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

Vzorové riešenia 1. kola zimnej časti

1. Zapáčiť sa Táni

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

Priemer sa rovná súčet deleno počet, takže stačí načítať vstup do poľa, vypočítať súčet a potom pre každého KSPáka zistiť, či je jeho šmrncovnosť vyššia ako priemer. Toto vieme spraviť na zopár `for`-cyklov, takže časová zložitosť je $O(n)$, a keďže šmrncovnosti si pamätáme v poli, pamäťová zložitosť je tiež $O(n)$.

Ak sa chceme vyhnúť reálnym číslam (lebo tam môžu vzniknúť zaokrúhľovacie chyby a je dobré používať ich iba tam, kde to fakt treba), máme viaceré možnosti: buď môžeme použiť celočíselné delenie so zaokrúhľovaním nadol (vyskúšajte si, že v tomto konkrétnom prípade to funguje aj vtedy, keď priemer vyjde necelý, lebo hľadáme *ostro väčšie* prvky) alebo môžeme delenie úplne obísť – z matematiky vieme, že $kspaci[i] > sucet/n$ (po slovensky *„i-ty KSPák je nadpriemerný“*) platí práve vtedy, keď $kspaci[i] \cdot n > sucet$, a pri rátaní $kspaci[i] \cdot n$ už žiadne reálne čísla nepotrebujeme.

Listing programu:

```
program tana;
var n, i, sucet, dobri: longint;
    kspaci: array[1..500000] of longint;
begin
  readln(n);
  for i := 1 to n do
    read(kspaci[i]);
  sucet := 0;
  for i := 1 to n do
    sucet := sucet + kspaci[i];
  dobri := 0;
  for i := 1 to n do
    if kspaci[i] * n > sucet then
      inc(dobri);
  writeln(dobri);
end.
```

2. Záhradné rošády

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

Pri tomto príklade je potrebné uvedomiť si niekoľko vecí. Ak v záhradke popresádzame na správne miesta iba jeden druh zeleniny, napr. petržlen, mrkvu už nemusíme presádzať. Keď je petržlen na správnom mieste, musí byť na správnom mieste aj mrkva. Niektorí z vás sa rozhodli presádzať tú zeleninu, z ktorej je menej kusov. V skutočnosti na tom nezáleží, keďže z oboch druhov zeleniny musí byť na nesprávnom mieste uložený rovnaký počet kusov. Ja som si vybrala, že budem presádzať petržlen.

Na presadenie petržlena z miesta i pôvodnej postupnosti na miesto j výslednej Rasťovej postupnosti je potrebné práve $|i - j|$ nocí. Zjavne nemá zmysel vymeniť dva susedné petržleny alebo mrkvy. Prvý petržlen pôvodnej postupnosti budem preto postupne presádzať na miesto prvého petržlena Rasťovej postupnosti, druhý petržlen pôvodnej postupnosti na miesto druhého petržlena Rasťovej postupnosti, ... Pri takomto prístupe som nemohla urobiť zbytočné presádzania.

Môj algoritmus bude teda vyzerať tak, že si načítam prvú postupnosť, ktorú si uložím do poľa. Potom v nej nájdem prvú pozíciu, na ktorej sa nachádza petržlen. Na tomto mieste začnem čítať druhú postupnosť (pre ušetrenie priestoru si ju nemusím pamätať) a načítavam, kým nedostanem prvý petržlen. Do celkového súčtu si započítam absolútnu hodnotu rozdielu pozícií týchto petržlenov. Následne pokračujem rovnakým spôsobom pre ostatné petržleny, teda až kým neprídem na koniec pôvodnej postupnosti.

Nakoniec mi stačí vypísať celkový počet presádzaní. Takýmto prístupom prejdem raz pôvodnú postupnosť a raz Rasťovu postupnosť, preto bude časová zložitosť $O(n)$ a pamäťová zložitosť taktiež $O(n)$.

Za riešenie s časovou zložitosťou $O(n^2)$ som strhávala 2 body a za chýbajúci alebo zlý odhad zložitosti som strhla jeden bod.

Listing programu:

```
program zahradne_rosady;
var n, i, j, suc: longint;
    p: array [1..90000] of char;
    rp: char;

begin
  readln(n);
  for i := 1 to n do read(p[i]);
  readln;
  j := 0;
  suc := 0;
  for i := 1 to n do
    if p[i] = 'P' then begin
      repeat
        read(rp);
        inc(j);
      until rp = 'P';
      suc := suc + abs(i - j);
    end;
  writeln(suc);
end.
```

opravovala Katka

3. Zbierka nepodarkov

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

Možno na prvý pohľad prekvapivým výsledkom je, že si vystačíme s odpoveďami -1, 1 a 2. Poďme sa pozrieť, prečo.

Najskôr uvažujme o situácii, keď je minimum z časov pre rozvarené porcie menšie ako minimum z časov pre nedovarené (preložené do ľudskej reči: nejakú porciu sme rozvarili za taký malý čas, že sa za neho nestihla dovariť žiadna z nedovarených porcií). To je ale v spore s tým, že z každej značky je aspoň jedna porcia rozvarená a aspoň jedna nedovarená. Zo značky, ktorá zodpovedá minimálnemu rozvarenému času, musí predsa existovať aj nedovarená porcia, a tá sa musela variť kratšie. Ale medzi časmi nedovarených porcií žiadny taký nie je. Preto takáto situácia nemohla nastať.

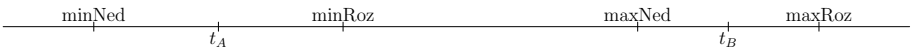
Podobne nemohol nastať ani prípad, že je maximum rozvarených časov menšie ako maximum nedovarených časov (preložené do ľudskej reči: nejakú porciu sme nedovarili za dlhší čas, ako sme potrebovali na rozvarenie ľubovoľnej porcie). Ďalej sa už týmito prípadmi nebudeme zaoberať.

Čo v prípade, keď je maximum nedovarených časov menšie ako minimum rozvarených (preložené do ľudskej reči: najskôr sme všetky porcie nedovarili a potom sme všetky už rozvarili)? Vezmime jednu značku, ktorej čas t je medzi týmito hodnotami. Potom všetky nedovarené porcie mohli byť tejto značky, lebo časy ich varenia sú menšie ako čas t. Zároveň aj všetky rozvarené porcie mohli byť tejto značky, lebo ich časy varenia sú väčšie ako čas t. Preto táto situácia mohla nastať a minimálny počet značiek je jedna.

Zamyslime sa nad zvyšnými situáciami. Keďže nenastáva žiadny z predchádzajúcich prípadov, tak musí platiť, že minimum z nedovarených časov $<$ minimum rozvarených $<$ maximum nedovarených $<$ maximum rozvarených. Jedna značka špagiet s časom t nám nestačí, lebo ak

čas t vyhovuje všetkým rozvareným porciám, tak t je menšie ako minimum rozvarených časov. Ale potom existuje nedovarená porcia s časom väčším ako t, a teda pre ňu t nevyhovuje. Podobne, keby t vyhovovalo všetkým nedovareným porciám, tak existuje rozvarená porcia, ktorej čas je menší ako t.

Preto potrebujeme aspoň dve značky špagiet, označme si ich ako A a B. Ich časy t_A a t_B si zvoľme tak, aby bolo t_A väčšie ako minimum nedovarených a menšie ako minimum rozvarených časov a t_B väčšie ako maximum nedovarených a menšie ako maximum rozvarených časov. Navyše, nech sa t_A ani t_B nerovná žiadnemu inému rozvarenému alebo nedovarenému času.



Potom všetky rozvarené porcie s časmi medzi t_A a t_B a aj všetky nedovarené porcie menšie ako t_A môžu byť značky A. Zvyšné porcie môžu byť značky B. Množina rozvarených porcií značky A (teda medzi časmi t_A a t_B) je neprázdna, lebo tam patrí rozvarená porcia s minimálnym časom, podobne ako aj množina nedovarených porcií značky A, kam patrí nedovarená porcia s minimálnym časom. Podobne množina nedovarených porcií značky B obsahuje nedovarenú porciu s maximálnym časom a množina rozvarených porcií značky B obsahuje rozvarenú porciu s maximálnym časom. Všetky porcie zo vstupu sme rozdelili medzi značky A a B, dostali sme teda korektnú situáciu s dvoma značkami špagiet.

Časová zložitosť je $O(n + r)$ a pamäťová je $O(1)$.

Listing programu:

```
program Zbierka_nepodarkov;
var n, r, maxNed, minNed, maxRoz, MinRoz, i, j: longint;

begin
  readln(n, r);
  read(j);
  maxNed := j;
  minNed := j;
  for i := 2 to n do begin
    read(j);
    if j < minNed then minNed := j;
    if j > maxNed then maxNed := j;
  end;
  read(j);
  maxRoz := j;
  minRoz := j;
  for i := 2 to r do begin
    read(j);
    if j < minRoz then minRoz := j;
    if j > maxRoz then maxRoz := j;
  end;
  if (minNed > minRoz) or (maxNed > maxRoz) then
    writeln('-1')
  else if minRoz > maxNed then
    writeln('1')
  else
    writeln('2');
end.
```

opravoval Mio

4. Zabaliť optimálne

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

Zo zadania jednoducho vyplývajú nasledovné pozorovania:

- do jedného balíčka vieme zabaliť ľubovoľne veľa rôznych čísel,
- do jedného balíčka nevieme zabaliť dve rovnaké čísla.

Minimálny počet balíčkov je teda maximum z počtov výskytov jednotlivých čísel. Ako ho zrátať? Riešenia, ktoré ste posielali, spadali do troch kategórií. Prvá skupina si pole čísel utriedila a potom postupným prechádzaním zisťovala maximálny počet výskytov čísel. Fungovalo to v čase $O(n \log n)$.

Druhá skupina používala pole, v ktorom si pamätala počet výskytov ku každému číslu. Tu musím podotknúť, že toto riešenie nebolo veľmi efektívne – jeho časová aj pamäťová zložitosť bola úmerná maximálnemu číslu na vstupe, čo mohlo byť až 10^9.

Tretia skupina bola dômyselnejšou verziou tej druhej. Namiesto poľa použili štruktúru `map`, čo je vyvažovaný binárny vyhľadávací strom. Potrebuje $O(n)$ pamäte a pristupuje k jednotlivým prvkom v čase $O(\log n)$, takže výsledná časová zložitosť bola $O(n \log n)$. Táto verzia sa ešte dá vylepšiť, aby fungovala v čase $O(n)$, viac na konci vzoráku.

Zostáva vyriešiť druhú časť úlohy. Máme počet balíčkov (označme ho p), ako teraz zistíme minimálnu veľkosť najväčšieho? Čísla do balíčkov vieme rozdeliť rovnomerne. Jedno také rozdelenie získame napríklad tak, že si utriedené čísla postupne dávame do balíčkov: prvé do prvého, druhé do druhého... p-te do p-teho, $p+1$ opäť do prvého... Čiže každé číslo bude v balíčku $1 + ((i-1) \bmod p)$, kde i je poradie čísla. Keďže rovnaké čísla nasledujú po sebe, žiadne dve nemôžu mať rovnaký zvyšok po delení p. Z toho vyplýva, že minimálna veľkosť najväčšieho balíčka je $\lceil n/p \rceil$.

Bodovanie za popis:

- Riešenie v $O(n)$ by získalo 10 bodov (škoda, že také nebolo).
- Riešenia v $O(n \log n)$ získali maximálne 9 bodov. Ten bod bol strhnutý symbolicky, aby ste si prečítali vzorák a videli, že existuje ešte trochu lepšie riešenie.
- Pomalšie riešenia získali najviac 6 bodov.
- Ďalej boli strhnuté body za chýbajúce alebo zlé odhady zložitostí, nedostatočný popis a nejaké drobnosti, ktoré si nájdete v opravených riešeniach.

Ako na riešenie v čase $O(n)$? Keby sme mali obmedzený rozsah čísel (napríklad do milióna), tak nemáme problém, vieme si počet každej veci udržiavať v poli.

Ale čo keď máme väčší rozsah a nechceme míňať veľa pamäte (teda nie rádovo viac ako n)? Pomôžeme si náhodou. Zoberieme si nejakú funkciu $h(x)$, ktorá každému číslu priradí nejaké číslo z rozsahu $0, 1, \ldots, kn$ (k môže byť napríklad 5). Hovoríme, že číslo „zahashujeme“.[1] Počet výskytov čísla x si zapíšeme do poľa na miesto $h(x)$. Toto pole sa zvykne nazývať hashovacia tabuľka.

Toto by mohlo fungovať, ale má to jeden drobný háčik. Veľmi ľahko sa nám môže stať, že dve veci budú mať rovnakú hash. Toto je celkom problém, ale vieme s ním bojovať. V praxi sa používajú dve možnosti:

- Na mieste $h(x)$ nebudeme mať len počet výskytov danej veci, ale spájaný zoznam dvojíc: (pôvodné číslo, počet výskytov). Potom keď niečo vkladáme do našej tabuľky, tak najprv číslo našej veci zahashujeme a potom ju ešte hľadáme v zozname. Ak tam nie je, pridáme ďalšiu vec na koniec zoznamu.

- Nebudeme mať funkciu $h(x)$, ale funkciu $h(x, i)$ (teda takú, čo papá dve čísla). Na políčkach tabuľky budeme mať dvojice (číslo predmetu, počet výskytov), prípadne záznam, že tam nič nie je. Vkladanie bude vyzerať nasledovne: Pozrieme sa najprv na pozíciu $h(x, 0)$. Ak tam nič nie je, resp. je tam práve náš prvok, tak sa tam vložíme. Ináč sa pozrieme na $h(x, 1)$. A tak ďalej. Táto metóda je použitá aj v jednom zo zdrojákov.

Dá sa dokázať, že pokiaľ je veľkosť tabuľky rozumne veľká (nejaký násobok počtu prvkov), tak jedno vloženie bude v priemere v čase $O(1)$. Dôkaz z dôvodu zachovania vášho psychického zdravia vynechávame.

Výnimočne pripájame zdrojáky aj v Pascale aj C++. Pascalovský má ukázať, že implementácia hashovacej tabuľky je pomerne bezbolestná vec, a C++ má ukázať, čo konkrétne v STL použiť.

[1] čítaj [zahešujeme]

Listing programu:

```
type entry=record
  co,kolko:longint;
end;
var table:array[0..999999] of entry;
    n,i,a,max,tsize:longint;

function h(x,i:longint):longint; begin
  h:=(47*x+2*i*(x+47)) mod tsize;
end;

procedure vloz(x:longint);
var j:longint;
begin
  j:=0;
  while (table[h(x,j)].co <> 0) and (table[h(x,j)].co <> x) do begin
    inc(j);
  end;
  table[h(x,j)].co := x;
  inc(table[h(x,j)].kolko);
end;

begin
  read(n);
  tsize:=10*n;
  for i:=1 to n do begin
    read(a);
    vloz(a);
  end;
  max := 0;
  for i:=0 to 10*n-1 do begin
    if table[i].kolko > max then
      max := table[i].kolko;
  end;
  writeln(max);
  writeln(1 + (n-1) div max);
end.
```

Listing programu:

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

int main() {
  int n, mx = 1;
  scanf("%d", &n);
  unordered_map<int, int> cisla;  // hashovacia tabulka, v ktorej si uchovavame pocty
jednotlivych cisel

  // zratame maximum poctu rovnakych cisel
  for (int i = 0; i < n; i++) {
    int c;
    scanf("%d", &c);
    if (cisla.find(c) != cisla.end()) {  // ak sme taketo cislo uz niekedy precitali
      cisla[c]++;  // zvysime pocet
      if (cisla[c] > mx)
        mx = cisla[c];  // nastavime nove maximum ak je treba
    } else {  // taketo cislo sme este nenacitali
      cisla[c] = 1;  // pridame ho do tabulky a nastavime pocet na 1
    }
  }

  printf("%d\n", mx);  // pocet balickov
  printf("%d\n", 1 + (n - 1) / mx);  // velkost najvacsieho balicka
}
```

5. O sedlákovi

opravoval Usamec
(max. 2 b za popis, 15 b za program)

Tento príklad dopadol vcelku veselo. Kto prišiel na správnu metódu, bol odmenený pomerne veľkých počtom bodov. Naopak, zlé metódy boli odmenené nulou.

Pred tým, než si povieme niekoľko spôsobov, ako sa úloha mala riešiť, ukážme si niekoľko príkladov, ako sa úloha riešiť nemala:

- Ak je objekt **presný** kruh, tak je to pomaranč.
- Ak je objekt **všade** konvexný, tak je to pomaranč.

- Ak objektu opíšeme obdĺžnik a dotýka sa ho **presne** v stredoch, tak je to pomaranč.

Všetky tieto metódy majú jednu spoločnú vlastnosť. Testujú vlastnosti objektov natvrdo bez akejkoľvek tolerancie. A to je problém. Akonáhle sa vo vstupe nájde nejaký drobný šum, nerovnosť, lístok, stopka, . . . , tieto metódy zlyhávajú. Dobré metódy vyzerajú skôr takto: Ak sa objekt **nie veľmi líši** od kruhu, tak je to pomaranč.

Ako otestovať svoju metódu a zistiť, či je dobrá? Veľmi jednoducho! Vyrobíme si sadu banánov a pomarančov rôznych veľkostí, otočení, vyfarbení... a na nej to jednoducho otestujeme.

Tak teraz, keď už máme približnú predstavu, čo chceme robiť, môžeme sa do toho pustiť. Najprv nájdeme *bounding box* pre objekt – najmenší obdĺžnik rovnobežný s osami, ktorý ho ohraničuje. To sa spraví jednoducho – nájdeme najhornejší, najdolnejší, najpravejší a najľavejší pixel obrázka.

A poďme overiť, či je objekt aspoň trochu podobný kruhu. Napríklad môžeme skúsiť testovať stredovú a osovú súmernosť. Táto vlastnosť je celkom dobrá, keďže banány veľmi stredovo súmerné nie sú. Test na pomaranč potom vyzerá asi nasledovne: Spočítame počet čiernych pixelov, ktoré na miestach, kde je ich stredovo/osovo súmerný obraz, nemajú čierny pixel. Ak je tento počet väčší ako $x\%$ z obsahu bounding boxu, objekt sa už priveľmi líši od kruhu a je to banán.

Ako určiť hodnotu x? Zoberieme našu pripravenú sadu a nájdeme vhodnú hodnotu.

Ešte ukážeme jedno iné a celkom jednoduché riešenie. Opäť nájdeme bounding box a potom len spočítame, koľko pixelov bounding boxu patrí ovociu. Ak je ich dosť veľa, tak máme pomaranč, ináč banán.

Všetky tieto riešenia ešte potrebujú doriešiť jednu vec. Ak je ovocie deravé, tak potrebujú zistiť, ktoré pixely sú vonkajšie a ktoré patria ovociu. V zadaní máme slušný predpoklad, že ovocie má celistvý okraj. Jednoduchšie ako zisťovať, čo je vnútri, je zistiť, čo je vonku.

Načítame si celý vstup do poľa čísel. A teraz pustíme „záplavu“ zvonku. Je jasné, že môžeme zaplaviť každé políčko, ktoré je na okraji a má hodnotu 0. Záplava funguje nasledovne: Svoje políčko označí číslom 2 a zaplaví susedné nulové políčka. Keďže každé políčko bude zaplavené najviac raz, časová zložitosť bude lineárna od veľkosti obrázka.

Po tomto postupe budú všetky políčka s hodnotou 0 alebo 1 patriť ovociu. Na toto potom môžeme aplikovať buď postup so symetriou alebo počtom políčok patriacich ovociu.

Listing programu:

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

using namespace std;

int smx[4] = {0,1,0,-1};
int smy[4] = {1,0,-1,0};

void flood(int y, int x, vector<vector<int> >& mriezka) {
  if (y < 0) return;
  if (y >= mriezka.size()) return;
  if (x < 0) return;
  if (x >= mriezka[0].size()) return;
  if (mriezka[y][x] == 1 || mriezka[y][x] == 2) return;
  mriezka[y][x] = 2;
  for (int i = 0; i < 4; i++)
    flood(y+smy[i], x+smx[i], mriezka);
}

void go() {
  int R, S; scanf("%d %d", &S, &R);
  vector<vector<int> > mriezka(R, vector<int>(S,0));
  int minY = 2000;
  int maxY = -1;
  int minX = 2000;
  int maxX = -1;
  // Popri nacitani si aj zistime v ktorej casti obrazku
  // je nieco vyznamne
  for (int i = 0; i < R; i++) {
    for (int j = 0; j < S; j++) {
      scanf("%d", &mriezka[i][j]);
```

```
      if (mriezka[i][j] == 1) {
        if (i < minY) minY = i;
        if (j < minX) minX = j;
        if (i > maxY) maxY = i;
        if (j > maxX) maxX = j;
      }
    }
  }
  // Zaplavime mriezku z kazdeho okraja
  for (int i = 0; i < R; i++) {
    flood(i, 0, mriezka);
    flood(i, S-1, mriezka);
  }
  for (int i = 0; i < S; i++) {
    flood(0, i, mriezka);
    flood(R-1, i, mriezka);
  }
  // Spocitame pocet policok ktore nie su 2 vo vyznamnej oblasti,
  // cize tie, ktore patria ovociu.
  int ovocie = 0;
  for (int i = minY; i <= maxY; i++) {
    for (int j = minX; j <= maxX; j++) {
      if (mriezka[i][j] != 2) ovocie++;
    }
  }
  if ((maxY-minY+1)*(maxX-minX+1)*0.68 > ovocie)
    printf("banan\n");
  else
    printf("pomaranc\n");
}

int main() {
  int t; scanf("%d", &t);
  for (int i = 0; i < t; i++)
    go();
}
```

opravoval Žaba

6. Obe(d/t)ovanie kravičiek

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

Sé onr sverdar sitja hvass!

Ahojte všetci. Práve ste si prečítali starodávne kúzlo, ktoré vás donúti dočítať tento vzorák až do konca. Teraz, keď už mám vašu plnú pozornosť, musím povedať, že ma trochu sklamalo, že bolo tak málo dobrých riešení. Ale zase úloha to nebola úplne jednoduchá.[2] Body som potom strhával za zlé alebo pomalé riešenia a po bode ste mohli stratiť za zlú alebo chýbajúcu časovú alebo pamäťovú zložitosť.

Teraz sa už poďme vrhnúť na vzorák. Označme si $p(x)$ najväčší počet kravičiek, ktorý vieme zjesť, ak máme práve k dispozícii x kráv. Hodnotu $p(x)$ zistíme tak, že sa pozrieme, čo môže drak spraviť. Môže zjesť nejaké menu alebo, ak je x kráľovo obľúbené číslo, vyhnať richtára po d kráv. Ak vyženie richtára, v tom kroku nezje žiadnu kravu a teda dokopy zje najviac $p(x+d)$ kráv. Ak mu dedinčania pripravia menu i, tak dokopy zje najviac $p(x-m_i)+m_i$ kráv.

Na zistenie $p(x)$ nám teda stačí nájsť maximum zo všetkých možností, ktoré sú prípustné (napr. nemôže zjesť viac kráv ako má). No a dostávame peknú rekurziu, lebo na zistenie $p(x)$ potrebujeme ďalšie hodnoty $p(y)$, ktoré vyrátame rovnakým spôsobom. Rekurzia je však veľmi pomalá, lebo ju pre tie isté hodnoty x rátame aj viac krát. Spôsob ako ju urýchliť je však jednoduchý a nazýva sa **memoizácia**. Teda budeme mať nejaké pole, do ktorého si budeme značiť vyrátané hodnoty $p(x)$. Keď sa spýtame na hodnotu, ktorú sme už predtým rátali, nerátame ju znova, iba sa pozrieme do nášho poľa.

Ešte nám zostáva otázka, ako zrátať, kedy môže drak žrať donekonečna. To nastane vtedy, ak začína s nejakým počtom kráv x (do ktorého sa vie dostať zo začiatočného počtu n) a postupnosťou zožratí kráv a vyhnaní richtára vie znova dosiahnuť počet x. Vtedy môže tento postup opakovať a jesť donekonečna. Kedy takýto prípad nastane zistíme tak, že si budeme pamätať, ktoré x máme v rekurzii aktuálne rozpracované. Keď sa rekurzia znova zavolá na nejaké rozpracované x, našli sme cyklus.

[2] Inak by to nebola šestka.

Zhrnutie: Použijeme rekurzívnu funkciu, ktorá dostane počet kráv a vypočíta najväčší počet kráv čo vie drak zjesť tak, že sa zavolá na ďalšie stavy. Pri tom si pamätá, čo už rátala a čo práve spracúva, aby vedela nájsť cyklus.

Keďže každý počet kráv spracuje najviac raz (kráv môže byť až $n+d$, ak drak vyhnal richtára hneď na začiatku) a na spracovanie jedného stavu sa potrebujeme pozrieť na $m+1$ stavov (všetky menu[3] plus vyhnanie richtára), časová zložitosť je $O(m(n+d))$. Pamätať si potrebujeme všetky menu, hodnoty $p(x)$ a niekoľko ďalších polí, dokopy však najviac $O(n+m+d)$.

Dodám ešte, že existovalo aj jedno pekné riešenie, kde sme si situáciu predstavili ako graf s orientovanými hranami a pomocou prehľadávania do šírky (DFS) zistili výsledok.

Listing programu:

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

vector<int> M;
bool nekonecno = false;
int A[40147];  // 0 = nespracovane, 1 = prave spracuvane, 2 = uz doratane
long long P[40147];  // zapamatane vysledky rekurzie
bool Oblubene[40147];
int n, m, s, d;

long long rekurzia(int v) {
  if (A[v] == 2)
    return P[v];
  if (A[v] == 1) {
    nekonecno = true;
    A[v] = 2;
    return 0;
  }
  A[v] = 1;
  for (int i = 0; i < m; i++) {
    int w = v - M[i];
    if (w >= 0)
      P[v] = max(P[v], rekurzia(w) + M[i]);
  }
  if (Oblubene[v])
    P[v] = max(P[v], rekurzia(v + d));
  A[v] = 2;
  return P[v];
}

int main() {
  scanf("%d %d %d %d ", &n, &m, &s, &d);
  M.resize(m);
  for (int i = 0; i < m; i++)
    scanf("%d", &M[i]);
  //spravim zvacsujuce hrany
  for (int i = 0; i < s; i++) {
    int x;
    scanf("%d", &x);
    Oblubene[x] = true;
  }
  long long vys = rekurzia(n);
  if (nekonecno)
    printf("nekonecno\n");
  else
    printf("%lld\n", vys);
}
```

opravoval JaNo

7. Obrovská šachovnica

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

Potešil som sa, že viac ako polovica z vás mala efektívne riešenie, plus-mínus pamäťová zložitosť a nejaké drobnosti. Potom bolo zopár pomalších riešení, ale tých bolo pomerne málo vzhľadom na to, že išlo o siedmy príklad.

Poďme sa pozrieť, ako by sa dalo nejaké riešenie vymyslieť.

[3] Všimli ste si, že slovo menu je nesklonné?

Každá šachovnica má svoj pravý dolný roh (ďalej PDR). Takže by sme mohli prejsť všetky políčka vstupu a zistiť, aká najväčšia šachovnica by mohla mať na tomto políčku svoj PDR. To by sa dalo vypočítať postupným skúšaním čoraz väčších šachovníc: najskôr skontrolujeme štvorec veľkosti 1×1, potom štvorec veľkosti 2×2, atď., až kým nenájdeme najmenší štvorec, ktorý už nie je šachovnica. Potom vieme, že štvorec o jedno menší je najväčšia šachovnica s PDR v danom políčku.

Toto riešenie je pomalé najmä preto, že mnohé štvorce kontroluje zbytočne veľa krát. Keď napríklad skontrolujeme štvorec 8×8 s PDR v políčku $[10, 20]$ a zistíme, že to nie je šachovnica, neskôr je zbytočné kontrolovať štvorec 9×9 s PDR v políčku $[11, 21]$ – tiež to nemôže byť šachovnica.

Skutočne väčšina odovzdaných riešení sa uberala tým smerom, že sa snažíme nekontrolovať dve rovnaké veci viackrát, ale nejako si zapamätať, čo už o vstupe vieme a nemusíme overovať.

Spôsobov ako to robiť je viacero, tak tu jeden z nich uvediem.

Trojica $[n][x, y]$ bude označovať štvorec s dĺžkou strany n políčok a PDR v bode $[x, y]$. Potom si môžeme všimnúť, že pre $n \geq 2$ je $[n][x, y]$ šachovnicou práve vtedy, keď sú šachovnicami štvorce $[n-1][x-1, y]$, $[n-1][x, y-1]$, $[n-1][x-1, y-1]$ a $[2][x, y]$. Možno si pomyslíte, že sa to vôbec nezlepšilo, veď namiesto jedného štvorca musím overovať štyri. Trik je ale v tom, že ak sa to rozumne naprogramuje, tak prvé tri štvorce už boli overené predtým a štvorec $[2][x, y]$ sú len 4 políčka, ktoré overíme v konštantnom čase.

Ak $S_{i,j}$ bude veľkosť (dĺžka strany) najväčšej šachovnice s PDR v $[i, j]$ a predchádzajúcu myšlienku dotiahneme do konca, dospejeme k takémuto záveru:

- ak $[2][i, j]$ nie je šachovnica, potom $S_{i,j} = 1$.
- ak $[2][i, j]$ je šachovnica, potom $S_{i,j} = \min(S_{i-1,j}, S_{i,j-1}, S_{i-1,j-1}) + 1$.

Prvý prípad je zrejmý, rozoberme si teda ten druhý. Ak je $[2][i, j]$ šachovnica, potom $S_{i,j}$ je určite aspoň 2. Štvorce $[S_{i,j} - 1][i - 1, j]$, $[S_{i,j} - 1][i, j - 1]$ a $[S_{i,j} - 1][i - 1, j - 1]$ musia byť potom tiež šachovnicami, preto sú hodnoty $S_{i-1,j}$, $S_{i,j-1}$ a $S_{i-1,j-1}$ aspoň $S_{i,j} - 1$, teda $S_{i,j} \leq \min(S_{i-1,j}, S_{i,j-1}, S_{i-1,j-1}) + 1$.

Platí aj opačná nerovnosť, keďže štvorec $[\min(S_{i-1,j}, S_{i,j-1}, S_{i-1,j-1}) + 1][i, j]$ je úplne pokrytý menšími prekrývajúcimi sa šachovnicami a teda je celý tiež šachovnicou.

Hodnoty $S_{i,j}$ budeme počítať postupne zhora nadol a zľava doprava – vďaka tomu už budeme v momente počítanie $S_{i,j}$ poznať potrebné hodnoty $S_{i-1,j}$, $S_{i,j-1}$ a $S_{i-1,j-1}$. Riešením úlohy je maximum zo všetkých $S_{i,j}$.

Pomocou nášho rekurentného vzťahu vieme vypočítať hodnotu pre jedno políčko $S_{i,j}$ v konštantnom čase; pre všetkých $r \times s$ políčok to bude čas $O(rs)$. Lepší čas dosiahnuť nemôžeme, lebo musíme načítať celý vstup. (Čo ak by práve posledné políčko rozhodlo o výsledku?)

Keď si všimneme, že nám stačí pamätať si iba posledné dva riadky, tak dosiahneme pamäťovú zložitosť $O(s)$.

Za vzorák bol plný počet – 12 bodov. Jeden bod sa dal stratiť za horšiu pamäťovú zložitosť, neporiadny popis alebo zlý odhad zložitosti. Pomalšie riešenia ako $O(rs)$ boli tiež primerane ohodnotené.

Listing programu:

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

#define MAXN 1247
char pole[2][MAXN];
int sach[2][MAXN];
int best, R, S;

int main() {
  scanf("%d %d", &R, &S);
  for (int i = 0; i < R + 2; ++i)
    pole[0][i] = 'x';
  best = 1;  // aspon jedno policko samotne je sachovnica
```

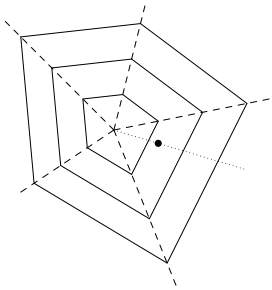
```
  for (int i = 0; i < R; ++i) {
    scanf("%s", pole[(i + 1) % 2] + 1);
    for (int j = 0; j < S; ++j) {
      // ak sedia farby policok
      if ((pole[i % 2][j] == pole[(i + 1) % 2][j + 1]) &&
          (pole[(i + 1) % 2][j] == pole[i % 2][j + 1]) &&
          (pole[(i + 1) % 2][j] != pole[i % 2][j])) {
        // vypocitaj velkost a updatni premennu best
        sach[(i + 1) % 2][j + 1] = min(sach[i % 2][j], min(sach[(i + 1) % 2][j], sach[i % 2][j +
1])) + 1;
        best = max(best, sach[(i + 1) % 2][j + 1]);
      } else {
        // policko samotne je sachovnica velkosti 1
        sach[(i + 1) % 2][j + 1] = 1;
      }
    }
  }
  printf("%d\n", best);
}
```

opravoval Bob

8. Ostreľovanie

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

Keď som sa dozvedel, že mi na opravovanie ostala už len táto geometrická úloha, vydal som srdcervúci výkrik. To zas bude zametania, epsilonov a skrytých okrajových prípadov... No povzbudilo ma, že niekoľkým z vás sa podarilo napísať program za plný počet bodov, a tak som sa aj ja osmelil a skúsil niečo vymyslieť.

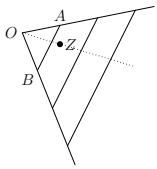


Zadanie sa nás pýta iba na celkový počet Zemčových trestných bodov, no aj tak bude asi treba spracovať každý zásah osobitne a trestné body sčítať. Oblasti terča môžu mať rôzne komplikované tvary, preto si hľadanie tej správnej rozdelíme na dva kroky: najprv nájdeme *výsek* roviny, v ktorom sa zásah nachádza. Výsekom som si nazval uhol, ktorý má vrchol v bode $[0,0]$ a na jeho ramenách ležia dva nasledujúce vrcholy zadaného n-uholníka.

Človek pozrie a vidí, no ako nájsť správny výsek algoritmicky? Mohli by sme použiť polárnu sústavu – každej polpriamke z bodu $[0,0]$ priradíme uhol, ktorý zviera s kladnou poloosou x proti smeru hodinových ručičiek (napríklad), a potom už len binárne vyhľadáme uhol polpriamky vedúcej do zásahu medzi uhlami polpriamok vedúcich do vrcholov n-uholníka.

V mojom riešení som sa snažil vyhnúť neceločíselnej aritmetike, preto som namiesto počítania uhlov použil vektorový súčin. Čo to je? Podrobnosti si môžete nájsť na Wikipédii (angl. cross product) alebo v nejakej učebnici matematiky. Nám stačí vedieť, že vektorový súčin dvojrozmerných vektorov $\vec{u} = (u_x, u_y)$ a $\vec{v} = (v_x, v_y)$ je $u_x v_y - u_y v_x$ (veľmi nepresne povedané), a že podľa jeho znamienka môžeme určiť vzájomnú polohu vektorov: ak je záporný, $\vec{v}$ leží napravo od $\vec{u}$; ak je kladný, $\vec{v}$ leží naľavo od $\vec{u}$; ak je rovný nule, $\vec{u}$ a $\vec{v}$ ležia na tej istej priamke.

S pomocou vektorového súčinu môžeme správny výsek binárne vyhľadať. Budeme si udržiavať interval výsekov, v ktorom sa nachádza zásah. Vezmeme si polpriamku z $[0,0]$ do vrcholu n-uholníka, ktorý leží v strede intervalu, a vektorovým súčinom zistíme, v ktorej polovici intervalu leží zásah.



Zostáva nám druhý krok: potrebujeme zistiť, v ktorej *vrstve* sa nachádza zásah. Označme si priesečník polpriamky OZ s úsečkou AB ako P. Počet trestných bodov je potom rovný dolnej celej časti pomeru $|OZ|/|OP|$ (tento vzťah sa dá ľahko dokázať využitím podobnosti trojuholníkov na obrázku).

Vzorové riešenie namiesto počítania priesečníka využíva inú myšlienku. Každú priamku v rovine môžeme zapísať všeobecnou rovnicou $ax+by+c=0$ pre nejaké hodnoty a, b, c. Keďže (a, b) je normálový vektor (t.j. kolmý na

priamku), dve rovnobežné priamky môžeme zapísať rovnicami, ktoré sa budú líšiť len v hodnotách c (označme ich c_1 a c_2). Vzdialenosť takýchto priamok je potom $\frac{|c_1-c_2|}{\sqrt{a^2+b^2}}$.

Zapíšme rovnicu priamky $AB = p_1$ ako $ax+by+c_1 = 0$. Smerový vektor priamky poznáme (je to $\overrightarrow{AB} = (x_B - x_A, y_B - y_A)$), normálovým vektorom je potom napríklad $(a, b) = (y_B - y_A, x_A - x_B)$. Zostáva vypočítať hodnotu c_1, no to ide ľahko dosadením bodu A do rovnice.

Veďme postupne bodmi O a Z priamky rovnobežné s p_1, označme ich p_0, p_z a ich rovnice zapíšme ako $ax+by+c_0 = 0$ a $ax+by+c_z = 0$. Pretože p_0 prechádza bodom $O = [0, 0]$, platí $c_0 = 0$; hodnotu c_z vypočítame dosadením bodu Z do rovnice.

Počet trestných bodov teraz vypočítame ako dolnú celú časť pomeru vzdialeností priamok p_z a p_1 od priamky p_0, teda:

$$\left\lfloor \frac{\frac{|c_z-c_0|}{\sqrt{a^2+b^2}}}{\frac{|c_1-c_0|}{\sqrt{a^2+b^2}}} \right\rfloor = \left\lfloor \frac{|c_z|}{|c_1|} \right\rfloor$$

Prvý krok má kvôli binárnemu vyhľadávaniu časovú zložitosť $O(\log n)$, druhý stíhame v konštantnom čase. Celková časová zložitosť je preto $O(n + m \log n)$. Pamätáme si len pozície vrcholov n-uholníka, takže pamäťová zložitosť je $O(n)$.

Hurá, nakoniec sa aj mne podarilo nakódiť funkčné riešenie! No a poučenie? Geometria je síce zlá a škaredá, ale niekedy nie príliš.

Listing programu:

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

struct Point {
  int x, y;
  Point(int ix = 0, int iy = 0) : x(ix), y(iy) {}
};

long long vektorovy_sucin(const Point& u, const Point& v) {
  return (long long) u.x * v.y - (long long) u.y * v.x;
}

int n, m;
vector<Point> P;

int vysek(const Point& Z) {
  int u = 0, v = n;
  while (v - u > 1) {
    int w = (u + v) / 2;
    if (vektorovy_sucin(P[v], P[w]) > 0) {  // uhol medzi w, v je < 180
      if (vektorovy_sucin(P[w], Z) <= 0 &&
        vektorovy_sucin(P[v], Z) >= 0)
        u = w;
      else
        v = w;
    } else {  // uhol medzi w, v je >= 180
      if (vektorovy_sucin(P[w], Z) <= 0 ||
        vektorovy_sucin(P[v], Z) >= 0)
        u = w;
      else
        v = w;
    }
  }
  return u;
}

long long trestne_body(const Point& Z, const Point& A, const Point& B) {
  long long a = (long long) B.y - A.y;
  long long b = (long long) A.x - B.x;
  long long c1 = abs(a * A.x + b * A.y);
  long long cz = abs(a * Z.x + b * Z.y);
```

```
  return cz / c1;
}

int main() {
  cin >> n;
  for (int i = 0; i < n; ++i) {
    int x, y;
    cin >> x >> y;
    P.push_back(Point(x, y));
  }
  P.push_back(P[0]);  // za poslednym vrcholom nasleduje prvy
  long long res = 0;
  cin >> m;
  for (int i = 0; i < m; ++i) {
    int x, y;
    cin >> x >> y;
    int w = vysek(Point(x, y));
    res += trestne_body(Point(x, y), P[w], P[w + 1]);
  }
  cout << res << endl;
}
```


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