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

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

# Vzorové riešenia 1. kola letnej časti

## 1. Zablatená cesta

opravovala Dominika
(max. 10 bodov)

Keďže toto je prvý príklad, nebol veľmi ťažký. Stačí mať základné poznatky z fyziky a vedieť upravovať lineárne rovnice. Vychádzame zo základného vzorca pre čas, rýchlosť a dráhu $t = s/v$.

Viacerí z vás upravovali rovnicu rôzne. Ja ukážem aspoň jeden spôsob, ako sa to dalo riešiť. Celkový čas sa rovná súčtu časov na jednotlivých úsekoch ($T = t_1 + t_2 + \cdots + t_N$) a Petržlenova rýchlosť je prenásobená príslušným koeficientom. Dostávame nasledujúci vzorec:

$$T = \frac{D}{v \cdot (1 - \frac{k_1}{4+k_1})} + \frac{D}{v \cdot (1 - \frac{k_2}{4+k_2})} + \cdots + \frac{D}{v \cdot (1 - \frac{k_N}{4+k_N})}$$

Keďže našou úlohou je vyrátať rýchlosť $v$, upravíme menovateľ každého člena:

$$T = \frac{D}{v \cdot \frac{4}{4+k_1}} + \frac{D}{v \cdot \frac{4}{4+k_2}} + \cdots + \frac{D}{v \cdot \frac{4}{4+k_N}}$$

Presunieme ho do čitateľa:

$$T = \frac{D \cdot \frac{4+k_1}{4}}{v} + \frac{D \cdot \frac{4+k_2}{4}}{v} + \cdots + \frac{D \cdot \frac{4+k_N}{4}}{v}$$

Vyjmeme $D/v$:

$$T = \frac{D}{v}\left(\frac{4+k_1}{4} + \frac{4+k_2}{4} + \cdots + \frac{4+k_N}{4}\right) = \frac{D}{v}\left(N + \frac{k_1 + k_2 + \cdots + k_N}{4}\right)$$

Teraz už stačí len vymeniť premenné $T$ a $v$ (vynásobiť rovnicu $v$ a vydeliť $T$):

$$v = \frac{D}{T}\left(N + \frac{k_1 + k_2 + \cdots + k_N}{4}\right)$$

Takto sme dostali vzorec na výpočet celkovej rýchlosti, ktorou má Petržlen jazdiť, aby mu to na danej dráhe trvalo presne čas $T$. Preto stačí pole koeficientov $k_i$ lineárne prejsť v čase $O(N)$ a rátať pri tom náš vzorec. Dokonca to môžeme robiť aj bez poľa, rovno pri načítavaní vstupu, a preto nám stačí konštantná pamäť $O(1)$.


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

### Listing programu:

```
program zablatena_cesta;
var T, N, D, S, k, i: integer;
    v: real;

begin
  ReadLn(T, N, D);
  S := 0;
  for i := 1 to N do begin
    Read(k);
    S := S + k;
  end;
  v := D * (N + S/4) / T;
  WriteLn(v:0:6);
end.
```

opravoval JMK
(max. 10 bodov)

## 2. Zmrzlináreň

Toto bol vcelku jednoduchý príklad. Vystačíme si pri ňom s jedným prechodom vstupu a s konštantnou pamäťou. Ako ste si mnohí všimli, zmrzliny sa dajú rozdeliť do štyroch skupín podľa drsnosti. Pri prechode budeme počítať počet zmrzlín v každej skupine a počet zatočení žalúdkom. Tento počet zvýšime pri každej novej zmrzline o súčet počtov zjedených zmrzlín z menej drsných skupín. Čas bude teda $O(N)$ a pamäť $O(1)$.

V našom riešení vlastne len simulujeme žalúdok, pričom nám stačí vedieť počet menej drsných zmrzlín ako je tá, ktorú práve jeme. Keďže vstup musíme načítať celý, čas algoritmu je optimálny. Za optimálne riešenie sa udeľovalo 10 bodov. Ak ste mali lineárnu pamäť alebo čas $O(N^2)$, dostali ste 7 bodov. Za chýbajúce odhady sme stŕhali 1 bod a za iné nedostatky podľa uváženia.

### Listing programu:

```
program zmrzlinaren;
const zmrzlina_meno: array [1..3] of string = ('karamelovo-muchotravkova',
'slivkovo-rybacia', 'puncovo-pastetova');
var zmrzlina_pocet: array [1..4] of longint;
    j, k, i, pocet_otoceni, N: longint;
    zmrzlina: string;

begin
  ReadLn(N);
  pocet_otoceni := 0;
  for i := 1 to 4 do
    zmrzlina_pocet[i] := 0;
  for j := 1 to N do begin
    ReadLn(zmrzlina);
    k := 1;
    while (k < 4) and (zmrzlina_meno[k] <> zmrzlina) do
      Inc(k);
    Inc(zmrzlina_pocet[k]);
    for i := k + 1 to 4 do
      pocet_otoceni := pocet_otoceni + zmrzlina_pocet[i];
  end;
  WriteLn(pocet_otoceni);
end.
```



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

vzorák písal kuko, opravoval U$Ama

# 3. Zvukové bariéry

(max. 10 bodov)

Samozrejme, brutálne a priamočiare riešenie je zobrať bitmapu – pole $V \times D$ a postupne do nej zakresľovať jednotlivé bloky. Na záver skontrolujeme, či sme vyfarbili všetky políčka. Nevýhody má toto riešenie aspoň dve:

1. na celý múr potrebujeme veľa pamäte,
2. vždy, keď chceme pridať ďalší blok, musíme hľadať výšku, kam patrí (čo je pomalé).

Druhý problém sa dá ľahko vyriešiť tak, že si pre každý stĺpec zapamätáme jeho výšku. Ak si však predstavíme, akým spôsobom sa bloky ukladajú zhora nadol, uvedomíme si, že tieto výšky sú už všetko, čo na popis múru (v každej fáze jeho výstavby) stačí: Aby sa múr dal dostavať, nesmú v ňom byť žiadne diery, ba ani záhyby – každý stĺpec musí pozostávať iba z jedného úseku, ktorý začína pri zemi. (Týmto by bol vyriešený aj prvý problém.)

Navyše rovnaká úvaha platí aj pre bloky: bloky nesmú mať diery ani záhyby – každý stĺpec sa musí skladať z jedného súvislého úseku. Preto si aj jednotlivé bloky môžeme pamätať jednoducho tak, že si pre každý stĺpec zapamätáme, kde začínajú `X`-ká a koľko ich v danom stĺpci je.

Uloženie bloku je jednoduché: blok musí na múr padnúť, prepytujem, ako riť na šerbeľ. Inými slovami reliéf nedostavaného múru sa musí zhodovať so spodkom bloku, alebo ešte inak: rozdiely výšok dvoch susedných stĺpcov a rozdiely výšok, v ktorých začínajú `X`-ká, sa musia zhodovať. Ak sa zhodujú, patrične zvýšime výšky. Na konci už len skontrolujeme, či má každý stĺpec múru rovnakú výšku, a to $V$.

Na rozdiel od prvého riešenia, čas potrebný na pridanie jedného bloku nie je úmerný veľkosti bloku, ale iba jeho šírke. Podobne čas kontroly, či je vyplnený celý múr, je úmerný šírke múru, nie jeho veľkosti $V \times D$. No a napokon potrebná pamäť je úmerná iba šírke blokov a múru, nie ich veľkostiam. (Super, nie?)

Pár slov k bodovaniu a vašim riešeniam. Keďže v tomto príklade nebola skrytá nejaká prevratná myšlienka, tak som očakával, že vaše programy budú bez problémov fungovať. Ale občas to tak nebolo. Preto predtým, než niečo do KSP pošlete, skúste si to na pár vstupoch otestovať. Určite tým nič nepokazíte a aspoň odchytíte pár základných chýb.

## Listing programu:

```
const MAXT = 20;
      MAXB = 50;
      MAXD = 10000;
      INFTY = maxint;
var D, V, T, N, i, j, k, w, h, p, r : integer;
    mur : array [1..MAXD] of integer;                    { vysky muru }
    sirka : array [1..MAXT] of integer;                  { sirky blokov }
    dol, vyska : array [1..MAXT, 0..MAXB] of integer; { relief spodku a vysky }
    s : array [0..MAXB] of string;

procedure die;
begin writeln ('NIE'); halt; end;

begin
  readln (D, V, T, N);
  for i := 1 to T do begin
    readln (w, h);
    for j := 0 to h-1 do
      readln (s[j]);
    sirka[i] := w;
```



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

```
    for j := 1 to w do begin
      k := 1;
      { predpokladam, ze ziadny stlpec nie je prazdny }
      while (s[h-k][j] = '-') do inc(k);
      dol[i,j-1] := k;
      while (k <= h) and (s[h-k][j] = 'X') do inc(k);
      vyska[i,j-1] := k - dol[i,j-1];
      while (k <= h) and (s[h-k][j] = '-') do inc(k);
      if k <= h then dol[i,1] := INFTY; { v bloku je diera; zabezpecime, aby sa
nedala pouzit }
    end;
  end;

  for i := 1 to N do begin
    readln (p, r);
    if (p < 0) or (p+sirka[r]-1 > D) then die;                    { vedla }
    for j := 1 to sirka[r]-1 do
      if mur[p+j]-mur[p+j-1] <> dol[r,j]-dol[r,j-1] then die; { vytvori sa diera }
    for j := 0 to sirka[r]-1 do
      inc (mur[p+j], vyska[r,j]); { upravime vysku muru }
  end;

  for i := 1 to D do
    if mur[i] <> V then die;

  writeln ('ANO');
end.
```

opravovalo sa samo, vzorák Tommy

## 4. Zaplať a leť

(max. 15 bodov)

Čo si bolo treba dobre všimnúť, je veta „lety vedú len na niektoré z ďalších letísk“ (t. j. na letiská s vyšším číslom). Rovnako ani zotriedenie vstupu nebolo vôbec náhodné. Pokiaľ totiž nejakým spôsobom spracujeme lety z prvých $i$ letísk, tak už určite vieme, či (a za koľko) sa vieme dostať na letisko $i+1$ (pretože všetky lety na letisko $i+1$ vedú z letísk 1 až $i$). Vstup – veľmi vhodne zotriedený – teda stačí spracovávať v poradí, v akom ho dostaneme.

Na začiatku stojíme na letisku 1 a ešte sme spoločnosti LAMA nemuseli nič zaplatiť. Na letisko 1 sa vieme dostať za sumu 0 (už tam sme) a na všetky zvyšné vôbec – budeme to reprezentovať nekonečnom. Ak existuje let z letiska 1 na letisko $b$ za cenu $c$, na letisko $b$ sa vieme dostať za sumu $c$. Keď si takto zapíšeme všetky lety, môžeme začať spracovávať lety z ďalšieho letiska – 2, 3, atď.

Ak stojíme na letisku $a$, na ktoré sme sa vedeli dostať najlacnejšie za sumu $S_a$, a existuje let z letiska $a$ na letisko $b$ za cenu $c$, tak s jeho využitím sa tam vieme dostať za sumu $S'_b = S_a + c$. Ak je táto suma menšia ako doteraz najlepšia ($S_b$), prepíšeme ju. To nám zaručí, že $S_b$ je najmenšia doteraz známa[1] suma, za ktorú sa vieme dostať na letisko $b$. Keď teda spracujeme všetky lety vedúce na letisko $b$, už vieme, za koľko sa na neho dá najlacnejšie dostať. Na konci nám už len stačí vypísať sumu $S_N$ (alebo `neexistuje`, ak $S_N = \infty$).

Potrebujeme si pamätať len sumy pre všetky letiská (lety spracovávame rovno, nemusíme si ich pamätať), teda pamäťová zložitosť je $O(N)$. Musíme tiež spracovať všetkých $K$ letov a nastaviť, že do všetkých letísk okrem prvého sa vieme dostať za sumu $\infty$, t. j. nevieme; časová zložitosť je potom $O(N+K)$.

[1] t. j. pomocou doteraz známych (spracovaných) letov



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

### Listing programu:

```
program zaplat_a_let;
const nekonecno = 1234567890;
var N, K, i, odkial, kam, cena: longint;
    D: array [1..1000000] of longint;

begin
  ReadLn(N, K);
  for i := 1 to N do
    D[i] := nekonecno;
  D[1] := 0;

  for i := 1 to K do begin
    ReadLn(odkial, kam, cena);
    if D[kam] > D[odkial] + cena then
      D[kam] := D[odkial] + cena;
  end;

  if(D[N] < nekonecno) then WriteLn(D[N])
  else WriteLn('neexistuje');
end.
```

opravoval U$Ama
(max. 15 bodov)

## 5. Opravovacia lotéria vracia úder

Tento príklad vyzeral na prvý pohľad strašne odstrašujúco. Takže základ úspechu bolo nezľaknúť sa a pomaly a isto rozbiť ten hnusný kód.

**Funkcia `countpoints`**

Pozrime sa na výpis výsledného počtu bodov. Chceme, aby funkcia `countpoints` vrátila čo najväčšie číslo. Za každú rovnosť pri porovnaní sa hodnota, ktorú vráti, zväčší o 1. Takže chceme týchto úspešných porovnaní čo najviac. Všimnime si polia `sbox1` a `sbox2`. Podmienka `sbox1[`$i$`]==sbox2[`$i$`]` platí len pre $i = 22$, čo predstavuje znak `w`. Takže na vstup `countpoints` chceme poslať 32 znakov `w`.

**Riešenie, ktoré beží asi 4 dni**

Máme vstup, ktorý chceme poslať funkcii `countpoints`. Stačí invertovať funkciu `round` (teda spraviť si funkciu, ktorá dostane požadovaný výstup funkcie `round` a povie, čo jej poslať na vstup) a pustiť ju patričný počet krát. Toto bude bežný počítač rátať asi 4 dni. Pokiaľ chcete vidieť rýchlejšie riešenie, čítajte ďalej.

**Funkcia `round` (v pascalovskom zdrojáku `kolo`)**

Občas nie je zlé si nakresliť, čo niektoré čudo robí. A navyše sa nám hodí zjednodušená notácia. Pod označením $S_1(x)$ budeme rozumieť `'a'+sbox1[`$x$`-'a']`. Uvedomte si, že $S_1$ je skoro to isté ako `sbox1`, až na to, že `sbox1` beží na číslach $0, 1, \ldots, 25$ a $S_1$ na písmenách `a`, `b`, ..., `z`. To isté platí aj pre $S_2(x)$.

Zoberme si teraz `in[0]` a `in[16]`. `in[0]` prežením cez $S_2$ a pošleme ho na `out[16]`. Ešte sčítame hodnoty `in[0]`, `in[16]` a `key[0]`, spravíme tomu zvyšok po delení 26, prežením cez $S_1$ a pošleme ako `out[0]`. Podobne to spravíme aj pre ostatné znaky vstupu. Podstatné je, že tieto dvojice znakov sa počítajú nezávisle od ostatných. Toto počítanie označme ako $f((\texttt{in[}i\texttt{]},\texttt{in[}i+16\texttt{]}),\texttt{key[}i\texttt{]}) = (\texttt{out[}i\texttt{]},\texttt{out[}i+16\texttt{]})$.

Spravme si teraz inverznú transformáciu. Najprv by bolo dobré invertovať $S_1$ a $S_2$. Toto je jednoduchá robota, popisovať ju tu nebudeme. Inverzie označme $S_1^{-1}$, $S_2^{-1}$. $S_1^{-1}(x)$ nám hovorí, čo máme poslať na vstup $S_1$, aby výstup bol $x$, formálne $S_1(S_1^{-1}(x)) = x$. Teraz môžeme písať inverznú transformáciu. Využijeme už nájdenú nezávislosť dvojíc.



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

Poznáme `out[`$i$`]`, `out[`$i+16$`]`, `key[`$i$`]`, chceme zistiť `in[`$i$`]` a `in[`$i+16$`]`. `in[`$i$`]` nájdeme ľahko, totiž `out[`$i+16$`]` $= S_2($`in[`$i$`]`$)$ a teda `in[`$i$`]`$= S_2^{-1}($`out[`$i+16$`]`$)$. Ďalej vieme `out[i]`$=$ $S_1($`in[`$i$`]`$+$`in[`$i+16$`]`$+$`key[`$i$`]`$)$, čiže $S_1^{-1}($`out[`$i$`]`$) =$`in[`$i$`]`$+$`in[`$i+16$`]`$+$`key[`$i$`]`, `in[`$i$`]` sme už spočítali a teda `in[`$i+16$`]`$= S_1^{-1}($`out[`$i$`]`$)-$`in[`$i$`]`$-$`key[`$i$`]`.

Celé toto označme jednoducho ako $inv($(`out[`$i$`]`,`out[`$i+16$`]`)`key[`$i$`]`$) =$ (`in[`$i$`]`,`in[`$i+16$`]`). Nech $a_{0,i} =$ (`w`,`w`), pre $i = 0, 1, \ldots, 15$. A nech $a_{j,i} = inv(a_{j-1,i}$,`key[`$i$`]`$)$. Potom vstup, ktorý máme zadať, zistíme z hodnôt $a_{k,i}$, kde $k =$ `NUMROUNDS` a $i = 0, 1, \ldots 15$ (samozrejme treba znaky umiestniť na správne miesta). Toto ale stále vyžaduje dlhé rátanie.

**Cykly**

Všimnime si, že dvojice $(x, x)$, kde $x$ je nejaký znak, môžu nadobúdať $26 \cdot 26 = 676$ rôznych hodnôt. Takže keď si vezmeme napr. postupnosť $a_{0,0}, a_{1,0}, a_{2,0}, \ldots$, tak sa po istom čase (najneskôr po 676 krokov) musíme dostať k dvojici znakov, ktorú sme už mali. Uvedomme si, že akonáhle $a_{j,i} = a_{k,i}$, tak aj $a_{j+1,i} = a_{k+1,i}$, atď.

Takže už len stačí nájsť cyklus. Dobrou správou je, že stačí nájsť prvé $a_{j,i}$ také, že $a_{0,i} = a_{j,i}$ a nemusíme sa namáhať metódou dvoch bežcov a inými zverinami. Je to totiž preto, že nemôže nastať napríklad takýto prípad: (`w`,`w`); (`c`,`d`); (`q`,`z`); (`e`,`f`); (`a`,`m`); (`q`,`z`); (`e`,`f`); (`a`,`m`); ... (teda máme najprv nejaký „chvost“ a potom sa cyklíme dookola). Problém je v tom, že hodnota $a_{j,i}$ okrem toho, že jednoznačne určuje hodnoty $a_{j+1,i}, a_{j+2,i}, \ldots$, určuje jednoznačne aj hodnoty $a_{j-1,i}, \ldots$, totiž $a_{j-1,i} = f(a_{j,i}$,`key[`$i$`]`$)$. V tomto príklade dvojici (`q`,`z`) predchádzajú 2 rôzne dvojice, čo je spor. A teda náš cyklus musí obsahovať aj štartovaciu hodnotu.

Takže stačí spočítať dĺžku cyklu pre všetky $i = 0, 1, \ldots, 15$. Označme ich $c_i$. Následne vieme, že $a_{k,i} = a_{k \bmod c_i,i}$, čo už zakončuje náš výpočet.

Výsledok bude: `unonecxsmomecnmmzesegftpexegfwel`.

**Bodovanie**

Okrem bodov, čo vám dala lotéria, sa dali získať nejaké bonusové body za rôzne drobnosti, ako je analýza funkcie `countpoints`, napísanie inverzného programu, atď.

## Listing programu:

```
#include <cstdio>

int inv1[] = {14,7,13,10,0,20,8,22,19,9,1,2,23,18,24,3,5,6,17,12,25,21,11,15,4,16};
int inv2[] = {1,25,0,7,6,23,13,22,21,8,17,4,19,3,15,11,14,5,9,24,2,20,16,10,12,18};

char key[] = "bazavektorovehopriestoru";

#define NUM_ROUNDS 2345678901231l

void backround(char a, char b, int k, char &oa, char &ob){
    oa = 'a'+inv2[b-'a'];
    int temp = inv1[a-'a'] - (oa-'a') - (key[k]-'a');
    while(temp < 0)
        temp += 26;
    ob = 'a'+temp;
}

void get(char a, char b, int k, char &oa, char &ob){
    char ca = a, cb = b;
    int len = 0;
    do{
        backround(ca, cb, k, ca, cb);
        len++;
    }while(ca != a || cb != b);

    int r = NUM_ROUNDS % len;
```



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

```
    oa = a, ob = b;
    for(int i = 0; i < r; i++)
        backround(oa, ob, k, oa, ob);
}

int  main(){
    char target[] = "wwwwwwwwwwwwwwwwwwwwwwwwwwwwww";
    char source[33];
    source[32] = 0;

    for(int i = 0; i < 16; i++)
        get(target[i], target[i+16], i, source[i], source[i+16]);

    printf("%s\n", source);
}
```

opravoval PPershing
(max. 20 bodov)

# 6. O kráľových jedoch

Poďme sa pustiť do tohto jedovatého problému ešte skôr, než kráľovi Baltazárovi dôjde trpezlivosť (a nám hlava). Máme nájsť niekoľko (pomerne málo) otrávených vín vo veľkom počte fliaš. Odpoveď na niektoré špeciálne prípady už vieme. Napríklad pre $k = 0$ je riešenie triviálne. Trochu horšie je to s $k = 1$. V tomto prípade ide o známe binárne vyhľadávanie – môžeme si to predstaviť ako hádanie čísla na intervale, kde každou ďalšou otázkou vieme interval rozdeliť na 2 časti, na jednu z nich sa spýtať a z toho zistiť, v ktorej polovici intervalu sa číslo nachádza. Prvou otázkou sa teda spýtame na polovicu fliaš, druhou otázkou sa spýtame na polovicu tej správnej polovice z minulého kola, ..., $n$-tou otázkou sa spýtame na $1/2^n$-tinu pôvodného počtu fliaš. Celkový počet otázok bude $\lceil \log_2 n \rceil$.

No fajn, to by sme mali pre $k = 1$. Čo by sme mohli spraviť pre $k > 1$? Keďže sa nám binárne vyhľadávanie pomerne oplatilo v predošlom prípade, mohli by sme ho nejakým spôsobom zovšeobecniť. Respektíve spraviť niekoľko binárnych vyhľadávaní.

Inu, na tento prístup sa môžeme pozrieť opäť rekurzívne – zoberme si náš interval a (tradične) ho rozdeľme na polovice. Následne sa opýtame na obe polovice, či obsahujú nejaký jed.[2] V prípade, že jedna (alebo obe) polovice obsahujú jed, zavoláme sa rekurzívne. Myšlienka jednoduchá, zložitosť neistá. Teda, až do doby, než sa nad ňou poriadne zamyslíme.

Predstavme si, že si našu rekurziu zakreslíme do stromu, ako sa vetvila. Ak by sme sa v každom kroku vetvili $2\times$ (napríklad, ak by platilo $n = k$, teda všetky vína by boli otrávené[3]), potom by náš strom mal na spodnej úrovni $n$ vrcholov a jeho hĺbka by bola $\log n$. Teraz si predstavme, že v tomto strome na konci vyznačíme $k$ jedov červenou farbou a tiež si tou farbou označíme každý vrchol, ktorého interval obsahuje nejaký jed. Môžete si rozmyslieť, že červenou farbou vyznačíme práve cestu od koreňa ku každému jedu. Spolu teda najviac $k \log n$ vrcholov. Označme si teraz ešte modré vrcholy – to budú práve tie, ktoré sú susedom červeného vrcholu. Týchto bude preto evidentne tiež (najviac) $k \log n$. Nuže, a teraz tvrdíme, že červené a modré vrcholy zodpovedajú práve otázkam položeným v našom rekurzívnom prehľadávaní – červené (až na koreň, na ktorý sa prirodzene pýtať nemusíme) sú otázky na polintervaly, pri ktorých sluha zomrel. Naopak, modré sú tie položené otázky, pri ktorých sluha prežil (a teda sme sa prestali rekurzívne volať).

[2] Je dôležité si uvedomiť, že na rozdiel od klasického binárneho vyhľadávania sa musíme pýtať na obe polovice, pretože z jednej otázky nevieme povedať nič o druhej polovici – môže a nemusí obsahovať ďalší jed.

[3] to, že je to v spore s predpokladom úlohy, neuvažujeme



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

Vidíme teda, že v najhoršom položíme $2k \log n$ otázok. To však zďaleka neznamená, že náš program bude mať danú časovú zložitosť – v zadaní sa totiž písalo, že treba vypísať všetky vína, ktoré chceme pri otázke zamiešať. No a to nie je to isté ako vypísať interval!

Aká je teda časová zložitosť tohoto postupu? Snáď ešte ťažšia otázka ako minule. Celkový čas si ale môžeme fikane rozdeliť na dve časti, megalomansky ich označíme ako prvú a druhú časť. Prvá časť bude zahrňovať prvých $\log k$ vnorení rekurzie, druhá časť bude zvyšných $\log n - \log k$.

Aký je horný odhad na čas prvej časti? Môžete si rozmyslieť, že je to $n \log k$ – na každej úrovni rekurzie vypíšeme najviac $n$ vín, ak sa postupne budeme pýtať na všetky malé intervaly zaradom.

Druhá časť bude zaujímavejšia – horný odhad na počet vypísaných vín bude najviac $k$-násobok horného odhadu na počet vypísaných vín v prípade, že je práve jedno „vylepšené“ nepriateľom. To je evidentné napríklad tak, že si našu rekurziu rozdelíme na $k$ ciest od listov ku koreňu a tieto sú v najhoršom prípade disjunktné (občas sa dokonca môže stať, že si polepšíme, ak sa spoja).

Jedinou nevyriešenou otázkou teda ostáva, čo v prípade jedného jedu. V tomto prípade je počet otázok $p = \sum_{i=\log k}^{\log n} n/2^i$. Inak zapísané $p = n/2^{\log k} \cdot (1+1/2+1/4+\ldots+1/2^{\log n - \log k})$. Toto môžeme odhadnúť zhora ako $p < n/k \cdot (1+1/2+1/4+\cdots) = 2n/k$. Výsledný čas druhej fázy je potom $k \cdot p = O(n)$. Spolu pre obe časti to dokopy dáva $O(n \log k)$.[4]

Zabudli sme na pamäťovú zložitosť, ale tá je jednoduchá – náš rekurzívny program si v každom kroku rekurzie pamätá iba konštantne veľa informácie, hĺbka rekurzie je $\log k$ a teda aj pamäť je $O(\log k)$.

No a posledná (a z nášho pohľadu najzaujímavejšia otázka) – aké je teoretické minimum počtu otázok? Je náš program dosť dobrý, alebo sme krvopotne vymysleli úplne mizerné riešenie? Začnime jednoducho. Položme si otázku, koľko možností vieme rozlíšiť na $0, 1, 2, 3, \ldots, x$ otázok? Keďže každá odpoveď na otázku je buď áno alebo nie, môžeme ňou rozlíšiť maximálne 2 prípady. Preto 2 otázkami vieme rozlíšiť 4 prípady, 3 otázkami najviac 8, a $k$-timi otázkami najviac $2^k$ možností. Opačným smerom – ak máme $t$ možností, potrebujeme aspoň $\lceil \log t \rceil$ otázok.

A sme doma – koľko možností existuje pre $k$ jedov z $n$ vín? Možno vám to niečo pripomína (a ak nie, tak šup zopakovať stredoškolskú kombinatoriku), ale je to počet kombinácií, čiže $\binom{n}{k}$. Počet otázok je teda aspoň $\log \binom{n}{k}$. No a odtiaľto začína nepovinný výlet do matematického upravovania:

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

Teraz využijeme znalosť, že pre veľké $n$ je hodnota $\log n!$ približne[5] $(n+1/2)\ln n - n + O(1)$. Dosadením dostávame

$$\begin{aligned}
\log \binom{n}{k} =& (n+1/2)\ln n - n - (k+1/2)\ln k + k \\
& - (n-k+1/2)\ln(n-k) + (n-k) - O(1) \\
>& (n+1/2)\ln n - (k+1/2)\ln k - (n-k+1/2)\ln n - O(1) \\
=& k \ln n - (k+1/2)\ln k - O(1) \\
\approx& k \ln n
\end{aligned}$$

Odpoveď teda je, že náš program je optimálny v asymptotickom zmysle.

[4] Môžeme si všimnúť, že v prípade, ak sú jedy distribuované rovnomerne, pri prvých $\log k$ krokoch naozaj nazbierame aspoň čas $n \log k$ a teda je to aj dolné ohraničenie (v najhoršom prípade).

[5] pozri `http://mathworld.wolfram.com/StirlingsApproximation.html`



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

Záver je teda ten, že sme našli veľmi jednoduchý program, ktorý splní kráľove požiadavky – je optimálny v asymptotickom zmysle; počet otázok, ktoré položí je $O(k \log n)$; časová zložitosť (počet zmiešaných vín) je $O(n \log k)$ a pamäťová zložitosť je $O(\log k)$.[6]

Ešte k bodovaniu – väčšina z vás prišla na správne a jednoduché riešenie. Body sa teda strhávali za drobné nezrovnalosti – za neprimerane veľkú pamäť ($O(n)$) ste si mohli obdržať $-1$ bod. Takisto, za zlý odhad počtu sluhov, času či pamäte ste si mohli prisúdiť $-1$ bod za každé. Chýbajúci program bol odmeňovaný približne $-3$ bodmi.

## Listing programu:

```
#include <cstdio>
#include <cstring>

// nasiel som jed. V pripade, ze chceme vypisovat iba nakoniec, iba si
// tu zapamatame jeho cislo
void vypis_jed(int i){
    printf("Vino %d je jedovate\n", i);
}

// spyta sa na to, ci interval <l,r) obsahuje jedy a vrati true ak ano
int query_interval(int l, int r){
    printf("Query: pocet %d, vina:", r - l);
    for(int i = l; i < r; i++) printf(" %d", i);
    printf("\n");

    char tmp[10];
    scanf("%s", tmp);
    return (strcmp(tmp, "zomrel") == 0);
}

// najdi vsetky jedy na intervale <l, r)
// predpokladame, ze je tam aspon 1
void najdi_jedy(int l, int r){
    if(l + 1 == r){
        vypis_jed(l);
        return;
    }
    int m = (l + r) / 2;

    if(query_interval(l, m)){
        najdi_jedy(l, m);
        if(query_interval(m, r)) najdi_jedy(m, r);
    }else{ // v pripade ze na prvom intervale neni jed, mozeme druhu
           // otazku rovno preskocit lebo vieme ze odpoved je kladna
        najdi_jedy(m, r);
    }
}

int main(){
    int n, k;
    scanf("%d %d", &n, &k);
    // ak nemame jedy, nema zmysel sa pytat
    if(k > 0)
        najdi_jedy(0, n);
}
```

[6] V prípade, že potrebujeme vypísať výsledok na konci, sa pamäť prirodzene zmení na $O(k)$.



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

# 7. O godzillích dostihoch

opravoval Mic
(max. 20 bodov)

Veľmi ste ma nepotešili, lebo prišlo málo riešení. Dosť z vás malo síce správnu myšlienku, ale len zopár si uvedomilo, ako jednoducho zlepšiť časovú aj pamäťovú zložitosť.

**Riešenie v $O(n^2)$**

Godzillie dostihy je konečná, symetrická hra s úplnou informáciou. Čo to znamená? Konečná znamená, že vždy skončí.[7] Symetrickosť hry sa prejaví tak, že keby sme vymenili toho, kto je na ťahu, tak by mal na výber z rovnakých ťahov (šach nie je symetrická hra, lebo v šachu každý hráč ťahá len figúrkami svojej farby). Hry s úplnou informáciou sú také, v ktorých každý hráč vidí všetko (napríklad poker nie je hra s úplnou informáciou). Tieto vlastnosti využijeme na nájdenie víťaznej stratégie.

Najskôr si formálne popíšeme túto hru. Pozíciou v hre rozumieme stav hry, čiže to, čo si musíme zapísať na papier (plus kto je na ťahu), aby sme mohli kedykoľvek v hre pokračovať. Pre Godzillie dostihy je to vzdialenosť od cieľa a aktuálna rýchlosť, preto pozície budeme označovať ako usporiadané dvojice $(n, v)$, kde $n$ je vzdialenosť do cieľa a $v$ je aktuálna rýchlosť. Ťahom sa rozumie zmena nejakej pozície na inú pozíciu. V prípade Godzillích dostihov sú prípustné ťahy z pozície $(n, v)$ len do $(n-v+1, v-1), (n-v, v), (n-v-1, v+1)$, ak $v > 1$. Ak $v = 0$, tak vieme ťahať len do pozície $(n-1, 1)$ a ak $v = 1$, tak nevieme ťahať do pozície $(n, 0)$. Zároveň z ľubovoľnej pozície $(n, v)$, kde $n \leq 0$ nevieme ťahať do žiadnej pozície. Tieto pozície budeme označovať ako konečné – kto sa do nich dostal, prehral, lebo nemá ťah.

Pozíciu nazveme prehrávajúcou ak platí, že nech by sme potiahli ľubovoľne, súper hrajúci optimálne nás v konečnom čase určite porazí. Vyhrávajúca pozícia je taká, z ktorej vieme vyhrať, nech by súper hral akokoľvek. Budeme postupovať nasledovne. Pozície, z ktorých už nevieme ťahať, označíme za prehrávajúce. Zoberieme si pozíciu $p = (n, v)$. Ak sa z pozície $p$ vieme dostať do pozície, ktorá je prehrávajúca, tak pozícia $p$ je vyhrávajúca (potiahneme, súper je v prehrávajúcej pozícii). Ak sa nevieme dostať do prehrávajúcej pozície, tak označíme našu pozíciu ako prehrávajúcu, lebo nech by sme ťahali akokoľvek, tak by sme súpera dostali do vyhrávajúcej pozície a ten by potiahol tak, že by sme boli zase v prehrávajúcej pozícii.

Ako bude vyzerať program? Spravíme si dvojrozmerné pole $D$ veľkosti $(n+1) \times n$. Prvý rozmer bude vzdialenosť od konca a druhý bude rýchlosť, čiže každé políčko poľa reprezentuje jednu pozíciu. Všetky pozície $p = (n, v)$, ktoré majú $n \leq 0$ označíme ako prehrávajúce (okrem pozície $(0, 0)$, lebo v nej sa dá iba začínať a vtedy sme vyhrali). Budeme postupovať po riadkoch[8] a podľa hore opísaného postupu zisťovať, či je daná pozícia vyhrávajúca alebo prehrávajúca. Keďže každým ťahom sa priblížime k cieľu, o každej pozícii, do ktorej sa vieme dostať, už máme vypočítané, či je vyhrávajúca alebo prehrávajúca. Týmto spôsobom zistíme typ pozície $(n, 0)$. Ak je vyhrávajúca, tak Mišo vyhrá. V opačnom prípade vyhrá Maják.

Časová aj pamäťová zložitosť tohto algoritmu je $O(n^2)$.

**Riešenie v čase $O(n \cdot \sqrt{n})$**

Niektorí z vás si všimli, že nie všetky pozície je možné dosiahnuť. Vždy totiž začíname na rýchlosti 0 a každým ťahom sa môže rýchlosť zvýšiť o 1. Akú najväčšiu rýchlosť vieme dosiahnuť? Ak by sme zrýchľovali neustále $k$ kôl (a dosiahneme rýchlosť $k$), tak prejdeme vzdialenosť

$$1 + 2 + 3 + \cdots + k = \sum_{i=1}^{k} i = \frac{k \cdot (k+1)}{2}$$

[7] Napríklad tenis nie je konečná hra.

[8] Riadok zodpovedá pozíciám s rovnakou vzdialenosťou od cieľa.



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

čo nesmie byť väčšie ako $n + k - 1$. To znamená, že počas hry nezrýchlime na viac ako $k = \sqrt{2n} + 1$ a preto nemá zmysel vyrátať tabuľku väčšiu ako $n \cdot k$, čím dosiahneme časovú aj pamäťovú zložitosť $O(n \cdot \sqrt{n})$.

**Bodovanie**

Za vzorové riešenie som udeľoval 20 bodov. Kvadratickým riešeniam som dával 14 bodov a za exponenciálne riešenia som dával 8 bodov. Za zlý popis som strhal 1–2 body, za nesprávne alebo chýbajúce odhady časovej či pamäťovej zložitosti som strhal po jednom bode, absencia popisu/kódu sa odmeňovala vydelením výsledného počtu bodov dvoma. Nefunkčné riešenia sa mohli tešiť maximálne jednému bodu.

A teraz si môžete pozrieť kód.

### Listing programu:

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

int main(){
    int N, M;
    cin >> N;
    N++;
    M = 2 * sqrt(N);
    vector<vector<bool> > D(N, vector<bool>(M + 1, false));
    D[0][0] = 1;

    for(int pozicia = 1; pozicia < N; pozicia++){
        for(int rychlost = 0; rychlost <= M; rychlost++){
            bool vyhravajuca = false;
            for(int novaRychlost = rychlost - 1; novaRychlost < rychlost + 2;
novaRychlost++){
                if(novaRychlost < 1 || novaRychlost > M) continue;
                if(pozicia - novaRychlost < 0){
                    vyhravajuca = true;
                    continue;
                }
                if(!D[pozicia - novaRychlost][novaRychlost]) vyhravajuca = true;
            }
            D[pozicia][rychlost] = vyhravajuca;
        }
    }

    if(D[N - 1][0])
        cout << "Vyhráva Mišo." << endl;
    else
        cout << "Vyhráva Maják." << endl;
}
```

## 8. Tragédia

vzorák písal Bob
(max. 25 bodov)

Tento príklad poskytol jedinečnú šancu všetkým zástancom nie-tak-úplne-korektných riešení a rôznych heuristík. Naozaj, (nejaké) body sa dali získať za (skoro) hocičo.

Vo vzoráku sa však budeme venovať len korektným riešeniam. Aj pre túto úlohu existuje jednoduchý backtrack s exponenciálnou časovou zložitosťou (približne za 8 bodov). Stačí



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

pomocou rekurzie vygenerovať všetky možné rozdelenia viet na scény a pre každé rozdelenie spočítať počet šliapnutí na nohu.

**Dynamika**

Ako to už býva, backtrack vedie k memoizácii a tá k riešeniu využívajúcemu dynamické programovanie. Prejdime preto rovno k nemu.

V súlade so zadaním označme $i$-te číslo danej postupnosti $h_i$ (je to číslo herca, ktorý má vyriecť $i$-tu vetu). Počet rôznych čísel v úseku $h_i, \ldots, h_j$ označme $u(i,j)$. Budeme počítať hodnoty $d(i)$ – najmenší počet šliapnutí, ktorý sa dá dosiahnuť vhodným rozdelením, ak uvažujeme iba prvých $i$ členov postupnosti. Výsledkom je teda $d(N)$.

Keď už máme vypočítané $d(1), \ldots, d(i-1)$, hodnotu $d(i)$ nájdeme tak, že vyskúšame všetky možnosti pre dĺžku posledného úseku v optimálnom rozdelení postupnosti $h_1, \ldots, h_i$. Ak by mal dĺžku $k$, potom by siahal od $h_{i-k+1}$ do $h_i$ a celkový počet šliapnutí by bol $d(i-k) + (u(i-k+1,i))^2$. Najmenšia z týchto hodnôt je práve hľadané $d(i)$.

Zostáva nám vyriešiť, ako budeme počítať $u(i,j)$. Použijeme pole `bool`ov, v ktorom si budeme pre každé číslo značiť, či už malo výskyt v prejdenom úseku postupnosti.

Na začiatku hľadania $d(i)$ pre ďalšie $i$ každú položku tohto poľa nastavíme na `false` – máme prázdny úsek. Ako postupne zväčšujeme dĺžku úseku $k$, pribúdajú nám nové čísla: $h_i, h_{i-1}, \ldots$ Ak pridané číslo ešte nemalo výskyt v tomto úseku, zväčšíme si počítadlo počtu rôznych čísel o 1 a poznačíme si jeho výskyt.

Toto riešenie má časovú zložitosť $O(N^2)$ – pre jedno $i$ vypočítame $d(i)$ v lineárnom čase. Mohlo získať okolo 18 bodov, s optimalizáciami aj viac.

**Tiež dynamika**

Keď sme pri hľadaní $d(i)$ skúšali všetky možnosti pre dĺžku posledného úseku, mohlo sa stať, že pri rôznych dĺžkach $k_1$ a $k_2$ boli hodnoty $u(i-k_1+1,i)$ a $u(i-k_2+1,i)$ rovnaké. Presnejšie, s rastúcim $k$ hodnota $u(i-k+1,i)$ neklesá a zväčší sa (o 1) práve vtedy, keď sa číslo $h_{i-k+1}$ nenachádza v úseku $h_{i-k+2}, \ldots, h_i$.

Na druhej strane, hodnota $d(i-k)$ s rastúcim $k$ nerastie. Myšlienka jednoduchého dôkazu: majme nejako rozdelenú postupnosť a odstráňme jej posledný prvok. Pri rovnakom rozdelení sa počet šliapnutí na nohu nemôže zväčšiť.

Ak teda zo všetkých $k$, pre ktoré je hodnota $u(i-k+1,i)$ rovnaká, vezmeme len to najväčšie[9], stále nájdeme optimálne rozdelenie. Preto nebudeme skúšať všetky možnosti pre dĺžku posledného úseku, ale možnosti pre počet rôznych čísel v ňom.

Označme $p(i,w)$ najväčšie také $k$, pre ktoré $u(i-k+1,i) = w$. Navyše $p(i,0) = 0$. Skúste si sami, ako by ste vypočítali $p(i,w)$, ak už poznáte hodnoty $p(i-1,w)$ a $p(i-1,w-1)$; bude sa vám ľahšie chápať nasledujúci odstavec.

Hodnotu $p(i,w)$ pre $w \geq 1$ nájdeme nasledovne. Sú dve možnosti: číslo $h_i$ sa v najdlhšom úseku končiacom $h_i$, v ktorom je práve $w$ rôznych čísel, nachádza iba raz ($p(i,w) = p(i-1,w-1)$), alebo sa v ňom nachádza viac ráz ($p(i,w) = p(i-1,w)$). Z možností si vyberieme podľa toho, či sa $h_i$ nachádza v úseku $h_{i-p(i-1,w)}, \ldots, h_{i-1}$ (to zistíme v konštantnom čase tak, že si budeme pre každé číslo pamätať jeho zatiaľ posledný výskyt).

Keď teraz prepíšeme vzťah, ktorým sme počítali $d(i)$ v predchádzajúcom riešení, dostaneme $d(i) = \min\{d(i-p(i,w)) + w^2 \mid w \geq 1\}$.

Skvelé! Skomplikovali sme si život a zatiaľ to nevyzerá o nič efektívnejšie. Keď si ale uvedomíme, že stačí skúšať hodnoty $w \leq \sqrt{N}$, získame riešenie s časovou zložitosťou $O(N\sqrt{N})$.

[9] to tiež znamená, že vezmeme najdlhší taký úsek $h_{i-k+1}, \ldots, h_i$



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

Prečo by to malo platiť? Optimálne riešenie totiž nemôže obsahovať scénu s viac ako $\sqrt{N}$ rôznymi hercami, pretože počet šliapnutí v nej by bol väčší ako $N$ a určite by existovalo lepšie delenie (napríklad venovať každej vete osobitnú scénu).

Mimochodom, Georgova hra bola nakoniec riadny prepadák. Divákom zjavne robilo problémy zapamätať si vzťahy medzi tými tisíckami postáv...

### Listing programu:

```
#include <iostream>
#include <vector>
#include <cmath>
#define INF 1023456789
using namespace std;

int main(){
    int N, M;
    cin >> N >> M;
    int S = sqrt(N);
    vector<int> H(N + 1);
    for(int i = 1; i <= N; ++i)
        cin >> H[i];

    vector<int> naposledy(M + 1, 0); // posledný výskyt čísla i
    vector<int> prva(S + 1, 1); // začiatok najdlhšieho úseku s i rôznymi číslami
    vector<int> riesenie(N + 1, INF); // d(i)
    riesenie[0] = 0;

    for(int i = 1; i <= N; ++i){
        for(int j = S; j >= 1; --j)
            if(naposledy[H[i]] < prva[j])
                prva[j] = prva[j - 1];
        prva[0] = i + 1;

        naposledy[H[i]] = i;

        for(int j = 1; j <= S; ++j)
            riesenie[i] = min(riesenie[i], riesenie[prva[j] - 1] + j * j);
    }

    cout << riesenie[N] << endl;
}
```

## 9. Tajný plán škorca Jonáša

opravoval MišoF
(max. 25 bodov)

Vitajte v našej záhradke. Posaďte sa do plátených skladacích stoličiek, onedlho na starú čerešňu priletí škorec Jonáš a to sa len budú diať divy!

#### Cesty po strome

Ako to už v informatike chodí, my si naše stromy kreslíme zásadne s koreňom hore. Celé riešenie teda začneme tým, že strom, ktorý sme dostali na vstupe, zoberieme a zavesíme ho za niektorý jeho vrchol – tak, aby všetky hrany smerovali dodola. Vrchol, za ktorý náš strom visí, voláme *koreň*. Vrchol $v$ voláme *predkom* vrcholu $u$, ak $v$ leží na ceste z $u$ do koreňa.

Pozrime sa teraz na to, ako vyzerajú cesty v takomto strome. Predstavme si, že ideme z nejakého vrcholu $u$ do nejakého vrcholu $v$. Ak je $v$ predkom $u$, cesta zjavne vyzerá tak, že ideme z $u$ dohora, kým neprídeme do $v$. V opačnom prípade musíme skôr či neskôr ísť



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

nejakou hranou dodola. Všimnime si prvý okamih, keď sa tak stane. Od tohto okamihu už musíme ísť stále dodola – totiž keby sme niekedy chceli zase ísť dohora, znamenalo by to, že sa vrátime po hrane, ktorou sme práve prišli.

Každá cesta v našom strome teda vedie najskôr dohora, kým neprídeme do prvého spoločného predka oboch koncov cesty, a z neho následne vedie dodola.

**Najbližší spoločný predok**

Ukážeme, že najťažšou časťou našej úlohy je práve nájdenie najbližšieho spoločného predka (least common ancestor) vrcholov, medzi ktorými chce Jonáš prejsť.

Začneme tým, že na našom strome spustíme z koreňa prehľadávanie do hĺbky, počas ktorého si pre každý vrchol $v$ predpočítame počet húseníc $c(v)$ na ceste z neho do koreňa. Toto vieme ľahko spraviť v lineárnom čase.

Ak sa nás teraz Jonáš opýta na cestu z $x$ do $y$, jediné, čo potrebujeme vedieť, je vrchol $z$, ktorý je najbližším spoločným predkom vrcholov $x$ a $y$. Akonáhle budeme poznať $z$, počet húseníc na ceste z $x$ do $y$ určíme ľahko.

Cestu z $x$ do $y$ rozdelíme na dva úseky: z $x$ do $z$ a z $y$ do $z$. Koľko húseníc je na ceste z $x$ do $z$? Predsa $c(x) - c(z)$: keď zoberieme cestu z $x$ do koreňa a odstrihneme z nej cestu zo $z$ do koreňa, zostane nám presne cesta z $x$ do $z$. Podobne na zvyšku cesty je $c(y) - c(z)$ húseníc, celková odpoveď bude teda $c(x) + c(y) - 2c(z)$.

**Prvý pokus: predrátanie $O(N)$, čas odpovede lineárny od dĺžky cesty**

Najjednoduchšou cestou, ako nájsť $z$ a neprezerať pri tom zbytočné časti stromu, je začať zostrojovať cesty z $x$ aj $y$ dohora.

Ešte počas úvodného prehľadávania do hĺbky si pre každý vrchol $v$ spočítame jeho bezprostredného predka („otca“) $p(v)$. Teraz vždy, keď nám príde nová otázka, inicializujeme dve premenné na $x$ a $y$. Obomi budeme teraz striedavo prechádzať vrcholy a značiť si v nich, že sme v nich v aktuálnej iterácii už boli. Striedavo do $x$ priradíme $p(x)$ a do $y$ zase $p(y)$, čím vlastne pôjdeme po oboch cestách rovnako rýchlo dohora. Akonáhle jedna z ciest príde do vrcholu, ktorý sme už pred tým označili, našli sme vrchol $z$. Je zjavné, že počet prezretých vrcholov je menej ako dvojnásobkom počtu vrcholov, ktoré skutočne ležia na ceste z $x$ do $y$, a teda je časová zložitosť odpovede na otázku lineárna od dĺžky uvedenej cesty.

Podotkneme ešte, že pri tomto riešení nie je potrebné predpočítať hodnoty $c(v)$, počet húseníc na ceste vieme počítať aj priamo počas jej hľadania.

Toto riešenie ešte stále nestačilo na zisk veľa bodov, lebo v najhoršom možnom prípade môže byť na zodpovedanie každej otázky potrebné prezrieť skoro celý strom.

**Druhý pokus: predrátanie $O(N \log N)$, odpoveď $O(\log N)$.**

Pokúsime sa teraz vymyslieť nejaký rýchlejší spôsob, ako nájsť najbližšieho spoločného predka. Jednou možnosťou, ak by sme ju vedeli realizovať, by bolo binárne vyhľadávanie. Zoberme si cestu z $x$ do koreňa. Najbližší spoločný predok vrcholov $x$ a $y$ určite leží na tejto ceste. A čo viac, má tú vlastnosť, že od neho vyššie sú všetko predkovia vrcholu $y$, zatiaľ čo od neho nižšie to predkovia nie sú.

Aby sme podľa tohto pozorovania vedeli spraviť efektívne riešenie, potrebujeme vedieť efektívne robiť dve veci: Prvou je povedať, či je vrchol $u$ predkom vrcholu $v$. Druhou je k vrcholom $u_1$ a $u_2$ (kde $u_2$ je predok $u_1$) nájsť vrchol, ktorý je na pol ceste medzi nimi.

Pozrime sa najskôr na prvý problém. Pomocou správneho triku vieme tieto otázky zodpovedať veľmi ľahko, dokonca v konštantnom čase. Opäť upravíme úvodné prehľadávanie do hĺbky. Budeme mať jedno globálne počítadlo počtu krokov. Na začiatku ho nastavíme na nulu a vždy, keď prejdeme po hrane (či už dodola alebo späť dohora) jeho hodnotu zvýšime. Pre každý vrchol $v$ si zapamätáme dve hodnoty: čas $s(v)$ jeho objavenia (hodnotu počítadla, keď sme doň prišli) a čas $t(v)$ jeho opustenia (hodnotu, keď z neho odchádzame dohora).



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

Teraz by malo byť zjavné, že ak $u$ je predkom $v$, tak $s(u) < s(v)$ a zároveň $t(v) < t(u)$. A naopak, ak obe tieto nerovnosti platia, tak $u$ musí byť predkom $v$. Takže pomocou hodnôt $s$ a $t$ vieme v konštantnom čase testovať, či je vrchol $u$ predkom vrcholu $v$.

A čo druhý problém? Nuž, ten tak ľahko riešiť nevieme. Ale ofintíme to trošku ináč. Namiesto klasického binárneho vyhľadávania budeme robiť po strome „skoky“, ktorých veľkosti budú mocniny dvojky.

Označme $p_i(x)$ vrchol vo vzdialenosti $2^i$ nad vrcholom $x$. Už vieme, že pre každé $x$ je $p_0(x) = p(x)$. No a ak poznáme všetky hodnoty $p_i(x)$, ľahko spočítame všetky hodnoty $p_{i+1}(x)$: zjavne platí $p_{i+1}(x) = p_i(p_i(x))$. (Slovne: ak chcem vedieť, čo je $2^{i+1}$ krokov nad $x$, pozriem sa, čo je $2^i$ krokov nad $x$, a následne čo je $2^i$ krokov nad dotyčným vrcholom.)

Zjavne žiaden vrchol nemá hĺbku väčšiu ako $N$, preto stačí $\log_2 N$ takýchto fáz, čas výpočtu týchto údajov je teda $O(N \log N)$.

A ako pomocou týchto údajov nájdeme najbližšieho spoločného predka vrcholov $x$ a $y$? Upravíme binárne vyhľadávanie. Najskôr overíme, či už $x$ nie je predkom $y$. Ak áno, odpoveď je $x$. Ak nie, nájdeme najmenšie $i$ také, že $p_i(x)$ je predkom $y$. Vieme teraz, že počet krokov z $x$ do najbližšieho predka $y$ je z intervalu $[1, 2^i]$, a na binárne vyhľadávanie na tomto intervale už máme všetko potrebné predpočítané.

**Hľadanie minima v úseku poľa**

Od optimálneho riešenia sme ešte ďaleko. Onedlho totiž ukážeme, že otázky budeme dokonca vedieť zodpovedať v konštantnom čase. Na to sa na našu úlohu potrebujeme pozrieť z trochu iného uhla.

Opäť raz začneme tým, že sa pozrieme na úvodné jednoduchučké prehľadávanie do hĺbky a zase raz si všimneme niečo ďalšie, čo nám pomôže.

Predstavme si, že vrcholy $x$ a $y$, ktoré nás zaujímajú, poznáme skôr, ako spustíme prehľadávanie. Teraz sa pozerajme na to, ako prehľadávanie do hĺbky beží, a všimnime si dva okamihy: ten, kedy prvýkrát objavíme $x$, a ten, kedy prvýkrát objavíme $y$. Ako spoznať ich najbližšieho spoločného predka $z$? Jednoducho: je to najvyššie položený zo všetkých vrcholov, cez ktoré sme prechádzali medzi objavením $x$ a $y$.

Ako nám toto pozorovanie pomôže? Tak, že našu úlohu „nájsť najbližšieho spoločného predka v strome“ prevedieme na jednoduchšiu úlohu „nájsť minimum v danom úseku poľa“.

*Hĺbkou* vrcholu $v$ nazveme dĺžku cesty z $v$ do koreňa. Počas prehľadávania do hĺbky si budeme pamätať hĺbku vrcholu, v ktorom práve sme, a vyplníme dve polia: $K(t)$ bude vrchol, v ktorom sme boli, keď bolo počítadlo krokov na hodnote $t$, a $H(t)$ bude hĺbka dotyčného vrcholu. Navyše si ku každému vrcholu zapamätáme hodnotu počítadla v okamihu, keď sme doň prvýkrát prišli.

Ako teraz budeme hľadať najbližšieho spoločného predka? Na vstupe dostaneme vrcholy $x$ a $y$. Zistíme si časy kedy boli objavené a označíme ich $t_1$ a $t_2$ tak, aby $t_1 \leq t_2$. Teraz už len stačí nájsť také $t$ z intervalu od $t_1$ po $t_2$ vrátane, pre ktoré je hodnota $H(t)$ najmenšia. Potom totiž, ako sme vyššie uviedli, platí $z = K(t)$.

Vo zvyšku tohto vzorového riešenia budeme teda riešiť túto zjednodušenú úlohu: ako v poli $H$ čo najrýchlejšie hľadať takéto minimá.

**Odbočka číslo 2: o chlp riešenie ako už máme**

V tomto okamihu vieme spraviť nové riešenie. Toto potrebuje čas $O(N)$ na predspracovanie a $O(\log N)$ na zodpovedanie každej otázky: stačí nad poľom $H$ postaviť intervalový strom, kde si v každom vrchole budeme pamätať index minima v úseku, ktorý danému vrcholu zodpovedá. Tento strom vieme vyrobiť v lineárnom čase a pomocou neho vieme nájsť minimum ľubovoľného úseku v logaritmickom čase.



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

**Tretí pokus: predrátanie $O(N \log N)$, odpoveď v konštantnom čase!**

Nový trik v tomto riešení: keď hľadáme minimum intervalu a pri tom ho delíme na menšie kusy, tieto menšie kusy nemusia byť disjunktné – aj keď sa budú prekrývať, všetko bude v poriadku, minimum nemáme ako minúť. Myšlienkovo bude zvyšok tohto riešenia podobný druhému pokusu o riešenie. Ale nepredbiehajme, poďme na to pekne postupne.

Pole $H$, v ktorom chceme hľadať minimá, má dĺžku $2N-1$. Pre jednoduchšie vyjadrovanie označme túto hodnotu $M$. (Všimnite si, že $N$ a $M$ sú rádovo rovnako veľké, keď teda budeme napr. hovoriť „lineárny“, je jedno, či myslíme závislosť času behu od $N$ alebo od $M$.) Tiež označme $Z = \lfloor \log_2 M \rfloor$.

Pre každé $m$ od 0 do $M-1$ a pre každé $z$ od 0 po $Z$ spočítame index $I(m, z)$, na ktorom sa nachádza minimum úseku, ktorý začína na pozícii $m$ a má dĺžku $2^z$. Inými slovami, pre každý možný začiatok nájdeme minimum úseku dĺžky 1, 2, 4, 8, atď.

Ako na to? Jednoducho. Na začiatku už priamo vieme tieto hodnoty pre $z = 0$, keďže minimom úseku dĺžky 1 je jeho jediný prvok. A akonáhle vieme všetky minimá pre úseky dĺžky $2^z$, ľahko v lineárnom čase spočítame všetky minimá pre úseky dĺžky $2^{z+1}$: Keď chceme zistiť $I(m, z+1)$, stačí sa pozrieť na $a = I(m, z)$ a $b = I(m + 2^z, z)$ a porovnať hodnoty $H(a)$ a $H(b)$. (Slovne: interval dĺžky $2^{z+1}$ jednoducho rozdelíme na dva intervaly dĺžky $2^z$. O každom z nich sme už spočítali, kde má minimum, takže už len stačí tieto dve minimá porovnať.)

No a ako nám tieto údaje pomôžu? Teraz príde k slovu trik, ktorý sme spomínali na začiatku tejto časti. Povedzme napríklad, že potrebujeme nájsť minimum úseku, ktorý začína na pozícii $m$ a má dĺžku 19. Na to sa stačí pozrieť na dva úseky: na úsek začínajúci na pozícii $m$ s dĺžkou 16 a (teraz to príde!) na úsek začínajúci na pozícii $m + (19 - 16) = m + 3$ s dĺžkou 16. Pre oba tieto úseky máme pozíciu minima predpočítanú, a teda všetko potrebné vieme spraviť v konštantnom čase.

Uverili ste? Zle robíte, odžubujem `:-)`. Jeden drobný krok ešte nie je jasný: ako sa z čísla 19 dopracovať k číslu 16? Teda presnejšie, k hodnote $z = 4$, aby sme vedeli pozrieť na správne miesto do predpočítanej tabuľky?

Niektoré procesory na takéto operácie majú inštrukcie, takže by sme sa mohli tváriť, že to v konštantnom čase ide. My sme ale frajeri a na takéto sa odvolávať nepotrebujeme – namiesto toho si to predpočítame!

Ako by vyzeralo pole, kde je ku každej dĺžke úseku napísané to správne $z$? Takto:

```
dlzka | 1 2 3 4 5 6 7 8 9 ...
------------------------
    z | 0 1 1 2 2 2 2 3 3 ...
```

Hodnota 1 je v ňom 2×, hodnota 2 je tam 4×, hodnota 3 zase 8×, atď. No a takéto pole si ľahko na začiatku vyplníme v lineárnom čase a potom sa už doň stačí len pozerať.

Takže zhrnutie tohto riešenia:

- Prehľadávaním do hĺbky zostrojíme polia $K$ a $H$.
- Predpočítame tabuľku, ktorú sme práve popísali.
- Predpočítame indexy $I(m, z)$, kde sú minimá úsekov dĺžky, ktorá je mocninou dvoch.
- Každú otázku teraz zodpovieme v konštantnom čase tak, že nájdeme správne $z$ a pozrieme sa na nanajvýš dve hodnoty v poli $I(m, z)$.

Najpomalšou časťou predrátania je spočítanie hodnôt $I(m, z)$, ktorých je $O(N \log N)$, a taká je aj jeho časová aj pamäťová zložitosť.

**Ako zlepšiť predrátanie?**

Predstavme si, že naše pole $H$, v ktorom hľadáme minimá, nasekáme na úseky dĺžky $D$. Keď teraz dostaneme nejaký interval $[t_1, t_2]$, sú dve možnosti:



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

- buď $t_1$ a $t_2$ ležia v tom istom úseku,
- alebo vieme interval $[t_1, t_2]$ rozdeliť na koniec úseku kde leží $t_1$, niekoľko celých úsekov a začiatok úseku kde leží $t_2$.

Prvou vecou, čo môžeme spraviť, je pre každý úsek dĺžky $D$ nájsť jeho minimum. Tieto minimá uložíme do nového poľa $H'$ dĺžky $M/D$. Na toto nám zjavne stačí lineárny čas.

V tomto poli $H'$ budeme opäť potrebovať vyhľadávať minimá. Použijeme naň teda predspracovanie z predchádzajúceho riešenia. Keďže toto pole má dĺžku $M/D$, bude toto predspracovanie trvať $O((M/D)\log(M/D))$.

Teraz prichádza dôležité pozorovanie: hodnotu $D$, teda dĺžku úseku, si my môžeme zvoliť ako sa nám páči. No a teraz vidíme, že jedna vhodná možnosť bude zvoliť $D$ rádovo rovné $\log M$. Keď toto $D$ dosadíme do odhadu zložitosti, zistíme, že dotyčné predspracovanie už bude mať časovú zložitosť lineárnu od $M$.

Zatiaľ sme teda použili len lineárny čas a už máme predpočítaného dosť na to, aby sme vedeli v konštantnom čase povedať minimum pre ľubovoľný interval, ktorý je tvorený presne niekoľkými celými úsekmi. Zostáva nám teraz doriešiť otázky, ktoré celé ležia vnútri jedného úseku – keď budeme vedieť v konštantnom čase odpovedať aj na tie, už sme vyhrali.

Jednou možnou cestou je využiť, že $H$ nie je len tak hocijaké pole: je to pole, ktoré obsahuje hĺbky vrcholov počas prehľadávania do hĺbky. A teda každá hodnota sa od predchádzajúcej líši buď o $+1$, alebo o $-1$.

Ku každému úseku môžeme zostrojiť postupnosť $D-1$ symbolov $+$ a $-$, ktorá popisuje, ako sa hodnoty v rámci úseku menia. No a len od tejto postupnosti závisí, kde sa v našom úseku nachádza minimum.

Ak teraz naopak zvolíme $D$ dostatočne malé, bude rôznych *typov* úsekov rozumne málo. Bude teda stačiť, keď si pre každý úsek zapamätáme jeho typ a pre každý typ úseku predpočítame odpovede na všetky možné otázky.

No ale čo je to „dostatočne malé“? Aby fungoval predchádzajúci trik, potrebujeme $D$ rádovo rovné $\log M$ alebo väčšie. Ukážeme teraz, že tým pravým kompromisom je napríklad hodnota $D = (\log_2 M)/2$.

Rôznych typov úsekov je $2^{D-1}$. Pre toto $D$ teda dostávame, že rôznych typov úsekov je rádovo $M^{1/2} = \sqrt{M}$. No a pre konkrétny typ úseku ľahko spočítame odpovede na všetky možné otázky v čase $O(D^2) = O(\log^2 M)$. Dokopy nám toto predpočítanie teda zaberie $O(\sqrt{M}\log^2 M)$, čo je ešte lepšie ako lineárne od $M$.

A tým sme definitívne vyhrali – máme riešenie, ktorému na predpočítanie stačí čas lineárny od veľkosti vstupu a následne vie na otázky odpovedať v konštantnom čase. A lepšie to už zjavne nepôjde.

**Pár slov na záver**

Vylepšenie, ktoré sme si predstavili v časti „Ako zlepšiť predrátanie?“, nemá veľký praktický význam – síce vedie k lepšej časovej zložitosti predrátania, ale toto zlepšenie sa pre vstupy praktickej veľkosti nestihne prejaviť. Koniec koncov, ani rozdiel medzi konštantou a logaritmom nie je až taký výrazný, obzvlášť nie v bežných úlohách na praktických súťažiach. Takže už riešenie uvedené v časti „Druhý pokus“ je veľmi dobré a na praktickej súťaži by pravdepodobne na plný počet bodov stačilo. Keďže je však táto úloha teoretická, pár bodov navyše za lepšiu zložitosť bolo. Naopak, body ste strácali hlavne kvôli chybám v odovzdaných programoch.

Z pedagogických príčin uvádzame predposledné riešenie – teda čas predspracovania $O(N \log N)$ a odpovedanie na otázky v konštantnom čase. Implementáciu posledného riešenia môžeme v prípade záujmu zverejniť na webe.



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

## Listing programu:

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

class hrana {
    public: int kam, kolko;
    hrana(int a, int b) : kam(a), kolko(b) {}
};

int N, M, L;
vector< vector<hrana> > strom;
vector< vector<int> > kde_min;
vector<int> suma, vrchol, hlbka, cas, krok;
int teraz=0, hlboko=0;

void load() {
    // nacitame strom a inicializujeme polia
    cin >> N;
    strom.resize(N); suma.resize(N); cas.resize(N);
    for (int n=0; n<N-1; ++n) {
        int x,y,d;
        cin >> x >> y >> d;
        strom[x].push_back( hrana(y,d) );
        strom[y].push_back( hrana(x,d) );
    }
}

void dfs(int kde, int rodic) {
    cas[kde] = vrchol.size();
    suma[kde] = teraz;
    ++hlboko;
    vrchol.push_back(kde);
    hlbka.push_back(hlboko);
    for (unsigned i=0; i<strom[kde].size(); ++i) {
        int kam = strom[kde][i].kam, kolko = strom[kde][i].kolko;
        if (kam == rodic) continue;
        teraz += kolko;
        dfs(kam,kde);
        vrchol.push_back(kde);
        hlbka.push_back(hlboko);
        teraz -= kolko;
    }
    --hlboko;
}

void predrataj() {
    // predratame minima pre useky dlzky 2^l
    M = hlbka.size();
    L = 2 + int(log(1.*N)/log(2.));
    kde_min.resize(L, vector<int>(M));
    for (int m=0; m<M; ++m) kde_min[0][m] = m;
    for (int l=1; l<L; ++l) for (int m=0; m<M; ++m) {
        int a = kde_min[l-1][m];
        int tmp = m + (1<<(l-1));
        int b = tmp<M ? kde_min[l-1][tmp] : a;
        kde_min[l][m] = hlbka[a]<hlbka[b] ? a : b;
    }
```



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

```
    // pre kazdu dlzku useku predratame spravnu mocninu 2
    krok.resize(2,0);
    int davam=1, kolko=2;
    while (krok.size() < hlbka.size()) {
        for (int i=0; i<kolko; ++i) krok.push_back(davam);
        ++davam; kolko*=2;
    }
}

int minimum(int t1, int t2) {
    if (t1 > t2) swap(t1,t2);
    int d = t2 - t1 + 1;
    int l = krok[d], z1 = t1, z2 = t2 - (1<<l) + 1;
    int a = kde_min[l][z1], b = kde_min[l][z2];
    return hlbka[a]<hlbka[b] ? a : b;
}

int main() {
    load();
    dfs(0,0);
    predrataj();
    int x,y;
    while (cin >> x >> y) {
        int z = vrchol[ minimum( cas[x], cas[y] ) ];
        cout << (suma[x] + suma[y] - 2*suma[z]) << endl;
    }
}
```

## 10. Túžba po majetku

opravoval Zemčo
(max. 25 bodov)

Počet riešení v tejto úlohe ma sklamal, pretože tento príklad bol na pomery úloh s číslom 10 ľahký a navyše existovalo úplne jednoduché riešenie s časom $O(NM)$ – pre každú nehnuteľnosť jednoducho prejsť všetky nároky a zistiť, či do nich patrí. Toto riešenie sa dá naprogramovať na tri riadky a bolo by ohodnotené približne ôsmimi bodmi. To je celkom výhodný pomer $\frac{\text{body}}{\text{námaha}}$, nie? V ďalšom budeme nehnuteľnosti nazývať bodmi (ich počet je $M$) a nároky obĺžnikmi (ich počet je $N$).

**Botanická záhrada**

Existuje veľké množstvo riešení s lepším časom ako priamočiary výpočet v odstavci vyššie. Veľmi stručne pre zaujímavosť popíšeme zopár takzvaných *ťažkých kladív*. Niektoré síce presahujú rámec stredoškolských súťaží, ale sme vo vzoráku tyče a preto nejaké poriadne náradie nezaškodí. Kto chce, môže túto časť preskočiť. Naše vzorové riešenie, ktoré je popísané nižšie, je technicky menej náročné.

Jednou z alternatív bolo použiť takzvaný *Quadtree*. Je to strom, v ktorom má každý vrchol štyroch synov a dal by sa nazvať najpriamočiarejším rozšírením klasického intervalového stromu (pozri sekcia nižšie) do dvoch rozmerov. Nech pracujeme s oblasťou, ktorá má štvorcový tvar. Každý vrchol tohto stromu prislúcha nejakej podčasti tejto oblasti.

Koreň zodpovedá celej uvažovanej oblasti. Tú môžeme rozdeliť na dve rovnako veľké polovice podľa zvislej čiary a každú z nich na dve polovice podľa vodorovnej. Dostávame 4 štvorcové oblasti, každú so štvrtinovým obsahom pôvodneho územia. Práve toto budú oblasti zabrané potomkami koreňa. Obdobne na štvrtiny sa delí aj územie týchto potomkov. Takto strom pokračuje ďalej až na štvorcové územia s nejakou konštantnou malou veľkosťou.



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

Náš algoritmus by túto dátovú štruktúru využil tak, že najprv by si do nej zaznamenal každý obdĺžnik postupným rekurzívnym prechodom od koreňa k listom. Ak by pre nejaký vrchol zistil, že celá oblasť pod ním patrí do daného obdĺžnika, už by nešiel hlbšie, ale zaznamenal by si to v počítadle v tomto vrchole. Potom by sa pre každý bod dokázal zo stromu v čase úmernom výške stromu (to je, hrubo povedané, logaritmus od veľkosti územia) dozvedieť odpoveď.[10]

V tejto úlohe mohol poslúžiť aj *dvojrozmerný intervalový strom*. Kto netuší, o čom by mohla byť reč, mal by si najprv prečítať sekciu nižšie o obyčajnom intervalovom strome. Veľmi stručne povedané, k súčtovému intervalovému stromu existuje jeho fínsky variant, ktorý sa implementuje a reprezentuje trošku ináč. Jeho výhodou je napríklad polovičná pamäťová zložitosť a veľmi krátky kód. No a tento diabolský Fín sa veľmi jednoducho rozširuje do ľubovoľného počtu rozmerov. Dostaneme tak dátovú štruktúru, ktorá nám umožňuje vedieť efektívne povedať súčet prvkov nejakej obdĺžnikovej podčasti dvojrozmerného poľa a tiež efektívne meniť hodnoty na jednotlivých políčkach.

Riešenie tohto problému by najprv prečíslovalo súradnice na čísla z menšieho rozsahu (pozri nižšie). Potom by vložilo obdĺžniky do stromu a následne by dokázalo v dobrom čase zistiť počet obdĺžnikov pre daný bod. Ak čitateľ nevie, ako by vyzeralo vloženie hraníc obdĺžnika do stromu, potom by si mal prečítať vzorové riešenie nižšie, kde sa rovnaká triková myšlienka používa v jednorozmernom intervalovom strome. Oproti vzorovému riešeniu má ale použitie dvojrozmerného fínskeho stromu trošku horší čas – $O((N+M)\log^2(N+M))$ a aj horšiu pamäť $O((N+M)^2)$. Oproti iným riešeniam v tomto prehľade má však jednu veľkú výhodu – jeho implementácia je dosť krátka.

Mimochodom, táto variácia intervalového stromu je vo svete známa pod menom *Fenwick Tree* a prívlastok fínsky dostala vďaka IOI 2001 vo Fínsku, kde sa objavila ako vzorák k jednej úlohe a tak sa dostala intenzívnejšie do povedomia.

Jednou z možností bolo použiť myšlienku z dátovej štruktúry s menom *KD-strom*.[11] Je to binárny strom, ktorý vo všeobecnosti slúži na uchovávanie množiny bodov v priestore s ľubovoľným rozmerom. V našom prípade sa pohybujeme v rovine. Každý vrchol tohto stromu zahŕňa niektorú súvislú podoblasť uvažovanej oblasti.

V koreni sú všetky body. Celú rovinu môžeme rozdeliť na dve polovice podľa ľubovoľnej priamky rovnobežnej s osou $x$, pričom niektoré body budú patriť do jednej časti a niektoré do druhej. Práve tieto dve časti budú dvaja synovia koreňa, ktorí sa následne rozdelia podľa priamky rovnobežnej s osou $y$. Nemusia si zvoliť rovnakú priamku. Deti synov si znova rozdelia body v sebe podľa nejakej priamky rovnobežnej s osou $x$ a takto sa postupuje až dovtedy, kým vrchol stromu neobsahuje len jeden bod a vtedy ostáva listom.

V tejto dátovej štruktúre sa teda strieda delenie oblastí podľa dvoch rozmerov. Ak delíme body rozumne (napríklad podľa mediánu na polovice), potom je strom relatívne plytký a stáva sa efektívnym nástrojom pri návrhu rôznych algoritmov.

Tobiáš túto myšlienku skombinoval s dvojrozmerným intervalovým stromom. Najprv si prečíslujeme súradnice (pozri nižšie) a potom skonštruujeme strom, v ktorom má koreň pod sebou celý uvažovaný štvorec, jeho synovia dve polovice podľa osi $x$, synovia týchto synov polovice podľa osi $y$ a tak ďalej, až kým nie je vrchol len jedno políčko. Celý strom má hĺbku $O(\log(N+M))$ a aby sme sa vyhli veľkej pamäťovej náročnosti, budeme strom budovať dynamicky – vrchol vznikne a bude udržovaný v pamäti len vtedy, ak bola oblasť súčasťou nejakého obdĺžnika. Takýchto vrcholov bude najviac $O((N\log(N+M))$, čo je o dosť menej ako teoretických $O((N+M)^2)$. Na začiatku si vložíme obdĺžniky a potom dokážeme pre jednotlivé body rýchlo cestovaním od koreňa k listom zistiť, do koľkých obĺžnikov patria.

[10] Zvedavý čitateľ môže nájsť trošku viac napríklad na adrese `http://en.wikipedia.org/wiki/Quadtree`.

[11] `http://en.wikipedia.org/wiki/Kd-tree`



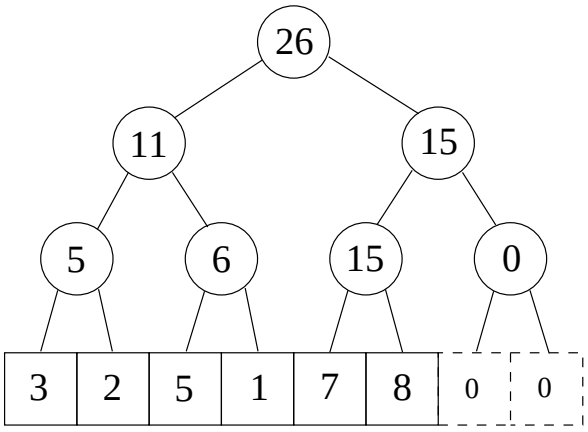
Táto práca bola podporená Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

**Intervalový strom**

V tejto sekcii si popíšeme dátovú štruktúru s názvom intervalový strom, ktorá je o dosť jednoduchšia ako tie v prehľade vyššie. Kto sa už s intervalovými stromami stretol, môže samozrejme tieto odstavce preskočiť.

Uvažujme jednoduchú úlohu: majme dané pole $A$, ktoré má dĺžku $N$ a v ktorom sú celé čísla. Radi by sme navrhli algoritmus, ktorý by mal na vstupe dva typy požiadaviek: povedať súčet ľubovoľného súvislého podintervalu tohto poľa a meniť hodnoty v poli. Tieto požiadavky sa na vstupe môžu všelijako striedať a každú musíme spracovať v reálnom čase bez informácie o nasledujúcich požiadavkách. Čas $O(N)$ na operáciu je príliš veľký a preto chceme obidve požiadavky riešiť v čase $O(\log N)$.

Pre jednoduchosť ďalších úvah zväčšime najskôr pole $A$ tak, aby jeho veľkosť bola najbližšia väčšia mocnina dvoch. Tým sa pole $A$ nanajvýš dvakrát predĺži, takže na časovej zložitosti výsledného algoritmu sa to neodrazí. Nech odteraz $N = 2^K$ je dĺžka predĺženého poľa. Novú časť poľa doplníme nulami a nebude pre nás vôbec dôležitá.



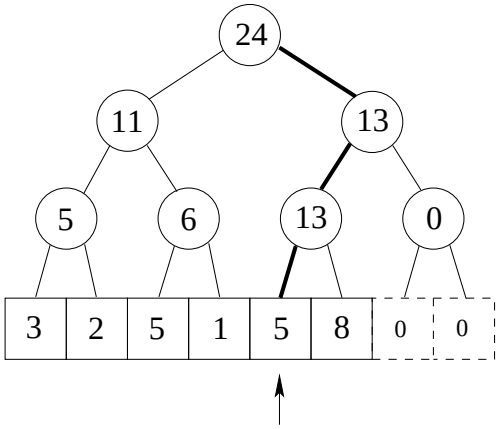

Predstavme si, že nad poľom $A$ vybudujeme úplný binárny strom. Jeho listy budú zodpovedať jednotlivým políčkam poľa $A$, každý vyšší vrchol tohoto stromu bude zodpovedať nejakému intervalu v poli $A$ (presnejšie bude zodpovedať políčkam určeným listami z jeho podstromu). V každom vrchole stromu si budeme pamätať súčet čísel v príslušnom intervale poľa. Takejto dátovej štruktúre budeme hovoriť intervalový strom.

V najspodnejšej vrstve nášho stromu je $N$ vrcholov, vo vyššej ich je $N/2$, v tretej odspodu $N/4$, atď. V celom našom strome je teda $2N - 1$ vrcholov, preto budeme potrebovať na jeho zapamätanie pamäť veľkosti $O(N)$.

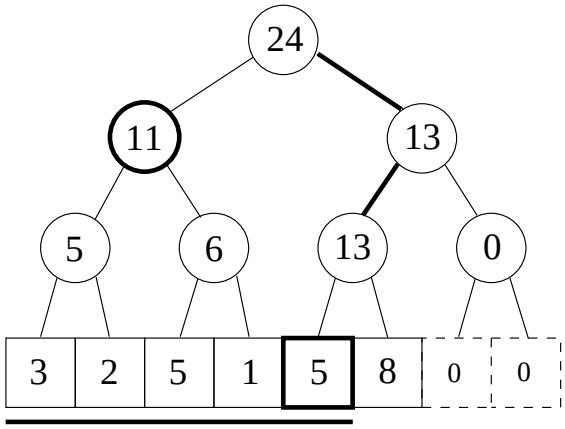
Čo sa stane s naším stromom, keď zmeníme hodnotu prvku $A[j]$? Musíme zmeniť zapamätané hodnoty pre všetky intervaly, ktoré zmenený prvok obsahujú. Tie ale zodpovedajú práve vrcholom na ceste z $j$-teho listu do koreňa. Je ich teda $K + 1 = O(\log N)$. Zmeniť hodnotu v poli $A$ teda vieme v logaritmickom čase.



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




Zmena hodnoty.




Počítanie súčtu.

Ešte ostáva ukázať, ako pomocou nášho stromu odpovedať na otázky o súčtoch. Riešme najskôr jednoduchšiu úlohu: Akú hodnotu má súčet $S(x) = A[1] + \ldots + A[x]$? Pozrime sa do koreňa nášho stromu. Sú dve možnosti: Ak interval od 1 po $x$ leží celý v ľavom podstrome, zavoláme sa rekurzívne naň. Ak nie, tak tento interval zaberá celý ľavý podstrom a kúsok pravého. Tak zoberieme súčet všetkých políčok v ľavom podstrome (ten máme spočítaný v ľavom synovi) a rekurzívne sa zavoláme na pravého syna a zvyšok intervalu.

Takto postupne v našom strome schádzame dole po ceste od koreňa do $x$-teho listu, na každej úrovni urobíme len konštantný počet operácií. Preto pre ľubovoľné $x$ vieme hodnotu $S(x)$ spočítať v čase $O(\log N)$. To je ale všetko, čo potrebujeme vedieť – totiž $A[x] + \ldots + A[y] = S(y) - S(x-1)$ (pričom definujeme $S(0) = 0$).

Najjednoduchšou implementáciou intervalového stromu je uložiť ho v jednom poli, podobne ako haldu. Teda synovia vrcholu $x$ sú na políčkach $2x$ a $2x+1$, pôvodné pole $A$ začína na pozícii $N$. Tento typ intervalového stromu nazveme súčtový. Čitateľ môže sám popremýšľať, ako by fungoval minimový a maximový intervalový strom a na čo by bol dobrý.

Riešme ešte jednu úlohu. Máme danú dlhú číselnú os a na nej máme vyznačené uzavreté intervaly, ktoré začínajú a končia v prirodzených číslach a všelijako sa môžu prekrývať. Radi by sme navrhli algoritmus, ktorý bude spracovávať dva typy požiadaviek:

1. Pre dané prirodzené číslo na osi povedať, v koľkých intervaloch sa nachádza (do intervalu sa číslo počíta aj keď je v koncovom bode).
2. Pridať nový interval alebo odobrať existujúci.

Nech $N$ je najväčšie číslo, ktoré sa nám pri požiadavke hociktorého typu vyskytne. Isto si viete predstaviť triviálne algoritmy, ktoré by prvú úlohu riešili v konštantnom čase a druhú



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

v lineárnom od $N$, prípadne opačne. Pre nás je ale lineárny čas opäť príliš veľký a preto chceme riešenie, ktoré nájde rovnováhu a obe požiadavky bude riešiť v logaritmickom čase. Na to nám môže opäť poslúžiť súčtový intervalový strom. Pole, nad ktorým postavíme tento strom, znovu označíme $A$. Bude dlhé $N$ a jeho jednotlivé políčka budú potenciálne otázky alebo začiatky a konce intervalov.

Do intervalového stromu si budeme značiť začiatky a konce intervalov. Čo by sa stalo, keby sme pre každý interval $<a, b>$ zvýšili o 1 hodnotu v poli na políčku s indexom $a$? Potom by bol pre bod $X$ súčet $S(X) = A[1] + A[2] + \ldots + A[X]$ vlastne počet intervalov, ktoré začali pred bodom $X$ alebo presne v ňom. To už pripomína želanú hodnotu. Avšak niektoré intervaly mohli pred bodom $X$ aj skončiť. Preto si ešte označíme aj koniec intervalu – hodnotu $A[b+1]$ znížime o 1 (v poli $A$ môžu byť aj záporné hodnoty). Potom bude platiť, že $S(X)$ je počet intervalov, ktoré do bodu $X$ už začali, ale ešte neskončili. Ak je na začiatku pole $A$ inicializované na nuly (na osi nie sú žiadne intervaly), potom po zodpovedajúcej úprave poľa pre interval $<a, b>$ dostaneme $S(0) = S(1) = \ldots = S(a-1) = 0$, $S(a) = S(a+1) = \ldots = S(b) = 1$ a $S(b+1) = \ldots\ S(N) = 0$. Takže súčet je jedna presne pre body, ktoré patria do intervalu. Takýmto spôsobom jednoducho pomocou dvoch úprav stromu dokážeme interval nielen pridať, ale ho aj odobrať. Pri odoberaní hodnoty upravíme obrátene – prvú znížime a druhú zvýšime.

**Prečíslovanie súradníc**

Uvažujme jednoduchý problém – sme organizátori veľkej hostiny a čakáme $N$ priateľov. Každý priateľ príde na svojom aute a my vieme, že sú všetky veľmi krásne a hodnotné. Pred našou vilou máme presne $N$ parkovacích miest v jednom dlhom rade. Z čírej roztopaše by sme chceli, aby boli autá po príchode našich priateľov zoradené a zaparkované podľa ceny. Problém ale je, že priatelia prichádzajú po jednom a my dopredu nevieme, kde je podľa usporiadania ich parkovacie miesto. Ak vyberieme náhodné, je veľmi pravdepodobné, že sa daný hosť bude musieť neskôr presúvať (napríklad preto, lebo príde nečakane veľa drahších alebo lacnejších áut). Je teda nemožné každému priateľovi pri príchode povedať, kde bude jeho parkovacie miesto.

Čo ak by ale boli ceny áut jeden milión, dva milióny, tri milióny, ..., $N$ miliónov? Ak by to tak bolo a my sme túto informáciu dopredu vedeli, potom by sme to mali ľahké – keď by prišiel hosť s autom s hodnotou $i$ miliónov, potom by sme ho nasmerovali na parkovacie miesto, ktoré je $i$-te v poradí od toho konca, kde bude lúzer s najlacnejším autom. Všimnite si, že táto informácia nám vlastne dopredu hovorí, koľké auto podľa ceny prichádza.

Čo ak máme náhodou dopredu zoznam hodnôt áut aj s cenami, ktoré ale nie sú násobkami milióna? Ak by sme dokázali každému autu priradiť jeho poradie od 1 po $N$, potom by sme dokázali prichádzajúcich hostí bezpečne nasmerovať na príslušné parkovacie miesto a mali istotu, že nebudeme musieť nikoho neskôr žiadať, aby si svojho tátoša preparkoval.

Formulujme si teda ešte raz úlohu: máme daných $N$ rôznych celých čísel. Chceli by sme im priradiť čísla od 1 po $N$ tak, aby priradenie zodpovedalo usporiadaniu. Ak dané priradenie označíme ako $f$, potom musí platiť, že ak vezmeme čísla $x$ a $y$ zo vstupu, potom $f(x) < f(y)$ vtedy a len vtedy, ak $x < y$.

Najjednoduchšou možnosťou, ako toto dosiahnuť, je jednoducho čísla utriediť a každému priradiť nové číslo podľa poradia v utriedenom poli. Časová zložitosť tohto prečíslovania je teda $O(N \log N)$ a lepšie to nejde.[12] Prečíslovanie môžeme prirodzene rozšíriť aj na prípad, v ktorom vstupné čísla nemusia byť rôzne. Vtedy navyše požadujeme, aby pôvodne rovnaké čísla obdržali rovnakú hodnotu. V takom prípade teoreticky nemusíme použiť pri prečíslovaní všetkých $N$ nových hodnôt, ale môže nám stačiť menej.

[12] Možno viete, že ak chceme triediť, tak sa nám to vo všeobecnosti nemôže podariť lepšie ako v čase $O(N \log N)$. Na domácu úlohu si môžete premyslieť, prečo to nejde ani pri úlohe prečíslovania. Ako pomôcka vám môže poslúžiť práve spomenutý fakt, že prečíslovanie veľmi úzko súvisí s triedením.



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

Trik prečíslovania využijeme aj v našej úlohe s bodmi a obdĺžnikmi. Neskôr vysvetlíme, na čo nám to bude dobré. Teraz si len povedzme, že ak hodnoty súradníc na vstupe prečíslujeme a situáciu si nakreslíme nanovo, potom sa nám určite nezmení odpoveď.

Budeme prečíslovávať súradnice obdĺžnikov aj bodov spolu. A budeme prečíslovávať súradnice $x$ a $y$ zvlášť. Ak teda napríklad máme odĺžnik s protiľahlými rohmi $[-1, 2]$ a $[4, 3]$ a bod $[2, 3]$, potom po prečíslovaní dostaneme situáciu s obdĺžnikom s rohmi v $[1, 1]$ a $[3, 2]$ a bodom $[2, 2]$. Všimnite si, že patričnosť bodu do obdĺžnika sa nám nezmenila a rovnako sa nám nezmenil ani fakt, že bod leží na hrane obdĺžnika.

Poďme si ukázať, že je to naozaj tak vždy. Uvažujme bod $[a, b]$ a obdĺžnik s rohmi $[x_1, y_1]$ a $[x_2, y_2]$. Predpokladajme, že $x_1 < x_2$ a $y_1 < y_2$. Ak bod patril do obdĺžnika, potom nutne $x_1 \leq a$ a zároveň $a \leq x_2$. Avšak ak prečíslujeme hodnoty $x_1, a, x_2$, potom sa nám tieto nerovnosti určite nepokazia. Navyše, po prečíslovaní nastane prípad rovnosti práve vtedy, ak boli čísla rovné aj pred prečíslovaním. Keďže analogicky platí aj tvrdenie pre $y$ súradnicu, potom dostávame, že bod patrí do obdĺžnika po prečíslovaní práve vtedy, ak tam patril aj pred prečíslovaním. Všimnite si, že platí aj fakt, že dva body sú totožné pred prečíslovaním práve vtedy, keď sú totožné po ňom.

### Riešenie zametaním

Zametanie je veľmi užitočná a obľúbená technika pri riešení rôznych úloh.[13] Všetci sme sa istotne stali v mladosti alebo detstve obeťami rodičovských príkazov a museli doma niečo pozametať.

Hlavná myšlienka je nasledovná: keď zametáme miestnosť, postupujeme systematicky z jednej strany do druhej a postupne rozširujeme už pozametanú časť dlážky. Tá je navyše v každom momente súvislá. Algoritmy založené na zametaní tiež systematicky (napríklad sprava doľava) spracovávajú situáciu. Schéma návrhu zametacích algoritmov vyzerá takto:

1. Ujasníme si, akým spôsobom (smerom) chceme situáciu zametať. Tento krok býva s istými skúsenosťami s návrhom zametacích algoritmov ľahký a priamočiary.

2. Definujeme si takzvané *udalosti* – momenty, kedy sa nám pri systematickom prechode deje niečo zaujímavé. Udalostiam sa niekedy hovorí *eventy*.

3. Navrhneme internú dátovú štruktúru, ktorá nám umožní získavať odpoveď na otázky a bude sa ľahko meniť.

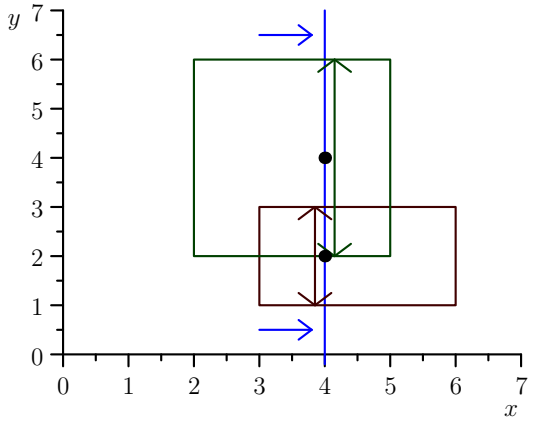
4. V prechode potom udalosti spracovávame v systematickom poradí.

Ak ste tejto schéme príliš neporozumeli, tak to určite bude lepšie po prečítaní zvyšku vzoráku. Zametací algoritmus je vhodný vtedy, ak vieme eventy spracovávať rýchlo. Naše riešenie začne tak, že si najprv prečísluje súradnice bodov aj obdĺžnikov. Teraz teda predpokladáme, že hodnota každej súradnice je najviac $N + M$. Budeme postupovať po území zľava doprava a postupne zisťovať pre body, v koľkých obdĺžnikoch ležia.

[13] Po anglicky sa algoritmu s technikou zametania hovorí *sweep line algorithm*.



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




Naším eventom bude výskyt bodu. Okrem toho bude eventom začiatok alebo koniec obdĺžnika. Budeme postupovať podľa rastúcej súradnice $x$. To znamená, že ak budeme mať obdĺžnik s rohmi $[0,0]$ a $[2,3]$ a bod $[3,0]$, dostaneme tri eventy: jeden na súradnici 0, druhý na súradnici 2 a tretí na súradnici 3. Pri riešení bodového eventu budeme pomocou dátovej štruktúry hľadať odpoveď na otázku, v koľkých obdĺžnikoch leží a pri obdĺžnikových udalostiach budeme meniť údaje v dátovej štruktúre. Asi neprekvapí, že touto dátovou štruktúrou bude intervalový strom.

Pozrime sa na situáciu zboku v smere zametania – otočme sa smerom rastu osi $x$ do nekonečna. Teraz sa na obdĺžniky môžeme dívať ako na intervaly. Pri tomto pohľade totiž zanedbávame ich $x$-ový rozmer a vidíme len ten $y$-ový. Ak je na ploche obdĺžnik s protiľahlými rohmi $[3,1]$ a $[6,3]$ a druhý s rohmi $[2,2]$ a $[5,6]$, tak z tohto pohľadu vidíme dva intervaly: jeden medzi $<1,3>$ a druhý medzi $<2,6>$. Ak by sme uvažovali bod $[4,2]$, tak ten by patril do oboch obdĺžnikov, pretože patrí do oboch uvedených intervalov. Avšak bod $[4,4]$ by patril len do druhého obdĺžnika.

Z tejto perspektívy teda pracujeme s intervalmi a pýtame sa, do koľkých z nich patria body. Tomuto príkladu zodpovedá obrázok vyššie. A čo sa stane, ak budeme prechádzať cez obdĺžnikový event? No predsa nejaký interval pribudne alebo ubudne. No a teraz by to už malo byť zrejmé a priamočiare použitie intervalového stromu podľa predchádzajúcej sekcie.

Zhrňme si činnosť algoritmu. Najprv si vytvorí $N + 2M$ eventov. Pri bodových udalostiach si musíme pamätať číslo bodu, ktorému event patrí, a $y$-ovú súradnicu. Pri obdĺžnikových eventoch si musíme pamätať dve $y$-ové súradnice – kde obdĺžnik začína a kde končí v $y$-ovom rozmere (kolmý na smer zametania).

Všetky udalosti utriedime podľa $x$-ovej súradnice – toto bude poradie, v akom ich budeme spracovávať. V prípade rovnosti treba najskôr spracovať udalosti, pri ktorých začínajú obdĺžniky, potom bodové eventy a potom tie, kde končia obdĺžniky. Totiž, ak nejaký bod je na hrane obdĺžnika, ktorý práve začína alebo končí, potom do neho ešte patrí. Spracovávajme teraz eventy jeden po druhom:

1. Máme event začiatku obdĺžnika s horizontálnymi hranicami $y_1, y_2$. V strome zvýšime hodnotu v liste $y_1$ o 1 a v liste $y_2 + 1$ znížime o 1.
2. Máme event bodu s súradnicou $y$. Zo stromu zistíme súčet hodnôt na políčkach 1 až $y$.
3. Máme event končiaceho obdĺžnika s horizontálnymi hranicami $y_1, y_2$. V strome znížime hodnotu v liste $y_1$ o 1 a v liste $y_2 + 1$ zvýšime o 1.

Teoreticky nám stačilo prečíslovať súradnice v rozmere $y$. Toto prečíslovanie nám spôsobilo, že veľkosť stromu stačí nastaviť rádovo $O(N + M)$, pretože také sú hodnoty $y$, s ktorými pracujeme pri eventoch.

A aká je časová zložitosť? Označme $K = N + M$. Prečíslovanie nás stojí $O(K \log K)$ a utriedenie eventov $O(K \log K)$. Spracovanie jedného eventu ľubovoľného typu nás stojí $O(\log K)$ a celkový počet eventov je $O(K)$. Celkový čas riešenia je teda $O((N+M)\log(N+$



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

$M$)) a pamäťové nároky sú $O(N + M)$. Náš kód je síce celkom dlhý, ale nie je príliš zložitý. Skladá sa z krátkej časti prečíslovania, krátkej časti implementácie intervalového stromu a zvlášť jednoduchej implementácie samotného zametania.

## Listing programu:

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

typedef pair<int,int> bod;
//typ - 0 - zaciatok obdlznika, 1 - otazka, 2 - koniec obdlznika
//poziadavka je cislo objektu, ktoreho sa udalost tyka. potrebne pre vystup
typedef struct{
    int typ, x, y1, y2, poziadavka;
} udalost;

//porovnavac udalosti. vrati true, ak je a skor ako b
bool comp(udalost a, udalost b){
    if (a.x > b.x) return false;
    if (a.x < b.x) return true;
    return a.typ < b.typ;
}

int N, M, pocetlistov;
//vstup. hranice obdlznikov a vyznamne objekty
vector<bod> poziadavky, uzemia1, uzemia2;
vector<udalost> udalosti;
//intervalovy strom a pole na vysledok
vector<int> strom, vysledok, X, Y;

void vlozudalost(int x, int y1, int y2, int typ, int poziadavka){
    udalost u;
    u.x = x;
    u.y1 = y1;
    u.y2 = y2;
    u.typ = typ;
    u.poziadavka = poziadavka;
    udalosti.push_back(u);
}

//inicializuje intervalovy strom
void init(int listov){
    int K = 1;
    while (K < listov) K*=2;
    strom.resize(K*2,0);
    pocetlistov = K;
}

//rekurzivna funkcia vrati sucet intervalu [1,pokial].
//prve tri parametre su vrchol a interval, ktory pod nim lezi
int sucet(int vrchol, int vrcholodkial, int vrcholpokial, int pokial){
    //cely interval pod tymto vrcholom je v dotazovanom intervale
    if (pokial >= vrcholpokial) return strom[vrchol];
    int vratim = 0;
    int stred = (vrcholpokial + vrcholodkial)/2;
    if (pokial > stred) vratim += sucet(vrchol*2+1, stred+1, vrcholpokial, pokial);
    vratim += sucet(vrchol*2, vrcholodkial, stred, pokial);
```



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

```
    return vratim;
}

//procedura upravi hodnotu v liste a zodpovedajuco aj cely strom
void uprav(int list, int zmena){
    int w = pocetlistov+list-1;
    while(w >= 1){
        strom[w] += zmena;
        w/=2;
    }
}

//metoda precisluje suradnice z lubovolnych na hodnoty od 1 do najviac 2N+M
void precisluj(){
    for(int i=0; i<N; i++){
        X.push_back(uzemia1[i].first);
        X.push_back(uzemia2[i].first);
        Y.push_back(uzemia1[i].second);
        Y.push_back(uzemia2[i].second);
    }
    for(int i=0; i<M; i++){
        X.push_back(poziadavky[i].first);
        Y.push_back(poziadavky[i].second);
    }
    sort(X.begin(), X.end());
    sort(Y.begin(), Y.end());
    //v utriedenom poli binarnym vyhladavanim najdeme miesto vyskytu danej hodnoty
    //nova hodnota je rozdiel medzi miestom zaciatku pola a najdenym pointrom
    for(int i=0; i<N; i++){
        uzemia1[i].first = (int)(lower_bound(X.begin(), X.end(), uzemia1[i].first) -
X.begin())+1;
        uzemia2[i].first = (int)(lower_bound(X.begin(), X.end(), uzemia2[i].first) -
X.begin())+1;
        uzemia1[i].second = (int)(lower_bound(Y.begin(), Y.end(),
uzemia1[i].second) - Y.begin())+1;
        uzemia2[i].second = (int)(lower_bound(Y.begin(), Y.end(),
uzemia2[i].second) - Y.begin())+1;
    }
    for(int i=0; i<M; i++){
        poziadavky[i].first = (int)(lower_bound(X.begin(), X.end(),
poziadavky[i].first) - X.begin())+1;
        poziadavky[i].second = (int)(lower_bound(Y.begin(), Y.end(),
poziadavky[i].second) - Y.begin())+1;
    }
}

int main(){
    cin >> N >> M;
    poziadavky.resize(M);
    uzemia1.resize(N);
    uzemia2.resize(N);
    for(int i=0; i<N; i++){
        cin >> uzemia1[i].first >> uzemia1[i].second;
        cin >> uzemia2[i].first >> uzemia2[i].second;
        //uistime sa, aby prvy bod obdlznikov bol ten s mensou x aj y
        if (uzemia1[i].first > uzemia2[i].first){
            int t = uzemia1[i].first;
            uzemia1[i].first = uzemia2[i].first;
            uzemia2[i].first = t;
        }
```



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

```
        if (uzemia1[i].second > uzemia2[i].second){
            int t = uzemia1[i].second;
            uzemia1[i].second = uzemia2[i].second;
            uzemia2[i].second = t;
        }
    }
    for(int i=0; i<M; i++){
        cin >> poziadavky[i].first >> poziadavky[i].second;
    }
    precisluj();
    for(int i=0; i<N; i++){
        vlozudalost(uzemia1[i].first, uzemia1[i].second, uzemia2[i].second, 0, -1);
        vlozudalost(uzemia2[i].first, uzemia1[i].second, uzemia2[i].second, 2, -1);
    }
    for(int i=0; i<M; i++){
        vlozudalost(poziadavky[i].first, poziadavky[i].second, -1, 1, i);
    }
    sort(udalosti.begin(), udalosti.end(), comp);
    //inicializacia intervaloveho stromu
    init(N + M + 2);
    vysledok.resize(M,0);
    for(int i=0; i<udalosti.size(); i++){
        //spracovavame zaciatok obdlznika
        if (udalosti[i].typ == 0){
            uprav(udalosti[i].y1, 1);
            uprav(udalosti[i].y2+1, -1);
        }
        //spracovavame otazku, v kolkych obdlznikoch je bod
        if (udalosti[i].typ == 1) vysledok[udalosti[i].poziadavka] = sucet(1, 1,
pocetlistov, udalosti[i].y1);
        //spracovavame koniec obdlznikov
        if (udalosti[i].typ == 2){
            uprav(udalosti[i].y1, -1);
            uprav(udalosti[i].y2+1, 1);
        }
    }
    for(int i=0; i<M; i++) cout << vysledok[i] << endl;
    return 0;
}
```



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

# Výsledková listina po 1. kole kategórie KSP-Z

**Chýba súbor Z!!!**

# Výsledková listina po 1. kole kategórie KSP-O

**Chýba súbor O!!!**

# Výsledková listina po 1. kole kategórie KSP-T

**Chýba súbor T!!!**


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