Riešenia kategórie B

B-I-1 Veže

Pre každú vežu vieme ľahko nájsť všetky s ktorými sa potencionálne ohrozuje. Takéto budú v tom istom riadku alebo stĺpci na šachovnici a budú opačnej farby. Nájdeme ich prejdením zoznamu veží. Každú takúto kolegyňu s ktorou sa potencionálne ohrozuje vieme skontrolovať. Ak medzi nimi leží akákoľvek ďalšia veža, nevidia na seba, inak sa naozaj ohrozujú.

Takto vyskúšame $\binom{v}{2}$ dvojíc veží. Pre každú dvojicu, ktorá sa „vidí“ (leží v tom istom riadku/stĺpci) a má opačné farby vieme v lineárnom čase prejsť všetky zvyšné veže a skontrolovať, či nestoja v ceste. Takto dostávame jednoduché riešenie s časovou zložitosťou $O(v^3)$. Celé riešenie pozostáva len z niekoľkých vnorených cyklov. Pämaťová zložitosť je $O(v)$ – stačí nám pamatať si jedno pole so súradnicami veží.

Pre jednoduchý prístup k popisu veží si údaje navonok zjednotíme v podobe štruktúry – v Pascale `record`, v C/C++ `struct`.

Podobné riešenie bez problémov vyrieši prvé tri vstupy, na zvyšné treba použiť efektívnejší prístup (alebo veľmi dlho čakať).

Listing programu (Pascal)

```
type veza = record
  riadok, stlpec, farba : longint;
end;

var n, v, i, j, k, vysledok : longint;
    ok : boolean;
    veze : array[1..100047] of veza;

function min(a, b : longint) : longint;
begin if a < b then min := a else min := b; end;

function max(a, b : longint) : longint;
begin if a > b then max := a else max := b; end;

function vidia_sa(var a, b : veza) : boolean;
begin vidia_sa := (a.riadok = b.riadok) or (a.stlpec = b.stlpec); end;

function medzi(var a, b, c : veza) : boolean;
begin
  medzi := false;
  if (a.riadok = b.riadok) and (b.riadok = c.riadok) then
    if (min(a.stlpec, b.stlpec) < c.stlpec) and (c.stlpec < max(a.stlpec, b.stlpec)) then medzi := true;
  if (a.stlpec = b.stlpec) and (b.stlpec = c.stlpec) then
    if (min(a.riadok, b.riadok) < c.riadok) and (c.riadok < max(a.riadok, b.riadok)) then medzi := true;
end;

begin
  readln(n, v);
  for i:=1 to v do
    readln(veze[i].stlpec, veze[i].riadok, veze[i].farba);
  vysledok := 0;
  for i:=1 to v do
    for j:=i+1 to v do if (veze[i].farba <> veze[j].farba) and (vidia_sa(veze[i], veze[j])) then begin
      ok := true;
      for k:=1 to v do if (medzi(veze[i], veze[j], veze[k])) then begin
        ok := false;
        break;
      end;
      if ok then inc(vysledok);
    end;
  writeln(vysledok);
end.
```

Jednoduché zlepšenie prináša pozorovanie že veža môže ohrozovať najviac štyri ďalšie: najbližšiu vľavo, vpravo, hore a dole. Pre každú vežu stačí medzi ostatnými v lineárnom čase nájsť týchto štyroch „susedov“ a skontrolovať, či sú to nepriatelia. (Počas spracúvania konkrétnej veže si pre každý smer pamätáme doteraz najbližšiu vežu od nej v tom smere.) Časovú zložitosť sme týmto zlepšili na $O(v^2)$, ale 4. a 5. vstup stále podobným prístupom rýchlo vyriešiť nevieme.

Môžeme však šikovne využiť to, že v prvých štyroch vstupoch je šachovnica malá a celá sa nám zmestí do pamäte ako dvojrozmerné pole. Môžeme si takéto pole v pamäti spraviť a zaznačiť doň všetky veže. Potom keď pre konkrétnu vežu chceme nájsť tie, ktoré ohrozuje, nemusíme prechádzať celý zoznam veží – stačí v našom poli prejsť riadok a stĺpec, v ktorých naša veža leží. Netreba zabúdať že každú dvojicu takto zarátam dvakrát – raz pre jednu vežu z dvojice, druhýkrát pre druhú, preto treba na konci vydeliť výsledný počet dvoma.

Toto riešenie má časovú zložitosť $O(nv)$. Jeho pamäťové nároky sú $O(n^2)$. Pre prvé štyri vstupy sa nám všetko do pamäte zmestí a takéto riešenie stačí na ich vyriešenie.

Ešte lepšie riešenie: Nemusíme pre každú vežu zas a znova prechádzať celý stĺpec a celý riadok. Keď už sme si veže vyplnili do dvojrozmerného poľa, stačí raz prejsť každý celý riadok a raz každý stĺpec a pre každý z nich spočítať všetky po sebe idúce dvojice veží opačnej farby. Takéto riešenie má časovú zložitosť $O(n^2)$. Nižšie uvádzame jeho implementáciu.

(Rovnakú časovú zložitosť by sme dostali, keby sme v predchádzajúcom riešení pre každú vežu prezerali príslušný riadok/stĺpec len dovtedy, kým nenarazíme na inú vežu alebo kraj šachovnice. To preto, že každým políčkom šachovnice by sme v každom smere išli najviac jedenkrát.)

Listing programu (Pascal)

```
var n, v, i, j, riadok, stlpec, farba, posledna, h, vysledok : longint;
    sachovnica : array[1..1047, 1..1047] of longint;

begin
  for i := 1 to 1047 do
    for j := 1 to 1047 do
      sachovnica[i][j] := -1;
  readln(n, v);
  for i:=1 to v do begin
    readln(riadok, stlpec, farba);
    sachovnica[riadok][stlpec] := farba;
  end;
  vysledok := 0;
  for i := 1 to n do begin
    posledna := -1; h := -1;
    for j := 1 to n do
      if sachovnica[i][j] <> -1 then begin
        if (posledna <> -1) and (h <> sachovnica[i][j]) then inc(vysledok);
        posledna := j;
        h := sachovnica[i][j];
      end;
  end;
  for j := 1 to n do begin
    posledna := -1; h := -1;
    for i := 1 to n do
      if sachovnica[i][j] <> -1 then begin
        if (posledna <> -1) and (h <> sachovnica[i][j]) then inc(vysledok);
        posledna := i;
        h := sachovnica[i][j];
      end;
  end;
  writeln(vysledok);
end.
```

Vzorové riešenie

Ako však vypočítať posledný vstup? Chceli by sme spraviť to isté ako v predchádzajúcom riešení – najskôr pre každý *neprázdny* riadok spočítať, koľko sa v ňom ohrozuje dvojíc veží, potom to isté spraviť pre každý *neprázdny* stĺpec.

Aby som mohol prezerať veže v poradí v akom idú na šachovnici v riadku alebo stĺpci podobne ako v predchádzajúcom riešení, potrebujem mať veže rozumne usporiadané. Usporiadajme teraz veže podľa čísla riadku a v prípade rovnosti podľa stĺpcu. V takto usporiadanom poli platí, že každý riadok zodpovedá nejakému súvislému úseku veží. Stačí teda prejsť po usporiadanom poli a pre každé dve po sebe nasledujúce veže skontrolovať, či sú v tom istom riadku a zároveň opačných farieb. Následne spravíme to isté pre stĺpce – usporiadame veže podľa stĺpcu a v prípade rovnosti podľa riadku, a potom usporiadané pole prejdeme a spočítame ohrozujúce sa dvojice.

Celkovo takéto riešenie spotrebuje $O(v)$ pamäte a časovú zložitosť bude mať $O(v \log v)$ kvôli triedeniu. V moderných programovacích jazykoch môžeme na triedenie použiť knižničnú funkciu:

Listing programu (Python)

```
def rataj_riadky(veze):
    veze.sort()
    answer = 0
    for i in range(len(veze)-1):
        if veze[i+1][0]==veze[i][0] and veze[i+1][2]!=veze[i][2]:
            answer += 1
    return answer

from sys import stdin
N = int(stdin.readline())
V = int(stdin.readline())

veze = [ tuple( int(x) for x in stdin.readline().split() ) for v in range(V) ]
total = rataj_riadky(veze)

veze = [ (s,r,c) for r,s,c in veze ]
total += rataj_riadky(veze)

print(total)
```

Listing programu (C++)

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

struct veza { int riadok, stlpec, farba; };

bool podla_riadkov(const veza &a, const veza &b){
  if (a.riadok == b.riadok)
    return (a.stlpec < b.stlpec);
  else
    return (a.riadok < b.riadok);
}

bool podla_stlpcov(const veza &a, const veza &b){
  if (a.stlpec == b.stlpec)
    return (a.riadok< b.riadok);
  else
    return (a.stlpec < b.stlpec);
}

vector<veza> veze;

int main(){
  int n, v;
  cin >> n >> v;
  veze.resize(v);
  for (int i = 0; i < v; i++)
    cin >> veze[i].riadok >> veze[i].stlpec >> veze[i].farba;
  int vysledok = 0;
  sort(veze.begin(), veze.end(), podla_riadkov);
  for (int i = 1; i < v; i++)
    if(veze[i-1].farba != veze[i].farba && veze[i-1].riadok == veze[i].riadok)
      vysledok++;
  sort(veze.begin(), veze.end(), podla_stlpcov);
  for (int i = 1; i < v; i++)
    if(veze[i-1].farba != veze[i].farba && veze[i-1].stlpec == veze[i].stlpec)
      vysledok++;
  cout << vysledok << endl;
}
```

Na utriedenie 100 000 prvkov potrebujeme dostatočne rýchly triediaci algoritmus, ideálne bežiaci v čase rádovo $n \log n$ pre n prvkov. Veľa programovacích jazykov má implementované takéto algoritmy, napríklad `sort` v C++. V ostatných jazykoch je treba naprogramovať vlastné rýchle triedenie. My si stručne popíšeme jeden zo zaužívaných prístupov – HeapSort, teda triedenie pomocou haldy.

Halda

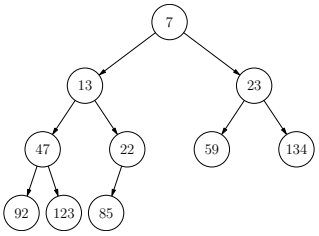
Predstavme si, že máme čiernu krabičku s tlačidlom, do ktorej vieme vhadzovať čísla a z ktorej po každom stlačení tlačidla vypadne najmenšie z čísel, ktoré sú práve v nej.

Takáto krabička vie byť celkom užitočná. Pomocou nej by sa nám napríklad ľahko triedilo – nahádžeme všetky čísla do nej, no a potom už len stláčame tlačidlo, kým postupne nevypadnú všetky v utriedenom poradí.

Jednou šikovnou implementáciou takejto krabičky je halda. Je to vlastne binárny strom, ktorý má každé poschodie úplne plné, možno okrem toho posledného, najspodnejšieho. Každý prvok, okrem tých najspodnejších,

má teda práve dvoch synov. Prvky musia byť usporiadané tak, aby platilo, že hodnota uložená v ľubovoľnom vrchole je menšia alebo rovná ako každá z hodnôt uložená v jeho synoch (ak nejakých má).

Takto môže vyzerať halda:



Všimnime si zaujímavú vlastnosť haldy: najmenšie číslo je určite v jej koreni. O ostatných číslach už toho tak veľa nevieme povedať – všimni si, že napríklad 22 je hlbšie ako 23.

Na to, že halda je strom, môžeme zase šťastne zabudnúť. Haldu si totiž vieme úplne jednoducho pamätať v poli. Stačí si vrcholy očíslovať po vrstvách. Na políčku s číslom x teda bude prvok, ktorý by sme prečítali ako x-tý, keby sme haldu „čítali po riadkoch“.

Všimnime si niektoré vlastnosti takto uloženej haldy:

- Ak je v halde n prvkov, v poli budú na políčkach 1 až n.
- Koreň haldy, teda najmenší prvok v nej, je na políčku s čísom 1.
- K-ta vrstva haldy sa v poli začína na políčku s indexom 2^{K-1}.
- Synovia vrcholu s číslom x majú vždy čísla $2x$ a $2x + 1$.
- A opačne, otec vrcholu s číslom x má číslo $\lfloor x/2 \rfloor$.

Halda z obrázku by vyzerala v poli takto:

1	2	3	4	5	6	7	8	9	10	11	12	
7	13	23	47	22	59	134	92	123	85			

Ukážeme teraz, ako do haldy efektívne pridávať nové čísla a ako z nej efektívne vyberať najmenšie.

Vloženie prvku

Ako teda vložíme nejaký prvok do haldy? Zaradíme ho do poľa na prvé voľné miesto. Jediné, čo nám teraz môže kaziť „haldovitosť“, je, že tento prvok môže byť menší od prvku nad ním. V tom prípade túto dvojicu prvkov vymeníme. Rozmyslite si, že opäť môže nastať jediný problém: nový prvok môže byť menší aj od svojho nového otca. V takomto prípade postup opakujeme. (Tomuto sa hovorí „bublanie prvku dohora“.)

Tento postup určite skončí, prinajhoršom vtedy, ked sa nový prvok dostane na úplný vrch haldy – do koreňa. Po jeho skončení je opäť celá halda v poriadku, úspešne sme teda vložili nový prvok.

Výber najmenšieho prvku

A čo s vybratím najmenšieho prvku? To bude fungovať podobne. Vieme, že najmenší prvok je ten na vrchu haldy. Tentokrát ho ale nemôžeme len tak odstrániť, vznikla by nám tam totiž diera. No a tú treba niečím zaplniť. Najjednoduchšie riešenie: Zoberieme posledný prvok v poli (t. j. najpravejší list v poslednej vrstve) a presunieme ten na začiatok poľa.

Touto zmenou sme opäť dosiahli, že čísla máme uložené na prvých niekoľkých políčkach poľa. Nemusí to ale ešte byť korektná halda. To, čo nám ju môže kaziť, je práve presunutý prvok. Preto zopakujeme niečo podobné, ako pri vkladaní. Tentokrát ale presunutý prvok môže byť len priveľký, preto ho budeme musieť „prebublať“ dodola. Toto bublanie treba robiť trochu šikovnejšie. Tentokrát totiž náš „zlý“ prvok môže mať až dvoch synov a byť väčší od každého z nich. Hravo ale zistíme, že riešenie je jednoduché: stačí ho vymeniť s menším z oboch synov.

Opäť, tento postup je konečný. Skončíme, ak už sú obidvaja aktuálni synovia väčší alebo rovní presunutému prvku, prípadne ak sa náš prvok prebublal až do najspodnejšej vrstvy. V každom prípade máme opäť korektnú haldu.

Odhad zložitosti

Všimnime si, čo sa deje pri jednom prebublaní prvku. Pri vkladaní prebubleme v najhoršom z poslednej vrstvy až do koreňa, pri vyberaní minima naopak, z koreňa až po najhlbšiu vrstvu. V obidvoch prípadoch je počet operácií úmerný hĺbke stromu, teda počtu vrstiev. A aká je tá hĺbka? Na zaplnenie k vrstiev haldy potrebujeme $2^k - 1$ prvkov, preto halda s n prvkami má približne $\log_2 n$ vrstiev. Každá operácia s haldou, v ktorej je n prvkov, má teda časovú zložitosť $O(\log n)$.

Triedenie pomocou haldy

Ako sme už spomínali na začiatku, pomocou haldy vieme ľahko napísať triedenie, známe pod menom HeapSort. (Heap je halda po anglicky.) Pri triedení n prvkov najskôr n-krát vložíme prvok do haldy, potom odtiaľ n-krát vyberieme najmenší. Počas celého tohto procesu nie je nikdy v halde viac ako n prvkov, preto časová zložitosť každej operácie s ňou je $O(\log n)$. Týchto operácií je $2n$, preto je výsledná časová zložitosť HeapSortu $O(n \log n)$. Pamäťová zložitosť haldy aj triedenia pomocou nej je samozrejme $O(n)$.

Listing programu (Pascal)

```
type veza = record
  riadok, stlpec, farba : longint;
end;

var n, v, i, vysledok : longint;
    veze : array[1..100047] of veza;

type func_t = function (var a, b : veza) : boolean;

function podla_riadkov(var a, b : veza) : boolean;
begin
  if a.riadok = b.riadok then podla_riadkov := (a.stlpec < b.stlpec)
  else podla_riadkov := (a.riadok < b.riadok);
end;

function podla_stlpcov(var a, b : veza) : boolean;
begin
  if a.stlpec = b.stlpec then podla_stlpcov := (a.riadok < b.riadok)
  else podla_stlpcov := (a.stlpec < b.stlpec);
end;

procedure swap(var a, b : veza);
var t : veza;
begin t := a; a := b; b := t; end;

procedure insert(i : longint; porovnaj : func_t);
begin
  while (i>1) and porovnaj(veze[i], veze[i div 2]) do begin
    swap(veze[i div 2], veze[i]);
    i := i div 2;
  end;
end;

procedure extract(pocet : longint; porovnaj : func_t);
var i, j : longint;
begin
  swap(veze[1], veze[pocet]);
  dec(pocet);
  i := 1;
  while true do begin
    j := i;
    if (2*i <= pocet) and porovnaj(veze[2*i], veze[j]) then j := 2*i;
    if (2*i+1 <= pocet) and porovnaj(veze[2*i+1], veze[j]) then j := 2*i+1;
    if j=i then break;
    swap(veze[i], veze[j]);
    i := j;
  end;
end;

procedure HeapSort(porovnaj : func_t);
var i : longint;
begin
  for i:=2 to v do insert(i, porovnaj);
  for i:=v downto 2 do extract(i, porovnaj);
```

```
end;

begin
  readln(n, v);
  for i:=1 to v do
    readln(veze[i].stlpec, veze[i].riadok, veze[i].farba);
  vysledok := 0;
  HeapSort(@podla_riadkov);
  for i:=2 to v do
    if (veze[i-1].farba <> veze[i].farba) and (veze[i-1].riadok = veze[i].riadok) then inc(vysledok);
  HeapSort(@podla_stlpcov);
  for i:=2 to v do
    if (veze[i-1].farba <> veze[i].farba) and (veze[i-1].stlpec = veze[i].stlpec) then inc(vysledok);
  writeln(vysledok);
end.
```

B-I-2 Dosah

Naša úloha sa skladá z viacerých otázok, ale sú od seba dosť nezávislé, preto každú takúto otázku budeme riešiť samostatne. Nech teda máme už danú mapu a konkrétne x, y a k. Chceme zistiť, kam všade sa vie tank dostať tak, aby neprešiel viac ako k krokov a zároveň neprešiel cez prekážku alebo nevyšiel z mapy.

Namiesto toho, aby sme z tohto políčka púšťali tančík a určovali mu všetky možné cesty, vylejme na tento štvorec vedro vody a sledujme čo sa deje. Na začiatku je voda, iba na políčku (x, y), postupne sa však začne vylievať aj na susedné políčka. Presnejšie bude sa snažiť vyliať na horné, dolné, pravé a ľavé políčko. Po prvom kroku je teda voda na pôvodných súradniciach (x, y) a tiež na nových súradniciach $(x+1, y), (x-1, y), (x, y+1)$ a $(x, y-1)$ – samozrejme ak na týchto miestach nie je prekážka alebo nebodaj nie sú mimo mapy. Toto vylievanie samozrejme pokračuje, tentoraz sa voda šíri ďalej zo všetkých už zaliatych políčok.

Teraz si všimnime, že pre každé n platí, že po n vyliatiach na susedné políčka je voda presne na tých miestach, kam sa vie dostať náš tank z (x, y) na najviac n posunutí. A keby sme nasimulovali toto rozlievanie vody, vieme potom ľahko zrátať odpoveď.

Pomalšie riešenie

Prvé, čo vieme spraviť, je naozaj priamočiare simulovanie vyššie popísaného procesu. Zoberieme si dvojrozmerné pole veľkosti $r \times s$ a do každého políčka si zaznačíme hodnotu -1, ktoré predstavuje, že na toto políčko sa voda ešte nedostala. Iba na políčko (x, y) si uložíme číslo 0, lebo tu je na začiatku voda. Ďalej postupne pre každé n od 1 po k zopakujem nasledovný postup:

Prejdem celé dvojrozmerné pole. Ak na nejakom políčku nájdem číslo od 0 po $n-1$, znamená to, že už je na tomto políčku voda, ktorá sa chce rozlievať ďalej. Všetkým jeho voľným susedom (teda nie je to prekážka, je na mape a ešte tam nebola voda – v poli mám uložené -1) zaznačím do poľa hodnotu n. To znamená, že toto políčko bolo zaliate v n-tom kroku.

Tento postup nám zaručí, že po n-tej iterácii máme v poli nezáporné čísla na tých miestach, kam sa vie tank dostať na najviac n krokov. Po kroku, v ktorom $n = k$, teda máme označené práve tie políčka, ktoré nás zaujímajú. Prejdeme celú mapu, spočítame, koľko ich je a vypíšeme toto číslo na výstup.

Bohužiaľ musíme počas simulácie rozlievania vody k-krát prejsť cez celú mapu, a teda máme časovú zložitosť $O(krs)$ na zodpovedanie jednej otázky.

Vzorové riešenie

Chceli by sme to teda nejak zrýchliť. Pozrime sa, či v predchádzajúcom programe nerobíme niečo zbytočne. Naozaj nám treba pozerať na všetky políčka? Ľahko nahliadneme, že nie. Ak sme v n-tej iterácii, nepotrebujem kontrolovať rozlievanie vody zo všetkých políčok, ale len z tých, kam sa voda prvýkrát dostala v predchádzajúcom, $(n-1)$. kroku. Všetko, čo sa dalo zaliať skôr, už dávno aj zaliate je. Inými slovami, nikdy sa nemôže stať, že sa čísla v dvoch susedných voľných políčkach budú líšiť o viac ako 1 – akonáhle niekedy zalejem jedno z nich, najneskôr v nasledujúcom kole (z neho) zalejem aj to susedné. V každej iterácii rozlievania nám teda stačí kontrolovať políčka s číslom $n-1$.

Otázkou však je, ako to robiť dostatočne rýchlo. Efektívne riešenie tejto úlohy je známe pod názvom prehľadávanie do šírky.

Prehľadávanie do šírky

Algoritmus je v podstate simuluje šírenie vody, akurát to robí veľmi šikovne. Podstatou je dátová štruktúra fronta, do ktorej vieme prvok vložiť a potom ho aj vybrať, pričom vo vnútri v dátovej štruktúre to funguje ako klasický rad na obed. Keď chceme vložiť nový prvok, zaradíme ho na koniec tohto radu, a keď prvok vyberáme, vyberieme vždy prvý prvok – ten, ktorý už najdlhšie čakal.

V našej fronte budú na spracovanie čakať políčka, z ktorých chceme rozlievať vodu. Na začiatku je tam teda jediné políčko: (x, y).

Kým je vo fronte nejaké políčko, robíme nasledujúcu vec: Vyberieme z fronty políčko, z ktorého sa ide šíriť voda a do premennej n si poznačím, v ktorom kroku som toto políčko zaplavil (to mám poznačené v poli). Pozrieme sa na susedné políčka. Ak je niektoré z nich voľné, zaznačíme si, že sme toto políčko objavili v kroku $n + 1$ a vložíme ho do fronty na spracovanie. (Samozrejme, ak $n = k$, už nové políčka na spracovanie do fronty nevkladáme.)

Po dobehnutí tohto algoritmu dostávame presne rovnaký výsledok ako pred tým, teda si zrátame počet zaplavených miest a vrátime ho ako výsledok.

A prečo to funguje? Všimnime si, že na začiatku sú vo fronte miesta, ktoré sa dajú zaplaviť na 0 krokov – také je jedno. Potom do fronty vložím všetky, ktoré sa dajú zaplaviť na 1 krok. Tieto postupne z fronty vyberám a do fronty *za ne* pridávam políčka, ktoré sú zo štartu dosiahnuteľné na 2 kroky. Keď teda spracujem všetky políčka dosiahnuteľné na 1 krok, vo fronte budú na spracovanie čakať práve všetky políčka dosiahnuteľné na 2 kroky. A tak ďalej. Teda robím to isté čo predtým, nepozerám sa však na všetky políčka znovu a znovu, ale každé políčko spracujem najviac jedenkrát – vtedy, keď to treba.

Zostáva už len určiť pamäťovú a časovú zložitosť. Pamäť je $O(rs)$, lebo si musím pamätať ako vyzerá celá mapa. Čas potrebný na odpovedanie na jednu otázku je $O(rs)$, lebo každé miesto zaplavím najviac raz a teda sa aj najviac raz ocitne vo fronte.

Listing programu (Python)

```
from sys import stdin
from queue import Queue
T = int(stdin.readline())

for t in range(T):
    R, S, Q = [ int(x) for x in stdin.readline().split() ]
    A = [ [ int(x) for x in stdin.readline().split() ] for r in range(R) ]

    for q in range(Q):
        r0, s0, krokov = [ int(x) for x in stdin.readline().split() ]
        dist = [ [ 1234567890 for s in range(S) ] for r in range(R) ]
        dist[r0-1][s0-1] = 0
        Q = Queue()
        Q.put( (r0-1,s0-1) )
        answer = 1

        while not Q.empty():
            cr,cs = Q.get()
            for nr,ns in [ (cr+1,cs),(cr-1,cs),(cr,cs+1),(cr,cs-1) ]:
                if nr<0 or ns<0 or nr>=R or ns>=S: continue
                if dist[nr][ns] <= dist[cr][cs]+1: continue
                if A[nr][ns] == 1: continue
                if dist[cr][cs] == krokov: continue
                answer += 1
                dist[nr][ns] = dist[cr][cs]+1
                Q.put( (nr,ns) )
        print(answer)
```

Listing programu (C++)

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

int A[2047][2047], T[2047][2047];

int main() {
    int t;
    scanf("%d␣",&t);
    while(t--) {
```

```
        int r,c,q;
        queue<int> Q;
        scanf("%d %d %d",&r,&c,&q);
        for(int i=0; i<r+2; i++) T[i][0]=T[i][c+1]=-2;
        for(int j=0; j<c+2; j++) T[0][j]=T[r+1][j]=-2;
        for(int i=1; i<=r; i++)
            for(int j=1; j<=c; j++) { scanf("%d",&A[i][j]); T[i][j]=-1; }
        for(int i=0; i<q; i++) {
            int x,y,r;
            scanf("%d %d %d",&y,&x,&r);
            int poc=0; T[y][x]=i;
            Q.push(y); Q.push(x); Q.push(0);
            while(!Q.empty()) {
                int y1=Q.front(); Q.pop();
                int x1=Q.front(); Q.pop();
                int kroky=Q.front(); Q.pop();
                poc++;
                if(kroky==r) continue;
                int dx[]={0,0,1,-1}, dy[]={1,-1,0,0};
                for(int j=0; j<4; j++) {
                    int y2=y1+dy[j],x2=x1+dx[j];
                    if(T[y2][x2]==i || T[y2][x2]==-2 || A[y2][x2]==1) continue;
                    T[y2][x2]=i;
                    Q.push(y2); Q.push(x2); Q.push(kroky+1);
                }
            }
            printf("%d\n",poc);
        }
    }
}
```

B-I-3 Súčet súčtov

Čo by ste robili vy, keby vám váš učiteľ matematiky dal za úlohu sčítať postupnosti čísel za celú triedu? Určite by sa mnohým z vás nechcelo robiť toľko sčítaní, a nejako by ste sa snažili ušetriť si robotu. Ale ako?

Najskôr sa pozrime, koľko výpočtov by žiaci spravili, keby počítali všetko poctivo tak, ako bolo uvedené v zadaní. Keď sa nehráme na detaily, tak máme rádovo n^2 žiakov, z ktorých každý spraví rádovo n sčítaní, čiže dokopy máme zložitosť $O(n^3)$. Koho to zaujíma, môže si spočítať, že spravia *presne* $n(n+1)(n+2)/6-1$ sčítaní.

Ako teda tento postup zefektívniť? Napríklad nemá veľký zmysel počítať niečo viackrát. Keď máte za úlohu sčítať prvých sedemnásť čísel a zistíte, že váš spolužiak už má vypočítaný súčet prvých šestnástich, tak nemusíte jeho robotu opakovať, ale sa ho môžete spýtať na jeho výsledok a pripočítať už len posledné, sedemnáste číslo.

Celý postup teda môžeme zlepšiť napríklad nasledovne: Žiaci rozdelia do skupín podľa toho, kde začína ich úsek. (Teda v prvej skupine budú tí, ktorí majú spočítať čísla od prvého po nejaké, v druhej skupine tí, ktorí začínajú sčitovať od druhého čísla, a tak ďalej.) V rámci skupiny sa potom postavia do radu podľa toho, ktorým číslom končia.

Počítanie bude teraz fungovať nasledovne: Prvý žiak v rade má sčítať len jednoprvkovú postupnosť. Teda rovno vie výsledok. Výsledok povie ďalšiemu žiakovi. Ten následne k nemu pripočíta nasledujúce číslo z postupnosti a tiež je hotový. Výsledok povie nasledujúcemu v rade, a tak ďalej. Každý žiak takto spraví len jedno jediné sčítanie – pripočíta svoje číslo ku súčtu žiaka pred ním. (Ak napríklad mám počítať súčet od 3. čísla po 11., počkám si, kým mi predo mnou stojaci žiak oznámi súčet od 3. po 10. číslo, pripočítam k nemu 11. číslo a som hotový.)

Takýmto spôsobom dokáže každý žiak zistiť svoj súčet v konštantnom čase. Keďže žiakov je rádovo n^2, spraví sa pritom len $O(n^2)$ operácií. Pri implementácii nám stačia dva vnorené cykly: vonkajší určuje začiatok úseku, vnútorný zase koniec (teda žiaka, ktorý práve počíta).

Listing programu (C++)

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

int main(){
    int n;
    cin >> n;
    long long celkovysucet = 0;

    //nacitame postupnost
    long long postupnost[n+1];
```

```
    for(int i = 1; i <= n; ++i) cin >> postupnost[i];

    for(int i = 1; i <= n; ++i) {
        // spocitame ziakov, ktorych postupnosti zacinaju i-tym prvkom
        long long aktualny = 0;
        for(int j = i; j <= n; ++j) {
            // nova hodnota bude hodnota ziaka predomnou plus j-ty clen postupnosti
            aktualny += postupnost[j];
            celkovysucet += aktualny;
        }
    }
    cout << celkovysucet << endl;
}
```

Ako sme videli, keď sa snažíme nejaký postup zrýchliť, tak sa zamyslíme, či nerobíme niečo zbytočne veľakrát. V predchádzajúcom riešení už toho zdanlivo nejde veľa zlepšiť – pre každého konkrétneho žiaka sme predsa jeho výsledok zistili v konštantnom čase a lepšie to už nemá ako ísť, nie?

Lenže ono to efektívnejšie pôjde. Trik bude v tom, že my vlastne jednotlivé medzivýsledky nepotrebujeme. Nás zaujíma len ich súčet. A nič nebráni tomu, aby sme tento súčet vedeli nejakým iným, šikovnejším spôsobom vypočítať efektívnejšie.

Predstavte si, že chcete sčítať 100-krát číslo 47. Čo by ste naozaj urobili? Urobili by ste 99 operácií sčítania alebo niečo iné? Asi nik by nesčítal, všetci predsa vedia, že výsledok bude 100 krát 47, čiže 4700.

Podobne aj v tejto úlohe budeme vo vzorovom riešení namiesto opakovaného sčítania radšej násobiť. Pre každý prvok postupnosti si jednoducho spočítame, koľkokrát sa objaví vo výsledku. Inými slovami, to, čo potrebujeme pre každý prvok zistiť, je počet žiakov, ktorí tento prvok majú vo svojom úseku.

Keď si zoberieme nejaké dve čísla $a, b; 1 \leq a \leq b \leq n$, tak existuje práve jeden žiak, ktorý mal za úlohu sčítať čísla z úseku od a po b vrátane. Zoberme si nejaký konkrétny, povedzme i-ty prvok postupnosti. Úsek od a po b, ktorý tento prvok obsahuje, musel zjavne začať najneskôr na i-tom mieste, teda musí platiť $a \leq i$. Zároveň tento úsek musí skončiť až po i-tom prvku, teda pre b platí $i \leq b$.

Úsek, ktorý obsahuje i-ty prvok, má teda i možných začiatkov a $n - i + 1$ možných koncov. Každý začiatok s každým koncom určuje jeden konkrétny úsek. Preto je takýchto úsekov presne $i \times (n - i + 1)$. Inými slovami, i-ty prvok sa vyskytne v celkovom súčte práve $i(n - i + 1)$-krát.

Program, ktorý rieši túto úlohu, teda postupne prečíta celý vstup od prvku s číslom 1 až po n, pre každé i vynásobí hodnotu i-teho prvku počtom $i \times (n - i + 1)$, a všetky tieto medzivýsledky sčíta.

Celková časová zložitosť bude $O(n)$. Tú už zjavne nevieme zlepšiť, lebo musíme načítať celý vstup. Pamäťová zložitosť bude $O(1)$, keď si uvedomíme, že pole si vlastne ani netreba pamätať. Jediné, čo potrebujeme, je aktuálne spracúvané číslo, jeho index, číslo n a doterajší súčet.

Listing programu (C++)

```
#include <iostream>

int main(){
    long long n;
    std::cin >> n;
    long long sucet = 0;
    for (int i = 1; i <= n; ++i) {
        long long prvok;
        std::cin >> prvok;
        sucet += prvok * i * (n-i+1);
    }
    std::cout << sucet << "\n";
}
```

B-I-4 Paralelné trampoty

V riešení tejto úlohy si ukážeme, že vo svete, v ktorom sa naraz deje viacero vecí, môže často vzniknúť nečakaný chaos.

Podúloha A – 6 bodov

Vyrobiť na tabuli číslo 900 je ľahké. Stačí napríklad, aby postupne každé dieťa 30× zopakovalo postup zo zadania. Presnejšie, akýkoľvek scenár, pri ktorom sa jednotlivé vykonania postupu zo zadania neprekrývajú,

vyrobí na tabuli číslo 900. Ak ho chceme zmenšiť, musíme teda nechať viac detí vykonávať ich postup v tom istom čase. Tu je prvé možné zlepšenie, ktoré nás dostane z 900 na 30:

- Postupne po jednom vôjde každé dieťa do triedy, prečíta z tabule 0 a zapamätá si ju.
- Všetky deti stoja pred triedou a usilovne počítajú, až kým každé z nich nedostane výsledok 1.
- Postupne po jednom vôjde každé dieťa do triedy, zotrie tabuľu a napíše na ňu svoj výsledok: číslo 1.
  V tomto okamihu už máme za sebou 30 vykonaní postupu zo zadania – no na tabuli namiesto čísla 30 svieti len číslo 1.
- Ešte 29× zopakujeme predchádzajúce tri body.

Celkom slušné zlepšenie, nie? Aj keby sme namiesto 30 detí mali v našej triede 1000, aj tak by sme skončili na čísle 30 (a nie 30000).

No ale lepšie to už zjavne nepôjde. Všimnime si totiž konkrétne dieťa. To svoj postup vykoná 30-krát, a to nutne postupne. Zakaždým pritom zvýši hodnotu na tabuli. A teda na konci musí byť hodnota na tabuli aspoň 30. A tým sme skončili – ukázali sme, že 30 dosiahnuť vieme, aj to, že menej nie.

Presvedčil vás predchádzajúci odsek? Ak tušíte zradu, tušenie vás veru neklame. V „dôkaze“, že menej ako 30 nevieme dosiahnuť, je totiž chyba. Kde? Pozrime sa na dotyčné jedno dieťa poriadnejšie. A vyberme si napríklad sedemnáste opakovanie jeho postupu. Naše dieťa vošlo do triedy, zapamätalo si číslo x z tabule, vyšlo von, prirátalo k nemu 1, znova vošlo dnu, zotrelo tabuľu, napísalo $x + 1$ a zase vyšlo von. A teraz prichádza kameň úrazu. Totiž skôr, ako toto dieťa začne svoje osemnáste kolo, môže sa pokojne stať, že do triedy vbehne iné dieťa so svojim výsledkom – a pokojne sa môže stať, že tento výsledok je menší ako $x + 1$.

Teda síce *je pravda*, že konkrétne dieťa postupne 30-krát zvýši aktuálnu hodnotu na tabuli, ale už *nie je pravda*, že ide o tú istú, postupne zvyšovanú hodnotu.

Ak to ešte nie je jasné, ukážeme to najskôr na malom príklade:

- Adamko bol v triede a zapamätal si 0.
- Betka bola v triede a zapamätala si 0.
- Cilka bola v triede a zapamätala si 0.
- Adamko bol v triede a zapísal 1.
- Dušanko bol v triede a zapamätal si 1.
- Dušanko bol v triede a zapísal 2.
- Betka bola v triede a zapísala 1.
- Dušanko bol v triede a zapamätal si 1.
- Dušanko bol v triede a zapísal 2.
- Cilka bola v triede a zapísala 1.
- Dušanko je už tretíkrát v triede po číslo... a po tretíkrát vidí na tabuli číslo 1.
- Dušanko tretíkrát zapísal na tabuľu číslo 2.

No a teraz už je čas na optimálne riešenie. Jeho hodnota je prekvapivá: najmenšie možné číslo, ktoré môže na konci ostať na tabuli, je číslo 2. Existuje viacero spôsobov, my si ukážeme jeden z nich. Dve z detí si nazveme Viktor a Zuzka, ostatných 28 ponecháme bez mena.

- Viktor bol v triede a zapamätal si 0.
- Zuzka bola v triede a zapamätala si 0.
- Všetky ostatné deti 30× vykonali celý svoj postup (v úplne ľubovoľnom poradí).
- Viktor bol v triede a zapísal 1.
- Viktor vykonal svoje 2. až 29. opakovanie. Na tabuli je číslo 29.
- Zuzka bola v triede a zapísala 1.
- Viktor bol v triede a zapamätal si 1.
- Zuzka vykonala svoje 2. až 30. opakovanie. Na tabuli je teraz číslo 30.

- No a úplne na záver prišiel do triedy Viktor a zapísal na tabuľu číslo 2.

Do kompletného riešenia už chýba len dôkaz, že menej ako 2 sa už naozaj dosiahnuť nedá. Ten je našťastie ľahký: Nula je na tabuli len na začiatku. Od okamihu, kedy prvýkrát niekto zapíše svoj výsledok, je číslo na tabuli stále kladné. Všimnime si dieťa, ktoré na konci zapísalo úplne poslednú hodnotu na tabuľu. Ide o jeho tridsiate opakovanie postupu, a teda číslo, ktoré v ňom prečítalo z tabule, bolo nutne už kladné – a teda aspoň 1. Tým pádom zapísaný výsledok je aspoň 2, č.b.t.d.

Podúloha B – 4 body

Pre pripomenutie, tu je ešte raz zadanie súťažnej úlohy. V Marienkinom domovskom adresári je súbor `/home/marienka/ponik3.png`, ku ktorému Janko nemá prístup. Správca Samko však spravil program, ktorý funguje nasledovne: keď ho užívateľ X spustí ako „`posli /nejaka/cesta/subor email`“, program postupne:

1. otvorí adresár `/nejaka/cesta` a skontroluje, či X má právo čítať `subor`.
2. ak áno, otvorí `subor`, načíta ho a prepošle ho na adresu `email`.

Trik je samozrejme vo vhodnom použití symbolických liniek, vysvetlených v zadaní. Asi najjednoduchšie riešenie vyzerá nasledovne:

1. Janko vyrobí symbolickú linku ukazujúcu na súbor v jeho domovskom adresári.
2. Janko spustí Samkov program, pričom ako prvý parameter použije práve dotyčnú symbolickú linku.
3. Samkov program skontroluje, že Janko má právo čítať dotyčný súbor. Všetko vyzerá OK.
4. Janko rýchlo zmení svoju symbolickú linku tak, aby ukazovala na Marienkin obrázok poníka.
5. Samkov program ho prečíta a pošle Jankovi mailom.

Napriek tomu, že ide o nesmierne jednoduchú formu útoku, sú takéto útoky na bezpečnosť systémov v praxi pomerne bežné. Mnohé *exploity* (programy zneužívajúce slabiny systémov) sú založené práve na tom, že útočník odhalí v systéme takúto chybu. V zadaní sme už spomínali, že sa takýmto chybám, vznikajúcim počas súčasného behu viacerých programov, zvykne hovoriť *race condition*. Slovo *race* (závod) je v názve práve preto, že sa programy pretekajú – stihne útočník v správnej chvíli urobiť potrebné zmeny?

Vo výhode je samozrejme útočník. Totiž často má k dispozícii pokusov, koľko len chce, pričom mu stačí uspieť jeden jediný raz. A skutočne aj mnohé reálne exploity z praxe fungujú jednoducho tak, že sa dokola skúšajú „trafiť do správneho okamihu“, až sa im to skôr či neskôr aj podarí.

DVADSIATY ÔSMY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh:
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2012