteLn(A + B + 1, ' ', I);
      { teraz je to A jednotiek, B dvojok, 1 jednotka, C trojok }
      if B <> 0 then WriteLn(A + 1, ' ', A + B + 1);
      { teraz je to A jednotiek, 1 jednotka, B dvojok, C trojok }
      A := A + 1;
    end;
```


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

```
    if p = 2 then begin
      { teraz je to A jednotiek, B dvojok, C trojok, 1 dvojka }
      if C <> 0 then WriteLn(A + B + 1, ' ', I);
      { teraz je to A jednotiek, B dvojok, 1 dvojka, C trojok }
      B := B + 1;
    end;
    if p = 3 then begin
      { teraz je to A jednotiek, B dvojok, C trojok, 1 trojka }
      C := C + 1;
    end;
    { a uz je to zase A jednotiek, B dvojok, C trojok }
  end;
end.
```

opravoval Piťo

6. O švédskych stoloch

(max. 20 bodov)

Ak by sme vedeli rozhodnúť, či sa niektorých k účastníkov vie najesť, mohli by sme to vyskúšať pre všetky k od 1 po N. Ak by to pre nejaké k bolo možné a pre $k+1$ už nie, potom k by bolo hľadané maximum. Prečo? Lebo ak pre ľubovoľných $k+1$ účastníkov nie je dosť jedla, tak pridaním ľubovoľného ďalšieho si všetci budú chcieť priložiť a aj on si bude chcieť nabrať, no nikto už nebude mať z čoho. Zároveň aj pre všetky menšie počty ako k sa vedia najesť, pretože stačí z tých k účastníkov niekoľko odobrať. Okrem toho, že oni nebudú jesť, aj tí zvyšní zjedia menej.

Aby sme vedeli rozhodnúť, či sa dokáže niektorých k účastníkov najesť, stačí si uvedomiť, že ak sa vie najesť niektorých k, určite sa vie aj tých k, čo zjedia najmenej. Vieme presne určiť, koľko každý zje, pretože ich počet – k – si pevne stanovíme.

Teraz už vieme napísať jednoduché riešenie. Postupne pre všetky k od 1 do N opakujeme nasledovný postup: Pre každého účastníka si vypočítame, koľko by zjedol, ak by bol pri stole spolu s $k-1$ inými účastníkmi. Vzostupne ich usporiadame a spočítame, či máme dosť jedla pre k najmenej pažravých. Hľadaným maximom je najväčšie k, pre ktoré bolo dosť jedla.

Vyskúšame $O(N)$ rôznych hodnôt k; množstvo jedla pre každého účastníka si spočítame v $O(N)$, takisto v $O(N)$ sčítame jedlo skonzumované k najmenej pažravými účastníkmi. Teda ak účastníkov usporiadame v čase $O(N \log N)$, bude celková časová zložitosť $O(N^2 \log N)$. Tieto riešenia dostali 14 bodov, pomalšie v $O(N^3)$ 12 bodov.

Ak nám nestačí tento algoritmus, môžeme sa zamyslieť a všimnúť si pár možností, ako ho zlepšiť. Keďže sa určite môže najesť ľubovoľný počet účastníkov menší ako hľadané maximum a zároveň sa väčší počet najesť nemôže, použijeme na nájdenie maxima binárne vyhľadávanie.

Binárne vyhľadávanie vo všeobecnosti funguje tak, že si vezmeme počiatočný interval a pozrieme sa na jeho stredný prvok. Ak je hľadaný výsledok v ľavej (pravej) polovici, vezmeme si ľavú (pravú) polovicu a opakujeme postup, kým nemáme interval obsahujúci len jeden prvok.

V tejto úlohe začneme s intervalom $1 \ldots N$ a vždy skontrolujeme, či sa dokáže najesť taký počet účastníkov, aký je stredný prvok intervalu. Ak áno, potom je hľadaný maximálny počet v pravej polovici; v opačnom prípade pokračujeme v hľadaní v ľavej polovici.

Pri binárnom vyhľadávaní je veľmi užitočné používať polootvorené intervaly. Teda hľadať výsledok na intervale $\langle$začiatok; koniec) – udržiavať si interval tak, aby prvky od začiatku (vrátane neho) po koniec hľadanú vlastnosť mohli mať, ale koniec už nie. V každom kroku si zúžime interval, kde sa môže nachádzať výsledok, na polovicu, čiže urobíme $O(\log N)$ krokov. V každom z nich vykonáme $O(N \log N)$ operácií, teda dostaneme výslednú časovú zložitosť $O(N \log^2 N)$. Tieto riešenia získali 17 bodov.

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

Všimnime si, že keď overujeme dostatok jedla pre k účastníkov, potrebujeme z nich len nájsť k najmenej pažravých, nepotrebujeme vedieť ich presné vzájomné poradie. Chceli by sme preto vymyslieť algoritmus, ktorý nám v čase lepšom ako $\Theta(N \log N)$ preusporiada pole tak, aby prvých k prvkov bolo najmenších k prvkov (nezaručí nám ale nič o ich poradí). Knižnica STL takýto algoritmus obsahuje, nazýva sa `nth_element`. My si popíšeme riešenie využívajúce Quick select, čo je vlastne upravený Quick sort.

Quick sort si vyberie niektorý prvok (voláme ho pivot) a jedným prechodom v $O(N)$ preusporiada celé pole tak, aby na jeho začiatku boli prvky menšie ako pivot, potom nasledovali prvky rovné pivotu a nakoniec tie väčšie ako pivot. Potom rekurzívne utriedi prvú a tretiu časť. Ak vyberie vhodný pivot (najlepšie medián – stredný prvok usporiadaného poľa), bude deliť pole približne na polovice a teda rekurzia sa bude vetviť len do hĺbky $O(\log N)$.

Pri jednom takomto preusporiadaní vieme podľa veľkostí častí presne povedať, v ktorej z nich sa bude nami hľadaný k-ty najmenší prvok nachádzať. Ak to bude tá stredná, vyhrali sme a pivot je to, čo hľadáme; inak sa rekurzívne zavoláme len na tú časť, kde sa nachádza hľadaný prvok.

Na rozdiel od Quick sortu v každej úrovni rekurzie pracujeme s čoraz menším úsekom poľa a celková časová zložitosť je v priemernom prípade $O(N)$. Existuje dokonca algoritmus, ktorý má zaručenú časovú zložitosť $O(N)$ aj v najhoršom prípade, nazýva sa Median of five. Je síce lineárny, ale s veľkou konštantou, v praxi je preto lepšie používať Quick select.

S použitím binárneho vyhľadávania a hľadania k-teho najmenšieho prvku sa nám nakoniec podarilo dosiahnuť časovú zložitosť $O(N \log N)$ vďaka $O(\log N)$ rozhodovaniam v čase $O(N)$. Pamäťová zložitosť všetkých algoritmov nemala dôvod prekročiť $O(N)$, potrebovali sme len konštantný počet polí dĺžky N a niekoľko premenných. Riešenia s touto zložitosťou (aj keď ju dosahovali len v priemernom prípade) dostali 20 bodov.

Listing programu:

```
#include <iostream>
#include <algorithm>
#include <cstdlib>
#include <ctime>
using namespace std;

#define MAX_N 1047

void quickSelect(int *A, int zac, int n, int kon) {
    if (kon - zac <= 1)
        return;

    int pivot = A[zac + rand() % (kon - zac)];
    int i = zac, j = kon - 1;

    while (i < j) {
        while (A[i] < pivot && i < j)
            ++i;
        while (A[j] >= pivot && j > i)
            --j;
        if (i < j)
            swap(A[i], A[j]);
    }

    if (n < i)
        quickSelect(A, zac, n, i);
    else {
        while (A[i] == pivot && i < kon)
            ++i;
```


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

```
        quickSelect(A, i, n, kon);
    }
}

int main() {
    srand(time(NULL));
    int A[MAX_N][2], B[MAX_N], N, M;
    int min = 987654321;
    cin >> N >> M;
    for (int i = 0; i < N; ++i) {
        cin >> A[i][0] >> A[i][1];
        if (min > A[i][0])
            min = A[i][0];
    }

    if (min > M) {
        cout << 0 << endl; // ak sa nenaje ani ten najmenej pažravý sám,
        return 0;          // tak potom nikto
    }

    int zac = 0, kon = N, str, sum;
    while (kon - zac != 1) {
        str = (zac + kon) / 2;
        for (int i = 0; i < N; ++i)
            B[i] = A[i][0] + str * A[i][1];

        quickSelect(B, 0, str, N);
        //nth_element(B, B + str, B + N); // STL

        sum = 0;
        for (int i = 0; i <= str; ++i)
            sum += B[i];

        if (sum <= M)
            zac = str;
        else
            kon = str;
    }

    cout << zac + 1 << endl;
}
```

opravoval Usamec
(max. 20 bodov)

7. Oktávy

Pred čítaním tohoto vzoráku je odporúčané si zohnať niečo pod zub, lebo kým ho dočítate, budete určite hladní.

Načítanie vstupu

Tento príklad bol mierne otravný tým, že vstup nebol v úplne ideálnej forme. To, čo by sme chceli, je miesto označenia tónov mať čísla podľa výšky tónu (`C0` dostane číslo 0, `Cis0` číslo 1, `C1` dostane 12, ...). Otázka znie, ako toto prečíslovanie spraviť bez toho, aby sme sa pritom zbláznili a nezapísali zbytočne veľa kódu. Prvý elegantný spôsob je nahádzať si všetky tóny do `unordered_map`[2], ktorá nám poskytne mapovanie zo stringov do čísel v dobrom čase

[2] v C++ treba includnúť `tr1/unordered_map`

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

(je implementovaná ako hash-mapa, kde operácie `insert` a `search` trvajú čas $O(1)$). Potom sa to už parsuje ľahko. Iný stručný kód parsovania je tu:

```
int noty[] = {9, 0, 0, 2, 4, 5, 7, 11};
int parse(char *s) {
  return noty[s[0] - 'A'] + (strlen(s) > 2) + 12 * (s[strlen(s) - 1] - '0');
}
```

Klasické delostrelectvo

Našou úlohou je nájsť nejaký pattern v texte s tým, že je dovolené ho ešte nejak upraviť. Na chvíľu zabudnime na úpravy a predstavme si dva klasické spôsoby, ako riešiť túto úlohu dostatočne rýchlo.

Prvou metódou bude hashovanie. Predstavme si, že každému kusu stringu S s dĺžkou l priradíme číslo $h(S)$ (hash) nasledovne:

$$h(S) = (S[l] + x \cdot S[l-1] + x^2 \cdot S[l-2] + \ldots + x^{l-1} \cdot S[1]) \bmod p$$

Kde x, p sú nejaké náhodne vybrané prvočísla (kto si chce zjednodušovať život a pokladať $p = 2^{32}$, bude zmlátený Chuckom Norrisom, takže to nerobte). Spočítame si hash patternu $h(P)$. A teraz ho chceme hľadať v texte T. Spočítame si hash prvých l znakov z T. Ak je iná ako $h(P)$, tak tu pattern isto nie je. Ak je rovnaká, tak tu asi máme pattern, ale pre istotu to ešte overíme. Teraz sa chceme v texte posunúť o jeden znak a vyrátať novú hash. Všeobecne chceme z hashe pre $T[i]T[i+1]\ldots T[i+l-1]$ vyrátať hash pre $T[i+1]T[i+2]\ldots T[i+l]$. Spravíme to nasledovne: Odpočítame $x^{l-1} \cdot T[i]$, vynásobíme to x a pripočítame $T[i+l]$ (všetko modulo p). Čitateľ môže ľahko overiť, že daná konštrukcia je skutočne funkčná a trvá konštantný čas (hodnotu x^{l-1} si samozrejme vypočítame raz a zapamätáme). Takto vieme overiť každý výskyt patternu. Pokiaľ nemáme príliš veľkú smolu, tak hashe sa budú rovnať len pre rovnajúce sa stringy a celkový čas hľadania bude lineárny od dĺžky patternu a textu.

Táto metóda síce vyžaduje generátor náhodných čísel a dostane menej bodov, ale zase programuje sa trochu ľahšie, je menej myšlienkovo náročná a umožňuje niekoľko zaujímavých trikov. Napríklad skúste si spočítať z hashe $h(P)$ hashe stringov takých, kde je každý znak posunutý o hodnotu $1, 2, 3, \ldots$. Alebo keď máme hash stringov A, B, tak vieme ľahko spočítať hash ich zreťazenia. A naopak ak máme hash stringu A a zreťazenia stringov A a B tak ľahko spočítame hash pre string B (toto si vyskúšajte za domácu úlohu).

Teraz si ukážeme inú metódu, ktorej sa v odborných kruhoch hovorí Knuth-Morris-Prattov algoritmus (skrátene KMP). Označme si prefixy patternu číslami 0, 1, 2, ..., l, ktoré vyjadrujú, koľko znakov zo začiatku patternu sme zobrali (takže pre pattern *abababc* je prefix 3 *aba*). Pre každú pozíciu v texte si teraz spočítame všetky možné prefixy, ktoré môžu na danej pozícii končiť. Napríklad v texte *qababab*d môžu na pozícii 7 končiť prefixy 6, 4, 2, 0. Ak niekde končí prefix l, tak sme našli pattern. Toto má samo o sebe príšernú časovú zložitosť, ale teraz vykonáme niekoľko vylepšení.

Prvé užitočné pozorovanie: Ak na pozícii i končí prefix p (iný ako 0), tak na pozícii $i-1$ musí končiť prefix $p-1$.

Druhé užitočne pozorovanie: Ak na pozícii i končí prefix p, tak všetky kratšie prefixy ako p, ktoré tam budú končiť, sú jednoznačne určené. Inými slovami, ak niekde končili prefixy 4, 2, 0 a o inej pozícii vieme, že tam končí 4, tak jasne vieme, že tam končí aj 2 a 0.

Vďaka druhému pozorovaniu si nám stačí pamätať pre každú pozíciu najdlhší prefix aký tam končí. A zároveň potrebujeme tabuľku, ktorá nám hovorí „ak tu končil prefix p, tak najdlhší kratší prefix, čo tu končí je q“, toto si označíme ako $B[p] = q$. Špeciálne $B[0] = 0$. Vďaka tejto tabuľke jasne vieme zrekonštruovať všetky prefixy, čo niekde končia.

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

Teraz budeme postupne zisťovať prefixy pre každú pozíciu v texte. Ak poznáme najdlhší prefix pre pozíciu $i-1$ (označme ho p), tak najdlhší prefix pre pozíciu i získame nasledovne:

```
kým p != 0 a P[p+1] != T[i]:
  p = B[p]
ak P[p+1] = T[i]:
  p++
```

Po tomto kuse kódu máme v hodnote p najdlhší prefix pre pozíciu i. Vlastne len postupne prezeráme všetky prefixy čo končia v $i-1$ a skúšame nájsť zhodu v znakoch. Ešte špeciálny je prefix 0, ten sa môže vyskytnúť vždy.

Teraz ešte ukážeme, že tento prechod textu má lineárnu zložitosť. Zjavne uvedený kus kódu sa zopakuje M krát (M je dĺžka textu). Operácia `p++` sa zopakuje tiež M krát. Operácia `p = B[p]` zníži hodnotu p aspoň o 1 a teda sa nemôže zopakovať viac ako M krát. A teda máme lineárnu zložitosť od dĺžky textu.

Teraz ešte treba ukázať ako vygenerovať hodnoty B. Presne tak isto ako sme počítali dĺžku najdlhšieho končiaceho prefixu v texte! Hodnota $B[p]$ vlastne znamená aký najdlhší prefix kratší ako p končí v patterne na pozícii p. A z $B[p-1]$ ju vygenerujeme presne rovnako ako pri texte. A ešte natvrdo nastavíme prvé 2 hodnoty $B[0] = B[1] = 0$. Podobne ako pri prechode textom aj tu sa dá ukázať, že časová zložitosť je lineárna od dĺžky patternu.

Pôvodná úloha

Najprv si ukážeme využitie hashovania. Spočítame si hash Halucinkinej melódie. A spočítame si hash rovnako dlhej melódie zo samých jednotiek (označíme $h(J)$). Prechádzanie textom spravíme skoro rovnaké ako pri klasickom hľadaní v texte, až na malú zmenu. Nebudeme kontrolovať presnú rovnosť hashí, ale pozrieme sa na ich rozdiel. Ak tento rozdiel je rozumný (od -36 do 36) násobok $h(J)$, tak máme kandidáta na dobré miesto. Rozumné násobky si uložíme do `unordered_mapy`, aby sme mali konštantný čas hľadania. Celkový čas je $O(N+M+T)$, kde N, M sú dĺžky melódií na vstupe.

A teraz si ukážeme trochu serióznejšie riešenie. Všimnime si, že nech melódiu posúvame akokoľvek, tak rozdiely medzi tónmi zostanú rovnaké. Preto z oboch melódií si spravíme postupnosti také, že ich prvky budú rozdiely dvoch po sebe idúcich tónov. A potom môžeme jednu postupnosť vyhľadávať v druhej. A to, čo nájdeme bude určite správna melódia. Celkový čas máme lineárny na prípravu a lineárny na hľadanie pomocou hashovania, či KMP algoritmu.

Listing programu:

```
#include <string>
#include <vector>
#include <cstdio>
#include <cstring>
using namespace std;

int noty[] = {9, 0, 0, 2, 4, 5, 7, 11};

int parse(char *s) {
    return noty[s[0] - 'A'] + (strlen(s) > 2) + 12 * (s[strlen(s) - 1] - '0');
}

int main() {
    vector<int> haluc, hermi;
    int n, last, cur;
    char buf[10];
    scanf("%d", &n);
    last = -47;
    for (int i = 0; i < n; i++) {
        scanf("%s", buf);
```


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

```
        cur = parse(buf);
        if (last >= 0)
            haluc.push_back(cur - last);
        last = cur;
    }
    scanf("%d", &n);
    last = -47;
    for (int i = 0; i < n; i++) {
        scanf("%s", buf);
        cur = parse(buf);
        if (last >= 0)
            hermi.push_back(cur - last);
        last = cur;
    }

    // Vyrobime si prefixovu tabulku
    vector<int> kmp;
    kmp.resize(hermi.size() + 1);
    kmp[0] = 0; kmp[1] = 0;
    for (int i = 2; i <= hermi.size(); i++) {
        int x = kmp[i - 1];
        // Indexujeme od 0, takze sme oproti vzoraku posunuti
        while (x != 0 && hermi[x] != hermi[i - 1])
            x = kmp[x];
        if (hermi[x] == hermi[i - 1])
            x++;
        kmp[i] = x;
    }

    // Pouzijeme ju
    int p = 0;
    for (int i = 0; i < haluc.size(); i++) {
        while (p != 0 && hermi[p] != haluc[i])
            p = kmp[p];
        if (hermi[p] == haluc[i])
            p++;
        if (p == hermi.size()) {
            printf("ANO\n");
            return 0;
        }
    }
    printf("NIE\n");
}
```

8. Obedy sa vydávajú do siedmej

vzorák písal Bob
(max. 25 bodov)

V tejto úlohe máme vlastne vypočítať dĺžku najdlhšej spoločnej podpostupnosti[3] zoznamu jedál a študentov v rade. Len pre upresnenie: na rozdiel od podreťazca, podpostupnosť nemusí byť súvislá.

Najdlhšia spoločná podpostupnosť

Riešenie založíme na dynamickom programovaní. Študentov a jedlá si označíme (podobne ako v zadaní) postupnosťami $A = (a_1, a_2, \ldots, a_m)$ a $B = (b_1, b_2, \ldots, b_n)$. Do dvojrozmerného

[3] http://en.wikipedia.org/wiki/Longest_common_subsequence

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

poľa D veľkosti $(m+1) \times (n+1)$ si na pozíciu $D[i][j]$ uložíme dĺžku najdlhšej spoločnej podpostupnosti prvých i študentov a prvých j jedál, teda $(a_1, \ldots, a_i)$ a $(b_1, \ldots, b_j)$.

Začnime pole D vypĺňať. Hodnoty $D[i][0]$ a $D[0][j]$ sú zjavne rovné 0 pre ľubovoľné i, j – spoločná podpostupnosť s prázdnou postupnosťou musí byť prázdna. Hodnotu $D[i][j]$ pre $i, j \geq 1$ nájdeme nasledovne: Ak sa prvky a_i a b_j rovnajú, potom ich môžeme pokojne vyhlásiť za posledný prvok najdlhšej spoločnej podpostupnosti a ďalej sa zaoberať len postupnosťami $(a_1, \ldots, a_{i-1})$ a $(b_1, \ldots, b_{j-1})$, teda $D[i][j] = 1 + D[i-1][j-1]$. V opačnom prípade nemôžu byť v najdlhšej spoločnej podpostupnosti oba prvky a_i a b_j naraz. Preto skúsime obe možnosti (ktorý z prvkov vyhodíme) a vyberieme z nich tú lepšiu, teda $D[i][j] = \max(D[i-1][j], D[i][j-1])$. (Prečo nemusíme brať do úvahy tretiu možnosť? Čo ak treba vyhodiť aj a_i aj b_j?)

Pri výpočte $D[i][j]$ už potrebujeme poznať hodnoty v D pre menšie i, j. Stačí preto vypĺňať pole D postupne po riadkoch alebo stĺpcoch. Nakoniec vypíšeme ako výsledok hodnotu $D[m][n]$. Skúste si premyslieť, ako by sa dalo vyplnené pole D využiť, ak by sme potrebovali vypísať aj celú najdlhšiu spoločnú podpostupnosť.

Časová aj pamäťová zložitosť tohto algoritmu je $\Theta(mn)$. Pri vhodnom poradí vypĺňania sa dá pamäť ušetriť – ak vypĺňame D napríklad po riadkoch, stačí si pamätať iba ten aktuálny a predchádzajúci. Takto dosiahneme pamäťovú zložitosť $\Theta(m+n)$.

Takéto riešenie je celkom fajn, dokonca aj nejaké body získalo. Z limitov pre m a n v zadaní však vidno, že na vyriešenie najväčšieho vstupu budeme potrebovať niečo rýchlejšie. A asi by sa zišlo využiť informáciu, že žiadny študent nestojí v rade na obed viackrát.

Prečíslovanie študentov

Pre našu úlohu je úplne nepodstatné, aké konkrétne čísla majú študenti priradené. Navyše môžeme rovno zahodiť jedlá, ktoré nikto zo študentov v rade nechce. Výsledok pre vstup $(47, 42, 1\,000\,000\,000)$ a $(1, 42, 13, 42, 47)$ je predsa ten istý, ako pre vstup $(1, 2, 3)$ a $(2, 2, 1)$.

Preto hneď po načítaní vstupu zmeníme študentom čísla tak, že študentovi a_i priradíme číslo i. Súčasne sa táto zmena musí prejaviť aj v postupnosti B. Vyrobíme z nej novú postupnosť $C = (c_1, \ldots, c_k)$: Ak je jedlo b_j pripravené pre nejakého študenta a_i v rade, potom do C vložíme prvok i (nové číslo študenta a_i); inak môžeme jedlo b_j zahodiť. Ako ale efektívne nájsť takú pozíciu i, že $a_i = b_j$? Pôvodnú postupnosť A utriedime (ku každému prvku si navyše zapamätáme jeho pôvodné umiestnenie) a potom môžeme b_j binárne vyhľadať.

Niektoré programovacie jazyky nám túto prácu ešte zjednodušia, lebo poskytujú tzv. asociatívne pole. Takéto pole sa dá indexovať napríklad aj reťazcami, desatinnými aj celými číslami z miliardového rozsahu, pričom v pamäti zaberá rádovo rovnako veľa priestoru ako obyčajné pole. Príklad použitia asociatívneho poľa v C++ (`map`) nájdete v kóde na konci vzoráku.

Časová zložitosť prečíslovania študentov je pri použití triedenia a binárneho vyhľadávania aj pri použití `map`-y z C++ rovnaká: $\Theta((m+n)\log m)$. Ak by sme mali asociatívne pole implementované hashovaním, dosiahli by sme očakávanú časovú zložitosť $\Theta(m+n)$.

Takže teraz hľadáme najdlhšiu spoločnú podpostupnosť postupností $(1, 2, \ldots, m)$ a C. Všimnime si, že každá ich spoločná podpostupnosť je rastúca a naopak, každá rastúca podpostupnosť C je spoločná s $(1, 2, \ldots, m)$. Stačí nám teda zistiť dĺžku najdlhšej rastúcej podpostupnosti[4] C.

Najdlhšia rastúca podpostupnosť

Opäť existuje jednoduchý algoritmus riešiaci túto úlohu založený na dynamickom programovaní. Tentoraz nám postačí jednorozmerné pole D veľkosti k. Postupne pre $i = 1, \ldots, k$

[4] `http://en.wikipedia.org/wiki/Longest_increasing_subsequence`

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

si na pozíciu $D[i]$ uložíme dĺžku najdlhšej rastúcej podpostupnosti C, ktorá **končí na pozícii** i. Aby sme túto hodnotu vypočítali, vyskúšame všetky možnosti pre predposledný prvok hľadanej podpostupnosti. Takže ak pre všetky $j < i$ platí $c_j \geq c_i$, potom $D[i] = 1$, inak $D[i] = 1 + \max_{j<i \,\wedge\, c_j<c_i} D[j]$. Výsledkom je najväčšia hodnota poľa D.

Priamočiarym prepísaním tejto myšlienky do kódu síce získame krátke riešenie, lenže beží v čase $\Theta(k^2)$ (a navyše potrebujeme prečíslovať študentov), takže sme si oproti tomu predchádzajúcemu veľmi nepomohli. Našťastie máme ešte kopu priestoru na zlepšovanie.

Rýchlejšie hľadanie maxima: pohľad cez hodnoty c_i

Pozrime sa bližšie, z ktorých hodnôt vyberáme maximum pri výpočte $D[i]$. Podmienku $j < i$ splníme jednoducho, stačí v čase výpočtu $D[i]$ uvažovať len prvky $(c_1, \ldots, c_{i-1})$, až potom k nim pridať c_i. S druhou podmienkou je to trochu horšie: tie c_j, ktoré sú menšie ako c_i, sa môžu nachádzať kdekoľvek medzi prvkami $(c_1, \ldots, c_{i-1})$.

Použijeme preto pomocné pole F s veľkosťou k inicializované nulami, do ktorého si budeme ukladať dĺžku zatiaľ nájdenej najdlhšej rastúcej podpostupnosti **končiacej v danej hodnote**. Všimnite si rozdiel oproti poľu D: F indexujeme hodnotami c_i, nie ich umiestnením i. Pri výpočte $D[i]$ budeme brať maximum z $F[1], \ldots, F[c_i - 1]$; potom upravíme hodnotu v $F[c_i]$ na $D[i]$ (premyslite si, že stará hodnota v $F[c_i]$ nemôže byť väčšia ako $D[i]$).

Hmm, potrebujeme efektívne zisťovať maximum z intervalu a tiež meniť hodnoty jednotlivých prvkov F – ako vhodné riešenie sa núka intervalový strom. V tomto prípade nám však postačí aj trochu slabšie kladivko, fínsky (Fenwickov) strom[5]. Ten má síce vo svojej maximovej verzii zopár obmedzení (pozná iba maximum prefixov a menený prvok nesmieme zmenšiť), ale pri tomto probléme nám to neprekáža.

Dostali sme teda riešenie, ktoré by malo získať plný počet bodov – jeho časová zložitosť (bez prečíslovania) je $\Theta(k \log k)$, keďže intervalovému aj fínskemu stromu trvá každá operácia $\Theta(\log k)$. Podrobnosti o tom, ako tieto stromy fungujú, tu nenájdete; bolo by to na dlhšie... a existuje riešenie, ktoré ich nevyužíva.

Rýchlejšie hľadanie maxima: pohľad cez hodnoty $D[i]$

Vráťme sa k pomalému spôsobu hľadania najdlhšej rastúcej podpostupnosti ($\Theta(k^2)$). Tentoraz budeme potrebovať pole E (za tými názvami sa naozaj žiadny hlbší zmysel neskrýva). Na pozícii $E[\ell]$ budeme mať zapísanú najmenšiu hodnotu, na ktorú končí nejaká zatiaľ nájdená rastúca podpostupnosť dĺžky ℓ (alebo ∞, ak sme takú dlhú rastúcu podpostupnosť ešte nenašli).

Prečo nám stačí uvažovať zo všetkých zatiaľ nájdených rastúcich podpostupností dĺžky ℓ práve tú s najmenšou hodnotou na konci? Tá je totiž „najlepšia“. Nech sme zatiaľ spracovali len prvky $(c_1, \ldots, c_i)$; zoberme si dve rastúce podpostupnosti P a Q rovnakej dĺžky, pričom nech P končí na p, Q na q a $p < q$. Potom vždy, keď budeme chcieť predĺžiť postupnosť Q nejakým prvkom c_j pre $j > i$, môžeme namiesto nej predĺžiť postupnosť P ($p < q < c_j$).

Ďalej si uvedomme, že prvky v poli E budú vždy v rastúcom poradí (až na tie ∞ na konci). Ak by totiž $E[\ell] \geq E[\ell + 1]$, potom by existovala rastúca podpostupnosť dĺžky ℓ končiaca na číslo menšie ako $E[\ell + 1]$ (zoberieme „najlepšiu“ podpostupnosť dĺžky $\ell + 1$ a vyhodíme jej posledný prvok) a teda aj menšie ako $E[\ell]$.

Pri výpočte hodnoty $D[i]$ binárne vyhľadáme v poli E najväčšie také ℓ, že $E[\ell] < c_i$, potom $D[i] = 1 + \ell$. Navyše musíme upraviť hodnotu $E[\ell + 1]$ na c_i, keďže sme našli rastúcu podpostupnosť dĺžky $\ell + 1$ s menším (presnejšie: nie väčším) prvkom na konci, ako mala doteraz „najlepšia“ nájdená podpostupnosť. (Prečo nemôže byť $E[\ell+1] < c_i$? Zachovali sme pôvodný význam poľa E?)

[5] `http://www.ksp.sk/ksp2.0/wiki/Kucharka/FinskyStrom`

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

Takže znova riešenie v čase $\Theta(k \log k)$ (k binárnych vyhľadávaní). Hoci vyzerá zložitejšie ako to predchádzajúce, kódi sa celkom jednoducho. Navyše nevyžaduje, aby boli hodnoty C z malého rozsahu (hoci v našom prípade kvôli prečíslovaniu študentov aj tak sú) – polia totiž indexujeme len číslami do k, nezávisle od hodnôt C.

Na záver som si nechal šikovnú implementáciu podľa riešenia Mareka Špana. Načo zbytočne písať binárne vyhľadávanie, keď máme v C++ k dispozícii `set`: prvky poľa E sú aj tak usporiadané, ∞ si nemusíme pamätať a `insert` vloží c_i presne tam, kam patrí.

Listing programu:

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

int main() {
    int m, n;
    cin >> m >> n;
    map<int, int> studenti;
    for (int i = 0; i < m; ++i) {
        int x;
        cin >> x;
        studenti[x] = i;     // študentovi a_i priradíme číslo i
    }

    set<int> najlepsie;
    for (int i = 0; i < n; ++i) {
        int x;
        cin >> x;
        if (studenti.count(x) == 0)
            continue;     // také jedlo nikto nechce

        set<int>::iterator it = najlepsie.lower_bound(studenti[x]);
        if (it != najlepsie.end())
            najlepsie.erase(it);     // vyhodíme starý najlepší koniec
        najlepsie.insert(studenti[x]);
    }

    cout << najlepsie.size() << endl;
}
```

opravoval Usamec, vzorák písal Bob

1::9. Tvoriví mládenci

(max. 25 bodov)

Hoci táto úloha nebola oveľa ťažšia ako priemerná sedmička či osmička, našli sa iba dvaja odvážni riešitelia. Vy ostatní: nebojte sa a skúste aj Téčko. Vďaka zmene v pravidlách môžete svoje riešenia po konzultácii s opravovateľom postupne vylepšovať.

Takže úlohou bolo nájsť čo najkratší zápis zadaného reťazca, pričom môžeme k výskytov reťazca X zakódovať ako `k[X]`. Keďže nevidno nijaký priamy spôsob, ako vygenerovať hľadaný zápis, obrátime sa na dynamické programovanie.

Označme zadaný reťazec $Z = z_1 z_2 \ldots z_n$. Použijeme pomocné dvojrozmerné pole D; na pozíciu $D[d][i]$ si uložíme dĺžku najkratšieho zápisu podreťazca Z začínajúceho i-tym znakom a dlhého d znakov, teda $z_i z_{i+1} \ldots z_{i+d-1}$. Tieto hodnoty budeme postupne počítať začínajúc od kratších podreťazcov.

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

Chceme teda zistiť hodnotu $D[d][i]$ a už máme vyrátané hodnoty D pre menšie dĺžky. Podreťazec $z_i z_{i+1} \ldots z_{i+d-1}$ sa dá v princípe zapísať troma spôsobmi:

- Zapíšeme ho nezmenený, v plnom znení. Dĺžka takého zápisu je d.
- Najprv zapíšeme najkratším spôsobom prvých j znakov (vyskúšame všetky j od 1 do $d-1$) a potom zvyšných $d-j$ znakov. Dĺžka bude $D[j][i] + D[d-j][i+j]$.
- Ak náš podreťazec má periódu dĺžky j (vyskúšame všetky j od 1 do $d/2$), potom ho zapíšeme ako opakovanie najkratšieho zápisu jeho periódy. Dĺžka tohto zápisu je $C[d/j] + 2 + D[j][i]$, kde $C[k]$ je počet cifier čísla k zapísaného v desiatkovej sústave.

Zo všetkých spôsobov zápisu si samozrejme vyberieme ten najkratší. Hodnoty $C[k]$ si predpočítame klasickým spôsobom (postupne delíme k desiatimi, kým nedosiahneme 0). Zostáva nám už len vedieť efektívne hľadať periódy v reťazci. Intuitívne sa zdá, že by nám stačilo nájsť len tú najkratšiu periódu (všetky ostatné sú jej násobkami), ale to by sme potrebovali dokázať a aj tak by to riešenie asymptoticky nezlepšilo.

Všeobecnejšiu úlohu „daný je reťazec S, nájdite všetky jeho periódy“ riešime tak, že hľadáme výskyty S v reťazci SS (S napísaný dvakrát za sebou). Skúste si rozmyslieť, čo nám takýto nájdený výskyt povie o perióde S. Pre dosiahnutie optimálnej časovej zložitosti môžeme na vyhľadávanie vzorky v texte použiť hashovanie alebo algoritmus KMP. V našom prípade sa však zaobídeme aj bez bez týchto hrôz.

Do poľa P si na pozíciu $P[d][i]$ predpočítame najväčšie také j, že podreťazec $z_i z_{i+1} \ldots z_{i+j-1}$ sa dá napísať ako niekoľko kópií $z_i z_{i+1} \ldots z_{i+d-1}$ za sebou, pričom tá posledná kópia nemusí byť úplná. Napríklad pre `abcabcabbb` je $P[3][1] = 8$. Tieto hodnoty vypočítame priamo (žiadna dynamika): postupne skúšame čoraz väčšie j a skončíme, keď narazíme na prvú nezhodu.

Teraz už vieme ľahko overiť, či má podreťazec $z_i z_{i+1} \ldots z_{i+d-1}$ periódu dĺžky j: j musí deliť d a $P[j][i]$ musí byť aspoň d.

Nakoniec nám ostal malý detail: úlohou bolo vypísať najkratší zápis daného reťazca, nie jeho dĺžku (ktorú už máme v $D[n][1]$). Mohli by sme si počas výpočtu jednotlivých hodnôt D zapamätať aj to, ktorým spôsobom vznikol ten najkratší zápis a potom ho pomocou týchto údajov rekurzívne vypísať. V mojej implementácii som ale využil iný prístup: znova vyskúšam všetky spôsoby a vyberiem z nich ten najkratší.

Výpočet hodnôt C trvá $\Theta(n \log n)$; hodnoty P aj D spočítame v čase $\Theta(n^3)$; výpis trvá určite $O(n^3)$. Spolu je teda časová zložitosť $\Theta(n^3)$, pamäťová zložitosť je $\Theta(n^2)$.

Listing programu:

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

string Z;
int n;
vector<int> C;
vector<vector<int> > P, D;

void vypis(int d, int i) {  // viac-menej okopírovaný kód, ktorý rátal D
    if (D[d][i] == d) {
        cout << Z.substr(i, d);
        return;
    }

    for (int j = 1; j <= d - 1; ++j)
        if (D[d][i] == D[j][i] + D[d - j][i + j]) {
```


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

```
            vypis(j, i);
            vypis(d - j, i + j);
            return;
        }

    for (int j = 1; j <= d / 2; ++j)
        if (d % j == 0 && P[j][i] >= d && D[d][i] == D[j][i] + C[d / j] + 2) {
            cout << d / j << '[';
            vypis(j, i);
            cout << ']';
            return;
        }
}

int main() {
    cin >> Z;
    n = Z.size();

    C.resize(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        int x = i;
        while (x > 0) {
            x /= 10;
            ++C[i];
        }
    }

    P.resize(n + 1, vector<int>(n));
    for (int d = 1; d <= n; ++d)
        for (int i = 0; i <= n - d; ++i) {
            P[d][i] = d;

            for (int j = d; i + j < n; ++j, ++P[d][i])
                if (Z[i + j] != Z[i + j % d])
                    break;
        }

    D.resize(n + 1, vector<int>(n));
    for (int d = 1; d <= n; ++d)
        for (int i = 0; i <= n - d; ++i) {
            D[d][i] = d;  // nezmenený

            for (int j = 1; j <= d - 1; ++j)  // zložený z dvoch
                D[d][i] = min(D[d][i], D[j][i] + D[d - j][i + j]);

            for (int j = 1; j <= d / 2; ++j)  // opakovanie periódy
                if (d % j == 0 && P[j][i] >= d)
                    D[d][i] = min(D[d][i], D[j][i] + C[d / j] + 2);
        }

    vypis(n, 0);
    cout << endl;
}
```


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

opravoval Zemčo
(max. 25 bodov)

1::10. Teória výberu suseda

Toto veru nebola jednoduchá úloha. Jej obtiažnosť spočívala najprv v myšlienkovom odhalení geometrickej podstaty (my v KSP dobre vieme, že ak chceme, aby ste riešili geometrickú úlohu, musíme ju dobre zamaskovať) a neskôr v na pohľad náročnej implementácii. Prišli dve riešenia s neoptimálnym časom a bez programu, ktoré dostali 12 bodov.

Kde sa nám v tejto úlohe zobrala geometria? Na túto otázku odpovieme trochu neskôr. Najprv si poďme napísať úlohu matematicky. Na vstupe máme tri špeciálne čísla - U$ámove množstvá potravín, tie označíme u_1, u_2, u_3. Potom máme $N-1$ ostatných hodnôt potravín, i-tu trojicu z nich označíme $a_{i,1}, a_{i,2}, a_{i,3}$. Ak majú nejaké hodnoty x, y, z spĺňať požiadavku zadania, potom musí platiť:

$$\begin{aligned} a_{1,1}x + a_{1,2}y + a_{1,3}z &< u_1x + u_2y + u_3z \\ a_{2,1}x + a_{2,2}y + a_{2,3}z &< u_1x + u_2y + u_3z \\ &\dots \\ a_{N-1,1}x + a_{N-1,2}y + a_{N-1,3}z &< u_1x + u_2y + u_3z \end{aligned}$$

Dostávame teda sústavu $N-1$ nerovníc. Po jednoduchej úprave dostaneme:

$$\begin{aligned} (a_{1,1} - u_1)x + (a_{1,2} - u_2)y + (a_{1,3} - u_3)z &< 0 \\ (a_{2,1} - u_1)x + (a_{2,2} - u_2)y + (a_{2,3} - u_3)z &< 0 \\ &\dots \\ (a_{N-1,1} - u_1)x + (a_{N-1,2} - u_2)y + (a_{N-1,3} - u_3)z &< 0 \end{aligned}$$

Všimnime si, že keď máme nerovnicu splnenú trojicou (x, y, z), potom bude splnená aj trojicou (rx, ry, rz) pre každé reálne $r > 0$. Dosadenie týchto hodnôt totiž nie je nič iné ako prenásobenie ľavej strany (respektíve celej nerovnice) konštantou r. Keďže r je kladné, potom sa nám to platnosť nerovnice určite neovplyvní. To ale znamená, že ku každej spĺňajúcej trojici nezáporných čísel (x, y, z) kde $z \neq 0$, existuje spĺňajúca trojica $(x/z, y/z, 1)$. Takže budeme hľadať riešenie, v ktorom je $z = 1$. Samozrejme, musíme overovať aj existenciu riešení, v ktorých $z = 0$. S tým si však vieme poradiť napríklad tak, že skonštruujeme druhú sériu nerovníc predpokladajúcich túto hodnotu a spustíme na nich rovnaký algoritmus. Pokračujme teda v prípade $z = 1$. Ak si teraz označíme hodnoty:

$$\begin{aligned} a_{i,1} - u_1 &= v_{i,1} \\ a_{i,2} - u_2 &= v_{i,2} \\ u_3 - a_{i,3} &= v_{i,3} \end{aligned}$$

potom môžeme sústavu napísať ako:

$$\begin{aligned} v_{1,1}x + v_{1,2}y &< v_{1,3} \\ v_{2,1}x + v_{2,2}y &< v_{2,3} \\ &\dots \\ v_{N-1,1}x + v_{N-1,2}y &< v_{N-1,3} \end{aligned}$$

Mimochodom, toto zbavenie sa hodnoty z má svoju peknú geometrickú reprezentáciu. Všetky prípustné trojice x, y, z tvoria akýsi podpriestor trojrozmerného priestoru. Položením parametra $z = 1$ dostávame prienik tohto podpriestoru s rovinou, čím spomedzi prípustných riešení začneme uvažovať len nejakú menšiu (a ľahšie popísateľnú) množinu reprezentantov. To len na okraj. Kto má záujem, môže si to skúsiť nakresliť a my ostatní pokračujme.

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

Keď už sme tu znova spomenuli tú geometriu, poďme sa jej držať. V mnohých prípadoch je totiž úloha, podobná tejto, pekne reprezentovateľná v dvojrozmernom alebo trojrozmernom priestore. To nám často pomáha, pretože ak si situáciu zredukujeme na známy geometrický problém, potom nám stačí použiť štandardný algoritmus na riešenie.

V našom prípade pracujeme s lineárnymi nerovnicami s dvoma parametrami x a y. To naznačuje, že sa budeme pohybovať v dvojrozmernom priestore. V prípade, že by sme mali rovnice boli by to obyčajné reprezentácie priamok. V prípade nerovníc to budú polroviny. A keďže nerovnosť je ostrá, budú to polroviny, do ktorých ich hranica patriť nebude. Ak hodnoty x, y nejakej konkrétnej nerovnici vyhovujú, znamená to, že bod $[x, y]$ patrí do príslušnej polroviny. Hľadáme teda bod, ktorý patrí do všetkých polrovín. Inými slovami, potrebujeme vypočítať prienik polrovín.

Tento prienik môže byť neohraničený, avšak vždy je súvislý a konvexný. Pripomeňme si, že útvar U (všeobecnejšie množina bodov, avšak útvar nie je nič iné ako množina bodov, ktorá mu patrí) je konvexný, ak pre každé dva body $A, B \in U$ platí, že množina $\{rA+(1-r)B \mid r \in \mathrm{R}, r \in [0,1]\} \subseteq U$. Inými slovami, že ak dva body patria do útvaru, tak do neho patrí celá úsečka spájajúca tieto dva body. To sú všetky body, ktoré sú ich (vhodnou) lineárnou kombináciou. Všimnite si, že nekonečnosť (respektíve neohraničenosť) útvaru nie je v nijakom spore s konvexnosťou podľa uvedenej matematickej definície.

Náš algoritmus bude používať metódu Divide & Conquer a bude mať veľmi jednoduchý priebeh:

Vstup: číslo K, polroviny $p_1, p_2, \ldots, p_K$.
Výstup: ich prienik.

1. ak $K = 1$, vráť útvar ohraničený p_1.
2. ak $K \geq 2$, nech $M = K/2$.
3. podprienik r_1 = spočítaj rekurzívne prienik $p_1, \ldots, p_M$.
4. podprienik r_2 = spočítaj rekurzívne prienik $p_{M+1}, \ldots, p_K$.
5. vráť $r_1 \cap r_2$.

Ľudskými slovami: prienik K polrovín dostaneme, ak zrátame prienik prvej polovice a druhej polovice osobitne a potom spravíme prienik týchto dvoch útvarov. Už nám zostalo len popísať, ako vypočítať prienik dvoch konvexných (potenciálne neohraničených) útvarov, ale o tom potom. Označme čas potrebný na vypočítanie tohto prieniku zatiaľ ako $T(n)$, kde n je súčet počtov vrcholov týchto útvarov. Všimnime si, že ak rátame prienik n polrovín, potom môže mať tento útvar najviac n vrcholov. Toto sa ľahko dokáže indukciou, pridaním novej polroviny do prieniku pribudne najviac jeden vrchol. Preto nám stačí uvažovať K ako jediný vstupný parameter tohto algoritmu. Ak označíme čas jeho behu $A(K)$, dostávame nasledovný rekurzívny vzťah:

$$A(K) = \begin{cases} 2 \cdot A(K/2) + T(K) + O(1), & \text{ak } K \geq 2; \\ O(1), & \text{ak } K < 2. \end{cases}$$

Analýzu časovej zložitosti dokončíme faktom, že prienik dvoch konvexných útvarov pri vhodnej reprezentácii vieme realizovať v lineárnom čase. Všetky implementačné detaily a algoritmus si môžete nájsť v príslušnom odstavci nižšie. Tu pokračujme v *pohľade zhora* a dokončime odhad časovej zložitosti. Lineárny algoritmus znamená, že počet vykonaných operácii sa dá zhora ohraničiť hodnotou cK pre nejakú konštantu c. Nahraďme teda $T(K)$ výrazom cK a rozpíšeme uvedený vzťah:

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

$$\begin{aligned} A(K) &= 2A(K/2) + cK \\ &= 2(2A(K/4) + c(K/2)) + cK = 4A(K/4) + 2c(K/2) + cK \\ &= 4(2A(K/8) + c(K/4)) + 2c(K/2) + cK = 8A(K/8) + 4c(K/4) + 2c(K/2) + cK \\ &\dots \end{aligned}$$

Ľahko vidno, že postupným rozvíjaním nám zakaždým pribudne najviac cK operácii a zároveň sa nám zmenší veľkosť v nerozvinutom člene na polovicu. V rozvoji by sme preto mohli pokračovať $\log K$ krát. Pri tejto analýze si pozorný čitateľ určite všimol, že sme zanedbali člen $O(1)$. Nie je zložité presvedčiť sa, že v konečnom dôsledku je naozaj zanedbateľný. Ak schováme konštantu c do O notácie, dostávame celkový čas behu algoritmu $O(K \log K)$. Pamäťové nároky sú lineárne od počtu polrovín. Na tomto mieste pohľad zhora končí. Snáď uznáte, že základná myšlienka nie je vôbec zložitá. Nasleduje už len popis algoritmu na prienik konvexných útvarov.

Reprezentácia polroviny a útvaru

Poznáma: budeme predpokladať, že čitateľovi nasledovných odstavcov sú pojmy ako *vektor* a *vektorový súčin* známe.

Máme viacero možností, ako si reprezentovať polrovinu. V našej implementácii si priamku, ktorá polrovinu ohraničuje, pamätáme vo forme dvoch bodov patriacich tejto priamke. Stranu roviny, ktorú tvorí naša polrovina, máme uloženú vo forme tretieho bodu, ktorý leží na správnej strane. Keď preto chceme zistiť o nejakom bode, či sa nachádza v danej polrovine, môžeme napríklad porovnať hodnotu vektorového súčinu smeru priamky a vstupného bodu a vektorového súčinu smeru priamky a bodu, ktorý máme uložený na identifikáciu polroviny.

Štandardný pohodlný spôsob reprezentácie geometrických útvarov ako zoznam bodov v poradí, v akom sa objavujú na obvode, tentokrát preferovať nebudeme. Jednou z príčin je, že náš útvar môže byť aj neohraničený a práca s ohraničujúcimi priamkami je teda prirodzenejšia. V ďalšom texte budeme niekedy stotožňovať pojmy ohraničujúca priamka a ohraničujúca polrovina, malo by to byť vždy zrozumiteľné.

Útvar si budeme reprezentovať v podobe dvoch zoznamov ohraničujúcich polrovín - ľavého a pravého okraja. Na každú polrovinu, ktorá nie je vodorovná s osou x, sa totiž môžeme pozrieť ako na časť roviny, ktorá je naľavo, respektíve napravo od svojej deliacej priamky. Roviny, ktoré majú deliaciu priamku rovnobežnú s osou x budeme ošetrovať úplne špeciálne a môžeme predpokladať, že sa nám v týchto zoznamoch neobjavia. Body si priamo nepamätáme, pretože v prípade potreby si ich vždy vieme vypočítať v konštantnom čase (ak nevieš ako, dozvieš sa v odstavci nižšie). V oboch zoznamoch sú polroviny v poradí od najvyššej - to znamená s najvyššou y súradnicou. Zoznam môže byť aj prázdny, vtedy je útvar na príslušnú stranu neohraničený. Iný spôsob, ako môže reprezentovaný útvar byť neohraničený, je, ak sa prvé priamky v ľavom a pravom zozname nepretínajú (prípadne posledné priamky z oboch zoznamov).

Ako vytvoriť nový útvar z jednej polroviny je, aspoň z teoretického hľadiska, jednoduché. Polrovinu dáme do príslušného zoznamu v závislosti od uhla jej priamky a toho, ktorú stranu tejto priamky reprezentuje (a druhý zoznam necháme prázdny). Technické detaily a rozbor prípadov necháme na čitateľa. Môžete nahliadnuť na našu implementáciu, kde sa príslušný kód nachádza v konštruktore útvaru.

Ako sa môžete presvedčiť v ďalšom čítaní, vďaka celočíselnému vstupu máme všetky hodnoty, ktoré sa nám v tomto vzorovom riešení vyskytnú, racionálne. Preto budeme používať racionálnu aritmetiku. Práca s racionálnou aritmetikou nás síce môže miestami spomaliť (hlavne pri redukovaní zlomku do základného tvaru po aritmetických operáciach), avšak zbavuje nás nepresností vyplývajúcich z používania reálnych čísel. Je prudko odporúčané vyhnúť sa reálnym číslam vždy, keď je to len trochu možné.

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

Pri analýze časovej zložitosti sme túto položku zanedbali. Keby sme chceli byť úplní, museli by sme náš odhad počtu operácii ešte vynásobiť časom potrebným na vykonanie jednej operácie s racionálnymi číslami (keďže pri normalizácii zlomkov používame Euklidov algoritmus[6], vyšiel by z toho rádovo logaritmus veľkosti uvažovaného čitateľa a menovateľa, čo je prinajhoršom maximum vstupnej súradnice). Tento detail je pre nás príliš nízkoúrovňový. V konečnom dôsledku by sme totiž, ak už chceme byť takí detailisti, potrebovali túto analýzu spraviť aj pri ľubovoľnej inej reprezentácii čísel. Bežne sa to pri analýze na vyššej úrovni pre jednoduchosť prehliada, hoci to často nie je konštantný čas. Na teoretickej úrovni je totiž zaujímavejšie sledovať počet operácii ako taký, nie počet strojových inštrukcii.

Počítanie prieniku dvoch priamok

V tejto časti si popíšeme jeden zo stavebných kameňov tohto aj mnohých iných geometrických algoritmov: pre zadané dve úsečky alebo priamky vypočítať bod, v ktorom sa stretnú (prípadne prehlásiť, že taký bod neexistuje). Tento odstavec bude viac analytická geometria ako programovanie.

Majme teda dve priamky p_1 a p_2. V našej reprezentácii máme na vstupe dva body priamky p_1, označme ich $A = [a_1, a_2]$ a $B = [b_1, b_2]$, obdobne dva body priamky p_2, ktoré označíme $C = [c_1, c_2]$ a $D = [d_1, d_2]$.

Ošetrime najprv prípad, kedy sú priamky rovnobežné alebo totožné. Je to práve vtedy, keď majú smerové vektory rovnaký smer. Smerové vektory u, v dostaneme jednoducho: $u = (u_1, u_2) = (a_1 - b_1, a_2 - b_2)$, $v = (v_1, v_2) = (c_1 - d_1, c_2 - d_2)$. Smerový vektor totiž nie je nič iné ako posun z jedného bodu priamky do druhého. Ak si vektor predstavíme ako šipku zakreslenú do roviny, tak existuje nekonečne veľa vektorov, ktoré majú rovnaký smer (uhol šipky) a líšia sa vo veľkosti (dĺžke šipky) alebo orientácii (orientáciu šipky na jednu z dvoch možných strán), v našej situácii nám stačí ľubovoľný.

Ak chceme zistiť, či majú tieto vektory rovnaký smer, použijeme vektorový súčin. Vypočítame teda hodnotu $u_1v_2 - u_2v_1$. Ak je táto hodnota nula, priamky su rovnobežné. Ak sú totožné, majú všetky body spoločné, ak sú iba rovnobežné, nemajú žiadny spoločný bod. Aby sme to rozlíšili, vezmeme ľubovoľný bod patriaci prvej priamke (napríklad jeden z dvoch, ktoré máme zapamätaný) a pomocou ďalšieho vektorového súčinu zistíme, či patrí druhej priamke. Ak sme zobrali bod C, tak ak má patriť priamke idúcej cez body A a B, musí byť smer z bodu A do bodu B rovnaký ako smer z bodu A do bodu C.

Ak nenastal prípad rovnosti smerových vektorov, potom existuje práve jeden bod prieniku. Aby sa nám lepšie pokračovalo, uvažujme bežné analytické vyjadrenie priamky ako vzťahu

$$ax + by + c = 0$$

Ak dosadíme bod $[x, y]$ do tejto rovnice a rovnosť bude platiť, znamená to, že bod patrí priamke (v opačnom prípade nie). Ak máme priamku danú dvoma bodmi $[a_1, a_2]$ a $[b_1, b_2]$, potom uvedený tvar dosiahneme položením

$$(b_2 - a_2)x + (a_1 - b_1)y + (a_1b_2 - b_1a_2) = 0$$

Prvé dva koeficienty získame dosadením normálového vektora. Ten je kolmý na smerový, a vieme, že ak máme vektor (a, b), vektor kolmý na neho má hodnotu napríklad $(-b, a)$. Posledný koeficient môžeme dosiahnúť napríklad všeobecným dosadením hodnôt $[a_1, a_2]$ za x a y do rovnice priamky, v ktorej potom ostane už len jedna neznáma c). Ak máme priamky vyjadrené ako

$$p_1x + p_2y + p_3 = 0$$

[6] http://sk.wikipedia.org/wiki/Euklidov_algoritmus

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

$$q_1x + q_2y + q_3 = 0$$

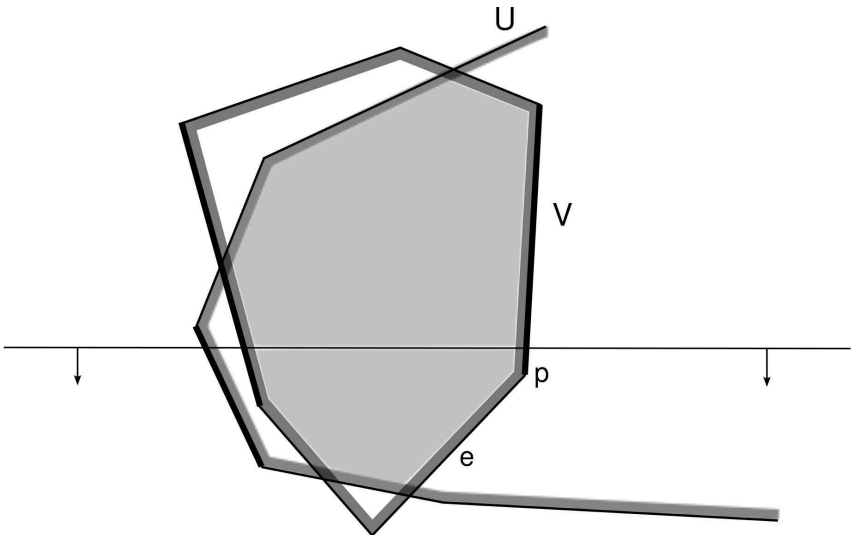
potom sa na situáciu môžeme dívať ako na sústavu dvoch rovníc o dvoch neznámych. Ich vyriešením môžeme dostať hodnoty $x = \frac{p_3q_2 - q_3p_2}{p_1q_2 - q_1p_2}$ a $y = \frac{q_1p_3 - p_1q_3}{p_1q_2 - q_1p_2}$. Všimnite si výraz v menovateli. Ten bude nulový práve vtedy, ak sú priamky rovnobežné a tento prípad sme už ošetrili.

Počítanie prieniku dvoch konvexných útvarov

Keď už sme sa rozcvičili na prieniku dvoch priamok, poďme si teraz povedať algoritmus na výpočet prieniku dvoch konvexných útvarov. Náš cieľ je, aby pracoval v lineárnom čase od počtu vrcholov vstupných útvarov. Ako sa to už v geometrii stáva, algoritmus bude zametací. Označme si vstupné útvary U a V.

Samotný algoritmus prebieha nasledovne: postupujeme smerom *zhora dole*, spracovávame hraničné zoznamy oboch vstupných útvarov L_U, P_U, L_V, P_V a postupne konštruujeme ľavú a pravú hranicu výstupného útvaru - L a P. Ak by sme si situáciu zakreslili, potom zametanie (spracovávanie vstupných útvarov) si môžeme znázorniť ako posúvanie vodorovnej čiary (budeme jej hovoriť *zametacia čiara*) smerom dole. To, čo je nad čiarou už máme spracované a to čo je pod ňou nás ešte len čaká.

Keďže útvary sú konvexné, v každom momente pretína z každého zoznamu našu zametaciu čiaru najviac jedna priamka. Preto pri implementácii nepotrebujeme používať žiadne zložité štruktúry. Bohato nám postačí pamätať si štyri čísla - indexy aktuálnych priamok (teda priamok, ktoré sú pretínané zametacou čiarou) v príslušných zoznamoch (alebo špeciálnu hodnotu, ktorá nám povie, že žiadna taká priamka neexistuje).



Najbližší zaujímavý bod je p, po posunutí zametacej čiary na tento bod budeme spracovávať hranu e. Ak indexujeme od 0, hodnoty jednotlivých ukazovateľov sú pre $L_U = 2$, $L_V = 1$, $P_U = NULL$, $P_V = 1$.

Ako začneme? Nech D_U je najvrchnejší bod prvého vstupného útvaru (ak neexistuje, tak špeciálne položme $D_U = [\infty, \infty]$), nech D_V je to isté pre druhý útvar. Nech M je minimum z y súradníc týchto bodov. Keďže prienik útvarov sú body, ktoré patria obom z nich, potom je ľahko vidno, že žiadny bod s vyššou súradnicou do tohto prieniku nemôže patriť. Preto odignorujeme všetko v zoznamoch L_U, L_V, P_U, P_V, čo sa odohráva nad M (ak bolo $M = \infty$, potom neodignorujeme nič). Najvyššie polroviny, ktoré sa oplatí uvažovať, sú tie, ktoré pretínajú zametaciu čiaru nastavenú na úrovni M.

Najbližšia zaujímavá udalosť sa udeje, až keď posunieme zametaciu čiaru na ďalší bod niektorej z hraníc. Vyberieme si teda najvyšší spomedzi týchto priesečníkov a poďme riešiť, ako bude ďalej vyzerať hranica výsledného útvaru. Keďže máme len štyri možnosti, potom

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

tento priesečník ľahko nájdeme vyskúšaním všetkých možností. Ak náhodou nastáva rovnosť y súradníc, potom budeme najskôr spracovávať priesečníky s nižšou x súradnicou.

Uvažujme situáciu, že najbližší spracovávaný priesečník je v zozname L_U. Označme tento bod ako p a hranu, ktorá z neho vychádza smerom nadol e. Pri vykonávaní nasledovných testov si treba dať pozor, že teraz už nepracujeme s celou hraničiacou priamkou, ale iba s jej úsekom, ktorý začína v bode p a pokračuje až do ďalšieho prieniku v hranici L_U (ak existuje) a samozrejme aj kontrolovať, či prienik s inou priamkou nastáva na mieste, kedy táto iná priamka ohraničuje svoj útvar. Môže sa totiž stať, že vypočítame prienik napríklad s hraničiacou priamkou z P_V, ale na mieste tohto prieniku už útvar V ohraničuje úplne iná priamka. Musíme teda skontrolovať nasledovné (v uvedenom poradí):

- Skontrolovať, či sa bod p nachádza medzi aktuálnou hranou L_V a P_V. Inými slovami, či sa nachádza tento bod vnútri druhého útvaru. Ak áno, potom je začiatok hrany e zjavne na hranici výsledného prieniku a teda hranu e treba pridať na koniec vznikajúceho zoznamu L.
- Zistiť, či sa hrana e pretína s aktuálnou hranou P_V. Ak áno, tak sa v úseku hrany e deje jedna z dvoch vecí: hranica P_V začína naľavo od e a po jej pretnutí končí napravo, alebo opačne. V prvom prípade je útvar V na súčasnej úrovni zametania naľavo od U, avšak nižšie, na úrovni konca hrany e, sa už vstupné útvary pretínajú. Preto treba pridať aktuálnu polrovinu z hranice P_V do zoznamu P a hranicu e do L. Práve toto miesto je totiž najvrchnejší bod vznikajúceho výstupného útvaru. V druhom prípade je situácia opačná - útvar prieniku nám zaniká, pretože na úrovni bodu p je ešte pravá hranica útvaru V napravo od ľavej hranice útvaru U, avšak v dolnom konci hrany e je už V naľavo od U.
- Zistiť, či sa hrana e pretína s aktuálnou hranou z L_V. Toto je trochu podobný prípad. Ak k tomuto prieniku dochádza, znamená to, že od tohto úseku sa zmení vstupný útvar, ktorý bude ohraničovať výstup zľava. Ak je bod p napravo od hrany z L_V, potom pridáme do L hranu z L_V, v opačnom prípade pridáme e.

Po vykonaní týchto troch testov sme pridali do L práve všetky hrany, ktoré bolo treba (premyslite si, že poradie týchto testov je dôležité a že keby sme ho nedodržali, mohol by nastať problém). Zvyšné tri prípady, keď najbližšie spracovávaný priesečník je na niektorej z iných hraníc, sa riešia analogicky. Po spracovaní posledného priesečníku máme rovno hotové aj priebežne budované hranice nového útvaru. Záverom konštatujeme, že dokážeme spracovať jedno posunutie zametacej čiary v konštantnom čase. Posunutí je len toľko, koľko je bodov vo vstupných útvaroch. Preto je celý tento algoritmus na prienik dvoch útvarov naozaj lineárny.

Na záver niekoľko poznámok k implementácii. Uvedený algoritmus musí špeciálne ošetrovať roviny, ktorých deliaca priamka je rovnobežná s osou x, pretože na viacerých miestach by pri zametaní skomplikovalo situáciu. Tieto priamky nám celý priestor, v ktorom hľadáme riešenie, ohraničujú do horizontálneho pásu - nech D je maximum z y súradníc priamok, ktoré ohraničujú situáciu zdola a H minimum z hodnôt y pre priamky, ktoré ohraničujú situáciu zhora. Preto pri čítaní vstupu budeme počítať tieto hodnoty a príslušné roviny zahadzovať. Potom zametanie upravíme, aby pracovalo len medzi týmito hodnotami.

Našou úlohou je vypísať nezáporné čísla. Aby sme toto dosiahli, pridáme do situácie ešte dve polroviny, $-y \leq 0$ a $-x \leq 0$. Prvá z nich je rovnobežná s osou x, preto len zmaximalizujeme hodnotu D s nulou. Tieto polroviny sú, na rozdiel od všetkých ostatných, uzavreté (majú neostrú nerovnosť). Keďže sa nám do zametania dostane len jedna z nich, nevadí to. Prienik priamky ohraničujúcej uzavretú polrovinu a priamky ohraničujúcej otvorenú polrovinu totiž do prienku polrovín stále nepatrí. Z podobnej príčiny nevadí ani uzavretosť dolnej polroviny.

Všimnime si napríklad špeciálne bod $[0, 0]$. Ak bol riešením iba on sám, tak je síce v prieniku našich dvoch pridaných polrovín, ale musel by tiež byť prienikom s nejakou treťou,

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

ktorá je už určite otvorená a do riešenia by nemohol patriť. Ak by taká tretia polrovina neexistovala, potom by musel byť riešením nielen tento bod, ale aj nejaké jeho okolie.

Listing programu:

```
#include<cstdio>
#include<iostream>
#include<cmath>
#include<vector>
#include<algorithm>
using namespace std;
#define INF 1000

//trieda na racionalnu aritmetiku
class Racio{
  public:
    long long citatel, menovatel;
    Racio(long long a, long long b){
      citatel = a;
      menovatel = b;
    }
    Racio(){
      citatel = 0;
      menovatel = 1;
    }
    //uprava zlomku na zakladny tvar, vyuzijeme funkciu __gcd, ktora sa nachadza v
<algorithm>
    void normalizuj(){
      while(true){
        long long d = __gcd( abs(citatel) , abs(menovatel) );
        if (d <= 1) break;
        citatel /= d;
        menovatel /= d;
      }
    }
    Racio operator+( const Racio b ) const{
      Racio vratim = Racio( citatel*b.menovatel+b.citatel*menovatel ,
menovatel*b.menovatel );
      vratim.normalizuj();
      return vratim;
    }
    Racio operator-( const Racio b ) const{
      Racio vratim = Racio( citatel*b.menovatel-b.citatel*menovatel ,
menovatel*b.menovatel );
      vratim.normalizuj();
      return vratim;
    }
    Racio operator*( const Racio b ) const{
      Racio vratim = Racio( citatel*b.citatel , menovatel*b.menovatel );
      vratim.normalizuj();
      return vratim;
    }
    Racio operator/( const Racio b ) const{
      Racio vratim = Racio( citatel*b.menovatel , menovatel*b.citatel );
      vratim.normalizuj();
      return vratim;
    }
    bool operator==( Racio b ) const{
      return citatel*b.menovatel == menovatel*b.citatel;
    }
```


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

```
    bool operator<( Racio b ) const{
      return citatel*b.menovatel < menovatel*b.citatel;
    }
    bool operator>( Racio b ) const{
      return citatel*b.menovatel > menovatel*b.citatel;
    }
    bool operator>=( Racio b ) const{
      return citatel*b.menovatel >= menovatel*b.citatel;
    }
    bool operator<=( Racio b ) const{
      return citatel*b.menovatel <= menovatel*b.citatel;
    }
    bool kladne(){
      return citatel*menovatel > 0;
    }
    bool zaporne(){
      return citatel*menovatel < 0;
    }
    bool nula(){
      return citatel == 0;
    }
};

typedef pair<Racio,Racio> bod;

//vektorovy sucin dvoch vektorov
Racio vs( Racio u1 , Racio u2 , Racio v1 , Racio v2 ){
  return u1*v2 - u2*v1;
}

//true, ak ma byt bod A spracovany pri zametani skorej ako bod B
//je to vtedy, ked ma A vyssiu y-ovu suradnicu, v pripade rovnosti mensiu x-ovu
bool skorej( bod A , bod B ){
  if (A.second == B.second){
    return A.first < B.first;
  }
  return A.second > B.second;
}

//struktura na hraniciacu polrovinu
struct priamka{
  //dva body, ktore urcuji hraniciacu priamku
  bod A,B;
  //bod, ktory lezi na strane priamky, kde je reprezentovana polrovina
  bod patri;
  //Test, ci je bod C v tejto polrovine.
  //Test robime podla vektoroveho sucinu, vektorovy
  //sucin smeroveho vektoru s vektorom do bodu C ma mat rovnake
  //znamienko ako sucin smeroveho vektoru a vektoru do bodu 'patri'.
  //Vieme, ze polrovina je otvorena, teda bod patriaci hraniciacej priamke do nej
nepatri,
  //preto neuvazujeme pripad, kedy VS vyde 0
  bool spravnastrana( bod C ) const{
    Racio u1 = A.first - B.first;
    Racio u2 = A.second - B.second;
    Racio v1 = A.first - patri.first;
    Racio v2 = A.second - patri.second;
    Racio w1 = A.first - C.first;
    Racio w2 = A.second - C.second;
    Racio vs1 = u1*v2 - u2*v1;
```


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

```
    Racio vs2 = u1*w2 - u2*w1;
    return  ( (vs1.kladne() && vs2.kladne()) || (vs1.zaporne() && vs2.zaporne()) );
  }
  //vrati bod, ktory patri priamke a nachadza sa na prislusnej
  //y-ovej suradnici. vstup predpoklada, ze priamka nie je rovnobezna
  //s osou x
  bod bodpriamky( Racio y ) const{
    //sklon - smernica priamky (ako velmi sa meni 'x' pri naraste 'y')
    Racio sklon = (A.first - B.first) / (A.second - B.second);
    Racio x = Racio( y - A.second ) * sklon + A.first;
    return bod( x , y );
  }
};

//struktura na utvar
struct utvar{
  //hranice, P = prava, L = lava
  vector<priamka> P,L;
  //vytvor utvar ohraniceny jedinou polrovinou
  utvar(priamka p){
    L = vector<priamka>(0);
    P = vector<priamka>(0);
    //rozbor pripadov, ci sa nachadza polrovina na pravo alebo na lavo od priamky
    if (p.A.second > p.B.second){
      Racio u2 = p.A.second - p.B.second;
      Racio u1 = p.A.first - p.B.first;
      if ( vs(u1,u2,p.patri.first-p.B.first,p.patri.second-p.B.second).kladne() ){
P.push_back( p ); }
      else{ L.push_back( p ); }
    }else{
      Racio u2 = p.B.second - p.A.second;
      Racio u1 = p.B.first - p.A.first;
      if ( vs(u1,u2,p.patri.first-p.A.first,p.patri.second-p.A.second).kladne() ){
P.push_back( p ); }
      else{ L.push_back( p ); }
    }
  }
  utvar(){
    L.resize( 0 );
    P.resize( 0 );
  }
};

//globalne premenne
int N;
priamka P[100000];
int u1,u2,u3;
int a[100000],b[100000],c[100000];
//hore, dole - hodnoty y, medzi ktorymi sa odohrava zametanie
//inicializuje sa podla rovin rovnobeznych s osou 'x'
Racio hore,dole;

//vrati bod, v ktorom sa pretnu priamky p1 a p2. do flagu sa nastavi, ci prienik existuje
bod prienik( const priamka &p1 , const priamka &p2 , bool &flag ){
  flag = true;
  //na rozdiel od textu vzoraku, tu si rovno pocitame normalove
  //vektory na zistenie rovnobeznosti
  //premyslite si, ze je to ekvivalentny test
  Racio u1 = p1.B.second - p1.A.second;
  Racio u2 = p1.A.first - p1.B.first;
```


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

```
  Racio v1 = p2.B.second - p2.A.second;
  Racio v2 = p2.A.first - p2.B.first;
  Racio D = u1*v2 - u2*v1;
  //su priamky rovnobezne?
  if ( D.nula() ){
    if ( vs(u1,u2,p2.A.first-p1.B.first,p2.A.second-p1.B.second).nula() ){
      return bod( p1.A.first , p1.A.second );
    }else{
      flag = false;
      return bod( Racio(0,1) , Racio(0,1) );
    }
  }
  Racio c1 = p1.A.first*p1.B.second - p1.A.second*p1.B.first;
  Racio c2 = p2.A.first*p2.B.second - p2.A.second*p2.B.first;
  Racio d1 = c1*v2 - c2*u2;
  Racio d2 = u1*c2 - v1*c1;
  return bod( d1/D , d2/D );
}

//pre dany zoznam hraniciacich priamok tvoriacich hranicu
//a index v tomto zozname vrati najblizsi priesecnik na tejto hranici.
//ak sme na poslednej priamke a teda tento bod neexistuje, vratime bod velmi daleko
dole
//na poslednej priamke
bod dalsiprienik( const vector<priamka> &hranica , int p ){
  bod vratim = hranica[p].bodpriamky( dole );
  bool flag;
  if (p < (int)hranica.size() - 1) vratim = prienik( (priamka&)hranica[p] ,
(priamka&)hranica[p+1] , flag);
  return vratim;
}

//pre dane dva intervaly (l1,p1) a (l2,p2) nastavime bod B,
//ktory ma x-ovu suradnicu v oboch intervaloch a y-ovu z parametra
//funkcia predpoklada, ze intervaly maju neprazdny prienik
//ak nastane specialna situacia, ze prienik je prave jeden bod, nenastavi sa nic
void nastavbod( Racio y , Racio l1 , Racio l2 , Racio p1 , Racio p2 , bod& B ){
  Racio vlavo = l1;
  if (l1 < l2) vlavo = l2;
  Racio vpravo = p1;
  if (p1 > p2) vpravo = p2;
  if ( vlavo == vpravo ) return;
  B = bod( (vpravo+vlavo)/Racio(2,1) , y );
}

//metoda na spracovanie usecky/polroviny, ktora ohranicuje situaciu zlava.
//horny bod tejto usecky je 'dalsi', samotna priamka je 'p'.
//hodnoty od,po oznacuju horizontalny pas, v ktorom tato usecka ohranicuje utvar
//ostatne vstupne parametre su vznikajuce zoznamy vystupneho utvaru a ostatne
vstupne ohranicujuce zoznamy.
//pre popis testov vid text vzoraku
void spracujlavu(bod dalsi, priamka p, Racio od, Racio po, const vector<priamka> &
prava1, int p1, const vector<priamka> &lava2, int l2, const vector<priamka> &prava2,
int p2, vector<priamka> &doplnany, vector<priamka> &doplnanyp, bod &bodvnutri){
  if ( po > hore || od < dole ) return;
  bool bolprienik = false;
  //test 1
  bool vlavo = true, vpravo = true;
  if ( l2 < lava2.size() ) if ( !lava2[l2].spravnastrana(dalsi) ) vlavo = false;
  if ( p2 < prava2.size() ) if ( !prava2[p2].spravnastrana(dalsi) ) vpravo = false;
```


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

```
  if ( vpravo && vlavo ){
    bolprienik = true;
    doplnany.push_back( p );
  }
  //test 2
  bool flag = true;
  bod B;
  if ( p2 < prava2.size() ){
    B = prienik( p , prava2[p2] , flag );
    if ( (flag == true) && (B.second <= od) && (B.second >= po) && (B.second >=
dalsiprienik(prava2,p2).second) && (B.second > dole) ){
      if ( prava2[p2].spravnastrana( dalsi ) ) return;
      bolprienik = true;
      doplnany.push_back( p );
      doplnanyp.push_back( prava2[p2] );
    }
  }
  //test 3
  flag = true;
  if ( l2 < lava2.size() ){
    B = prienik( p , lava2[l2] , flag );
    if ( (flag == true) && (B.second <= od) && (B.second >= po) && (B.second >=
dalsiprienik(lava2,l2).second) && (B.second > dole) ){
      bolprienik = true;
      if ( lava2[l2].spravnastrana( dalsi ) ){
        doplnany.push_back( lava2[l2] );
      }else{
        doplnany.push_back( p );
      }
    }
  }
  //ak sme na tejto y-ovej urovni zistili, ze vstupne utvary tu maju neprazdny prienik,
potom skusme
  //vypocitat priklad bodu vo vyslednom utvare
  if (bolprienik){
    Racio vpravo1 = Racio(INF,1), vpravo2 = Racio(INF,1), vlavo2 = Racio(-INF,1);
    if ( p1 < prava1.size() ){
      vpravo1 = prava1[p1].bodpriamky( dalsi.second ).first;
    }
    if ( p2 < prava2.size() ){
      vpravo2 = prava2[p2].bodpriamky( dalsi.second ).first;
    }
    if ( l2 < lava2.size() ){
      vlavo2 = lava2[l2].bodpriamky( dalsi.second ).first;
    }
    nastavbod( dalsi.second , dalsi.first , vlavo2 , vpravo1 , vpravo2 , bodvnutri );
  }
  return;
}

//metoda na spracovanie novej usecky pravej, analogicka interpretacia ako pri metode
'spracujlavu'
void spracujpravu(bod dalsi, priamka p, Racio od, Racio po, const vector<priamka>
&lava1, int l1, const vector<priamka> &lava2, int l2, const vector<priamka> &prava2,
int p2, vector<priamka> &doplnany, vector<priamka> &doplnanyl, bod &bodvnutri){
  if ( po > hore || od < dole ) return;
  bool bolprienik = false;
  //test 1
  bool vlavo = true, vpravo = true;
  if ( l2 < lava2.size() ) if ( !lava2[l2].spravnastrana(dalsi) ) vlavo = false;
```


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

```
  if ( p2 < prava2.size() ) if ( !prava2[p2].spravnastrana(dalsi) ) vpravo = false;
  if ( vpravo && vlavo ){
    bolprienik = true;
    doplnany.push_back( p );
  }
  //test 2
  bool flag = true;
  bod B;
  if ( l2 < lava2.size() ){
    B = prienik( p , lava2[l2] , flag );
    if ( (flag == true) && (B.second <= od) && (B.second >= po) && (B.second >=
dalsiprienik(lava2,l2).second) && (B.second > dole) ){
      if ( lava2[l2].spravnastrana( dalsi ) ) return;
      bolprienik = true;
      doplnany.push_back( p );
      doplnanyl.push_back( lava2[l2] );
    }
  }
  //test 3
  flag = true;
  if ( p2 < prava2.size() ){
    B = prienik( p , prava2[p2] , flag );
    if ( (flag == true) && (B.second <= od) && (B.second >= po) && (B.second >=
dalsiprienik(prava2,p2).second) && (B.second > dole) ){
      bolprienik = true;
      if ( prava2[p2].spravnastrana( dalsi ) ){
        doplnany.push_back( prava2[p2] );
      }else{
        doplnany.push_back( p );
      }
    }
  }
  //bol neprazdny prienik vstupnych utvarov?
  if (bolprienik){
    Racio vlavo1 = Racio(INF,1), vpravo2 = Racio(INF,1), vlavo2 = Racio(-INF,1);
    if ( l1 < lava1.size() ){
      vlavo1 = lava1[l1].bodpriamky(dalsi.second).first;
    }
    if ( p2 < prava2.size() ){
      vpravo2 = prava2[p2].bodpriamky(dalsi.second).first;
    }
    if ( l2 < lava2.size() ){
      vlavo2 = lava2[l2].bodpriamky(dalsi.second).first;
    }
    nastavbod( dalsi.second , vlavo1 , vlavo2 , dalsi.first , vpravo2 , bodvnutri );
  }
  return;
}

//metoda vypocita prienik dvoch utvarov a ak je neprazdny, nastavi bod 'b'
//ako priklad bodu, ktory je vnutri vystupneho utvaru
//hodnota tohto bodu sa pocita pri priebeznom spracovani useciek
utvar * prienik( const utvar &p1 , const utvar &p2 , bod &b ){
  utvar * vratim = new utvar();
  //inicializujeme zametaciu ciaru na nizsi z najvyssich bodov vstupnych utvarov
  //sweep - y-ova hodnota tohto bodu
  Racio sweep = hore;
  if ( (p1.L.size() > 0) && (p1.P.size() > 0) ){
    bool flag = true;
    bod p = prienik( (priamka&)p1.L[0] , (priamka&)p1.P[0] , flag );
```


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

```
    if (flag == true && p.second < sweep) sweep = p.second;
  }
  if ( (p2.L.size() > 0) && (p2.P.size() > 0) ){
    bool flag = true;
    bod p = prienik( (priamka&)p2.L[0] , (priamka&)p2.P[0] , flag );
    if (flag == true && p.second < sweep) sweep = p.second;
  }
  //indexy, kde sa aktulane v hraniciacich zoznamoch nachadzame
  int p1l = 0, p1p = 0, p2l = 0, p2p = 0;
  bool flag;
  //ignorujeme vsetko nad hodnotou sweep
  while( p1l < (int)p1.L.size()-1 ){
    if ( dalsiprienik(p1.L,p1l).second > sweep ){ p1l++; }
    else{ break; }
  }
  while( p1p < (int)p1.P.size()-1 ){
    if ( dalsiprienik(p1.P,p1p).second > sweep ){ p1p++; }
    else{ break; }
  }
  while( p2l < (int)p2.L.size()-1 ){
    if ( dalsiprienik(p2.L,p2l).second > sweep ){ p2l++; }
    else{ break; }
  }
  while( p2p < (int)p2.P.size()-1 ){
    if( dalsiprienik(p2.P,p2p).second > sweep ){ p2p++; }
    else{ break; }
  }
  //na uvod spracujeme hraniciace priamky na urovni sweep
  //pri vsetkych spracovavaniach musime vypocitat hodnoty 'zaciatok' a 'koniec',
  //to je vertikalny pas, kde je prislusna polrovina ohranicuje utvar
  if ( p1l < p1.L.size() ){
    bod zaciatok = p1.L[p1l].bodpriamky( hore );
    if ( p1l > 0 ) zaciatok = prienik( (priamka&)p1.L[p1l-1] , (priamka&)p1.L[p1l] ,
flag );
    bod koniec = dalsiprienik( p1.L , p1l );
    spracujlavu( zaciatok , p1.L[p1l] , zaciatok.second , koniec.second , p1.P , p1p
, p2.L , p2l , p2.P , p2p , vratim->L , vratim->P , b );
  }
  if ( p2l < p2.L.size() ){
    bod zaciatok = p2.L[p2l].bodpriamky( hore );
    if ( p2l > 0 ){ zaciatok = prienik( (priamka&)p2.L[p2l-1] , (priamka&)p2.L[p2l] ,
flag ); }
    bod koniec = dalsiprienik( p2.L , p2l );
    spracujlavu( zaciatok , p2.L[p2l] , zaciatok.second , koniec.second , p2.P , p2p
, p1.L , p1l , p1.P , p1p , vratim->L , vratim->P , b );
  }
  if ( p1p < p1.P.size() ){
    bod zaciatok = p1.P[p1p].bodpriamky( hore );
    if ( p1p > 0 ) zaciatok = prienik( (priamka&)p1.P[p1p-1] , (priamka&)p1.P[p1p]
, flag );
    bod koniec = dalsiprienik( p1.P , p1p );
    spracujpravu( zaciatok , p1.P[p1p] , zaciatok.second , koniec.second , p1.L ,
p1l , p2.L , p2l , p2.P , p2p , vratim->P , vratim->L , b );
  }
  if ( p2p < p2.P.size() ){
    bod zaciatok = p2.P[p2p].bodpriamky( hore );
    if ( p2p > 0 ){ zaciatok = prienik( (priamka&)p2.P[p2p-1] , (priamka&)p2.P[p2p]
, flag ); }
    bod koniec = dalsiprienik( p2.P , p2p );
```


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

```
    spracujpravu( zaciatok , p2.P[p2p] , zaciatok.second , koniec.second , p2.L ,
p2l , p1.L , p1l , p1.P , p1p , vratim->P , vratim->L , b );
  }
  //a samotne posuvanie sa a zametanie
  //vybrany - identifikator hranice, z ktorej pochadza bod, ktory najblizsie treba
spracovat
  int vybrany = 0;
  while(vybrany != -1){
    bod dalsi = bod( Racio(INF,1) , dole );
    vybrany = -1;
    if ( p1l < (int)p1.L.size() - 1 ) if ( skorej( prienik((priamka&)p1.L[p1l] ,
(priamka&)p1.L[p1l+1] , flag ) , dalsi) ){
      dalsi = prienik( (priamka&)p1.L[p1l] , (priamka&)p1.L[p1l+1] , flag );
      vybrany = 0;
    }
    if ( p1p < (int)p1.P.size() - 1 ) if ( skorej( prienik((priamka&)p1.P[p1p] ,
(priamka&)p1.P[p1p+1] , flag ) , dalsi) ){
      dalsi = prienik( (priamka&)p1.P[p1p] , (priamka&)p1.P[p1p+1] , flag );
      vybrany = 1;
    }
    if ( p2l < (int)p2.L.size() - 1 ) if ( skorej( prienik((priamka&)p2.L[p2l] ,
(priamka&)p2.P[p2l+1] , flag ) , dalsi) ){
      dalsi = prienik( (priamka&)p2.L[p2l] , (priamka&)p2.L[p2l+1] , flag );
      vybrany = 2;
    }
    if ( p2p < (int)p2.P.size() - 1 ) if ( skorej( prienik((priamka&)p2.P[p2p] ,
(priamka&)p2.P[p2p+1] , flag ) , dalsi) ){
      dalsi = prienik( (priamka&)p2.P[p2p] , (priamka&)p2.P[p2p+1] , flag );
      vybrany = 3;
    }
    //ak je prienik uplne dole, uz ho nespracovavame
    //(v pripade rovnosti urcite nove usecky neprispeju do utvaru, kedze polroviny su
otvorene)
    if (dalsi.second <= dole) return vratim;
    if (vybrany == 0){
      p1l++;
      bod koniec = dalsiprienik( p1.L , p1l );
      spracujlavu( dalsi , p1.L[p1l] , dalsi.second , koniec.second , p1.P , p1p ,
p2.L , p2l , p2.P , p2p , vratim->L , vratim->P , b );
    }
    if (vybrany == 1){
      p1p++;
      bod koniec = dalsiprienik( p1.P , p1p );
      spracujpravu( dalsi , p1.P[p1p] , dalsi.second , koniec.second , p1.L , p1l ,
p2.L , p2l , p2.P , p2p , vratim->P , vratim->L , b );
    }
    if (vybrany == 2){
      p2l++;
      bod koniec = dalsiprienik( p2.L , p2l );
      spracujlavu( dalsi , p2.L[p2l] , dalsi.second , koniec.second , p2.P , p2p ,
p1.L , p1l , p1.P , p1p , vratim->L , vratim->P , b );
    }
    if (vybrany == 3){
      p2p++;
      bod koniec = dalsiprienik( p2.P , p2p );
      spracujpravu( dalsi , p2.P[p2p] , dalsi.second , koniec.second , p2.L , p2l ,
p1.L , p1l , p1.P , p1p , vratim->P , vratim->L , b );
    }
  }
  return vratim;
```


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

```
}

//zakladna funkcia algoritmu. vstup je niekolko polrovin danych
//usekom pola polroviny na indexoch [odkial,pokial] vratane.
//uloha je vypocitat utvar ktory je ich prienikom, popritom
//sa do bodu 'a' nastavi priklad bodu patriaci do tohto utvaru
utvar * f( priamka polroviny[] , int odkial , int pokial , bod &a ){
  if (pokial ==  odkial){
    a = (bod&)polroviny[odkial].patri;
    return new utvar( polroviny[odkial] );
  }
  int m = (odkial+pokial)/2;
  bod temp;
  //pre vacsie mnozstvo polrovin ratame prienik prieniku prvej polovice a prieniku
druhej polovice
  return prienik( *f(polroviny,odkial,m,(bod&)temp) ,
*f(polroviny,m+1,pokial,(bod&)temp) , a );
}

int main(){
  int rovin = 0;
  hore = Racio(INF,1);
  dole = Racio(-INF,1);
  scanf("%d ",&N);
  scanf("%d %d %d ",&u1,&u2,&u3);
  for( int i=0; i<N-1; i++ ){
    //nacitavame a transformujeme vstup na nerovnice, v ktorych z = 1
    //priklad bodu patriaceho do danej polroviny zistujeme analyzou pripadov
    //overujucemu citatelovi odporucame napisat si na papier
    scanf("%d %d %d ",&a[rovin],&b[rovin],&c[rovin]);
    a[rovin] -= u1;
    b[rovin] -= u2;
    c[rovin] = u3 - c[rovin];
    if ( a[rovin] != 0 ){
      if ( b[rovin] != 0 ){
        P[rovin].A = bod( Racio(0,1) , Racio(c[rovin],b[rovin]) );
        P[rovin].B = bod( Racio(1,1) , Racio(c[rovin]-a[rovin],b[rovin]) );
      }else{
        P[rovin].A = bod( Racio(0,1) , Racio(0,1) );
        P[rovin].B = bod( Racio(0,1) , Racio(1,1) );
      }
      rovin++;
    }else{
      //specialny pripad = rovina rovnobezna s osou 'x'.
      //nepridame ju medzi ostatne, ale spracujeme osobitne
      if ( b[rovin] == 0 ){
        //je aj druhy koeficient 0? potom alebo do nej patria vsetky body, alebo ziadny
        if ( c[rovin] <= 0 ){
          printf("Misof sa neda nastavit!\n");
          return 0;
        }
      }
      if ( b[rovin] < 0 ){
        if ( Racio(c[rovin] , b[rovin]) > dole ) dole = Racio( c[rovin] , b[rovin] );
      }
      if (b[rovin] > 0){
        if ( Racio(c[rovin] , b[rovin]) < hore ) hore = Racio( c[rovin] , b[rovin] );
      }
    }
  }
```


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

```
  //nastavime horizontaly pas na len kladne y-ony, ak este nebol
  if ( dole < Racio(0,1) ) dole = Racio(0,1);
  if ( dole >= hore ){
    printf("Misof sa neda nastavit!\n");
    return 0;
  }
  //doplnime body patriace polrovinam
  for(int i=0;i<rovin;i++){
    Racio y = (dole+hore)/Racio(2,1);
    if (a[i] > 0){
      P[i].patri = bod( (Racio(c[i],1) - Racio(b[i],1)*y)/Racio(a[i],1) - Racio(1,1)
, y );
    }else{
      P[i].patri = bod( (Racio(c[i],1) - Racio(b[i],1)*y)/Racio(a[i],1) + Racio(1,1)
, y );
    }
  }
  //pridame rovinu, ktora obmedzi riesenia len na tie s nezapornymi x-ami
  //bezne su roviny sice otvorene, avsak toto nam nevadi
  P[rovin].A = bod( Racio(0,1) , Racio(0,1) );
  P[rovin].B = bod( Racio(0,1) , Racio(1,1) );
  P[rovin].patri = bod( Racio(1,1) , (dole+hore)/Racio(2,1) );
  rovin++;
  bod B;
  utvar * p = f( P , 0 , rovin-1 , B );
  if ( p->L.size() + p->P.size() > 0 ){
    B.first.normalizuj();
    B.second.normalizuj();
    printf("Vyhovujuce su napriklad hodnoty [%lld/%lld,%lld/%lld,1]\n", B.first.citatel ,
B.first.menovatel , B.second.citatel , B.second.menovatel );
    return 0;
  }
  //druha verzia vstupu, kedy pracujeme s pripadom z = 0
  for( int i=0; i<rovin-1; i++ ){
    P[i].A = bod( Racio(0,1) , Racio(0,1) );
    P[i].B = bod( Racio(b[i],a[i]) , Racio(1,1) );
    Racio y = ((dole+hore)/Racio(2,1));
    if ( a[i] < 0 ){
      P[i].patri = bod( Racio(b[i],a[i])*y - Racio(1,1) , y );
    }else{
      P[i].patri = bod( Racio(b[i],a[i])*y + Racio(1,1) , y );
    }
  }
  p = f( P , 0 , rovin-1 , B );
  if ( p->L.size() + p->P.size() > 0 ){
    B.first.normalizuj();
    B.second.normalizuj();
    printf("Vyhovujuce su napriklad hodnoty [%lld/%lld,%lld/%lld,0]\n", B.first.citatel ,
B.first.menovatel , B.second.citatel , B.second.menovatel );
    return 0;
  }
  printf("Misof sa neda nastavit!\n");
  return 0;
}
```


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

opravovalo sa samo, vzorák Usamec
(max. 25 bodov)

2::9. Tesnotka pred kontrolou

Príklad na prvý pohľad vyzeral vcelku odpudzujúco. Máme rozdeliť vstup na presne rovnaké polovice a navyše máme minimalizovať súčet dvoch podielov. Fuj...

Poďme najprv vykonať niekoľko užitočných pozorovaní. Pozorovanie prvé: Keď viem cenu a počet samohlások v jednej polovici, viem to automaticky aj pre druhú (odčítam od celkovej ceny a počtu samohlások). Pozorovanie druhé: Súčet cien a súčet počtu samohlások nie je až tak vysoký.

Takto sa už dá zostaviť celkom veľarozmerná dynamika. Budeme chcieť zistiť, či vieme vybrať K kníh s celkovo S samohláskami a celkovou cenou C. To dáva celkové rozmery $26 \times 25001 \times 25001$. Programovať sa to bude celkom jednoducho. Budeme postupne používať knihy. Na začiatku sme nepoužili žiadnu, takže vieme mať akurát 0 kníh s celkovou cenou a počtom samohlások 0. Teraz zoberieme knihu. Pozrieme sa na všetky možné kombinácie, čo zatiaľ môžeme spraviť. Nech má kniha cenu c a počet samohlások s. Potom ak vieme spraviť doteraz bez nej kombináciu (K, S, C), tak s ňou vieme mať kombinácie (K, S, C) a ($K+1$, $S+s$, $C+c$). Keď už vyskúšame všetky knihy, tak sa len pozrieme na možné dosiahnuteľné stavy s polovicou kníh v jednej polovici a vyberieme najlepší.

Stále toto je ale príliš pomalé. Pozrime sa, ako sa počíta naše výsledné číslo, ak v jednej kope máme knihy s cenou C a počtom samohlások S (celková cena je označená ako Σ_C a celkový počet samohlások ako Σ_S):

$$\frac{C}{S} + \frac{\Sigma_C - C}{\Sigma_S - S}$$

Fixujme teraz hodnotu počtu samohlások v kope a položme si otázku, aká je najvýhodnejšia cena jednej kopy. Tento výraz je lineárna funkcia od C. Takže je výhodné mať C čo najmenšie, alebo čo najväčšie. Takže môžeme teraz počítať nasledovné: Nech v jednej polovici je K kníh s S samohláskami. Za akú najnižšiu a najvyššiu cenu to vieme dostať? Toto sa už dá programovať a ľahko počítať. Ale spravíme ešte jedno zjednodušenie. Budeme si počítať iba najnižšiu cenu. Totiž ak je výhodná najvyššia cena pri počte samohlások S, tak je rovnako výhodná najnižšia cena pri počte samohlások $\Sigma_S - S$. Túto najnižšiu cenu si označíme ako $m(K, S)$.

Pre úplnosť ešte popíšeme ako bude vyzerať výsledná dynamika: Opäť budeme postupne používať knihy. Na začiatku vieme akurát dosiahnuť počet kníh 0, počet samohlások 0 s minimálnou cenou 0 (takže máme $m(0,0) = 0$. Teraz dostaneme knihu s cenou c a počtom samohlások s. A teraz skúsime vylepšiť každú dvojicu $m(K, S)$. Vyberieme menšiu hodnotu z dvojice $m(K, S)$ a $m(K-1, S-s)+c$. Samozrejme si dáme pozor, aby sme vylepšovali v rozumnom poradí (nech jednu knihu nepoužijeme 25-krát).

Pamäte potrebujeme teraz už len $O(N^2 \cdot max_S)$ (max_S je maximálny počet samohlások v knihe), reálne 25×25000. Času potrebujeme $O(N^3 \cdot max_S)$, čo je pri zadaných hodnotách zvládnuteľné.

Listing programu:

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

int main() {
    int n;
    scanf("%d", &n);
    vector<pair<int, int > > book;
    book.resize(n);
    for (int i = 0; i < n; i++)
```


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

```
        scanf("%d", &book[i].first);
    for (int i = 0; i < n; i++)
        scanf("%d", &book[i].second);

    vector<vector<int> > data;
    data.resize(27);
    for (int i = 0; i < 27; i++)
        data[i].resize(25100, 1000000000);

    data[0][0] = 0;
    for (int i = 0; i < n; i++)
        for (int j = n / 2 - 1; j >= 0; j--)
            for (int k = 0; k < 25100; k++)
                if (data[j][k] < 1000000000)
                    data[j + 1][k + book[i].first] = min(data[j + 1][k +
book[i].first], data[j][k] + book[i].second);

    int sums = 0, sumc = 0;
    for (int i = 0; i < n; i++) {
        sums += book[i].first;
        sumc += book[i].second;
    }

    long double mi = 1000000000;
    for (int i = 0; i < 25100; i++) {
        if (data[n / 2][i] >= 1000000000)
            continue;
        long double av = (long double) data[n / 2][i] / (long double) i + (long
double) (sumc - data[n / 2][i]) / (long double) (sums - i);
        if (av < mi)
            mi = av;
    }
    printf("%.10Lf\n", mi);
}
```

opravoval Tomi
(max. 25 bodov)

2::10. Turbulencia a krevety

Na úlohy, kde je veľa vecí v rovine, ale majú celočíselné súradnice, existuje niekoľko všeobecných trikov. Tu použijeme dva z nich – prečíslovanie súradníc a zametanie.

Vďaka prečíslovaniu nebudeme mať súradnice od mínus milióna po plus milión, ale od nuly po (rádovo) N. Zvlášť sa to hodí v úlohách, kde vôbec nezáleží na obsahu a vzdialenostiach, lebo to ani nemusíme neskôr prečíslovať naspäť, aby sme vyrátali výsledok. V našom prípade nie je dôležité, aké sú tie štvorce (teda po prečíslovaní už väčšinou obdĺžniky) veľké, iba to, kde majú najväčšie pokrytie (krevetami).

Zametanie[7] sa dá často použiť aj vtedy, keď súradnice nie sú celočíselné. Zoberme nekonečne dlhú metlu (čiže vlastne priamku), začneme niekde ďaleko a postupne zametieme všetko, čo v rovine je. Keď tou metlou niečo nájdeme, volá sa to udalosť. Tuto pustíme metlu rovnobežne s nejakou súradnicovou osou, preto udalosti nebudú samostatné body, ale „našiel som úsečku, ktorou začína/končí štvorec“.

Kostra riešenia

[7] http://en.wikipedia.org/wiki/Sweep_line_algorithm

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

Riešenie v čase $O(N^2)$ by vyzeralo tak, že prečíslujeme súradnice a zrátame pre celú rovinu dynamickým programovaním, ktoré políčko je ako zasiahnuté. Ale tu si ukážeme lepšie riešenie, ktoré beží v $O(N \log N)$ čase s $O(N)$ pamäťou.

Keby sme úlohu riešili rovno pre celú rovinu, aj s prečíslovaním súradníc to nezlepšíme pod $O(N^2)$. Ale vďaka zametaniu nám bude stačiť pamätať si, ako vyzerajú štvorce, ktoré sú práve pod metlou. Metla je priamka, takže nám stačí vedieť spraviť „teraz odtiaľto potiaľto pribudol/odbudol štvorec“ a „zisti, aké je pod metlou najväčšie pokrytie“. Situáciu pod priamkou ešte k tomu prečíslujeme, aby boli tie „odtiaľto“ a „potiaľto“ od nula po rádovo N.

Udalostí (t.j. začiatok alebo koniec štvorca) je $O(N)$, takže aby sa to celé zmestilo do $O(N \log N)$, spracovanie jednej udalosti môže trvať najviac $O(\log N)$. Vlastne máme pole dlhé $O(N)$ a potrebujeme spôsob, ako robiť dve veci: naraz ku všetkým prvkom poľa od a po b pripočítať v, a zistiť, aká je práve najväčšia hodnota v celom poli. A oboje musí ísť v $O(\log N)$. O chvíľu si ukážeme rôzne dátové štruktúry, čo to dokážu, ale predtým ešte musíme vyriešiť pár detailov.

Po prvé, podľa zadania máme stredy štvorcov vo výsledku ignorovať. Tak každý zarátame ako 1×1 štvorec, ktorý nie je za „+1“ bod, ale „$-\infty$“. (V praxi to nie je ∞, ale také malé číslo, aby sa ten štvorec zaručene nedostal na výstup – takže niečo menšie ako $-N$.)

Po druhé, musíme odignorovať miesta, čo nie sú vnútri lietadla. Preto pri načítavaní osekáme všetky štvorce tak, aby sa zmestili dnu. Jano Hozza má ešte iný spôsob: všade okolo lietadla pridáme ďalšie „$-\infty$“.

Sme pripravení, ide sa na vec:

Tomiho súčtovo-maximový intervalový strom

Kódi sa takmer rovnako ako normálny intervalový strom[8] a má sklony byť užitočný v nečakaných situáciách.

Intervalové stromy vedia väčšinou dve veci, obidve v logaritmickom čase: zmeniť hodnotu $A[i]$ o v, a...

- súčtový: zistiť, koľko je súčet $A[0] + A[1] + \ldots + A[i-1]$ (to budeme volať $P[i]$).
- maximový: zistiť maximum z $A[0], A[1], \ldots, A[i]$.
- súčtovo-maximový: zistiť maximum z $P[0], P[1], \ldots, P[i]$.

Na čo je to dobré? Presne na to, čo potrebujeme. Keď pridávame štvorec, čo ide od y_1 po y_2, tak na $A[y_1]$ pridáme $+1$ a na $A[y_2]$ pridáme -1. Potom hodnota $P[y]$ vyjadruje okrevetenosť pozície y. A keď chceme vedieť najväčšieho chudáka, tak chceme vedieť maximum zo všetkých $P[y]$.

Ako sa to naprogramuje? Rovnako, ako bežný súčtový alebo maximový intervaláč, ale v každom vrchole si pamätáme dve rôzne čísla, súčet a maximum, a počítame ich týmito vzorcami:

```
sucet[ja] := sucet[lavysyn] + sucet[pravysyn]
maximum[ja] := max(maximum[lavysyn], sucet[lavysyn]+maximum[pravysyn])
```

Aby sme si zachovali $O(logN)$ čas, tak „maximum“ vrcholu `V` nemôže závisieť od ničoho, čo nie je v jeho potomkoch. Preto sa pri rátaní `maximum[V]` chvíľu tvárime, že `V` je koreň a nič vľavo od neho neexistuje. Keď sa potom zisťujeme maximum rodiča, musíme to dať na rovnakú mieru tak, že ku maximu pravého syna ešte pripočítame `sucet[lavysyn]` (zase ako keby `ja` bol koreň a nič vľavo od neho neexistovalo).

V tejto úlohe stačí vedieť zisťovať maximum z celého stromu, čo už je v `maximum[koreň]`, ale strom zvládne s menšími úpravami zisťovať aj maximum z $P[0 \ldots b]$, a s väčšími aj maximum z $P[a \ldots b]$.

Lazy-loading intervalový strom

[8] lebo som lenivý, tak si ho naštudujte vo vzoráku 27. ročníka, prvého letného kola

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

Lazy-loading stromy sú síce štandardnejšie riešenie na takéto úlohy, ale ja ich veľmi nemusím... našťastie to už odovzdal Mišo Anderle, takže som to nemusel vymýšľať :-)

Použijeme maximový lazy-loading strom, ktorý vie jednak zisťovať maximum zo všetkých $A[i]$, a druhak vie všetky $A[a,\ldots,b]$ naraz zmeniť o v. Dokáže to tak, že nemení zaradom $A[a]$, $A[a+1]$, atď až do $A[b]$, ale v každom vrchole `V` si pamätá ešte hodnotu `lazy[V]`, ktorá znamená „všetky Áčka, čo sú mojimi potomkami, by mali byť ešte o toľkoto väčšie, ale teraz sa mi to tam nechce zapisovať“. Vždy, keď idem hlbšie ako do `V`, tak môj `lazy` presuniem do svojich potomkov, nech to tam dlho nestraší.

Listing programu:

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

int N, W, H, D;

// prečíslovanie
set<int> ypsilony;
map<int,int> skomprimuj;

// zametacie udalosti
struct event {
    int x, y1, y2, hodnota;
    event(int _x, int _y1, int _y2, int _hodnota) {
        x = _x; y1 = _y1; y2 = _y2; hodnota = _hodnota;
    }
};

bool operator < (const event& a, const event& b) {
    return a.x < b.x;
}

vector<event> eventy;

// štruktúra stromu:
//   koreň je vrchol 1
//   potomkov majú vrcholy 1 až listov-1   (je ich dokopy listov-1)
//   listy sú vrcholy listov až listov*2-1   (je ich dokopy listov)
int listov;
#define LEFT(v)   ((v) * 2)
#define RIGHT(v)  ((v) * 2 + 1)
#define PARENT(v) ((v) / 2)

#ifdef CHCEM_TOMIHO_STROM
// súčtovo-maximový intervaláč
////////////////////////////////////////////////////////////

vector<int> sucet, maximum;

void inicializuj_strom(int size) {
    // listov == najmensia mocnina dvojky co je aspon size
    listov = 1;
    while (listov < size)
        listov *= 2;
    // vyrobíme prázdne sucet a maximum
```


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

```
    sucet.clear(); sucet.resize(listov*2);
    maximum.clear(); maximum.resize(listov*2);
}

void zmen_list(int v, int val) {
    maximum[v] = sucet[v] = sucet[v] + val;
    for (v = PARENT(v); v >= 1; v = PARENT(v)) {
        sucet[v] = sucet[LEFT(v)] + sucet[RIGHT(v)];
        maximum[v] = max(maximum[LEFT(v)], sucet[LEFT(v)] +
maximum[RIGHT(v)]);
    }
}

void zmen_interval(int begin, int end, int val) {
    zmen_list(listov + begin, val);
    zmen_list(listov + end, -val);
}

int maximum_stromu() {
    return maximum[1];
}

#else
// lazy-loading maximový intervaláč
//////////////////////////////////////////////////////////

vector<int> maximum, lazy;
vector<int> lavy_okraj, pravy_okraj;

void inicializuj_strom(int size) {
    // listov == najmensia mocninca dvojky co je aspon size
    listov = 1;
    while (listov < size)
        listov *= 2;
    // vyrobíme prázdne maximum, lazy, lavy_okraj a pravy_okraj
    maximum.clear(); maximum.resize(listov * 2);
    lazy.clear(); lazy.resize(listov * 2);
    lavy_okraj.clear(); lavy_okraj.resize(listov * 2);
    pravy_okraj.clear(); pravy_okraj.resize(listov * 2);
    // ľavý a pravý okraj == kam až siahajú moji všetci potomkovia
    // najprv to vyrátam pre listy (tam je to vždy iba ten list samotný)
    for (int i = 0; i < listov; i++) {
        lavy_okraj[listov + i] = i;
        pravy_okraj[listov + i] = i + 1;
    }
    // a potom zdola nahor pre všetky ostatné vrcholy
    for (int i = listov-1; i >= 1; i--) {
        lavy_okraj[i] = lavy_okraj[LEFT(i)];
        pravy_okraj[i] = pravy_okraj[RIGHT(i)];
    }
}

void zmen_interval(int begin, int end, int val, int ja = 1) {
    if (begin >= pravy_okraj[ja] || end <= lavy_okraj[ja]) {
        // ak som ja úplne mimo intervalu, nič nerob
    } else if (begin <= lavy_okraj[ja] && end >= pravy_okraj[ja]) {
        // ak som úplne vnútri intervalu, zapíš to do lazy
        lazy[ja] += val;
    } else {
        // inak svoj lazy konečne daj potomkom a zavolaj sa na nich
```


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

```
        lazy[LEFT(ja)] += lazy[ja];
        maximum[LEFT(ja)] += lazy[ja];
        lazy[RIGHT(ja)] += lazy[ja];
        maximum[RIGHT(ja)] += lazy[ja];
        lazy[ja] = 0;
        zmen_interval(begin, end, val, LEFT(ja));
        zmen_interval(begin, end, val, RIGHT(ja));
    }
    // nakoniec updatnem svoje maximum. musím pripočítať aj lazy[ja], lebo
    // "lazy" znamená "tvárme sa že všetci podo mnou sú o toľkoto väčší" a preto
    // sa to ráta aj do môjho maxima.
    // (maximum svojich potomkov pridám, iba ak nejakých mám.)
    maximum[ja] = lazy[ja] +
        (ja >= listov ? 0 : max(maximum[LEFT(ja)], maximum[RIGHT(ja)]));
}

int maximum_stromu() {
    return maximum[1];
}

#endif
//////////////////////////////////////////////////////////////

void pridaj_stvorec(int stredx, int stredy, int velkost, int hodnota) {
    // všetky intervaly sú tu polootvorené, takže na všetky konce pridám +1.
    int l = max(1, stredx - velkost), r = min(W + 1, stredx + velkost + 1);
    int u = max(1, stredy - velkost), d = min(H + 1, stredy + velkost + 1);
    eventy.push_back(event(l, u, d, hodnota));
    eventy.push_back(event(r, u, d, -hodnota));
    ypsilony.insert(u);
    ypsilony.insert(d);
}

int main() {
    int i;
    cin >> H >> W >> N >> D;
    for (i = 0; i < N; i++) {
        int x, y;
        cin >> y >> x;
        pridaj_stvorec(x, y, D / 2, 1);
        pridaj_stvorec(x, y, 0, -2 * N);   // -nekonecno
    }
    // zaradom všetkým ypsilónom, aké sme videli, priradíme nové hodnoty
    i = 0;
    for (typeof(ypsilony.begin()) it = ypsilony.begin(); it != ypsilony.end(); it++) {
        skomprimuj[*it] = i;
        i++;
    }
    inicializuj_strom(ypsilony.size());
    // zoradíme zametané udalosti - je ich rádovo N (konkrétne 4N)
    sort(eventy.begin(), eventy.end());
    // spracujeme eventy a hľadáme najväčšieho chudáka
    int best = 0;
    for (i = 0; i < eventy.size(); i++) {
        event e = eventy[i];
        zmen_interval(skomprimuj[e.y1], skomprimuj[e.y2], e.hodnota);
        // najprv spracujeme všetky eventy s rovnakou x-súradnicou, a až potom
        // hľadáme najväčších chudákov, aby nám to nepokazil nejaký medzistav.
        if (i != eventy.size() - 1 && eventy[i].x == eventy[i + 1].x)
            continue;   // tohto preskočíme
```


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

```
        best = max(best, maximum_stromu());
    }
    cout << best << endl;
}
```


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