Korešpondenčný seminár z programovania
XXIX. ročník, 2011/12
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 2. kola zimnej časti

1. Žiariaci gitaristi

opravoval Žaba
(max. 7 b za popis, 3 b za program)

No, myslím si, že prvá úloha splnila svoj zámer a naozaj ju väčšina z vás zvládla na plný počet bodov. Prekvapivo sa však vyskytlo veľké množstvo úplne rôznych riešení. Poďme sa teda pozrieť na podľa mňa najbezbolestnejšie z nich.

Samozrejme, struny sú od seba nezávislé. Preto sa stačí sústrediť na jednu z nich – s ostatnými budeme robiť to isté. To hlavné, čo nás zaujíma, je ako hrať čo najlenivejšie. Ideálne by bolo nespraviť ani jeden zbytočný pohyb.

Pokúsme sa teda o to, že struny budeme stláčať a púšťať až vtedy, keď to bude úplne nutné. Predstavme si, že už máme na strune stlačené nejaké pražce. Teraz nám príde požiadavka zahrať nový tón. Ak je to najvyšší, ktorý držíme, rovno ho zahráme. Ak potrebujeme strunu stlačiť na vyššom mieste od všetkých, ktoré držíme, stlačíme ju aj tam (a nič nepustíme). Toto stačí na zahranie správneho tónu.

No a ak je nový tón nižší ako najvyšší práve stlačený, tak potrebujeme pustiť všetky tóny, ktoré sú vyššie od nášho, inak tento tón nezaznie. (A potom prípadne chytiť ten tón, ktorý máme zahrať.) Všimnite si, že nerobíme žiaden zbytočný pohyb, iba také, ktoré musíme spraviť.

Ako teda vyzerajú pohyby na strune? Občas stlačíme nejaký nový tón, občas nejaký pustíme, ale vždy, keď stláčame, stláčame vyšší pražec ako ostatné práve držané, a tiež keď púšťame, púšťame od najvyšších pražcov. Teda celá struna funguje ako dátová štruktúra LIFO[1], tiež známa ako stack alebo zásobník.

Zásobník vieme ľahko implementovať pomocou poľa a jednej premennej. V premennej si budeme pamätať, koľko prvkov máme v zásobníku. No a tie prvky budeme mať uložené na začiatku poľa – tak, aby bol prvý ten, ktorý je úplne naspodku. Keď teraz chceme pridať nový prvok do zásobníka, zvýšime počítadlo a na príslušné miesto poľa ten prvok uložíme. A odobratie prvku samozrejme funguje presne opačne. Obe tieto operácie vieme teda robiť v konštantnom čase.

Program riešiaci úlohu bude teda vyzerať nasledovne. Bude mať šesť zásobníkov – pre každú strunu jeden. Prvky v zásobníku budú čísla práve stlačených pražcov. Keď nám príde nová požiadavka, tak sa pozrieme na vrch príslušného zásobníka. Kým tam bude hodnota väčšia ako chceme, budeme ju z neho vyberať (a prirátavať pohyby). Keď sa väčšie hodnoty minú, buď tam je hľadaná hodnota, alebo nejaká menšia. V tom druhom prípade ešte navrch pridáme želanú hodnotu (a opäť prirátame pohyb).

Čo sa týka zložitosti, tak pamäťová bude $O(n)$, lebo si budeme musieť pamätať najviac n tónov. Celková časová zložitosť bude tiež $O(n)$, lebo každý tón na vstupe najviac raz pridáme do zásobníka a najviac raz ho odtiaľ (niekedy neskôr) vyberieme.

Listing programu:

```
var Struny: array [1..6,1..1000000] of longint; { pre kazdu strunu jedno pole ako zasobnik }
    pocet: array [1..6] of longint; { udava pocet prvkov v danom zasobniku }
    pohyby,n,i,struna,prazec: longint;
```

[1] last in first out

```
begin
  readln(n);
  for i:=1 to n do
    begin
      read(struna); readln(prazec);
      while((pocet[struna]<>0) and (Struny[struna,pocet[struna]]>prazec)) do
begin
  pohyby:=pohyby+1;
  pocet[struna]:=pocet[struna]-1;
end;
      if((pocet[struna]=0) or (Struny[struna,pocet[struna]]<prazec)) then
begin
  pohyby:=pohyby+1;
  pocet[struna]:=pocet[struna]+1;
  Struny[struna,pocet[struna]]:=prazec;
end;
    end;
  writeln(pohyby);
end.
```

Listing programu:

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

stack<int> Struny[6];

int main() {
    int n, pohyby=0;
    scanf("%d ",&n);
    for(int i=0; i<n; i++) {
        int struna,prazec;
        scanf("%d %d ",&struna,&prazec);
        struna--;
        while (!Struny[struna].empty() && Struny[struna].top()>prazec) {
            Struny[struna].pop(); pohyby++;
        }
        if (Struny[struna].empty() || Struny[struna].top()<prazec) {
            Struny[struna].push(prazec); pohyby++;
        }
    }
    printf("%d\n",pohyby);
}
```

2. Zaraďte sa, prosím

opravoval Usamec
(max. 7 b za popis, 3 b za program)

Táto úloha nebola veľmi ťažká, akurát si bolo treba veci poriadne premyslieť a, ako sa medzi nami hovorí, „neodrbať sa“.

Pre začiatok si môžeme pre každého študenta spočítať, kedy sa skončí jeho vybavovanie, Usámu zatiaľ zanedbáme. Pôjdeme postupne po študentoch a budeme si pamätať, kedy ktorý skončil. To, kedy skončil $(i-1)$-vý, využijeme pri výpočte toho, kedy skončil i-ty študent.

Ak $(i-1)$-vý bol vybavený najneskôr vtedy, kedy sa i-ty postavil do radu, tak i-ty pôjde dnu hneď keď príde. Jeho vybavovanie teda skončí v čase $t_i + d_i$. Takúto situáciu budeme volať medzera. Všimnite si, že medzi časom, kedy skončilo vybavovanie $(i-1)$-vého a časom, kedy začalo vybavovanie i-teho študenta, sa Usáma vie kedykoľvek bez čakania dostať dnu.

A naopak, ak bol $(i-1)$-vý študent vybavený neskôr ako v čase t_i, bude musieť i-ty študent čakať. Dnu pôjde až v tom okamihu, kedy bude $(i-1)$-vý vybavený. Čas vybavenia i-teho zistíme tak, že k času vybavenia $(i-1)$-vého pripočítame d_i.

Kam teraz do radu napchať Usámu? Ak v čase medzi a a b existuje nejaká medzera, nájdeme prvú z nich a vtedy ho pošleme dnu bez čakania. (Presnejšie, dnu ho pošleme na začiatku príslušnej medzery, resp. v čase a, ak dotyčná medzera začala skôr. Tiež nezabudneme skontrolovať medzeru po spracovaní posledného zo študentov.)

Ak tam žiadna medzera nie je, tak tvrdíme, že sa Usámovi určite oplatí prísť spolu s niektorým iným študentom. Totiž keby prišiel medzi nejakými študentmi $(i-1)$ a i, bude čakať až do vybavenia $(i-1)$-vého študenta. Vtedy isto by sa dostal na rad aj keby prišiel až naraz s i-tym študentom, a zároveň by kratšie čakal. Počas simulácie teda postupne pre všetky

možné i zistíme, ako dlho by Usáma čakal, ak by prišiel spolu s i-tym študentom. Vyberieme si najlepšiu z tých možností, ktoré ležia v predpísanom intervale časov od a po b.

Celé toto riešenie sa dá naprogramovať ako jeden prechod cez vstup. Stačí si pritom pamätať, kedy skončilo vybavovanie predchádzajúceho študenta. Takže máme časovú zložitosť $O(n)$ a pamäťovú $O(1)$.

Listing programu:

```
var n,a,b,vybavenie_predchadzajuceho,best_cakanie,best_miesto,t,d,i:longint;
    cakanie:longint;

function max(x,y:longint):longint;
begin
  if x > y then exit(x);
  exit(y);
end;

begin
  read(n,a,b);
  vybavenie_predchadzajuceho := 0;
  best_cakanie := 1000000123;
  best_miesto := 1000000123;

  for i:=1 to n do begin
    read(t,d);
    { Overime pritomnost medzery }
    if vybavenie_predchadzajuceho <= t then begin
      { Overime, ci sa medzera da pouzit }
      if (t >= a) and (vybavenie_predchadzajuceho <= b) then begin
        { Rovno ju mozeme vypisat a skoncit, nic lepsie nenajdeme }
        writeln(max(vybavenie_predchadzajuceho, a), ' 0');
        exit;
      end;
    end;
    { Ak prichod spolu s i-tym studentom ma zmysel }
    if (vybavenie_predchadzajuceho <= b) and (t >= a) then begin
      cakanie := vybavenie_predchadzajuceho - t;
      { Keby sme mali rovnost, tak prechadzajuca varianta urcite pride skor }
      if cakanie < best_cakanie then begin
        best_cakanie := cakanie;
        best_miesto := t;
      end;
    end;
    if vybavenie_predchadzajuceho < t then vybavenie_predchadzajuceho := t + d
    else vybavenie_predchadzajuceho := vybavenie_predchadzajuceho + d;
  end;
  { Este overime, ci sa neda zaradit na koniec }
  if vybavenie_predchadzajuceho <= b then begin
    writeln(max(vybavenie_predchadzajuceho, a), ' 0');
    exit;
  end;
  if best_cakanie = 1000000123 then writeln('ZAJTRA')
  else writeln(best_miesto, ' ', best_cakanie);
end.
```

Listing programu:

```
#include <cstdio>
#include <algorithm>

using namespace std;

int n, a, b;

int main() {
  scanf("%d %d %d", &n, &a, &b);
  int vybavenie_predchadzajuceho = 0;
  int best_cakanie = 1000000123;
  int best_miesto = 1000000123;

  for (int i = 0; i < n; i++) {
    int t, d; scanf("%d %d", &t, &d);
    // Skontrolujeme, ci mame medzeru
    if (vybavenie_predchadzajuceho <= t) {
      // Skontrolujeme, ci medzera (vybavenie_predchadzajuceho, t) sa da pouzit
      if (t >= a && vybavenie_predchadzajuceho <= b) {
        // A rovno to vypiseme a skoncime (nic lepsie nenajdeme)
        printf("%d 0\n", max(a, vybavenie_predchadzajuceho));
        return 0;
```

```
      }
    }
    // Ak sa da do radu zaradit s i-tym studentom
    if (vybavenie_predchadzajuceho <= b && t >= a) {
      int cakanie = vybavenie_predchadzajuceho - t;
      // V pripade rovnosti to predtym ziskane bude mat lepsi cas
      if (cakanie < best_cakanie) {
        best_cakanie = cakanie;
        best_miesto = t;
      }
    }
    if (vybavenie_predchadzajuceho < t)
      vybavenie_predchadzajuceho = t+d;
    else
      vybavenie_predchadzajuceho += d;
  }
  // Vsetci skoncili a da sa este zaradit
  if (vybavenie_predchadzajuceho <= b) {
    printf("%d 0\n", max(a, vybavenie_predchadzajuceho));
    return 0;
  }
  if (best_cakanie == 1000000123)
    printf("ZAJTRA\n");
  else
    printf("%d %d\n", best_miesto, best_cakanie);
}
```

opravoval Žaba

3. Zlenivel nám Luxusko

(max. 7 b za popis, 3 b za program)

Vitajte pri ďalšej časti vášho obľúbeného seriálu Kontrola Správnych Programov. V dnešnej časti si povieme niečo o riešení tretej úlohy.

Spoznať miestnosti spojené priamou chodbou nie je ťažké – veď predsa informácie o chodbách dostaneme priamo na vstupe. K úspešnému riešeniu tejto úlohy si potom bolo treba uvedomiť už len jednu dôležitú vec, ktorú sa zadanie snažilo čo najviac skryť. A to tú, že z miestnosti x do miestnosti y sa dvomi chodbami dostaneme práve vtedy, ak existuje miestnosť z, s ktorou obe susedia. Teraz sa teda riešenie mení na zistenie toho, či majú dve miestnosti spoločného suseda. Otázkou však zostáva, ako to robiť efektívne.

Mnohí z vás si zobrali tabuľku $n \times n$, do ktorej si zapísali maticu susednosti. (Teda ak vedie chodba z i-tej miestnosti do j-tej, tak si na políčko v i-tom riadku a j-tom stĺpci dáme 1, inak tam bude 0.) Pomocou nej už vedeli v lineárnom čase zistiť, či existuje miestnosť z, s ktorou susedia aj miestnosť x, aj miestnosť y. (Prejdeme všetky možné z a pre každé sa pozrieme, či susedí aj s x, aj s y.) Toto riešenie však malo pamäťovú zložitosť $\Theta(n^2)$, za čo som strhával jeden bod.

Vzorové riešenie používa dynamické polia, ktorým sa dá nastavovať veľkosť počas programu. To znamená, že nebudeme mať zbytočne veľa nevyužitého miesta v pamäti.[2] Pomocou takýchto polí si zapamätáme graf tak, že v i-tom poli budú čísla tých miestností, do ktorých sa dá dostať z i-tej miestnosti. Keď dostaneme otázku, či sa dá dostať z miestnosti x do miestnosti y dvomi chodbami, tak to skontrolujeme takto: Najskôr si do pomocného poľa dĺžky n poznačíme, do ktorých miestností sa dá dostať z miestnosti x. Následne prejdeme miestnosti, do ktorých sa dá dostať z y, a zistíme, či je niektorá z nich poznačená v našom pomocnom poli.

Dynamické polia môžeme použiť aj v Pascale, len je to trochu zložitejšie, lebo ich za behu nevieme naťahovať. (Teda vedeli by sme, ale len za cenu horšej časovej zložitosti.) Musíme to teda spraviť na dve fázy. V prvej fáze si načítame všetky chodby do poľa. Počas toho si pre každú miestnosť spočítame, koľko chodieb z nej vychádza. Keď prvá fáza skončí, naalokujeme (pomocou `SetLength`) pre každú miestnosť jej pole na správnu veľkosť. Potom prejdeme cez zoznam načítaných chodieb a rozmiestnime miestnosti do správnych polí.

Pamäťová zložitosť bude teda $O(m+n)$, lebo si pamätáme každú chodbu (ktorých je dokopy m) a ešte jedno pomocné pole dĺžky n. Časová zložitosť bude $O(m + kn)$, lebo potrebujeme

[2] Nejaké nevyužité miesto mať môžeme, napr. `vector` v C++ funguje tak, že má v sebe nejakú rezervu navyše. Ale nevyužitého miesta bude menej ako využitého, teda nám to rádovú pamäťovú zložitosť nepokazí.

načítať graf, na čo treba $O(m)$ krokov, a potom každú z k otázok spracujeme v čase lineárnom od počtu vrcholov, teda v čase $O(n)$.

Posledná vec je, že zatiaľ sme len vyriešili úlohu, ako zistiť, či sú miestnosti vzdialené *práve* dve chodby. V zadaní však bolo *nanajvýš* dve. Teda musíme nájsť aj situácie, kedy sú x a y spojené chodbou, alebo dokonca $x = y$. To by sme samozrejme vedeli v programe ošetriť, vieme si však namiesto toho pomôcť úplne super fintou. Stačí, keď si upravíme graf tak, že ku každej miestnosti pridáme slučku – teda chodbu z tej miestnosti do tej miestnosti. Čo sa tým zmenilo? Zjavne nepribudli žiadne nové dvojice miestností, ktoré sú spojené najviac dvomi chodbami. Ale keď sa pozrieme na nejakú dvojicu miestností x a y, ktoré boli spojené práve jednou chodbou, tak sú teraz spojené aj práve dvomi: $x \to x \to y$. A aj každá miestnosť x je sama so sebou spojená práve dvomi chodbami: $x \to x \to x$. A toto malé zlepšenie sa mi hrozne páči, lebo ušetrí kopec škaredých if-ov.

Listing programu:

```
var graph: array [0..1047] of array of longint;
    pocet: array [0..1047] of longint;
    vstup1: array [0..10047] of longint;
    vstup2: array [0..10047] of longint;
    T:array [0..1047] of boolean;
    n,m,k,i,j,a,b: longint;
    ano: boolean;

begin
  read(n); read(m); readln(k);
  for i:=1 to n do pocet[i]:=1;
  for i:=1 to m do
    begin
      read(a); readln(b);
      inc(pocet[a]);
      inc(pocet[b]);
      vstup1[i]:=a; vstup2[i]:=b;
    end;
  for i:=1 to n do begin
    setlength(graph[i],pocet[i]);
    pocet[i]:=0;
  end;
  for i:=1 to m do begin
    graph[vstup1[i],pocet[vstup1[i]]]:=vstup2[i];
    graph[vstup2[i],pocet[vstup2[i]]]:=vstup1[i];
    inc(pocet[vstup1[i]]);
    inc(pocet[vstup2[i]]);
   end;
  for i:=1 to n do begin
    graph[i,pocet[i]]:=i; inc(pocet[i]);
  end;
  for i:=1 to k do begin
    read(a); readln(b);
    for j:=1 to n do T[j]:=false;
    for j:=0 to pocet[a]-1 do T[graph[a,j]]:=true;
    ano:=false;
    for j:=0 to pocet[b]-1 do begin
      if(T[graph[b,j]]=true) then ano:=true;
    end;
  if(ano=true) then writeln('Ano')
  else writeln('Nie');
  end;
end.
```

Listing programu:

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

vector<vector<int> > G;

int main()
{
  int n,m,k;
  scanf("%d %d %d ",&n,&m,&k);
  G.resize(n+42);
  for(int i=1; i<=n; i++) G[i].push_back(i);
  for(int i=0; i<m; i++)
```

```
  {
    int a,b;
    scanf("%d %d",&a,&b);
    G[a].push_back(b); G[b].push_back(a);
  }
  for(int i=0; i<k; i++)
  {
    vector<bool> T;
    T.resize(n+42,false);
    int x,y;
    scanf("%d %d",&x,&y);
    for(int j=0; j<G[x].size(); j++) T[G[x][j]]=true;
    bool t=false;
    for(int j=0; j<G[y].size(); j++) {if(T[G[y][j]]) t=true;}
    if(t) printf("Ano\n");
    else printf("Nie\n");
  }
  return 0;
}
```

opravoval Imp, časti pre odvážnych dodal MišoF

4. Zápasy s hokejkami

(max. 10 b za popis, 5 b za program)

Už limity v tejto úlohe nám mohli trocha napovedať, že sa budeme hrať s binárnymi zápismi čísel. Totiž, $n \leq 30$, teda hráčov je najviac 2^{30}. To sa ešte s malou rezervou zmestí do 32-bitovej znamienkovej premennej.[3] No na druhej strane, je to dosť veľké číslo a nie je preto dobrý nápad si celý strom vytvoriť v pamäti. Poďme radšej nájsť nejaký spôsob, ako z čísla hráča vypočítať jeho postup v pavúku.

V prvom kole súperia dvojice $0 \leftrightarrow 1$, $2 \leftrightarrow 3$, $\ldots$, $2k \leftrightarrow 2k+1$, $\ldots$ a v každej dvojici vyhrá ten s menším číslom. A to je zároveň ten, ktorý má číslo deliteľné dvoma. Do druhého kola teda postúpia hráči s číslami $0, 2, 4, 6, \ldots$ a zápasia v ňom proti sebe dvojice v tvare $4k \leftrightarrow 4k+2$. Vyhráva ten s menším číslom, čiže s číslom, ktoré je deliteľné štyrmi. Pokračovaním tejto úvahy sa dostaneme k tomu, že hráč vyhrá i-te kolo práve vtedy, ak je deliteľný číslom 2^i. Navyše, ak súťažiaci vyhral i kôl, v jeho podstrome pavúka sa nachádza (vrátane neho) práve 2^i hráčov. Zo všetkých týchto hráčov musí mať najdlhšiu hokejku, ináč by sa nebol prepracoval na vrch tejto skupiny. To znamená, že musí mať hokejku dĺžky aspoň 2^i.

Teraz by sme mohli vysloviť hypotézu, že práve 2^i je najkratšou možnou dĺžkou hokejky každého z hráčov, ktorí vypadnú po i kolách. Treba ale zdôrazniť, že toto tvrdenie sme zatiaľ nedokázali. Musíme ešte ukázať, že naozaj každý z týchto hráčov mohol túto hokejku naozaj mať. Presnejšie, pre každé $x > 0$ potrebujeme nájsť také rozdanie hokejok hráčom, pri ktorom má hráč x skutočne má hokejku dĺžky 2^i, kde $i = \max\{j : j \geq 0 \wedge 2^j \mid x\}$, a celý turnaj prebehne ako má.

Ako na to? Najskôr zoberme hráčov v podstrome pod hráčom x, teda hráčov s číslami $x, \ldots, x + 2^i - 1$. Im zaradom priradíme hokejky s číslami 2^i až 1. Tým zabezpečíme dve veci: každý zápas v tomto podstrome dopadne ako má, a zároveň sme hráčovi x dali hokejku dĺžky 2^i, teda najkratšiu možnú. Teraz zoberme ostatných hráčov (v poradí od 0 po $2^n - 1$ a rozdajme im zvyšné hokejky (v poradí od najväčšej po najmenšiu). Takto opäť zabezpečíme dve veci: každý zápas medzi zvyšnými hráčmi dopadne ako má, a tiež zápas, v ktorom x mal prehrať, x naozaj aj prehrá, lebo jeho protihráč práve dostal dlhšiu hokejku ako on.

Otázkou ešte zostáva, ako číslo $i = \max\{j : j \geq 0 \wedge 2^j \mid x\}$ nájsť. Jednou možnosťou je x postupne deliť dvoma, kým sa to dá bezo zvyšku. Číslo x sa dá deliť dvoma bez zvyšku najviac $\lfloor \log_2 x \rfloor$-krát. A keďže $x < 2^n$, tak $\log_2 x < \log_2 2^n = n$.

Iná možnosť je robiť bitový AND čísla x postupne s maskami 2^i pre rastúce i, až kým nenarazíme na nenulový výsledok, čo znamená, že sme našli jednotkový bit (rozmyslite si, prečo). Musíme pretestovať všetky bity čísla, ktorých je opäť $\lfloor \log_2 x \rfloor + 1 \leq n$.

Samozrejme, v oboch prípadoch treba špeciálne ošetriť hráča 0, ktorý nikdy neprehrá. O tom je ale jasné, že vždy musí mať hokejku dĺžky presne 2^n.

[3] No dobre, tá rezerva je celkom veľká, ale všimnite si, že 2^{31} by sa už nezmestilo.

Zostáva nájsť maximálnu dĺžku hokejky pre hráča x. O hráčovi 0 už vieme, že nemôže mať inú hokejku ako 2^n. Hráč, ktorý s nim priamo prehral, pochopiteľne, môže mať hokejku najviac dĺžky $2^n - 1$. S ním zase prehral ďalší, ktorý už môže mať dĺžku hokejky najviac $2^n - 2$. Nahliadnime teraz, ako sa dá zistiť, s kým hráč x prehral. Už vieme, že ak si definujeme číslo i tak, ako v predchádzajúcom texte, tak hráč x najskôr vyhral i kôl a potom v kole $(i + 1)$ prehral. No ale to znamená, že v kole $(i + 1)$ hral hráč x s hráčom y, ktorý má od x číslo o 2^i *menšie*.

To isté si môžeme sformulovať aj v dvojkovej sústave. Číslo x sa končí presne i nulami a pred nimi je jednička. Číslo y hráča, s ktorým x prehral, dostaneme tak, že túto poslednú jedničku zmeníme na nulu. Ukážeme si túto úvahu na príklade. Uvažujme hráča $53 = 110101_2$. Ten prehral hneď v prvom kole s hráčom 110100_2. Ten prehral v treťom kole s hráčom 110000_2, ktorý prehral v piatom kole s 100000_2. Nakoniec, v poslednom šiestom kole tento hráč prehral s 000000_2, ktorý je absolútnym víťazom.

Vidíme, že počet jednotkových bitov v binárnom zápise čísla x zodpovedá počtu súťažiacich, ktorí sú určite lepší ako x. (Inými slovami, reťaz víťazov od x až po 0.)

Nech b je počet jednotkových bitov v čísle x – teda počet súťažiacich, o ktorých sme našou úvahou zistili, že musia mať určite dlhšiu hokejku ako x. Potom zjavne platí, že x nemôže mať hokejku dlhšiu ako $2^n - b$. Ukážeme, že takú dlhú hokejku naozaj môže mať. Najskôr zoberieme hokejky dĺžky $2^n - b$ až 2^n a rozdáme ich ako treba – hráč x dostane hokejku dĺžky $2^n - b$, hráč, ktorý ho porazil, o jedno dlhšiu, a tak ďalej. No a následne zoberieme zvyšné hokejky a rozdáme ich zvyšným hráčom opäť podľa čísla – čím väčšie číslo hráča, tým kratšiu zo zvyšných hokejok dostane.

Počet jednotkových bitov v čísle x môžeme nájsť podobne ako pri počte núl na jeho konci: buď postupným delením (kde si pamätáme, koľkokrát nám vznikol zvyšok 1) alebo bitovým ANDom. Obe metódy vedú k časovej zložitosti $O(n)$.

Keďže musíme odpovedať na m otázok, celková časová zložitosť programu je $O(mn)$. Pamäťová zložitosť je konštantná, lebo si stačí pamätať konštantne veľa čísel.

Podrobnejšie o zložitosti

Táto časť vzorového riešenia je **len pre odvážnych**. Ak pôsobí odstrašujúco, nič si z toho nerob a pokojne ju preskoč.

Vyššie popísané odhady časovej a pamäťovej zložitosti platia za jedného veľmi dôležitého predpokladu: že sa nám n-bitové číslo zmestí do celočíselnej premennej. Ak by sme totiž túto úlohu riešili pre väčšie n, už by sa veľkosť čísel hráčov prejavila na časovej aj pamäťovej zložitosti. Očividnejšie je to u pamäťovej: na zapamätanie jedného n-bitového čísla potrebujeme $\lceil n/8 \rceil$ bajtov pamäte. Lenže následne sa to prejaví aj na zložitosti časovej: načítame síce číslo v desiatkovej sústave, zaujímajú nás ale jeho cifry v sústave dvojkovej, musíme ho teda do tejto sústavy previesť. A časová zložitosť tohto prevodu bude závisieť (aspoň lineárne!) od počtu desiatkových cifier. No a ten je priamo úmerný číslu n, teda počtu cifier dvojkových.

A nie len to. Následne by sa ukázalo, že oba postupy, ktoré sa nám zdali ekvivalentné, sú vlastne rozlične dobré. Zatiaľ čo delenie dvomi by postupne zmenšovalo celé číslo a priamočiara implementácia by bola (pre najhoršie možné x) až kvadratická od n, použitie bitového ANDu vieme robiť postupne po bajtoch, a teda časová zložitosť naozaj ostane lineárna od n.

Optimalizácia konštánt

Aj táto časť vzorového riešenia je **len pre odvážnych**. Ale zatiaľ čo tá predchádzajúca bola prudko teoretická, táto bude prudko praktická.

Na zistenie, koľkými nulami končí bitový zápis n-bitového čísla, nepotrebujeme n operácií. Ak sa nám celé číslo zmestí do premennej, stačí nám tých operácií $O(\log n)$. Prečo? Počet núl na konci čísla totiž môžeme binárne vyhľadávať.

Na to, aby sme vedeli binárne vyhľadávať, potrebujeme pre dané i zistiť, či číslo x končí aspoň i nulami. To ale vieme spraviť, a to dokonca v konštantnom čase. Napríklad tak, že otestujeme, či `((x >> i) << i) == x`. (Teda x najskôr vhodným bitovým posunom celočíselne vydelíme 2^i, potom vynásobíme 2^i a potom zistíme, či sa niečo pri delení stratilo.) Alebo tak, že otestujeme, či `(x & ((1 << i)-1)) == 0`. (Teda vyrobíme si bitovú masku, ktorá má jednotky na posledných i pozíciách, a logickým ANDom otestujeme, či sú všetky tieto pozície v x nulové.

Aj druhá operácia, ktorú potrebujeme (spočítať jednotkové bity), sa dá implementovať pomocou $O(\log n)$ aritmetických a logických operácií. Tento algoritmus je však dosť komplikovaný, preto ho nebudeme uvádzať.

Existuje však ešte efektívnejšie riešenie. Obe tieto operácie sú natoľko zaujímavé a bežne používané vo výpočtoch, že ich mnohí výrobcovia procesorov implementujú hardvérovo. Taký procesor potom má na tieto operácie priamo inštrukcie, ktoré môžeme v programe v konštantnom čase použiť.

Ako to spraviť? Napríklad v `gcc` a `g++` nájdeme dve užitočné funkcie: `__builtin_ctz(x)` a `__builtin_popcount(x)`. Prvá z nich (Count Trailing Zeros) vracia počet nulových bitov, ktorými končí x, druhá (POPulation COUNT) vracia počet jednotkových bitov v x – čiže presne tie veci, ktoré potrebujeme.[4] No a kompilátor sa už o low-level detaily postará za nás. Ak kompilujeme na architektúre, ktorá má danú procesorovú inštrukciu, priamo ju použije, a inak je táto funkcia implementovaná ako makro, ktoré porobí čo najmenej aritmetických a logických operácií a vyrobí nám želaný výsledok.

Listing programu:

```
program Priklad4;

var N, M: longint;
var x, k, i, min, max: longint;

begin
  Read(N, M);

  for k := 1 to M do begin
    Read(x);

    min := 1 shl N;
    max := 1 shl N;

    for i := N - 1 downto 0 do
      if x and (1 shl i) <> 0 then begin
        dec(max);
        min := 1 shl i;
      end;

    WriteLn(min, ' ', max);
  end;
end.
```

Listing programu:

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

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

  while (M--) {
    int x;
    scanf("%d", &x);

    int min = 1 << N, max = 1 << N;

    for (int i = N; i--; )
      if (x & 1 << i) {
        --max;
        min = 1 << i;
      }
```

[4] U oboch funkcií ide o tzv. gcc rozšírenie – nie sú teda súčasťou štandardu jazyka C ani C++.

```
    printf("%d %d\n", min, max);
  }

  return 0;
}
```

Listing programu:

```
#include <iostream>

int main() {
    int n, m, x;
    std::cin >> n >> m;
    while (m--) {
        std::cin >> x;
        std::cout << (1<<__builtin_ctz(x)) << " " << ((1<<n)-__builtin_popcount(x)) << "\n";
    }
}
```

opravoval Usamec

5. Oprav chybu

(max. 5 b za popis, 10 b za program)

Táto úloha sa dala riešiť minimálne dvoma rozumnými spôsobmi.[5]

Obidva spôsoby majú ale spoločný úvod. Potrebujeme získať dostatočné množstvo vhodného slovenského textu a vhodne ho spracovať. Slovenský text sa dá získavať rôznymi spôsobmi. Môžeme si buď ručne kopírovať text z Wikipédie či nejakých novín do textového súboru, alebo si napísať nejaký skript, ktorý to spraví za nás. Každopádne aj prvá možnosť funguje celkom dobre a netrvá až tak dlho – samozrejme pokiaľ kopírujeme dosť dlhé texty.

Ďalšou časťou je predspracovanie textu. Obe naše metódy požadujú text bez diakritiky a nezaujímajú ich iné znaky ako písmená a medzery. Navyše ani nebudeme rozlišovať medzi veľkými a malými písmenami. Preto sa zíde napísať si program, ktorý číta vstup po znakoch a upravuje ich: tie, čo nechceme (napríklad čísla a interpunkciu) mení na medzery, písmenká mení z veľkých na malé a zbavuje ich diakritiky. Možné vylepšenie: nevypisovať viac medzier po sebe.

Inou možnosťou, ako spracovať takýto text, je ručný search and replace v nejakom textovom editore alebo dobrý linuxový skript. (Odporúčame pohrať sa s programom `iconv` resp. `cstocs` pri zmene kódovania alebo odstránení diakritiky, a s programom `tr` pri mazaní, resp. `sed` pri komplikovanejšom editovaní textu.)

Poďme sa teraz už pozrieť na riešenia úlohy. Prvým pomerne dobrým spôsobom je vytvoriť si databázu najčastejších slov, tú potom vložiť priamo do zdrojáku a následne využiť. Databáza sa dá vytvoriť jednoducho: Zoberieme očistený text, rozsekáme ho na slová, tie vložíme do poľa a toto pole usporiadame. (Iné spôsoby zahŕňajú použitie dátových štruktúr, akými sú písmenkový strom alebo hashovacia tabuľka.) Potom vieme na jeden prechod spočítať početnosť každého slova a rovno si ju vypísať na výstup vo vhodnom tvare, ktorý vložíme do zdrojáku. Navyše vyhodíme slová, ktoré sa vyskytujú málo často (aby sme sa zmestili do limitu na veľkosť zdrojového súboru).

Ako môžeme takýto slovník využiť? Budeme si počítať pre každé písmeno skóre vyjadrujúce ako veľmi sa toto písmeno hodí na doplnenie. Nakoniec vyberieme to s najvyšším skóre. Vždy, keď narazíme na slovo, ktoré obsahuje _, postupne vyskúšame doplniť každé písmeno. Ak sa nám podarí vytvoriť známe slovo, tak nejakým vhodným spôsobom upravíme skóre patričného písmenka (napríklad k nemu pripočítame jednotku, počet výskytov slova, alebo nejakú kombináciu dĺžky slova a počtu výskytov).

Tu si treba ešte uvedomiť jednu vec. Overovať, či slovo existuje, by sme chceli rýchlejšie ako prechádzaním celého slovníka. Najľahší spôsob je mať zoznam slov utriedený a v ňom binárne vyhľadávať (pozrieť sa do stredu a podľa toho sa rozhodnúť, kam sa pozrieme ďalej). Opäť sa dajú využiť aj iné dátové štruktúry.

[5] čítaj: „autorovi vzoráku sú známe tieto dva“

Celé to nakoniec bude mať časovú zložitosť $O(zn \log s)$ alebo $O(zn)$ (podľa toho, aké dátové štruktúry používame) a pamäťovú zložitosť $O(s)$, kde z je počet písmen abecedy, n je veľkosť vstupu a s je veľkosť slovníka.

Hlavnou výhodou tejto metódy je jej jednoduchosť a pomerne vysoká účinnosť. Na druhej strane, v niektorých prípadoch nefunguje vôbec – napríklad keď máme chyby v relatívne neznámych slovách.

Iný spôsob je nepozerať sa na slová, ale využiť štatistické závislosti medzi jednotlivými písmenami. Napríklad to, že pravdepodobnosť výskytu písmena v slovenskom texte závisí od predchádzajúcich písmen. Napríklad po písmenách `og` bude veľmi pravdepodobné `a` (napr. máme slovo `doga`) či `r` (`program`), ale málo pravdepodobné `h` (ak práve nepíšeme o Vincentovi van Goghovi, také slovo asi nemáme). Ale naopak, po písmenách `vc` bude `h` omnoho pravdepodobnejšie ako inokedy.

Poďme teda zistiť, ako tieto závislosti fungujú. Rovnako ako v predchádzajúcom riešení si zoženieme dostatočne dlhý slovenský text, zbavíme ho interpunkcie a diakritiky a všetky písmenká zmeníme na malé. Takto dostaneme text, v ktorom už je len 27 rôznych znakov: malé písmená a medzera. Teraz si spravíme pole A rozmerov $27 \times 27 \times 27$ a v ňom si pre každý 3-písmenkový podreťazec zapamätáme, koľkokrát sa vyskytoval. (Toto je okolo 19 tisíc čísel. Celý obsah poľa sa nám neskôr zmestí ako konštanta do programu.)

To, čo nás bude zaujímať, sú vyššie popísané pravdepodobnosti. Presnejšie, keď čítam zaradom slovenský text, bude ma zaujímať, aká je pravdepodobnosť konkrétneho nasledujúceho písmena za predpokladu, že poznáme jemu predchádzajúce dve písmená. Označovať to budeme nasledovne: pravdepodobnosť, že uvidíme `b`, ak sme práve videli `ka`, označíme $P(\texttt{b} \mid \texttt{ka})$.

No a pomocou poľa A vieme ľahko vypočítať tieto pravdepodobnosti. Ak sa napríklad z trojíc tvaru `vc?` v našom slovenskom texte vyskytlo len `vca` (100×), `vce` (70×), `vci` (50×), `vco` (40×), `vch` (30×) a `vcu` (10×), tak máme dokopy 300 výskytov dvojice `vc`. Ak teraz čítame iný slovenský text a prečítame písmenká `vc`, v približne 1/3 prípadov po nich bude nasledovať `a`, zatiaľ čo v približne 1/10 prípadov to bude `h`. Teda budeme počítať s tým,[6] že $P(\texttt{a} \mid \texttt{vc}) = 1/3$ a $P(\texttt{h} \mid \texttt{vc}) = 1/10$.

Načo nám je toto dobré? Je to istý model správania sa slovenčiny. Nie je úplné presný, ale na druhej strany je dosť presný na to, aký je jednoduchý. Vieme pomocou neho napríklad generovať text. Zoberieme prvé dve písmená, aké chceme, náhodne vygenerujeme (so správnymi pravdepodobnosťami!) tretie, opäť náhodne vygenerujeme (s pravdepodobnosťami podľa druhého a tretieho písmena) štvrté, a tak ďalej. Ak by sme to naozaj skúsili, zistili by sme, že výstup síce nedáva zmysel, ale vedeli by sme ho už čítať „po slabikách“ bez toho, aby nám príliš lámal jazyk.

No ale ako nám to pomôže nájsť chýbajúce písmenko? Trik je v nasledujúcej úvahe: Majme nejaký text. Čím viac tento text „znie ako slovenčina“, tým je väčšia pravdepodobnosť, že ho vyššie popísaný náhodný generátor vygeneruje.

A presne toto použijeme. Postupne vyskúšame všetky možnosti, čím nahradiť chýbajúce písmeno. Zakaždým dostaneme nejaký text. Spomedzi týchto textov chceme vybrať ten, ktorý najviac znie ako slovenčina. A teda ten, ktorý má najväčšiu pravdepodobnosť, že ho vygenerujeme. Ale túto pravdepodobnosť predsa vieme ľahko spočítať!

Ukážeme si to na príklade. Pravdepodobnosť toho, že keď začneme s písmenami `ja`, vygenerujeme text `jano␣vari` (pričom znak ␣ predstavuje medzeru), spočítame nasledovne:

$$P(\texttt{n} \mid \texttt{ja}) \cdot P(\texttt{o} \mid \texttt{an}) \cdot P(\texttt{␣} \mid \texttt{no}) \cdot P(\texttt{v} \mid \texttt{o␣}) \cdot P(\texttt{a} \mid \texttt{␣v}) \cdot P(\texttt{r} \mid \texttt{va}) \cdot P(\texttt{i} \mid \texttt{ar})$$

Toto riešenie má ešte jeden technický zádrheľ: násobenie veľa malých pravdepodobností nám kvôli zaokrúhľovacím chybám zväčša rado vyrobí nulu. Tu si ale vieme pomôcť tak, že namiesto

[6] V skutočnosti sa oplatí drobná nepresnosť: každej trojici písmen napr. pridáme 0.01 výskytu. Toto je dobré na to, aby nás príliš nezaskočilo, ak sa v novom texte vyskytne trojica písmen, ktorá za sebou v starom texte nikdy nebola.

pravdepodobnosti vygenerovania slova budeme počítať jej logaritmus. No a logaritmus súčinu čísel je súčet logaritmov tých čísel.

Samotné riešenie používa navyše ešte jednu drobnosť. Nepočíta zbytočne pravdepodobnosti, ktoré sú pre každú možnosť rovnaké, alebo inými slovami povedané počíta len pravdepodobnosti okolo vynechaných znakov.

Celková časová zložitosť bude celkom príjemná: $O(zn)$. Pamäťové nároky sú $O(z^3)$.

Listing programu:

```
int counts[] =
{0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,
  /* a dalsia kopa cisel, na webe sa da stiahnut cely zdrojak */};

#include <cstdio>
#include <cstdio>
#include <cmath>
#include <vector>

using namespace std;
#define BUFSIZE 2000
char buf[BUFSIZE];

int main() {
  // Pamet na posledne 2 znaky (0-25 su a-z, 26 je medzera, 27 vynechane pismeno)
  char last2 = 26;
  char last = 26;
  // Nacitanie vstupu, pocas neho si zapametame, kazdu trojicu, ktora obsahuje
  // vynechanie
  vector<vector<int> > triplets;
  while (true) {
    int rc = fread(buf, sizeof(char), BUFSIZE, stdin);
    if (rc <= 0) break;
    for (int i = 0; i < rc; i++) {
      char cur = buf[i];
      if (cur == '_') {
        cur = 27;
      } else if (cur >= 'a' && cur <= 'z') {
        cur = cur - 'a';
      } else if (cur >= 'A' && cur <= 'Z') {
        cur = cur - 'A';
      } else {
        cur = 26;
      }
      if (last2 == 27 || last == 27 || cur == 27) {
        vector<int> triplet(3);
        triplet[0] = last2; triplet[1] = last; triplet[2] = cur;
        if (last == 26 && cur == 27)
          triplet[0] = 26;
        if (last == 26 && last2 == 27)
          triplet[2] = 26;
        triplets.push_back(triplet);
      }
      last2 = last;
      last = cur;
    }
  }

  // Pseudopocet (smoothing factor) - nechceme veciam davat 0 pravdepodobnost
  double smooth = 0.1;

  // Spocitame pravdepodnost dvojic
  double doubs[27][27];
  for (int i = 0; i < 27; i++) {
    for (int j = 0; j < 27; j++) {
      doubs[i][j] = 0;
      for (int k = 0; k < 27; k++)
        doubs[i][j] += counts[27*27*i+27*j+k]+smooth;
    }
  }

  bool first = true;
  double mc = 0;
  int mi = 0;

  // Pre kazde pismeno spocitame relevantnu cast pravdepodobnosti a vyberieme
  // najlepsiu.
  for (int i = 0; i < 26; i++) {
    double pc = 0;
```

```
    for (int j = 0; j < triplets.size(); j++) {
      vector<int> tr = triplets[j];
      for (int k = 0; k < 3; k++) {
        if (tr[k] == 27)
          tr[k] = i;
      }
      pc += log(counts[tr[0]*27*27+tr[1]*27+tr[2]]+smooth)-log(doubs[tr[0]][tr[1]]);
    }

    if (pc > mc || first) {
      first = false;
      mc = pc;
      mi = i;
    }
  }
  printf("%c\n", 'a'+mi);
}
```

opravoval JaNo

6. Odplata a omeleta

(max. 13 b za popis, 7 b za program)

Ako býva v takýchto úlohách zvykom, skúsime spraviť z konkrétnej otázky všeobecnú. Nebudeme sa pýtať, koľko nás stojí trafenie všetkých n vtákov, ale **za akú najmenšiu omeletovú cenu vieme trafiť prvých i z nich** (počítané zľava). Toto číslo si označme $C[i]$.

Prvých 0 vtákov dokážeme zasiahnuť za cenu 0, preto $C[0] = 0$.

Predstavme si, že už máme našu otázku zodpovedanú od 0 po $i-1$ a teraz na ňu chceme odpovedať pre i. Vieme, že i-teho vtáka musí zasiahnuť nejaké vajce, inak by ostal nezasiahnutý. Keďže vtáci napravo od neho nás zatiaľ nezaujímajú, môžeme toto vajce hodiť čo najviac doľava tak, aby len tesne zasiahlo aj i-teho vtáka. Nemá totiž zmysel špiniť drôt za ním.

Nech i-teho vtáka trafíme j-tym druhom vajca. Týmto vajcom možno navyše zasiahneme ešte niekoľko ďalších vtákov naľavo od i-teho. Prvého, ktorého už nezasiahneme, označme p; ak zasiahneme všetkých, bude $p = 0$. Inými slovami, p je najmenšie také číslo, že $(p+1)$-vého vtáka trafí vajce hodené na i-teho vtáka.

Prvých p vtákov vieme pokryť s cenou $C[p]$ (keďže $p < i$, túto hodnotu už máme vypočítanú), teda ak použijeme j-te vajce, tak $C[i]$ bude $C[p] + o_j$. Vypočítame túto hodnotu pre každý z m druhov vajec a vyberieme najmenšiu.

Na samom konci budeme mať našu otázku zodpovedanú aj pre n, čo je presne odpoveď na pôvodnú úlohu.

Jediným problémom je, ako rýchlo a efektívne vypočítať p. Toto bola hlavná pointa, v ktorej sa vaše riešenia líšili. Niektorí z vás vo for-cykle postupne skúšali čoraz menšie hodnoty p, čo ale môže trvať až $O(n)$, preto ich riešenie malo časovú zložitosť $O(mn^2)$. (Toto bolo za 9 bodov.) Iní použili šikovnejšie binárne vyhľadávanie, čo ich doviedlo k zložitosti $O(mn \log n)$ (11 bodov).

Vzorák: Uvedomme si, že s rastúcim i hodnota p pre fixný druh vajca neklesá – interval, ktorý zasiahne vajce určené i-temu vtákovi, sa posúva doprava. To môžeme využiť na zrýchlenie riešenia.

Pre každý typ vajca si zapamätáme naposledy nájdenú hodnotu p. Keď potom zväčšíme i (úsek vtákov, ktorých chceme zasiahnuť), novú hodnotu p pre j-ty druh vajca nájdeme postupným zväčšovaním starej hodnoty.

Ako sme na tom s časovou zložitosťou? Môže sa nám stať, že budeme pre nejaké i a j zväčšovať hodnotu p o „veľa“ (teoreticky až o n). Dokopy však pre každý druh vajca zvýšime hodnotu p najviac n-krát, čo je spolu $O(mn)$ krokov.

Časová zložitosť celého algoritmu je preto $O(mn)$. Ak ste navyše spotrebovali $O(m+n)$ pamäte, dostali ste plný počet bodov.

Listing programu:

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

```
#define MAXN 50047
#define MAXM 147
#define LINF 1000111000111000111LL
typedef long long ll;

ll n, m, X[MAXN], D[MAXM], O[MAXM], C[MAXN], P[MAXM];
// n, m, pozicia vtaka, rozsah a cena vajca, cena za prvych k vtakov, P

int main() {
    scanf("%lld %lld", &n, &m);
    for (int i = 0; i < n; ++i) {
        scanf("%lld", X + i);
        C[i + 1] = LINF;
    }
    C[0] = 0;
    for (int i = 0; i < m; ++i) {
        scanf("%lld %lld", D + i, O + i);
        P[i] = 0;
    }
    for (int i = 1; i <= n; ++i)
        for (int j = 0; j < m; ++j) {
            // tu sa starame o P
            while (X[P[j]] + D[j] < X[i - 1])
                P[j]++;
            C[i] = min(C[i], C[P[j]] + O[j]);
        }
    printf("%lld\n", C[n]);
}
```

opravoval Luxusko

7. Opičí hnev

(max. 12 b za popis, 8 b za program)

Hra, ktorú Mišo a Jano hrajú, je *kombinatorická*. Kombinatorickú hru spoznáte tak, že:

- ju hrajú dvaja hráči, ktorí sa striedajú,
- v každom momente hry vedia obaja hráči všetko o aktuálnom stave hry,[7]
- pre každú možnú pozíciu v hre sú pevne určené pozície, na ktoré sa z nej dá dostať,
- sú definované pozície, v ktorých hra končí; posledný hráč, ktorý ťahal, vyhral/prehral (závisí od pravidiel),
- hra vždy skončí po konečnom počte ťahov.

Skontrolujme našu hru – striedajú sa dvaja hráči; pozícia v hre je daná len nahnevanosťou obete; stav hry je jednoznačne určený pozíciou a hráčom na ťahu a tieto údaje vedia obaja; z pozície x sa vieme dostať len do nejakej pozície z množiny $\{2x, 3x, \ldots, 9x\}$; hra končí v pozíciách s nahnevanosťou väčšou alebo rovnou n, pričom posledný ťahajúci vyhral; naša hra evidentne niekedy skončí, nakoľko nahnevanosť vždy aspoň zdvojnásobíme.

V takýchto hrách väčšinou existujú pozície, z ktorých si hráč na ťahu dokáže vhodnou stratégiou vynútiť víťazstvo, budeme ich nazývať *vyhrávajúce* (V), ostatné pozície budú *prehrávajúce* (P).

My potrebujeme zistiť, či je počiatočná pozícia (1) V alebo P, a podľa toho vyhrá Mišo alebo Jano. Platí, že pozície, v ktorých hra končí, sú P[8] – hráč v predošlom ťahu nahneval obeť aspoň n a hráč na ťahu prehral. Ak z pozície x existuje ťah do P pozície, vieme tam potiahnuť a dostať súpera do pozície, z ktorej nevie vyhrať, preto si x označíme ako V. Ak z pozície x nevieme potiahnuť do žiadnej P pozície, znamená to, že každý ťah vedie do pozície, z ktorej súper vie vyhrať, preto x bude prehrávajúca pozícia.

Teraz už vieme priamočiaro napísať rekurzívny algoritmus, ktorý zistí, či je pozícia V alebo P: ak sme v pozícii väčšej alebo rovnej n, vrátime P. Inak sa zavoláme na všetky pozície, do ktorých sa vieme dostať, a ak je aspoň jedna z nich P, vrátime V, inak P.

[7] žiadne karty, ktoré druhý hráč nevidí, ani nič podobné

[8] Špeciálny prípad nastáva, keď $n = 1$. Vyhrať by mal vtedy podľa mňa Mišo, napokon on bude stále ten, čo obeť ako prvý nahnevá aspoň n. Nestrhával som ale body ani keď ste tvrdili, že vyhrá Jano, ani keď ste sa nad tým nezamysleli a ani v testovacích vstupoch tento zákerný prípad nebol.

Tento algoritmus s exponenciálnou časovou zložitosťou sa dal ľahko zmeniť na lineárny pridaním tabuľky veľkosti n, či už memoizáciou (nebudeme tú istú pozíciu zisťovať viackrát, ale iba prvýkrát a výsledok si zapamätáme) alebo dynamickým programovaním odzadu (vždy sa potrebujeme pozrieť iba na 8 väčších pozícií, ktoré už budeme mať zistené).

Lepšie riešenie využíva to, že V a P pozície sa budú striedať v pekných intervaloch. Všetky pozície z intervalu $[n, \infty)$ sú P. Ak je interval $[x, y)$ P a $y > 2(x-1)$, tak interval $[\lceil x/9 \rceil, x)$ bude V, lebo čísla z neho vieme prenásobiť tak, aby sme súpera dostali do P. Ak je interval $[x, y)$ V a $y > 9(x-1)$, tak interval $[\lceil x/2 \rceil, x)$ bude celý P, lebo keď ktorékoľvek číslo z neho vynásobíme ktorýmkoľvek možným číslom $(2, 3, \ldots 9)$, dostaneme súpera do V. Všimnite si, že nové intervaly, ktoré týmto postupom získavame, spĺňajú potrebné nerovnosti, a preto môžeme postup na nich zopakovať.

Pôjdeme odzadu od n, budeme postupne spracovávať celé intervaly naraz a pozerať, či sú P alebo V. Nakoniec zistíme, v akom type intervalu sa nachádza pozícia 1. Keďže spodnú hranicu vždy znížime aspoň o polovicu, vykonáme $O(\log n)$ krokov, čo bude aj časová zložitosť algoritmu, keďže každý interval spracujeme v $O(1)$.

Alternatívne riešenie od Jakuba Šafina spočíva vo vygenerovaní si možných pozícií, na ktoré sa vôbec dá z 1 násobením číslami $2, 3, \ldots, 9$ dostať. Všetky takéto čísla sú v tvare $2^a \cdot 3^b \cdot 5^c \cdot 7^d$ (2, 3, 5 a 7 sú prvočísla menšie ako 10, žiadne iné prvočíslo nemôže deliť pozíciu, na ktorú sa dá dostať naším spôsobom), pričom pre exponenty platí $a, b, c, d \leq \log m$, kde m je rovné najväčšej hodnote n na vstupe. Potom hrubý horný odhad počtu rôznych pozícií do m je $(\log m)^4$, keď si povieme, že každé z a, b, c, d môže nadobúdať hodnoty $1, \ldots, \log m$ nezávisle na sebe.

Pre každú dosiahnuteľnú pozíciu x spustíme naše predchádzajúce riešenie a výsledok (kto vyhrá v prípade, že limit nepríčetnosti chudáka je x) si zapamätáme. Pole dosiahnuteľných pozícií zotriedime podľa ich veľkosti. Potom keď budeme chcieť odpovedať, kto vyhrá pre zadané n, pozrieme sa na najmenšiu dosiahnuteľnú pozíciu n', ktorá nie je menšia ako n (nájdeme ju binárnym vyhľadávaním). Hodnotu n totiž opičiaci prekonajú práve vtedy, keď prekonajú hodnotu n'.

Predpočítanie bude trvať $O((\log m)^5)$ a na jednu otázku vieme odpovedať v čase $O(\log \log m)$. Pre veľa otázok nám teda rýchlejšie odpovedanie na ne vynahradí čas strávený predpočítavaním. Pamäťová zložitosť je $O((\log m)^4)$.

Listing programu:

```
#include <iostream>

using namespace std;

int main(void){
    long long n;
    while(cin >> n){
        bool v;
        if(n == 1)
            v = 1;
        else{
            v = 0;
            while(n > 1){
                if (v)
                    n = (n+1)/2;
                else
                    n = (n+8)/9;
                v ^= 1;
            }
        }
        if (v)
            cout << "Miso" << endl;
        else
            cout << "Jano" << endl;
    }
}
```

8. O slonoch v škodovke

opravoval Tomi
(max. 15 b za popis, 10 b za program)

V úlohách, kde sa hľadá najkratšia cesta, sa väčšinou dá momentálna situácia vystihnúť jediným údajom – miestom, kde sa práve nachádzame. Tu sú tie údaje dva: kde sa nachádzame, a koľko vodičákov nám ešte zostalo.

Predstavme si $k+1$ kópií pôvodného grafu usporiadaných do vrstiev; každú vrstvu asociujeme s určitým počtom vodičákov (od 0 do k). Keď sa rozhodneme prejsť po nejakej ceste, máme dve možnosti: buď počkáme na zelenú, čím sa dostaneme do vrcholu s rovnakým počtom vodičákov (zostaneme v tej istej vrstve), alebo prejdeme na červenú – prídeme o vodičák, ale bude to rýchlejšie.

Tým pádom máme normálny orientovaný graf s $n \cdot (k+1)$ vrcholmi a s rôznymi (ale nezápornými) dĺžkami hrán, v ktorom chceme nájsť najkratšiu cestu do nejakého cieľa. To znie ako práca pre Dijkstrov algoritmus. Ten vyzerá takto:

Počas behu algoritmu sa vrcholy delia do troch skupín. Buď sme k vrcholu už našli zaručene najkratšiu cestu (budeme ich volať *hotové*), alebo sme k nemu našli zatiaľ len najkratšiu z ciest vedúcich iba cez hotové vrcholy (len nevieme, či je to aj úplne najkratšia cesta), alebo ešte vôbec nevieme, ako sa k nemu dostať. Na začiatku je hotový iba vrchol, z ktorého začíname (a vzdialenosť k nemu je nula). Všetkým ostatným vrcholom dáme vzdialenosť nekonečno (resp. také veľké číslo, že to je prakticky nekonečno).

Potom kým ešte sú nejaké nehotové vrcholy, opakujeme toto: spomedzi vrcholov z tej strednej skupiny vyberieme ten s najmenšou vzdialenosťou od štartu. Cesta, čo sme k nemu našli, už je určite najlepšia (pretože keby sa k nemu dalo dostať aj skratkou, tá skratka by musela ísť nejakým iným nehotovým vrcholom, a potom by sme vybrali ten iný vrchol, a nie tento). Takže tento vrchol môžeme prehlásiť za hotový. Ak má nejakých nehotových susedov, ku ktorým sa cez tento vrchol dá dostať rýchlejšie, ako sme si zatiaľ mysleli, tak si to zapamätáme.

Praktická implementácia Dijkstru vyzerá väčšinou takto: z tých troch skupín vrcholov nás zaujíma iba tá stredná, lebo hotové už sú hotové, a o tých neznámych nič nevieme. Vrcholy z tejto strednej skupiny si budeme pamätať v prioritnej fronte zoradenej podľa ich vzdialenosti. Vďaka tomu budeme vedieť rýchlo zisťovať, ktorý vrchol z tej strednej skupiny je najbližšie. (Prioritnú frontu si pozrite na Wikipédii, ja ju neviem.)

Niekedy sa stane, že vrchol už v prioritnej fronte je, ale chceli by sme mu zmenšiť vzdialenosť, a naša prioritná fronta to nedokáže. V takom prípade tam napríklad môžeme ten vrchol vložiť druhýkrát, vyberieme ich už v správnom poradí. (Teda najprv ten s menšou vzdialenosťou. Keď vyberieme vrchol a zistíme, že už sme preňho našli aj lepšiu vzdialenosť, tak ho odignorujeme.)

Lenže moment. Dijkstrov algoritmus predpokladá, že hrany majú vopred určené vzdialenosti, ale v našom grafe sú semafory. Keď prechádzame z bodu A do bodu B, závisí to nielen od vzdialenosti medzi nimi, ale aj od času, kedy sme prišli do bodu A. Dá sa Dijkstrov algoritmus napriek tomu použiť?

Dá. Semafory sa našťastie správajú pomerne rozumne – keď meriame dĺžku cesty z A do B a chceme sa ňou do B dostať čo najskôr, vždy sa nám oplatí čo najskôr vyraziť z A. (Nemáme nikdy dôvod „len tak“ čakať.) A túto záruku nám Dijkstrov algoritmus vie dať: pozerá sa na dĺžky hrán iba vtedy, keď je v nejakom hotovom vrchole a pozerá sa, kade ďalej. Vtedy už je isté, že poznáme najkratšiu cestu do A (preto je to hotový vrchol). V tejto situácii z toho teda nevznikne žiaden problém a inokedy sú mu dĺžky hrán ukradnuté, takže bude fungovať ako normálne.

Zložitosť Dijkstrovho algoritmu je normálne $O((E+V)\log V)$, kde V je počet vrcholov (u nás $n(k+1)$) a E je počet hrán (u nás $m(2k+1)$), takže celková zložitosť je $O(k(n+m)\log(kn))$.

Listing programu:

```
#include <cstdio>
#include <vector>
#include <queue>
```

```
using namespace std;

typedef long long ll;

struct Hrana {
  int kam;
  ll cena;
  ll start;
  ll inter;
};

struct Vrchol {
  vector<Hrana> next;
  ll best;
  Vrchol() : best(1000000000000000000ll) {}
};

int main() {
  int n, m, k; scanf("%d %d %d", &n, &m, &k);
  k++;
  // v poli "v" usporiadame vrcholy po k vrstvach (cisla 0 az n-1 budu stavy,
  // kde sme stratili 0 vodicakov, n az 2n-1 budu 1 vodicak, atd.)
  vector<Vrchol> v(n*k);

  for (int i = 0; i < m; i++) {
    Hrana h;
    int a, b;
    scanf("%d %d %Ld %Ld %Ld", &a, &b, &h.cena, &h.start, &h.inter);
    a--; b--;
    // hrany vo vrstve
    for (int i = 0; i < k; i++) {
      h.kam = i*n+b;
      v[i*n+a].next.push_back(h);
      h.kam = i*n+a;
      v[i*n+b].next.push_back(h);
    }
    // hrany do dalsej vrstvy (start a inter dame tak, aby sme nikdy necakali)
    Hrana h2;
    h2.cena = h.cena;
    h2.start = 0;
    h2.inter = 1;
    for (int j = 0; j < k-1; j++) {
      h2.kam = (j+1)*n+b;
      v[j*n+a].next.push_back(h2);
      h2.kam = (j+1)*n+a;
      v[j*n+b].next.push_back(h2);
    }
  }

  // do haldy budeme davat minus vzdialenost, lebo priority_queue normalne vybera najvacsi prvok
  priority_queue<pair<ll, int> > halda;
  halda.push(make_pair(0, 0));
  while (!halda.empty()) {
    ll dist = -halda.top().first;
    int x = halda.top().second;
    halda.pop();
    if (v[x].best <= dist) {
      continue;   // uz sme tento vrchol navstivili lepsou cestou
    }
    v[x].best = dist;
    for (int i = 0; i < v[x].next.size(); i++) {
      // vypocitame, kolko treba cakat
      ll d = v[x].next[i].start;
      ll d2 = dist+v[x].next[i].cena;
      ll wait;
      if (d > d2) {
        wait = d-d2;
      } else {
        ll e = v[x].next[i].inter;
        ll mod = (d2-d)%e;
        if (mod == 0) mod = e;
        wait = e - mod;
      }
      d2 += wait;
      // ak sme nasli kratsiu cestu, zapamatame si ju a pridame ju do haldy
      if (d2 < v[v[x].next[i].kam].best) {
        v[v[x].next[i].kam].best = d2+1;
        halda.push(make_pair(-d2, v[x].next[i].kam));
      }
    }
  }
```

```
  ll best = v[n-1+(k-1)*n].best;
  for (int i = 0; i < k; i++) {
    best = min(best, v[n*i+n-1].best);
  }
  printf("%Ld\n", best);
  return 0;
}
```

vzorák písal Žaba

1::9. Tisíce podpostupností

(max. 10 b za popis, 15 b za program)

Túto úlohu sme síce označili číslom 9, ale bola naozaj ľahká. Poďme sa teda spolu pozrieť, ako sa mala riešiť.

Aby sme zarátali každú skáčucu podpostupnosť práve raz, budeme ich počítať systematicky – podľa toho, v ktorom prvku pôvodnej postupnosti končia. Počet skáčucich podpostupností končiacich v k-tom prvku označme d_k. Najprv zistíme hodnotu d_1, potom d_2, ... až d_n, pričom si budeme pomáhať už vypočítanými hodnotami (ako pri dynamickom programovaní).

Nech už máme vypočítané $d_1, \ldots, d_{k-1}$ a zaujíma nás hodnota d_k. Jednou zo skáčucich podpostupností končiacich v k-tom prvku je samotný k-ty prvok. Ostatné vzniknú tak, že vezmeme nejakú skáčucu podpostupnosť končiacu v j-tom prvku a doplníme za jej koniec k-ty prvok, pričom musí platiť $j < k$ a $a_j \neq a_k$.

Koľko je takých podpostupností, ktoré môžeme doplniť? Jednoduchým, ale pomalým riešením by bolo prejsť všetky prvky $a_1, \ldots, a_{k-1}$ a sčítať hodnoty d_j pre tie j, pre ktoré $a_j \neq a_k$. Pozrime sa na tento súčet z opačnej strany: sú tam vlastne všetky d_j okrem tých, kde $a_j = a_k$.

Budeme si preto pamätať súčet $s = d_1 + \ldots + d_{k-1}$ a tiež pre každú *hodnotu* x počet skáčucich podpostupností končiacich v nejakom prvku a_j s hodnotou x (teda súčet vhodných d_j), ktorý označíme p_x. Hodnotu d_k potom dostaneme ako $1 + s - p_{a_k}$. Následne k premenným s a p_{a_k} pričítame d_k (aby sme zachovali ich význam) a môžeme pokračovať výpočtom d_{k+1}.

Treba ešte doriešiť, ako si budeme pamätať p_x – hodnoty a_k sú totiž z priveľkého rozsahu na to, aby nám stačilo obyčajné pole. Slušní ľudia by použili `map`-u alebo by si hodnoty a_k prečíslovali do rozsahu $1, \ldots, n$, ale keďže U$amec je zvrhlík zaťažený na hash tabuľky, na plný počet bodov ste museli použiť tie.

Očakávaná časová zložitosť je v tomto prípade $O(n)$, lebo na výpočet každej z hodnôt d_k potrebujeme (vďaka hash tabuľke) len konštantný čas. Pamäťová zložitosť je tiež $O(n)$, práve kvôli spomínanej hash tabuľke. Všetky čiastkové počty podpostupností budeme počítať modulo 1 000 000 007, preto sa nám pohodlne zmestia do 32-bitových premenných.

Listing programu:

```
#include <cstdio>
#include <algorithm>
#include <tr1/unordered_map>
using namespace std;
using namespace std::tr1;

#define MOD 1000000007

int main() {
    int n, sucet = 0;
    scanf("%d", &n);
    unordered_map<int, int> P;
    for (int i = 0; i < n; i++) {
        int a;
        scanf("%d", &a);
        int d = (1 + sucet - P[a] + MOD) % MOD;  // "+ MOD" aby sme nedostali zaporne cislo
        sucet = (sucet + d) % MOD;
        P[a] = (P[a] + d) % MOD;
    }
    printf("%d\n", sucet);
}
```

1::10. Totálna fúzia

opravoval Usamec
(max. 15 b za popis, 10 b za program)

Upozornenie: v tomto vzoráku predpokladáme, že už vieš, ako sa efektívne hľadá minimálna kostra grafu (a čo to vlastne je). Ak túto podmienku nespĺňaš, odporúčame ti najprv si naštudovať Primov alebo Kruskalov algoritmus.

Úlohu si prevedieme na graf nasledovným spôsobom: Vrcholy grafu (množina V) budú jednotlivé letiská. Hrany grafu (množina E) budú tvorené letmi a ich cena bude rovná nákladom na presun letu do inej spoločnosti. Navyše si dodefinujeme ďalšie množiny hrán: $E_1, E_2, \ldots, E_\ell$, kde v množine E_i sú všetky lety prevádzkované spoločnosťou i. Všetky hrany v E_i majú cenu 0.

Teraz chceme zistiť, koľko prostriedkov musíme vynaložiť na to, aby sme po zrušení všetkých spoločností okrem i-tej dostali súvislý graf letov. To, čo v tomto prípade hľadáme, je minimálna kostra grafu $G_i = (V, E \cup E_i)$ (niektoré lety budeme mať dvakrát, to nám však riešenie nepokazí, lebo ak by sme taký let chceli použiť, vyberieme si tú verziu s cenou 0). Minimálna kostra nie je nič iné ako najlacnejší podgraf G_i, v ktorom existuje cesta medzi každou dvojicou vrcholov. Minimálnu kostru vieme hľadať v čase $O(f \log f)$ pomocou Kruskalovho alebo Primovho algoritmu. Keby sme vyskúšali všetky možné i, tak týmto spôsobom by sme dostali riešenie, ktoré beží v čase $O(\ell f \log f)$, čo je v niektorých prípadoch celkom veľa.

Rýchlejšie riešenie dostávame pomocou nasledujúceho tvrdenia:
Nech K je nejaká minimálna kostra grafu (V, E). Potom existuje taká minimálna kostra grafu $G_i = (V, E \cup E_i)$, že každá jej hrana patrí do E_i alebo do kostry K.

Dôkaz urobíme sporom. Nech K_i je taká minimálna kostra grafu G_i, ktorá obsahuje čo najmenej *zlých* hrán (t. j. takých, ktoré patria do $E \setminus K$). Nech e je nejaká zlá hrana, teda leží v minimálnej kostre K_i a zároveň nepatrí do E_i ani do kostry K.

Keď odstránime hranu e z kostry K_i, graf sa nám rozpadne na dva komponenty A, B. Pozrime sa na hrany z kostry K, ktoré vedú medzi A a B. Každá z týchto hrán má ostro väčšiu cenu ako e – v opačnom prípade by sme totiž mohli najlacnejšiu z nich použiť namiesto hrany e v kostre K_i a dostali by sme tak novú kostru grafu G_i, ktorá by nebola drahšia ako K_i a navyše by obsahovala menej zlých hrán, čo je spor.

Pozrime sa teraz na kostru K. Keď do nej pridáme hranu e, vznikne v nej cyklus. Na tomto cykle je určite ešte aspoň jedna hrana (okrem e), ktorá vedie medzi vrcholom z A a vrcholom z B, označme ju h. V predchádzajúcom odseku sme si ukázali, že takéto hrany sú drahšie ako e, preto keď vymeníme v kostre K hranu h za hranu e, dostaneme lacnejšiu kostru grafu (V, E) ako K, čo je spor.

Dokázané tvrdenie nám dáva celkom jasný návod, ako postaviť rýchlejšie riešenie. Dalo by sa zhrnúť nasledovne:

- Nájdi najlacnejšiu kostru K grafu (V, E).
- Pre každé i nájdi najlacnejšiu kostru grafu $(V, K \cup E_i)$ a z nich vyber najlepšie riešenie.

Hlavné zlepšenie spočíva v tom, že v druhom kroku už neberieme v každej iterácii do úvahy všetkých f hrán z E, ale len $n - 1$ hrán kostry K.

Prvý krok vieme urobiť v čase $O(f \log f)$ pomocou Kruskalovho alebo Primovho algoritmu.[9] Keby sme bez rozmyslu robili druhý krok priamo napr. Primovým algoritmom, tak dostaneme zložitosť $\sum_i O((n - 1 + |E_i|) \log(n - 1 + |E_i|)) = O((\ell n + f) \log f)$. Pri priamom použití Kruskalovho algoritmu by zložitosť bola viac-menej rovnaká.

Tu ale istým spôsobom robíme zbytočnú robotu navyše. V každej iterácii znovu triedime hrany z kostry K. To robiť netreba. Navyše cena hrán z E_i je vždy nula, takže ich môžeme dať dopredu. Keď si hrany z kostry K utriedime vopred, tak jediné čo nám zostáva, je použiť

[9] Síce existujú aj rýchlejšie algoritmy, ale tie sú už strašný grc...

union-find a to bude mať zložitosť: $O((\ell n + f) \cdot \alpha(f))$, kde α je inverzná Ackermannova funkcia (niečo veľmi pomaly rastúce; pre rozumne veľké čísla je to menej ako 5).

Celkovo máme časovú zložitosť $O(f \log f + (\ell n + f) \cdot \alpha(f))$.

Listing programu:

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

using namespace std;

int boss[2000];
int size[2000];

int clear(int n) {
  for (int i = 0; i < n; i++) {
    boss[i] = i;
    size[i] = 1;
  }
}

int find(int x) {
  if (boss[x] == x) return x;
  return boss[x] = find(boss[x]);
}

bool join(int a, int b) {
  int ba = find(a);
  int bb = find(b);
  if (ba == bb) return false;
  if (size[ba] > size[bb]) {
    boss[bb] = ba;
    size[ba] += size[bb];
  } else {
    boss[ba] = bb;
    size[bb] += size[ba];
  }
  return true;
}

int main() {
  // Pre kazdu aerolinku zoznam letov
  vector<vector<pair<int, int> > > aerolinky;

  // (cena, (odkial, kam))
  vector<pair<int, pair<int, int> > > hr;
  vector<pair<int, pair<int, int> > > kostra;
  int n, f, l;
  scanf("%d %d %d", &l, &n, &f);
  aerolinky.resize(l);
  for (int i = 0; i < f; i++) {
    int a, b, c, p;
    scanf("%d %d %d %d", &a, &b, &p, &c);
    a--; b--; p--;
    hr.push_back(make_pair(c, make_pair(a,b)));
    aerolinky[p].push_back(make_pair(a,b));
  }

  // Prvy kruskal na globalnu kostru
  clear(n);
  sort(hr.begin(), hr.end());
  for (int i = 0; i < hr.size(); i++) {
    if (join(hr[i].second.first, hr[i].second.second)) {
      kostra.push_back(hr[i]);
    }
  }
  // Bonus: Kostru uz mame usortenu :)

  int best = 2000000123;

  for (int i = 0; i < l; i++) {
    clear(n);
    int cena = 0;
    // Najprv prihadzeme hrany s cenou 0
    for (int j = 0; j < aerolinky[i].size(); j++) {
      join(aerolinky[i][j].first, aerolinky[i][j].second);
    }

    // Potom zbytok
```

```
    for (int j = 0; j < kostra.size(); j++) {
      if (join(kostra[j].second.first, kostra[j].second.second)) {
        cena += kostra[j].first;
      }
    }

    best = min(best, cena);
  }
  printf("%d\n", best);
}
```

2::9. Trans-formácia

opravoval MišoF a bolelo ho to
(max. 8 b za popis, 17 b za program)

Vitajte pri čítaní tohto vzorového riešenia!
Čo v ňom nájdete:
– prečo máme všetci (vrátane organizátorov) za túto úlohu nula bodov
– keby všetko fungovalo, ako rýchlo a bezbolestne získať aspoň nejaké body
– ako opraviť nefunkčné riešenia a dostať riešenie správne a takmer dostatočne rýchle

Prečo máme všetci nulu?

Preto, lebo túto úlohu nevieme spravodlivo ohodnotiť. Stala sa totiž vec nemilá, všetci – ako vy, riešitelia, tak aj my, organizátori – sme sa nechali oklamať riešením, ktoré je síce elegantné a efektívne, ale hlavne ***nesprávne***. No a čo čert nechcel, prejavilo sa to až na veľkých vstupoch, ktoré sme už nevedeli overiť pomalým správnym riešením. No a keďže sme tým ovplyvnili aj vás, ťažko už teraz povedať, čo by sa zmenilo, keby boli počas času na riešenie v testovači správne výstupy.

V rámci tohto vzoráku si okrem iného popíšeme, ako to rýchle riešenie fungovalo, prečo nebolo správne, a ako sa dá (takmer) opraviť. V súčasnosti by už v testovači mali byť správne výstupy. Ak si vyskúšate odovzdať úlohu teraz, rýchle nesprávne riešenie nedostane skoro žiadne body. Ale o tom viac neskôr.

Odhady zložitosti

Okrem premenných p a z (počet predpôn a počet slovných základov) budeme v našich odhadoch zložitosti používať ešte aj iné premenné: maximálnu dĺžku slov ℓ, veľkosť použitej abecedy σ a maximálny počet „vnorených“ predpôn / slovných základov α.

Poslednú z premenných vysvetlíme podrobnejšie. Občas sa nám stane, že niektorá predpona je prefixom inej, napr. `pre` a `pred`. Práve z takýchto dvojíc predpôn vznikajú situácie, kedy to isté výsledné slovo vieme naskladať viacerými spôsobmi. No a občas sa stane, že existuje aj postupnosť viacerých predpôn taká, že každá z nich je prefixom všetkých nasledujúcich. A to isté pre slovné základy a ich sufixy. Napríklad slovné základy `side`, `roadside` a `broadside`. Premenná α bude udávať dĺžku najdlhšej takejto postupnosti. (Pre práve uvedený príklad by bolo $\alpha = 3$. Zjavne vždy $\alpha \leq \ell$.)

V zadaní úlohy bolo $\ell = 50$ a $\sigma = 26$. Naše opravené riešenie nie je dostatočne rýchle vo *všetkých* takýchto prípadoch, len v tých, kedy je navyše α rozumne malá.

Rýchle body

Táto úloha patrí k úlohám, za ktoré je (ak samozrejme všetko funguje, ako má) hanba mať nulu. Riešenie „vyskúšam všetky možnosti“ je aj v silách riešiteľa kategórie Z.

Aj skúšať všetky možnosti sa však dá rôzne efektívne. Úplne najviac priamočiare riešenie by vyzeralo: „pre všetky predpony P, pre všetky základy Z, ak som slovo PZ ešte nevyrobil, pridaj ho do poľa už vyrobených slov“. Takéto riešenie by za normálnych okolností získalo tak 2 body.

Ostrieľanejší riešiteľ už určite vidí, kde robíme zbytočnú prácu: za slovami „ak som slovo PZ ešte nevyrobil“ sa skrýva prechádzanie celého poľa a kontrolovanie. Takéto riešenie má teda časovú zložitosť $O(p^2z^2\ell)$, lebo v najhoršom prípade vyrobíme pz rôznych slov, každé s každým porovnáme a na každé porovnanie treba $O(\ell)$ krokov.

A prečo je táto práca zbytočná? Lebo v dobrom programovacom jazyku vieme *s ešte menšou námahou* vyrobiť *efektívnejšie riešenie*: stačí vyrobené slová nahádzať do setu (množiny). Set sa nám sám postará o odstránenie duplikátov, a navyše to spraví rádovo efektívnejšie.

Ak použijeme usporiadanú množinu (ako napr. `set` v C++ alebo `TreeSet` v Jave – implementované ako vyvažovaný binárny strom), dostávame riešenie so zaručenou časovou zložitosťou $O(pz\log(pz)\ell)$. Ak použijeme neusporiadanú množinu (ako napr. `set` v Pythone, `unordered_set` v C++ alebo `HashSet` v Jave – implementované ako hashovacia tabuľka), dostaneme riešenie s očakávanou časovou zložitosťou $O(pz\ell)$.

Takéto riešenia by za normálnych okolností získali 4-5 bodov, plus ďalšie 2 za popis.

Listing programu:

```
#include <iostream>
#include <vector>
#include <string>
#include <tr1/unordered_set> // v novom C++11 standarde staci <unordered_set>, namespace std;
using namespace std;
using namespace std::tr1;

int main() {
    int P; cin >> P;
    vector<string> prefixy(P); for (int p=0; p<P; ++p) cin >> prefixy[p];
    int Z; cin >> Z;
    vector<string> zaklady(Z); for (int z=0; z<Z; ++z) cin >> zaklady[z];

    unordered_set<string> slova;
    for (int p=0; p<P; ++p) for (int z=0; z<Z; ++z) slova.insert( prefixy[p] + zaklady[z] );
    cout << slova.size() << endl;
}
```

Listing programu:

```
from sys import stdin
P = int(stdin.readline())
predpony = [ stdin.readline().strip() for x in range(P) ]
Z = int(stdin.readline())
zaklady = [ stdin.readline().strip() for x in range(Z) ]

slova = set()
for predpona in predpony:
    for zaklad in zaklady:
        slova.add( predpona + zaklad )
print len(slova)
```

Hlbšia analýza

Ako to už chodí, lepšie riešenie nájdeme tak, že sa podrobnejšie pozrieme na štruktúru problému, ktorý riešime.

Čomu sa rovná odpoveď? No, je to zhruba pz, mínus nejaké duplikáty. Keď pochopíme, ako vznikajú, možno nám to pomôže spočítať, koľko ich je.

Kedy sa môže stať, že nám to isté slovo vznikne viacerými spôsobmi? Potrebujeme na to dve rôzne predpony, ktoré rovnako začínajú (presnejšie, jedna musí byť prefixom druhej) a zároveň dva rôzne slovné základy, ktoré rovnako končia (presnejšie, jeden musí byť sufixom druhého). A navyše musí platiť, že dlhšia predpona „prečnieva“ o ten istý reťazec ako dlhšia prípona.

Príklad: predpony `k` a `kol`, základy `olmica` a `mica`. Dlhšia predpona + kratší základ vyrobia to isté slovo ako kratšia predpona + dlhší základ.

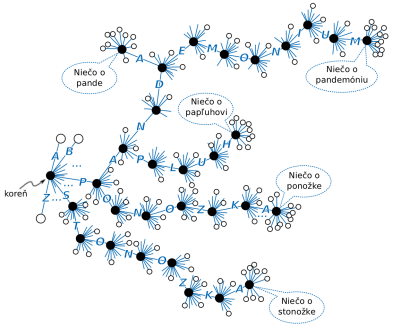
Bude teda treba nejak vhodne identifikovať takéto situácie. Potom sa zamyslíme ďalej, čo s nimi vlastne spraviť.

Ako teda nájsť všetky dvojice predpôn, z ktorých jedna je prefixom druhej? Vo všeobecnosti to príliš dobre nejde: keď máme množinu rôzne dlhých reťazcov, môže sa stať, že budú tvaru napr.: `a`, `ab`, `abc`, ... a teda každá dvojica bude k sebe pasovať.

V našom prípade nám však pomôže, že naše reťazce sú krátke. Ku každej predpone môže existovať nanajvýš $\ell - 1$ iných predpôn, ktoré sú jej prefixami. (Stručný odhad: pre $\ell \leq 50$ a $p \leq 100\,000$ to znamená, že v ľubovoľnom testovacom vstupe bude menej ako 5 miliónov dvojíc „k sebe pasujúcich“ predpôn. To je ešte stále rozumne málo.)

Písmenkové stromy

Rozumným spôsobom, ako uložiť množinu reťazcov, je písmenkový strom (trie): zakorenený strom, v ktorom má každý vrchol toľko synov, koľko písmen má použitá abeceda. Každá cesta z koreňa dodola teda zodpovedá jednému reťazcu. Na obrázku si môžete pozrieť umelecké stvárnenie tejto dátovej štruktúry.



Obr. 1: Rozkošné znázornenie písmenkového stromu od Kamily Součkovej.

Všetky dvojice k sebe pasujúcich predpôn vieme teraz zostrojiť nasledovne: Najskôr vložíme do písmenkového stromu postupne všetky predpony. Potom pre každú predponu prejdeme cestu z jej vrcholu do koreňa. Cestou stretneme práve všetky vrcholy, ktoré zodpovedajú iným predponám, ktoré sú jej prefixami.

Takto teda vieme nájsť všetky dvojice „k sebe pasujúcich“ predpôn v zjavne optimálnom čase $O(p\ell)$.

To isté vieme spraviť aj so slovnými základmi, s jedinou zmenou: keďže chceme, aby na seba pasovali ich konce, do druhého písmenkového stromu budeme slovné základy vkladať odzadu, od posledného písmenka k prvému.

Spojenie polovíc dokopy NESPRÁVNE

Potiaľto sa nás ešte dostalo veľa. A tu sme sa všetci nechali zlákať na tú istú nesprávnu cestu. Ako vyzerala?

Pre ľubovoľný reťazec x označme $\pi(x)$ počet dvojíc predpôn, ktoré k sebe pasujú a dlhšia z nich sa od kratšej líši práve o reťazec x. Podobne označme $\rho(x)$ počet dvojíc slovných základov, ktoré k sebe pasujú a líšia sa o x. Napríklad pre slovné základy `liste`, `iste`, `ste`, `ihla` a `hla` by bolo $\rho(\texttt{li}) = 1$ a $\rho(\texttt{i}) = 2$.

Nesprávne riešenie teraz prehlásilo nasledovné: Celkový počet duplikátov môžeme spočítať tak, že cez všetky možné x sčítame $\pi(x) \cdot \rho(x)$.

Na prvý pohľad to vyzerá správne: Máme $\pi(x)$ dvojíc predpôn takých, že jedna je o reťazec x dlhšia od druhej, podobne máme $\rho(x)$ dvojíc slovných základov takých, že jeden je o x dlhší od druhého. No a vždy, keď si vyberieme jednu z $\pi(x)$ dvojíc predpôn P_1, P_2 a jednu z $\rho(x)$ dvojíc slovných základov Z_1, Z_2, tak z nich budeme vedieť vyrobiť nejaké jedno slovo dvoma rôznymi spôsobmi.

Nesprávne riešenie teda na výstup vypísalo hodnotu $pz - \sum_x \pi(x)\rho(x)$.

V čom je problém? V tom, že toto riešenie nespočíta počet duplikátov. Spočíta len počet spôsobov, ako vyrobiť duplikát. A to nie je to isté. Rozdiel vzniká, akonáhle sa to isté výsledné slovo dá vyrobiť *viac ako dvoma* spôsobmi.

Pozrime sa napríklad na nasledovný vstup: predpony `a`, `ab` a `abc`; slovné základy `bcd`, `cd` a `d`. Nesprávne riešenie by dospelo k nasledovnému záveru: $\pi(\texttt{b}) = \pi(\texttt{c}) = \pi(\texttt{bc}) = 1$ a zároveň $\rho(\texttt{b}) = \rho(\texttt{c}) = \rho(\texttt{bc}) = 1$. Preto existujú 3 spôsoby ako vyrobiť duplikát, a teda máme $3 \cdot 3 - 3 = 6$ rôznych slov.

To ale nie je pravda. V tomto prípade totiž nejde o tri rôzne slová, z ktorých sa každé dá vyrobiť dvomi spôsobmi. Ide tu o jedno slovo (slovo `abcd`), ktoré sa dá vyrobiť tromi rôznymi spôsobmi: ako `a` + `bcd`, ako `ab` + `cd` aj ako `abc` + `d`. A v takomto prípade už vyššie popísaný vzorec nedá správnu odpoveď. V skutočnosti pre tento vstup bude mať výsledný jazyk nie 6, ale 7 rôznych slov.

Spojenie polovíc dokopy SPRÁVNE

Vyššie uvedený vzťah nebol zase úplne zlý – vedie aspoň k správnemu riešeniu pre $\alpha = 2$. Len ho ešte potrebujeme rozšíriť na situácie, kedy sa môže stať, že to isté slovo vieme naskladať viac ako dvomi spôsobmi.

Na to, aby sme si boli istí výsledkom, musíme slová výsledného jazyka spočítať nejak systematicky. Jednou možnosťou, ako takýto systém môže vyzerať, je nasledovný algoritmus: Pre každú predponu P spočítame, koľko je slov, ktoré sa dajú vyrobiť z predpony P a nejakého slovného základu, ale nedajú sa vyrobiť zo žiadnej kratšej predpony.

Napríklad v príklade z predchádzajúcej časti by sme výsledné slovo `abcd` zarátali len pre predponu `a`, ale už nie pre predponu `ab` ani pre `abc`.

Trik je samozrejme v tom, že to celé musíme spraviť efektívne. Nemôžeme si napríklad dovoliť pre každú predponu prejsť všetky slovné základy a vyhádzať tie z nich, ktoré už nevyrobia nové slová.

Tým správnym nástrojom bude kombinatorický trik nazvaný *princíp inklúzie a exklúzie*. Keďže to bude trochu komplikovanejšie, celé si to ukážeme na príklade.

Predpokladajme, že medzi mnohými predponami máme aj tieto štyri: `a`, `ab`, `abc` a `abcd`. Ukážeme si, ako spočítať, koľko nových slov vznikne z `abcd`. („Nových“ znamená takých, ktoré sa nedajú vyrobiť z `a`, `ab`, ani `abc`.)

Všetkých slov, ktoré vieme vyrobiť z `abcd`, je samozrejme z – keď za `abcd` pridáme rôzne slovné základy, dostaneme rôzne slová. Od týchto z slov ale potrebujeme odčítať tie, ktoré sa dajú vyrobiť aj z kratšej predpony. Takže:

Init. Počet nových slov inicializujeme na z.
M1. Odčítame počet slov, ktoré sa dajú vyrobiť z `a` aj `abcd`.
M2. Odčítame počet slov, ktoré sa dajú vyrobiť z `ab` aj `abcd`.
M3. Odčítame počet slov, ktoré sa dajú vyrobiť z `abc` aj `abcd`.

Lenže samozrejme toto ešte nie je úplne ono. Uvažujme napríklad situáciu, kedy medzi slovnými základmi máme aj tieto tri: `bcdsrnka`, `dsrnka` a `srnka`. Potom sme ale slovo `abcdsrnka` odčítali dvakrát: aj v kroku M1, aj v kroku M3.

Takéto slová, ktoré sa dali vyrobiť presne tromi spôsobmi, musíme zase pripočítať (aby neboli odpočítané dvakrát, ale len jedenkrát):

P1. Pripočítame počet slov, ktoré sa dajú vyrobiť z `a`, `ab` aj `abcd`.
P2. Pripočítame počet slov, ktoré sa dajú vyrobiť z `a`, `abc` aj `abcd`.
P3. Pripočítame počet slov, ktoré sa dajú vyrobiť z `ab`, `abc` aj `abcd`.

A už je to dobre? Ešte nie. Problém je so slovami, ktoré mohli vzniknúť všetkými štyrmi spôsobmi – napr. slovo `abcdzajac`, ak by sme mali slovné základy `bcdzajac`, `cdzajac`, `dzajac` aj

`zajac`. Slovo `abcdzajac` sme trikrát odčítali (M1, M2, M3) a potom trikrát pripočítali (P1, P2, P3), takže zatiaľ vlastne nie je zohľadnené vôbec. Aby sme dostali správny výsledok, musíme takéto slová odčítať od aktuálneho počtu:

M4. Odčítame počet slov, ktoré sa dajú vyrobiť z `a`, `ab`, `abc` aj `abcd`.

Ostáva nám vymyslieť, ako tieto počty slov rátať rýchlo. Naše riešenie teda bude zložené z dvoch fáz: najskôr predspracujeme slovné základy, a potom pre každú predponu použijeme vyššie popísaným spôsobom princíp inklúzie a exklúzie. Obe tieto fázy budú mať rádovo rovnakú časovú zložitosť: počet predpôn (resp. základov), krát 2^α, krát niečo rozumne malé.

Ako spoznáme slová, ktoré sa dajú vyrobiť napr. z predpôn `jelena`, `jelenabc` aj `jelenabcd`? Zjavne spoločný prefix (v tomto prípade `jelena`) je irelevantný. Takéto slovo vznikne vtedy a len vtedy, keď máme vhodnú trojicu slovných základov: tretí (ten najkratší) je ľubovoľný, druhý je `d`+tretí a prvý je `bc`+druhý.

Takže keby sme si nejako pamätali počet reťazcov x takých, že x, `d`x aj `bcd`x sú platné slovné základy, tak by sme vedeli odpoveď na našu otázku.

Najstručnejší spôsob, ako popísať takúto trojicu reťazcov, je „`bc,d`“ – teda postupne zľava doprava vymenujeme reťazce, o ktoré sa jednotlivé slovné základy líšia. Toto bude teda *kľúč* do dátovej štruktúry, v ktorej si budeme ku nemu pamätať počet výskytov v našom zozname slovných základov.

Ukážme si teda na väčšom príklade, ako budú vyzerať dáta, ktoré chceme naším predspracovaním vyrobiť. Uvažujme osem slovných základov: `havran`, `bcdsrnka`, `dsrnka`, `srnka`, `bcdzajac`, `cdzajac`, `dzajac` a `zajac`. Potom chceme, aby napríklad platilo:

„`b,c,d`“ je priradený počet 1 (je len jedno x také, že x, `d`x, `cd`x aj `bcd`x je slovný základ)
„`bc,d`“ je priradený počet 2 (sú dve také x: `srnka` aj `zajac`)
„`bcd`“ je priradený počet 2 (opäť sú dve také x: `srnka` aj `zajac`)
„`bc`“ je priradený počet 2 (a opäť sú dve také x: `dsrnka` aj `dzajac`)
„`b,cd`“ je priradený počet 1 (tentokrát vyhovuje len `zajac`)

Ako túto hrôzu efektívne vyrátať? Pre každý slovný základ x vygenerujeme a zarátame tie *kľúče*, pre ktoré je x najdlhším slovným základom. Keď teda máme konkrétne x, zistíme si, ktoré iné slovné základy sú jeho sufixami. (Ak máme slovné základy napr. uložené v písmenkovom strome od konca, toto vieme spraviť v $O(\ell)$.) Takto dostaneme maximálny kľúč pre tento slovný základ. Všetky ostatné z neho vzniknú tak, že vynecháme niektoré čiarky a prípadne všetko od niektorej čiarky ďalej.

Napr. pre `bcdsrnka` by sme v našom príklade dostali maximálny kľúč „`bc,d`“ a z neho následne vyrobili ešte „`bcd`“ a „`bc`“. Pre `bcdzajac` by sme dostali maximálny kľúč „`b,c,d`“ a z neho vyrobili: „`b,cd`“, „`bc,d`“, „`bcd`“, „`b,c`“, „`bc`“ a „`b`“.

Takto teda spracujeme všetky slovné základy. Následne budeme prechádzať všetky predpony a v princípe pre ne budeme robiť presne to isté: Pre danú predponu nájdeme všetky iné, ktoré sú jej prefixami. Tým dostaneme maximálny kľúč, z ktorého vyrobíme všetky menšie. Každý z nich zodpovedá jednému pričítaniu/odčítaniu pri výpočte správneho počtu nových slov. Znamienko určíme podľa počtu čiarok.

Keďže to celé znie pomerne komplikovane, odporúčame stráviť nejaký čas štúdiom nasledujúceho programu, ktorý tento algoritmus implementuje. Namiesto písmenkového stromu sme v ňom použili hashmapy – sú o niečo pomalšie, ale výrazne pohodlnejšie a navyše nežerú toľko pamäte ako (neoptimalizovaný) písmenkový strom.

Listing programu:

```
from sys import stdin

# nacitame vstup:
P = int(stdin.readline())
predpony = set([ stdin.readline().strip() for x in range(P) ])
Z = int(stdin.readline())
zaklady = set([ stdin.readline().strip() for x in range(Z) ])
```

```
# urobime si hashmapu, ktora ku kazdemu klucu priradi pocet prislusnych n-tic slovnych zakladov
from collections import defaultdict
pocty = defaultdict(int)

# combinations: pre dany maximalny kluc "zoz" vyrobi vsetky kluce
# odporucame spustit a porovnat vystupy:
# print [ x for x in combinations( ["aa","bb","cc"], True ) ]
# print [ x for x in combinations( ["aa","bb","cc"], False ) ]

def combinations(zoz,mazat_spredu):
    n = len(zoz)
    if n==0: return
    for i in range(2**(n-1)):
        ans, cur = [], zoz[0]
        for j in range(n-1):
            if i & (2**j):
                ans.append(cur)
                cur = zoz[j+1]
            else:
                cur += zoz[j+1]
        ans.append(cur)
        yield ans
    if mazat_spredu:
        for x in combinations(zoz[1:],True): yield x
    else:
        for x in combinations(zoz[:-1],False): yield x

# prejdeme slovne zaklady, pre kazdy z nich vyrobime a zaratame vsetky mozne kluce
for z in zaklady:
    # najdeme indexy do "z" kde zacinaju slovne zaklady
    pre = [0]
    for i in range(1,len(z)):
        suffix_z = z[i:]
        if suffix_z in zaklady: pre.append(i)

    # vyrobime z nich maximalny kluc
    pre = [ z[pre[x]:pre[x+1]] for x in range(len(pre)-1) ]

    # zapocitame vsetky kluce ktore z neho vzniknu
    for key in combinations(pre,False): pocty[ tuple(key) ] += 1

# v tomto okamihu moze byt poucne dat si vypisat vypocitane pocty:
# for x in pocty: print x,'->',pocty[x]

# prejdeme predpony a pre kazdu vypocitame pocet slov, ktore idu vyrobit z nej,
# ale nejdu vyrobit zo ziadnej kratsej
answer = 0
for p in predpony:
    answer += Z
    # najdeme indexy do "p" kde koncia predpony
    suf = []
    for i in range(1,len(p)):
        prefix_p = p[:i]
        if prefix_p in predpony: suf.append(i)
    suf.append(len(p))

    # vyrobime z nich maximalny kluc
    suf = [ p[suf[x]:suf[x+1]] for x in range(len(suf)-1) ]

    # pre kazdy kluc, co sa da z neho vyrobit, pri/odpocitame prislusny pocet slov
    for key in combinations(suf,True): answer += (-1)**len(key) * pocty[ tuple(key) ]

# vypiseme vysledok
print answer
```

opravoval Bob

2::10. Tiramisu

(max. 10 b za popis, 15 b za program)

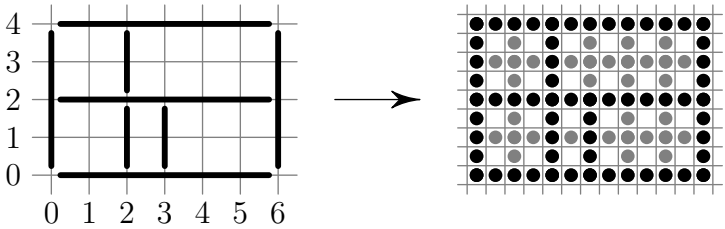
Táto úloha sa pôvodne vyskytla v súťaži Spotify Programming Challenge 2011 pod názvom Pie Squares. V mene KSP ďakujem spoločnosti Scrool za jej poskytnutie[10] a Lukášovi za pomoc pri komunikácii, lebo ja po švédsky neviem ani slovo.

[10] Aby som licenčným podmienkam učinil zadosť, úloha je ďalej distribuovaná pod CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0/) a kompletnú sadu testovacích vstupov si môžete stiahnuť na http://www.ksp.sk/wiki/uploads/Zadania/vstupy29-2-10.zip.

Jednoduché riešenie – stačí si to nakresliť

Na vstupe máme zadanú sadu úsečiek a našou úlohou je vypísať obsahy vzniknutých obdĺžnikov. Aby nám neprekážali záporné súradnice, posuňme si všetky úsečky tak, aby bol ľavý dolný roh tiramisu (to je ten s najmenšími súradnicami) v bode $[0, 0]$.

Úsečky si „nakreslime“ do dvojrozmerného poľa. Bodu $[x, y]$ bude zodpovedať políčko $[2x, 2y]$, vodorovnej úsečke medzi bodmi $[x, y]$ a $[x+1, y]$ políčko $[2x+1, 2y]$ a zvislej úsečke medzi bodmi $[x, y]$ a $[x, y + 1]$ zase políčko $[2x, 2y + 1]$. Napríklad vstup zo zadania by vyzeral takto:



Teraz už len potrebujeme spočítať obsahy jednotlivých obdĺžnikov. Môžeme to urobiť vhodným prehľadávaním (flood fill), no v našom prípade stačí pomocou dvoch while-cyklov nájsť ku každému ľavému dolnému rohu obdĺžnika zodpovedajúci pravý horný roh.

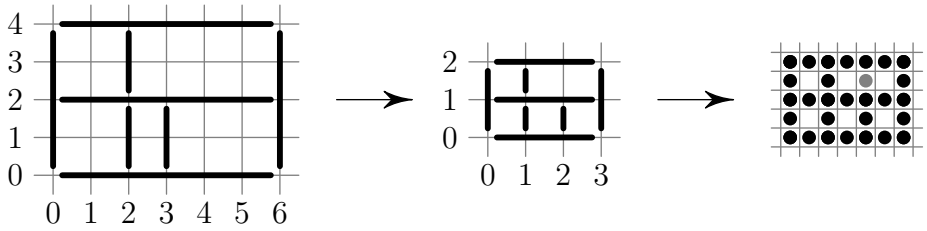
Označme si rozmery tiramisu w a h. Pretože obsahy obdĺžnikov musíme nakoniec ešte usporiadať podľa veľkosti triedením, časová zložitosť tohto riešenia je $O(wh+n \log n)$. Pamäťová zložitosť je $O(wh)$.

Takéto riešenie je pre rozmery tiramisu, aké sa mohli vo vstupoch vyskytnúť ($2 \cdot 10^9 \times 2 \cdot 10^9$), zjavne príliš neefektívne. Dá sa však ľahko nakódiť a získa 3 body (plus niečo za popis).

Kompresia súradníc

Všimnite si, že vo vstupe zo zadania sú súradnice koncov úsečiek zbytočne „nafúknuté“. Žiadna úsečka napríklad nemá koncový bod s x-ovou súradnicou rovnou 1. Môžeme preto všetky body s x-ovou súradnicou väčšou ako 1 posunúť o jeden doľava. Tým sme zmenšili celkovú šírku tiramisu, no zmenili sme aj plochu niektorých obdĺžnikov. Aby sme to napravili, budeme si pamätať, že pôvodná šírka štvorčekov medzi súradnicami 0 a 1 bola dva, nie jeden (a teda že to vlastne nie sú štvorčeky, ale obdĺžničky).

Takúto úvahu môžeme urobiť aj vo všeobecnosti. Najprv zistíme, na ktorých x-ových súradniciach naozaj ležia nejaké konce úsečiek; označme si ich $x_0, x_1, \ldots, x_p$ tak, že platí $x_0 < x_1 < \ldots < x_p$. Potom x-ové súradnice koncov úsečiek skomprimujeme: z x_i sa stane i. Navyše si zapamätáme, že vzdialenosť medzi novými súradnicami i a $i + 1$ je $x_{i+1} - x_i$. Rovnaký postup použijeme aj na y-ové súradnice.



Rôznych x-ových súradníc, na ktorých naozaj ležia konce úsečiek, je najviac $2n$; rovnako vieme ohraničiť aj počet rôznych y-ových súradníc. Po kompresii má preto tiramisu rozmery najviac $2n \times 2n$ a časová zložitosť nášho algoritmu sa zlepší na $O(n^2)$. Riešenie hodné 6 bodov.

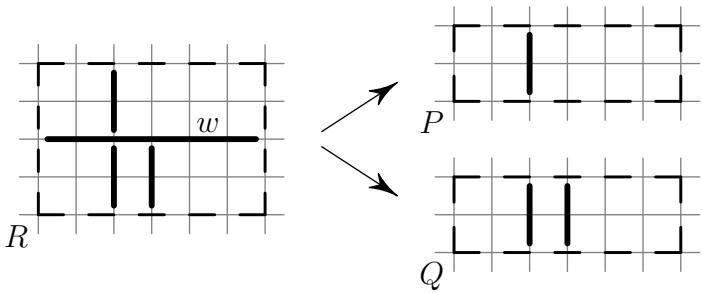
V stopách Georgovho noža

Pod týmto poetickým nadpisom sa skrýva rekurzívne riešenie, ktoré sa bude snažiť zopakovať Georgov postup pri krájaní tiramisu. Odteraz už nebudeme rezy kresliť do poľa. Najprv nájdeme štyri úsečky, ktoré ohraničujú celý obdĺžnik, a vyhodíme ich. Na zvyšok zavoláme funkciu, ktorej úlohou bude vygenerovať zoznam kúskov.

Funkcia dostane ako vstupné argumenty obdĺžnik R a zoznam úsečiek (rezov), ktoré ho delia na menšie časti. Ak je tento zoznam prázdny, výstupom funkcie bude jediný kúsok – celý zadaný obdĺžnik R.

V opačnom prípade vznikol niektorý z rezov ako prvý, označme ho w. Tento rez rozdelil obdĺžnik R na dve časti: na ľavú a pravú (ak bol zvislý), alebo na hornú a dolnú časť (ak bol vodorovný); označme ich P, Q. Koncové body w teda ležia na protiľahlých stranách R. Mohlo by sa stať, že takých rezov bude viacero, no vtedy budú mať rovnaký smer (inak by sa pretínali) a my si vyberieme ľubovoľný z nich.

Nájdeme rez w a rozdelíme zvyšné rezy do dvoch skupín podľa toho, či ležia v obdĺžniku P alebo Q. Na oba obdĺžniky a ich prislúchajúce skupiny úsečiek potom zavoláme našu rekurzívnu funkciu.



Ako sme na tom s časovou zložitosťou? V každom volaní funkcie prejdeme lineárne cez všetky úsečky, ktoré máme v argumente. Výsledná zložitosť teda závisí od spôsobu, ako sa úsečky delia do skupín P a Q – v ideálnom prípade rovnomerne, vtedy sa nám bude zoznam rezov zmenšovať na polovicu a dosiahneme zložitosť $O(n \log n)$.

Ak by sa však úsečky delili napríklad spôsobom „žiadna a všetky“, v každej úrovni rekurzie by sa ich počet zmenšil len o jednu (konkrétne w). V najhoršom prípade má teda toto riešenie zložitosť $O(n^2)$. Na väčšine vstupov však beží celkom svižne – u nás by získalo 9 bodov, v Spotify Programming Challenge dokonca plný počet.

Zametanie

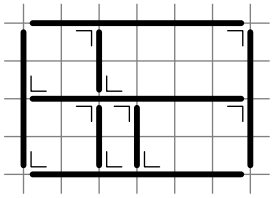
Nakoniec sa dostávame k vzorovému riešeniu. Ako už napovedá nadpis, budeme zametať, čo sa pri podobných úlohách stáva často. Aj väčšina vašich riešení bola z tohto súdka.

Začnime triviálnymi pozorovaniami: Každý kúsok rozrezaného tiramisu má svoj ľavý dolný roh a svoj pravý horný roh. Tieto dva rohy ho jednoznačne určujú. V každom bode môže mať svoj ľavý dolný roh najviac jeden obdĺžnik (v tom istom bode ale môže mať iný obdĺžnik svoj pravý horný roh), podobne pre pravý horný roh.

Až na malý počet špeciálnych prípadov (tie štyri rezy, ktoré ohraničujú celé tiramisu) platí, že ľavý koniec každej vodorovnej úsečky a dolný koniec každej zvislej úsečky je ľavým dolným rohom nejakého obdĺžnika (a nijak inak ľavý dolný roh nevznikne). Podobné kritérium platí aj pre pravé horné rohy.

Nájsť všetky ľavé dolné a všetky pravé horné rohy teda vieme, zostáva posledná úloha – určiť každému rohu jeho zodpovedajúci náprotivok.

No a tu sa dostáva k slovu zametanie. Zotriedime všetky vygenerované rohy podľa ich x-ovej súradnice – zľava doprava; v prípade rovnosti súradníc pôjdu najprv pravé horné, až potom ľavé dolné rohy. V tomto poradí ich budeme postupne spracovávať (zametať) a počas toho upravovať množinu „neuzavretých obdĺžnikov“ S (t. j. ľavé dolné rohy, ktoré sme už spracovali, ale zatiaľ sme nespracovali im zodpovedajúce pravé horné rohy).



Ľavý dolný roh spracujeme jednoducho: pridáme ho do S.

Keď príde rad na pravý horný roh r, musíme mu nájsť v S jeho náprotivok ℓ. To bude ale vďaka poradiu, v akom zametáme rohy, ľahké: ℓ musí mať y-ovú súradnicu menšiu ako r, no čo najväčšiu (určite neexistuje v S taký ľavý dolný roh, ktorý by mal y-ovú súradnicu medzi ℓ a r). Potom vypočítame plochu obdĺžnika určeného rohmi $[\ell, r]$ a vymažeme ℓ z S.

Množinu S si v C++ reprezentujeme dátovou štruktúrou `map`, v ktorej budú rohy usporiadané podľa y-ovej súradnice, aby sme vedeli hľadať ℓ príkazom `lower_bound` v logaritmickom čase. Celková časová zložitosť (triedenie rohov, zametanie a triedenie výstupu) preto bude $O(n \log n)$, pamäťová zložitosť $O(n)$ a riešenie získa 15 bodov.

Záver

Fajn, tak ešte nie úplne záver, ešte tu máme jedno riešenie. Ale už len krátko, sľubujem.

Prejdeme postupne všetky ľavé dolné rohy a ku každému nájdeme zodpovedajúci ľavý horný a pravý dolný roh, z týchto údajov už vieme vypočítať plochu obdĺžnika. Zodpovedajúci ľavý horný roh nájdeme binárnym vyhľadávaním vo vhodne zotriedenom zozname všetkých ľavých horných rohov – má rovnakú x-ovú súradnicu ako ľavý dolný roh a je od neho prvý v smere rastúcej y-ovej súradnice. Zodpovedajúci pravý dolný roh nájdeme podobne.

Toto riešenie má tiež zložitosť $O(n \log n)$ a na rozdiel od predchádzajúceho sa dá pohodlne implementovať aj v jazykoch, ktoré neponúkajú ekvivalent dátových štruktúr `set`/`map`.

A ešte jedna drobnosť: výsledné plochy sa nemusia vojsť do 32-bitových premenných, bolo preto treba použiť `long long`-y.

Tak. Dúfam, že ste počas čítania tohto vzoráku skonzumovali nejaké to tiramisu (ja som počas jeho písania zošrotoval minimálne päť). Ak aj nie, pochutnajte si aspoň na kóde zametacieho riešenia:

Listing programu:

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

struct Event {  // udalostami pri zametani budu lavy dolny a pravy horny roh
    int x, y;
    bool start;  // lavy dolny: true, pravy horny: false
    Event(int ix, int iy, bool istart) : x(ix), y(iy), start(istart) {}
};

bool operator < (const Event &a, const Event &b) {
    return a.x < b.x || (a.x == b.x && !a.start && b.start);
}

int main() {
    int n;
    scanf("%d", &n);
    vector<int> X1(n), Y1(n), X2(n), Y2(n);
    for (int i = 0; i < n; ++i) {
        scanf("%d%d%d%d", &X1[i], &Y1[i], &X2[i], &Y2[i]);
        if (X1[i] > X2[i])
            swap(X1[i], X2[i]);
```

```
        if (Y1[i] > Y2[i])
            swap(Y1[i], Y2[i]);
    }
    // pozicia celeho tiramisu (obdlznik pred delenim)
    int sx = *min_element(X1.begin(), X1.end()), ex = *max_element(X2.begin(), X2.end());
    int sy = *min_element(Y1.begin(), Y1.end()), ey = *max_element(Y2.begin(), Y2.end());

    vector<Event> E;
    E.push_back(Event(sx, sy, true));
    E.push_back(Event(ex, ey, false));
    for (int i = 0; i < n; ++i)
        // styri okrajove usecky sme uz vyriesili osobitne
        if (!(X1[i] == sx && Y1[i] == sy) && !(X2[i] == ex && Y2[i] == ey)) {
            E.push_back(Event(X1[i], Y1[i], true));
            E.push_back(Event(X2[i], Y2[i], false));
        }
    sort(E.begin(), E.end());

    map<int, int> S;  // y-ovym suradniciam lavych dolnych rohov priraduje ich x-ove suradnice
    vector<long long> result;
    for (int i = 0; i < (int) E.size(); ++i)
        if (E[i].start) {
            S[E[i].y] = E[i].x;
        } else {
            map<int, int>::iterator el = S.lower_bound(E[i].y);
            --el;  // chceme prvy ostro nizsi roh
            result.push_back((long long) (E[i].x - el->second) * (E[i].y - el->first));
            S.erase(el);
        }

    sort(result.rbegin(), result.rend());
    for (int i = 0; i < (int) result.size(); ++i)
        printf("%lld\n", result[i]);
}
```


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