Korešpondenčný seminár z programovania
XXX. ročník, 2012/13
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 1. kola letnej časti

vzorák napísala Anička

1. Zázračný školský autobus

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

V tejto úlohe si bolo treba uvedomiť jednu podstatnú vec o autobusoch. Konkrétne, že autobusy majú dvere iba na jednej strane. Preto o smere letu autobusu vieme rozhodnúť tak, že sa pozrieme, či vidíme alebo nevidíme dvere. Teraz, keď už vieme túto podstatnú informáciu, ľahko si uvedomíme, že keď vidíme dvere, autobus letí doprava, a keď nevidíme dvere, letí doľava.

Stačilo teda pozrieť, či je na vstupe znak `H`.

Takéto riešenie má časovú zložitosť $O(nm)$, pretože v najhoršom prípade musíme prečítať celý vstup. Pamäťová zložitosť je $O(1)$, keďže si stačí pamätať len aktuálny znak.

Listing programu (Pascal)

```
var znak : char;
begin
    while(not eof(input)) do begin
        read(znak);
        if znak = 'H' then begin
            writeln('doprava');
            exit;
        end;
    end;
    writeln('dolava');
end.
```

Listing programu (Python)

```
import sys
if 'H' in sys.stdin.read():
    print 'doprava'
else:
    print 'dolava'
```

vzorák napísala Peťa

2. Známka

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

Na začiatok je dobrý nápad zistiť, aký je priemer aktuálnych známok. (Je rozumnejšie zisťovať, či platí $1z_1 + 2z_2 + \cdots + 5z_5 = 2(z_1 + \cdots + z_5)$ než či platí $\frac{1z_1+2z_2+\cdots+5z_5}{z_1+\cdots+z_5} = 2$, vyhneme sa tak reálnym číslam a teda aj zaokrúhľovacím chybám. Okrem toho, z prvého výpočtu hneď vidno, koľko chýba k cieľovému súčtu známok, a to budeme využívať.) Označme si aktuálny súčet známok $A = 1z_1 + 2z_2 + \cdots + 5z_5$ a cieľový $C = 2(z_1 + \cdots + z_5)$.

Ak $A = C$, netreba nič meniť, inak sa treba rozhodnúť podľa toho, či je A priveľa, alebo primálo.

Pokiaľ je A priveľa, musíme nejakú známku zlepšiť. Dôležité je uvedomiť si, že sa nám oplatí zmeniť čo najhoršiu známku, a to preto, že ak vieme znížiť A zmenou jednej známky, tak určite vieme znížiť A o tú istú hodnotu aj zmenou horšej známky – napríklad zmenou trojky na jednotku vieme hodnotu A znížiť o dva, no A sa dá znížiť o dva aj zmenou štvorky na dvojku alebo päťky na trojku. Naopak to však neplatí.

Navyše sa nám oplatí meniť najhoršiu známku práve na jednotku (lebo chceme A znížiť, čo najviac), okrem prípadu, kedy by sme takto dosiahli $A < C$. Vtedy treba najhoršiu známku zmeniť na $A - C -$ `hodnota najhoršej známky`, pretože tým sa A zmení presne na C a chod vesmíru bude zachránený.

Pokiaľ je A primálo, musíme mať k dispozícii aspoň jednu jednotku. Z rovnakých dôvodov ako minule sa nám oplatí meniť čo najlepšie známky na čo najhoršie. Meníme teda jednotky na päťky, pokiaľ rozdiel medzi C a A je väčší alebo rovný 4. Inak, ak stále neplatí $A = C$, zmeníme jednotku na známku s hodnotou $C - A + 1$ a dosiahneme $A = C$. (Pripomíname, že $A = C$ práve vtedy, keď je priemer známok 2.)

Na dosiahnutie $A = C$ stačí opakovať vyššie popísaný prístup: vždy najprv zistiť, v akom vzťahu sú A a C a potom náležite zmeniť jednu zo známok.

Časová zložitosť tohto algoritmu je lineárna od počtu známok: každú známku totiž zmeníme nanajvýš raz. (V skutočnosti nikdy nebude potrebné meniť všetky známky, avšak na zákerných vstupoch obsahujúcich samé päťky ich bude treba zmeniť viac ako polovicu. Nuž a O-notácia hovorí, že $O(\frac{n}{2}) = O(n)$.) Pamäťová zložitosť je konštantná, stačí si pamätať čísla $z_1, \ldots, z_5$, A a C.

Pozorní čitatelia si v tomto momente ale určite všimli, že vyššie popísaný algoritmus robí čosi navyše: napríklad pokiaľ je $A = 13$ a $C = 26$ (čiže máme k dispozícii trinásť jednotiek), algoritmus najprv zmení jednu z jednotiek na päťku, vďaka čomu A narastie na 17. Následne zmení jednu z jednotiek na päťku, vďaka čomu A narastie na 21. Následne zmení jednu z jednotiek na päťku, vďaka čomu A narastie na 25. No a napokon zmení jednu z jednotiek na dvojku a konečne zmení A na vytúžených 26.

Teraz je už jasne vidieť, čo robí vyššie popísaný algoritmus navyše: namiesto toho, aby priveľký rozdiel A a C zmenšil naraz (napríklad zmenou troch jednotiek na päťky), zmenšuje ho postupne. Stačí ho upraviť tak, aby zistil, aký najväčší násobok rozdielu menenej známky na (správnym smerom) najodlišnejšiu možnú známku sa zmestí do rozdielu A a C, a výrazne sa zlepší rýchlosť – časová zložitosť takto upraveného algoritmu bude z nasledujúcich dôvodov konštantná.

Pre každú z piatich hodnôt známok (dokonca v praxi len troch, nie piatich – ak je $A < C$, stačí meniť jednotky, ak je $A > C$, stačí meniť päťky, štvortky a nanajvýš trojky) totiž urobí len konštantný počet operácií: zistí, aký je rozdiel medzi A a C, zistí, koľko známok jedného druhu môže zmeniť na najrozdielenjšiu hodnotu, aby sa A priblížilo k C, no nepreskočilo ho, a pokiaľ sa A priblížilo k C už dostatočne blízko, zmení vhodnú známku o vhodnú hodnotu, čím docieli $A = C$. Teda progam spraví menej ako $5k$ krokov, kde k je nejaká konštanta. Preto jeho časová zložitosť je $O(1)$. Pamäťová zložitosť je zjavne tiež $O(1)$.

Listing programu (C++)

```
#include<iostream>
#define For(i, n) for(int i = 0; i<(n); ++i)
using namespace std;
typedef long long ll;
ll a[10], sum, pocet;

int main(){
    // pre jednoduchosť, známky budú 0..4, tým pádom priemer chceme 1
    For(i,5) {
        cin >> a[i];
        pocet+=a[i];
        sum+=a[i]*i;
    }
    // ak je priemer primalý, spočítame, koľko jednotiek treba meniť
    if (sum<=pocet) {
        cout << (pocet-sum+3)/4 << endl;
        return 0;
    }
    // inak postupne meníme 4ky, 3ku, 2ky. (pozor, známky sú 0..4)
    ll zmena = 0;
    int znamka = 4;
    while(sum>pocet) {
        // koľko známok danej hodnoty musím zmeniť
        ll kolko = min((sum-pocet+znamka-1)/znamka, a[znamka]);
        zmena += kolko;
        sum -= znamka*kolko;
        znamka--;
    }
    cout << zmena << endl;
}
```

vzorák napísal Jano

3. Zemčo a trojuholníky

(max. 8 b za popis, 4 b za program)

Našou úlohou je zisiť, koľko najmenej paličiek musíme odobrať, aby bola krabica bezpečná. Preto by sme si mali najprv rozmyslieť, čo to znamená bezpečná krabica – teda ako čo najrýchlejšie povedať, či nejaká množina paličiek obsahuje nejakú trojicu, z ktorej sa trojuholník poskladať nedá.

Trojicu nazývame bezpečná, ak sa z nej dá poskladať trojuholník, inak ju nazývame nebezpečná. Množinu paličiek nazývame bezpečná, ak všetky trojice v nej sú bezpečné, inak je nebezpečná. Keď v tejto úlohe bude slovo trojica, máme na mysli trojicu rôznych paličiek, teda nemôžeme mať jednu paličku v trojici viackrát. (Ale ak máme viac paličiek rovnakej dĺžky, tie v trojici byť môžu.)

Bezpečnosť

V tejto časti sa naučíme, ako čo najrýchlejšie povedať o množine paličiek, či je bezpečná.

Množina veľkosti 1 alebo 2 je bezpečná, lebo neobsahuje žiadnu trojicu (a teda všetky trojice sú bezpečné). Množina veľkosti tri, alebo konktrétna trojica, je bezpečná práve vtedy, keď pre ich dĺžky platí trojuholníková

nerovnosť. Presnejšie, ak si dĺžky označíme od najkratšej po najdlhšiu písmenami a, b a c, $(a \leq b \leq c)$, tak musí platiť $a + b > c$.

Posledný prípad je, že máme v množine viac ako tri paličky. Skontrolovať, či sú bezpečné všetky trojice, si nemôžeme dovoliť, lebo ich je rádovo n^3. Preto musíme vymyslieť nejaký prefíkanejší (a hlavne rýchlejší) spôsob, napríklad namiesto kontrolovania všetkých trojíc skontrolovať len jednu trojicu.

Keď budeme chvíľu rozmýšľať, zistíme, že tá správna trojica je nasledovná: najmenšie dve čísla, označme ich a, b, a najväčšie číslo, označme ho c. Totiž, ak neplatí $a + b > c$, tak sme našli nebezpečnú trojicu a nemusíme ďalej kontrolovať. A ak platí, $a + b > c$, tak budú všetky trojice bezpečné, lebo pre ľubovoľnú trojicu d, e, f platí, že $d + e \geq a + b$ (lebo a, b sú dve najmenšie čísla v množine) a $f \leq c$ (lebo c je najväčšie číslo v množine). Potom ale z $a + b > c$ dostáme $d + e > c$, potom $d + e > f$, a teda trojica d, e, f je bezpečná.

Zhrnutie – množina paličiek neobsahuje nebezpečnú trojicu práve vtedy, keď najkratšie dve paličky spolu s najdlhšou nie sú nebezpečná trojica. Bezpečnosť množiny teda vieme overiť v čase $O(1)$ vďaka tomu, že čísla sú na vstupe usporiadané a nájsť najmenšie alebo najväčšie čísla nie je problém.

Zjednodušenie

Teraz sa budeme snažiť zistiť, ako nájsť najväčšiu bezpečnú množinu paličiek, lebo n mínus jej veľkosť je odpoveď na našu úlohu.

Vďaka predošlému zhrnutiu vieme, že bezpečnosť množiny závisí len od dvoch najkratších a jednej najdlhšej paličky. Preto, keď sa pozrieme na paličky v utriedenom poradí, (a my ich už na vstupe máme utriedené) tak sa nám oplatí nechať si nejaký súvislý úsek. Prečo? Keby najlepším riešením bol nejaký nesúvislý úsek, tak je v ňom nejaká diera – palička, ktorú sme z množiny vyhodili, ale dĺžkovo patrí medzi najkratšiu a najdlhšiu. Potom ale môžeme túto paličku nechať v množine a namiesto nej vyhodiť najkratšiu, a určite tým nič nepokazíme (rozmyslite si prečo). A zmenšíme dieru. Takto môžeme dieru zmenšovať, až kým nedostaneme súvislý úsek, ktorý bude tiež najlepším riešením.

Takže vieme, že sa stačí zaoberať len súvislými úsekmi. Tých je oproti všetkým množinám veľmi málo.

Riešenie

V poslednej časti sa pozrieme na to, ako nájsť najdlhší bezpečný úsek.

Mohli by sme napríklad vyskúšať všeky úseky, a pre každý overiť v čase $O(1)$, či je bezpečný. To by bolo riešenie s časovou zložitosťou $O(n^2)$. To je však príliš pomalé.

Všimneme si niekoľko vecí. Ak máme nejaký úsek, ktorý je bezpečný, tak všetky úseky, ktoré začínajú neskôr (a končia spolu s pôvodným úsekom alebo skôr), sú tiež bezpečné. Rovnako tie, ktoré končia skôr (a začínajú spolu s pôvodným úsekom alebo neskôr), sú tiež bezpečné. Obrátene, ak máme nejaký nebezpečný úsek, tak všetky úseky, ktoré začínajú skôr alebo vtedy, keď daný úsek, a končia neskôr alebo vtedy, keď aj on, sú tiež nebezpečné. To znamená, že pre konkrétny začiatok úseku Z existuje nejaká zlomová hranica H taká, že všetky úseky, ktoré začínaju v Z, ale končia skôr než H, sú bezpečné, a tie čo končia neskôr sú nebezpečné. (Napríklad majme čísla 7 8 9 9 10 11 13 13 14 15 16 17 17. Ako Z si zvolíme číslo 7. Potom úseky, ktoré končia číslom 14 alebo skôr, sú bezpečné, lebo $7 + 8 > 14 \geq 13 \geq 13 \geq 11 \ldots$, ale úseky, ktoré sú dlhšie a siahajú až po 15-ku alebo ďalej, sú nebezpečné, lebo $7 + 8 \not> 15$)

Preto nemusíme pozerať všetky úseky, ale pre každý začiatok Z stačí pozrieť len jeden – najdlhší bezpečný, teda ten po hranicu H. Túto zlomovú hranicu H by sme mohli napríklad binárne vyhľadávať s celkovým časom algoritmu $O(n \log n)$, ale my si ukážeme niečo ešte lepšie.

Keď sa pozrieme na dva začiatky, Z_1 a Z_2, pričom Z_1 je skôr ako Z_2, a ich príslušné zlomové hranice H_1 a H_2, tak zrejme H_1 nebude neskôr ako H_2. Preto môžeme riešiť úlohu metódou dvoch bežcov. Teda budeme mať dvoch bežcov, ktorí obaja začnú na začiatku poľa. V každom kroku sa prvý bežec pohne o 1 políčko doprava, jeho pozícia sa stane začiatkom Z. Druhý bežec sa bude chcieť postaviť na H. Z vyššie uvedených dôvodov stačí, ak sa rozbehne zo svojho predošlého políčka smerom doprava a bude bežať políčko po políčku, pokiaľ je úsek bezpečný.

Overíme, či úsek $Z - H$ nie je najdlhší a pokračujeme (prvý bežec o 1 políčko doprava, druhý zasa beží, pokiaľ je nový úsek bezpečný), celé to opakujeme až kým obaja nedobehnú do konca. Celé kúzlo je v časovej zložitosti – keďže každý bežec beží len doprava, mohol sa pohnúť najviac n-krát, a pohyb ľubovoľného bežca o políčko doprava nás stojí konštantný čas, tak celková časová zložitosť je $O(n)$. Pamäťová zložitosť je tiež $O(n)$.

Listing programu (C++)

```
#include<cstdio>
#define For(i, n) for(int i = 0; i<(n); ++i)

int n, A[200047], najlepsi, bezec;
int main(){
```

```
    scanf("%d", &n);
    najlepsi = min(n,2);
    For(i,n) scanf("%d",A+i);
    For(i,n){
        while (bezec<n && A[i]+A[i+1]>A[bezec]) bezec++;
        najlepsi = max(najlepsi,bezec-i);
    }
    printf("%d\n",n-najlepsi);
}
```

vzorák napísal Syseľ

4. Zahrajme sa

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

Na začiatok poznamenám, že pokiaľ budem hovoriť o mriežke, budem používať smery „hore“ a „dole“ zodpovedajúce popisu mriežky na vstupe napriek tomu, že mriežka je v skutočnosti vodorovná a teda by som mal používať „dopredu“ a „dozadu“.

Máme kváder, ktorý je niekde položený na mriežke. Všímajme si napríklad jeho ľavý horný roh. Zaberá niekoľko riadkov a niekoľko stĺpcov a má nejakú výšku. Celé toto nám presne popíše situáciu, a budeme to nazývať umiestnenie kvádra. Ako si neskôr ukážeme, spôsobov ako môže byť kváder natočený je len 6 a pozícii pre ľavý horný roh je najviac toľko ako políčok mriežky, teda celkovo máme najviac $6 \times r \times s$ rôznych umiestnení. Medzi niektorými dvoma umiestneniami sa vieme pohybovať „prevalením“ kvádra okolo nejakej hrany. Takéto umiestnenia budeme nazývať susedné.

Umiestnenia si môžeme predstaviť ako vrcholy grafu, pričom medzi susednými je hrana. Tie, v ktorých kváder leží na nejakej diere, vylúčime. Teraz už vieme pomocou prehľadávania do šírky zistiť, na koľko najmenej prevalení sa do nejakého umiestnenia vieme dostať. Ostáva nám poriešiť niekoľko implementačných detailov.

Najskôr popíšem, čo je to prehľadánie do šírky. Môžeme si to predstaviť ako rozdeľovanie všetkých umiestnení do skupín. Do skupiny 0 zaradíme iba pôvodné umiestnenie. Potom sa pozrieme na všetky susedné umiestnenia pôvodného a zaradíme ich do skupiny 1. Tu budú umiestnenia do ktorých sa vieme dostať na jedno prevalenie. Potom prejdeme všetky umiestnenia v skupine 1, pozrieme sa na ich susedov, ktorých sme ešte nevideli, a pridáme ich do skupiny 2. To budú umiestnenia, do ktorých sa vieme dostať na minimálne 2 prevalenia. Takto pokračujeme, kým nejaká skupina nebude prázdna – ďalej sa už dostať nevieme. Teda si musíme pre každé umiestnenie pamätať, či sme ho už videli a niekde si pamätať jednotlivé skupiny.

Skupiny sa najlepšie pamätajú v rade (fronte, queue). Umiestnenia stoja v rade podobnom ako je ten pred školským bufetom. Náš algoritmus vždy obslúži prvé umiestnenie a celý rad sa posunie. Ak bude chcieť v budúcnosti obslúžiť nejaké ďalšie umiestnenia (pretože patria do nasledujúcej skupiny), pošle ich na koniec radu. Občas sa na začiatku radu objaví oddeľovač – ten povie, že umiestnenia, ktoré sú za ním patria do nasledujúcej skupiny. Vtedy znova vložíme oddeľovač na koniec, aby sme všetko, čo vygenerujeme počas prehľadávania novej skupiny, poslali až do tej ďalšej. Ak sa v rade už nič okrem oddeľovača nenachádza, skončili sme.

Rad môžeme implementovať ako veľmi dlhé pole (také, aby sa doň zmestili všetky prvky), pričom si budeme pamätať, kde má začiatok a kde koniec. Pri odoberaní posunieme začiatok, pri pridávaní posunieme koniec. V prípade, že sa dostaneme ku koncu poľa, pokračujeme od začiatku, kde sú už staré, uvoľnené pozície. V kóde na obsluhu radu slúžia funkcie „`push(X)`“, ktorá pridá X na koniec, „`pop()`“, ktorá odoberie a vráti prvý prvok, a „`empty()`“, ktorá zistí, či je prázdny.

Ďalší problém je efektívne zistenie, či sa v pod kvádrom nachádza diera. Inak povedané, či sa v nejakom obdĺžniku nad mriežkou nachádza nejaké špeciálne políčko. Diery môžeme reprezentovať ako 1-tky, normálne políčka ako nuly. V tom prípade nás zaujíma, či je súčet všetkých políčok obdĺžnika nenulový. (Ak by bol nulový, boli by tam len normálne políčka.) Toto vieme zistiť pomocou prefixových súčtov. Prefix mriežky je každý obdĺžnik začínajúci v ľavom hornom rohu. Najskôr si pre prefix spočítame súčet jeho prvkov. Prechádzame všetkými možnými pravými dolnými rohmi prefixu a vždy k hodnote tohto rohu pripočítame súčty prefixov s pravým dolným rohom o políčko nad našim a naľavo od nášho, ktoré sme už vypočítali pred chvíľou. Takto sme ale započítali prefix o políčko diagonálne naľavo nahor od nášho dvakrát, teda ho raz musíme odrátať.

Keď už máme prefixové súčty, vieme v konštantnom čase zistiť súčet prvkov ľubovolného obdĺžnika. Zoberieme taký prefix, že náš obdĺžnik sa nachádza v jeho pravom dolnom rohu. Od neho odpočítame prefixy nad a naľavo od nášho obdĺžnika a pripočítame prefix diagonálne nad našim obdĺžnikom, ktorý sme odpočítali dvakrát.

Posledný problém, s ktorým sa treba popasovať, je natočenie kvádra a zmena súradnice ľavého horného rohu pri prevaľovaní. Rozmery kvádra si označíme číslami 0 až 2 tak, že nultý je počet stĺpcov mriežky, na ktorých kváder leží, prvý je počet riadkov mriežky, na ktorých kváder leží, a druhý je výška kvádra. Kváder môže byť otočený len 6 rôznymi spôsobmi. Každý z troch rozmerov môže byť jeho nultým rozmerom a pre každý nultý rozmer vieme meniť prvý a druhý rozmer. Otočenia si môžeme očíslovať 0 až 5 a do dvojrozmerného konštantného poľa P si potom pre každé natočenie a každý rozmer tohto natočenia predpočítame, ktorému

rozmeru pôvodného natočenia zodpovedá. Pôvodné natočenie má číslo 0, na číslovaní zvyšných nezáleží. A nakoniec si očíslujeme (0 až 3) hrany okolo ktorých možno kváder prevaliť. Potom pre každé natočenie a každú hranu si do poľa J predpočítame, do ktorého natočenia sa po prevalení po tejto hrane dostaneme. Polia DS a DR nám pre každú hranu povedia, či sa po prevalení po nej číslo riadka/stĺpca kvádra zvýši (`+1`), zníži (`-1`) alebo nezmení (`0`). V rade si ku každému umiestneniu pamätáme 3 čísla: číslo riadku a stĺpca ľavého horného rohu a natočenie. Vďaka tomuto vieme vždy jednoducho for-cyklom prejsť 4 hrany okolo ktorých sa chceme prevaliť a vygenerovať pozície a natočenia susedných umiestnení kvádra.

Zostáva nám určiť časovú a pamäťovú zložitosť. Celý graf má $O(rs)$ vrcholov a vrchol najviac 4 hrany, teda aj hrán je $O(rs)$. Prehľadávanie do šírky trvá lineárne od súčtu počtu hrán a vrcholov, prefixové sumy robíme tiež lineárne od počtu políčok, a teda časová zložitosť je $O(rs)$. Pamäťová zložitosť je tiež $O(rs)$, keďže veľkosť polí aj fronty je úmerná počtu políčok.

Po vyčerpávajúcom výklade si zaslúžite listing vzorového programu:

Listing programu (C++)

```
#include<iostream>
#include<string>

using namespace std;

// Predpočítané polia pre ľahšiu prácu s otáčaním
int P[6][3] = {
        {0, 1, 2},
        {0, 2, 1},
        {1, 0, 2},
        {1, 2, 0},
        {2, 0, 1},
        {2, 1, 0}
};

int DS[4] = {0,1,0,-1};
int DR[4] = {-1,0,1,0};

int J[6][4] = {
        {1,5,1,5},
        {0,3,0,3},
        {3,4,3,4},
        {2,1,2,1},
        {5,2,5,2},
        {4,0,4,0}
};

// Globálne premenné, ktoré používa fronta
const int queue_size = 1<<22;
int queue_MEM[queue_size];
int queue_front = 0;
int queue_last = 0;

// Funkcie fronty
void push(int value){
        queue_MEM[queue_last]=value;
        queue_last = (queue_last+1)%queue_size;
}

bool isEmpty(){
        return (queue_front == queue_last);
}

int pop(){
        if(isEmpty()) return -1;
        int res = queue_MEM[queue_front];
        queue_front = (queue_front+1)%queue_size;
        return res;
}

int main(){
        int r,s; // riadky, stĺpce
        int A[3]; // pôvodné rozmery
        int xr,xs; // pozícia červeného tlačidla
        string riadok; // buffer na čítanie vstupu po riadkoch
        cin >> r >> s;
        for(int i = 0; i < 3; i++) cin >> A[i];
        int SUM[r+1][s+1]; // Pole na prefixové sumy.
        bool BOL[r][s][6]; // Pamätáme si, či sme dané umiestnenie už videli
        for(int i = 0; i < r; i++) SUM[i][0] = 0;
        for(int j = 0; j < s; j++) SUM[0][j] = 0;
        for(int i = 0; i < r; i++){
                cin >> riadok; // Načítame riadok
                for(int j = 0; j < s; j++){
                        SUM[i+1][j+1] = SUM[i+1][j] + SUM[i][j+1] - SUM[i][j]; // Spočítame aktuálny prefix
                        if(riadok[j] == '#') SUM[i+1][j+1]++; // A ak sme na diere tak ju k prefixu pripočítame
                        if(riadok[j] == 'X'){ // Našli sme červené tlačidlo!
                                xr = i;
                                xs = j;
                        }
                        for(int k = 0; k < 6; k++) BOL[i][j][k] = false; // Ešte sme nikde neboli...
                }
        }
        int round = 0; // Kolo BFS, alebo ako ďaleko sme sa dostali
```

```
        push(0);
        push(0);
        push(0);
        push(-1); // Oddeľovač
        BOL[0][0][0] = true; //Boli sme na začiatku
        int actR, actS, actP, actRE, actSE, nextR, nextS, nextP, nextRE, nextSE;
        while(true){
                actR = pop(); // Aktuálny riadok
                if(actR == -1){ // A nie je to oddeľovač?
                        if(isEmpty()){ // A je ešte niečo v rade?
                                round = -1; // Nie je ... asi sa na červené tlačidlo nedostaneme
                                break;
                        } else {
                                round++; // Je, ideme do ďalšieho kola
                                push(-1);
                                continue;
                        }
                }
                actS = pop(); // Aktuálny stĺpec
                actP = pop(); // Aktuálne natočenie
                actRE = actR+A[P[actP][1]]; // Prvý riadok pod kvádrom
                actSE = actS+A[P[actP][0]]; // Prvý stĺpec za kvádrom
                if( (actR <= xr) && (actS <= xs) && (xr < actRE) && (xs < actSE) ){
                        break; // Našli sme červené tlačidlo!
                }
                for(int i = 0; i < 4; i++){ // Prejdeme 4 hrany
                        nextP = J[actP][i]; // Nasledujúce natočenie
                        // Kam sa posunie ľavý horný roh?
                        if(DR[i] != -1) {
                                // Ak sa riadok nemení, DR[i] je 0,
                                // Ak sa zvyšuje, tak o počet riadkov čo teraz kváder zakrýva
                                nextR = actR + DR[i]*A[P[actP][1]];
                        } else {
                                // Pokiaľ sa otáčame hore, klesne riadok o výšku
                                nextR = actR - A[P[actP][2]];
                        }
                        if(DS[i] != -1) {
                                // Ak sa stĺpec nemení, DS[i] je 0,
                                // Ak sa zvyšuje, tak o počet stĺpcov čo teraz kváder zakrýva
                                nextS = actS + DS[i]*A[P[actP][0]];
                        } else {
                                // Pokiaľ sa otáčame vľavo, klesne stĺpec o výšku
                                nextS = actS - A[P[actP][2]];
                        }
                        // A kde budú nové hranice kvádra ?
                        nextRE = nextR+A[P[nextP][1]];
                        nextSE = nextS+A[P[nextP][0]];
                        if( (nextR < 0) || (nextS < 0) || (r < nextRE) || (s < nextSE) ) continue; // Sme na ploche?
                        if(BOL[nextR][nextS][nextP]) continue; // Je pozícia nová
                        BOL[nextR][nextS][nextP] = true; // Teraz už nie ... :)
                        // A nie sme nad dierou?
                        if( SUM[nextRE][nextSE] + SUM[nextR][nextS] != SUM[nextR][nextSE] + SUM[nextRE][nextS] ) continue;
                        // Ak je všetko v poriadku, toto umiestnenie chceme v budúcnosti spracovať
                        push(nextR);
                        push(nextS);
                        push(nextP);
                }
        }
        // A hurá vypísať odpoveď
        cout << round << endl;
        return 0;
}
```

opravovalo sa samo, vzorák napísal MišoF
(max. 0 b za popis, 12 b za program)

5. Odchod o 5 minút

V tejto úlohe nešlo ani tak o to napísať funkčný program. Na to samozrejme existuje milión rôznych postupov a netreba pri nich riešiť žiaden algoritmický problém. Hlavným cieľom tejto úlohy bolo skúsiť napísať program, ktorý bude stručný, jednoduchý a ktorého písanie prebehne pokiaľ možno bezbolestne. Koľko riadkov kódu má vaše riešenie? Vzorové ich má zhruba desať.

V tomto vzorovom riešení použijeme Python3 a odrodu regulárnych výrazov ním podporovanú. Riešenie s podobnou myšlienkou sa určite (aj keď možno v trochu zložitejšej podobe) dá napísať aj v ľubovoľnom inom jazyku podporujúcom regulárne výrazy.

Rozsekanie textu stránky na popisy spojení

Na nájdenie potrebných údajov vo vstupe použijeme regulárne výrazy. Najskôr si napíšeme jeden, ktorému bude zodpovedať kus stránky popisujúci jedno možné spojenie – teda jednu postupnosť liniek, ktorá nás dovezie zo štartu do cieľa.

Keď si otvoríme niektorý z príkladov vstupu v bežnom textovom editore, ľahko si napríklad všimneme, že každé spojenie má na svojom začiatku tag „`<a␣name="spojenieX"></a>`“, na ktorý ukazujú linky umiestnené inde na stránke. Podľa tohto tagu sa teda bude dať dobre spoznať, kde začína nový popis spojenia. A kde končí? Tam, kde nasledujúci začína. Teda až na posledný popis spojenia, tomu musíme nájsť nejaký iný text, ktorý

nám povie, kde ten popis končí. Prvé, čo sa za jeho popisom nachádza, je reťazec „`<div␣class="netlacit">`“, použijeme teda ten.

V nasledujúcom regulárnom výraze je použitá jedna z moderných fičúrii, ktoré sme si v predchádzajúcich sériách neukázali. Volá sa *look-ahead assertion*, čo si môžeme preložiť ako „kontrola nasledujúceho textu“. V regulárnych výrazoch v Pythone sa zapisuje nasledovne: „`(?=text)`“. Táto vzorka zodpovedá presne tomu istému, čomu by zodpovedala vzorka „`text`“, s jediným rozdielom: text zodpovedajúci takto skontrolovanej vzorke sa *nepovažuje* za prečítaný. Ak by teda vzorka pokračovala, alebo (ako je to aj v našom prípade) sme hľadali nasledovný výskyt tej istej vzorky, text skontrolovaný pomocou look-ahead assertion sa môže zároveň stať súčasťou nasledujúceho výskytu vzorky.

Najskôr si to ukážeme na jednoduchom príklade: Máme v riadku niekoľko záznamov, pričom pred aj za každým je bodkočiarka. Chceme napísať regulárny výraz, ktorý nájde všetky záznamy. Nájsť jeden záznam je ľahké: nájdeme bodkočiarku, ľubovoľne veľa iných znakov, a znova bodkočiarku. Ak ale dáme nájsť v riadku všetky výskyty tohto regulárneho výrazu, nedostaneme to, čo chceme:

```
>>> riadok = ';10;20;30;40;50;'
>>> re.findall( r';[^;]*;', riadok )
[';10;', ';30;', ';50;']
```

Čo sa stalo? Prvý výskyt nám „zožral“ aj bodkočiarku, ktorou začínal výskyt obsahujúci číslo 20. A práve toto vieme elegantne napraviť pomocou look-ahead assertion:

```
>>> re.findall( r';[^;]*(?=;)', riadok )
[';10', ';20', ';30', ';40', ';50']
```

Vyššie použitý regulárny výraz teda môžeme čítať: „bodkočiarka, ľubovoľne veľa iných znakov, a za nimi vidím, ale nespracujem ďalšiu bodkočiarku“.

Ešte jedna časť vyššie uvedeného regulárneho výrazu sa dá zapísať aj krajšie (alebo aspoň ináč – krása je vecou vkusu). Ďalšia užitočná syntax, ktorú si ukážeme, je iná verzia iterácie. Kvantifikátor „`*`“, použitý v Pythonovskom regulárnom výraze, sa správa „pažravo“: ak má na výber viac možností, vyberie si tú, pri ktorej mu bude zodpovedať čo najviac textu. O tom sa ľahko presvedčíme ďalším experimentom:

```
>>> re.findall( r';.*;', riadok )
[';10;20;30;40;50;']
```

Existuje však aj opačná verzia tohto kvantifikátora: „`*?`“. Tej vždy zodpovedá najmenší možný úsek textu. Napríklad vzorku „`;.*?;`“ môžeme čítať nasledovne: „bodkočiarka, čo najmenej ľubovoľných znakov, ďalšia bodkočiarka“.

Keď teda spojíme tento kvantifikátor a look-ahead assertion, dostávame nové riešenie:

```
>>> re.findall( r';.*?(?=;)', riadok )
[';10', ';20', ';30', ';40', ';50']
```

Regulárny výraz, ktorý nájde popis jedného spojenia, teraz môžeme napísať napr. nasledovne:
„`<a␣name="spojenie.*?(?=<a␣name="spojenie|<div␣class="netlacit")`“.

V tomto okamihu je dobré skontrolovať si, či je všetko v poriadku. Napíšeme teda program, ktorý načíta celý vstup a spočíta v ňom všetky výskyty tejto vzorky:

```
import re, sys
page_contents = sys.stdin.read().replace('\n','␣')
connection_pattern = r'<a␣name=”spojenie.*?(?=<a␣name=”spojenie|<div␣class=”netlacit”)'
print( len( re.findall( connection_pattern, page_contents ) ) )
```

Keď tento program spustíme, vypíše nám číslo 5, takže všetko je, ako má byť.

Posledná vec, ktorú zmienime v tejto časti: všimnite si príkaz `replace('\n','␣')`, ktorým sme zmenili všetky konce riadkov na medzery. V Pythone totiž pri štandardných nastaveniach znaku „`.`“použitému v regulárnom výraze zodpovedá akýkoľvek znak *okrem* konca riadku. A keďže nám na riadkoch nezáleží, jednoducho sa koncov riadkov zbavíme. (Alternatívnym riešením bolo použiť v príkaze `re.findall` prepínač `re.DOTALL`.)

Rozsekanie jedného spojenia na údaje o linkách

Tu už potrebujeme len spraviť to isté v bledomodrom: nájsť v popise spojenia všetky čísla liniek a tiež všetky časy príchodu/odchodu a im zodpovedajúce zastávky. Ide to samozrejme aj jedným regulárnym výrazom, nám sa však viac páčilo použiť dva stručnejšie: jedným si nájdeme všetky čísla liniek, druhým zas všetky navštívené zastávky, a na konci to už len dáme celé dokopy a vypíšeme.

Čísla liniek sú ľahké. Tie v popise spojenia vyzerajú nasledovne:
`<span␣class="linka␣ba␣l53"␣style="font-size:␣2em;">53</span>`
Vieme ich teda nájsť napríklad nasledujúcim regulárnym výrazom: „`<span␣class="linka.*?>(.*?)<`“. Číslo linky bude zodpovedať časti vzorky, označenej zátvorkou použitou v našom regulárnom výraze.

Alternatívnym riešením by bolo použiť okrem look-ahead aj tzv. look-behind assertion a napísať vzorku, ktorej bude zodpovedať len samotné číslo linky. Je však lepšie zvyknúť si časti vzorky, ktoré ma skutočne zaujímajú, označiť zátvorkami. Oceníme to, akonáhle bude takých častí viac ako jedna – teda hneď v nasledujúcom odseku.

Popis času a zastávky vyzerá takto: `<b>16:51</b>␣</td><td␣width="200"><b>Vajnory</b>␣</td>`. Zase teda napíšeme nejakú vzorku, ktorá nájde takto vyzerajúce kusy popisu spojenia a vyberie z nich len čas a názov zastávky. Celé to môže vyzerať napríklad takto: „`<b>([0-9]+:[0-9]+)</b>␣</td><td.*?><b>(.*?)</b>')`“. Prípadne to môžeme ďalej zjednodušiť, ako je ukázané nižšie.

A to už je naozaj všetko.

Listing programu (Python)

```
import re, sys
page_contents = sys.stdin.read().replace('\n','␣')
conn_pattern = r'<a␣name="spojenie.*?(?=<a␣name="spojenie|<div␣class="netlacit")'

for num,conn in enumerate(re.findall( conn_pattern,page_contents )):
    if num>0: print()
    L = [ x.group(1) for x in re.finditer(r'<span␣class="linka.*?>(.*?)<',conn) ]          # all the lines
    S = [ x.groups() for x in re.finditer(r'<b>([0-9]+:[0-9]+)</b>.*?<b>(.*?)</b>',conn) ] # all the stops
    for i in range(len(L)):
        print('{}\n{}␣{}\n{}␣{}'.format(L[i], S[2*i][0], S[2*i][1], S[2*i+1][0], S[2*i+1][1]))
```

vzorák napísal Žaba

6. Olívia vstupuje na scénu

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

Dúfam, že sa vám zadanie s Olíviou páčilo, určite to nie je posledný raz, čo ste sa s ňou stretli. Poďme teraz rovno na vysvetľovanie vzoráku tejto úlohy. Nebola to úloha príliš zložitá, stačilo si všimnúť zopár maličkostí, z ktorých sa riešenie vykľulo aj samo.

Najdôležitejšie pozorovanie, ktoré potrebujeme k riešeniu tejto úlohy, je to, že jednotlivé presuny vieme trochu prehadzovať bez toho, aby sme zmenili dané riešenie. Zoberme si nejaké optimálne riešenie a nech to začína na plošine x a končí napravo od plošiny x. Pohyby, ktoré sme spravili naľavo od x, vytvoria niekoľko súvislých úsekov, ktoré začínajú aj končia v x. A rovnako aj na pravej strane nám vznikne zopár súvislých úsekov pohybov, ktoré začínajú aj končia v x (s výnimkou posledného).

To ale znamená, že vieme naše optimálne riešenie ľahko prerobiť tak, že najskôr sa pohybujem naľavo od x a keď už nemôžem, tak až vtedy prejdem doprava a už sa naľavo od x nikdy neposuniem. To nám dáva návod na to, ako by sme mohli vyriešiť našu úlohu. Pre každú plošinu x si spočítam štyri informácie:

- Koľko najviac mostov viem prejsť na plošinách 1 až x tak, aby som skončil na plošine x.
- Koľko najviac mostov viem prejsť na plošinách 1 až x tak, že môžem skončiť kdekoľvek.
- Koľko najviac mostov viem prejsť na plošinách x až n tak, aby som skončil na plošine x.
- Koľko najviac mostov viem prejsť na plošinách x až n tak, že môžem skončiť kdekoľvek.

S týmito informáciami nie je problém zistiť výsledok. Skúsime začať na každej plošine a vieme, že optimálne riešenie bude vyzerať tak, že najskôr pôjdeme jedným smerom, vrátime sa a potom pôjdeme druhým smerom. S informáciami, ktoré máme, je to ľahké spočítať. Potom len zoberieme maximum z výsledkov.

Posledný problém je, ako porátať dané hodnoty. Použijeme dynamické programovanie. Najskôr prejdeme pole zľava doprava a zrátame prvé dve odrážky. Ak začíname v x a chceme sa do x vrátiť, tak to znamená, že po moste medzi $x-1$ a x prejdeme párny počet krát (samozrejme najväčší možný) a plus čo najviac sa pohnem naľavo od plošiny $x-1$, pričom sa však musím vrátiť, čo je hodnota, ktorú už máme predrátanú. Druhú hodnotu získame tak, že prejdeme po moste medzi x a $x-1$ nepárny počet krát a potom pôjdeme z $x-1$ kamkoľvek. Plus netreba zabudnúť, že občas sa nám oplatí aj vrátiť, keďže môžeme skončiť kdekoľvek, ale to už máme vyrátané. To isté spravíme aj v opačnom smere.

Keďže len párkrát prejdeme celé pole a na rátanie hodnôt spotrebúvvame konštantný čas, časová zložitosť bude $O(n)$. Rovnako aj pamäťová, keďže si minimálne celé pole musíme pamätať.

Listing programu (C++)

```
#include <cstdio>
#include <iostream>
#include <string>
#include <algorithm>
```

```
#include <vector>
#include <cmath>
#include <queue>
#include <set>
#include <map>
using namespace std;

#define For(i,n) for(int i=0; i<(n); i++)
typedef long long ll;

ll P1[100047][3];
ll P2[100047][3];

int main()
{
    int n;
    scanf("%d␣",&n);
    vector<int> A; A.resize(n-1);
    For(i,n-1) scanf("%d␣",&A[i]);
    For(i,n-1)
    {
        P1[i+1][0]=P1[i][0]+A[i]-A[i]%2;
        P1[i+1][1]=max(P1[i][0],P1[i][1])+A[i]-(A[i]+1)%2;
        P2[n-i-2][0]=P2[n-i-1][0]+A[n-i-2]-A[n-i-2]%2;
        P2[n-i-2][1]=max(P2[n-i-1][0],P2[n-i-1][1])+A[n-i-2]-(A[n-i-2]+1)%2;
        if(A[i]==1) P1[i+1][0]=0;
        if(A[n-i-2]==1) P2[n-i-2][0]=0;
    }
    ll vys=0;
    For(i,n) vys=max(vys,max(P1[i][0]+max(P2[i][0],P2[i][1]),P2[i][0]+max(P1[i][0],P1[i][1])));
    cout << vys << endl;
return 0;
}
```

vzorák napísal Bob

7. Odfotiť si škôlku

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

Na siedmej priečke vás čakala vcelku jednoduchá úloha... Pravda, ak ste dostali správny nápad.

Zoberme si ľubovoľnú dvojicu detí u, v a pozrime sa na ich vzájomnú polohu v rade (ktoré dieťa je naľavo a ktoré napravo). V permutáciách zadaných na vstupe sa vzájomná poloha u a v môže líšiť od tej v permutácii π, no nie príliš často. Totiž aby sa deti u a v na fotke vymenili, musí aspoň jedno z nich vyjsť z radu a zaradiť sa na zlé miesto – to sa však vďaka garanciám v zadaní môže stať najviac dvakrát.

Na základe tejto myšlienky vieme o každej dvojici detí ľahko rozhodnúť, ktoré z nich má byť vo výslednom poradí π viac vľavo: Stačí sa pozrieť na päť zadaných permutácií a zvoliť si tú možnosť, ktorá sa v nich opakuje častejšie (teda aspoň trikrát). Toto rozhodovanie môžeme implementovať v konštantnom čase, ak si pre každý prvok zapamätáme, na ktorých pozíciách sa v zadaných permutáciách nachádza.

Vieme teda zistiť vzájomnú polohu ľubovoľnej dvojice detí. Ako z toho naskladať kompletnú permutáciu? Mohli by sme porovnať každý prvok s každým; na i-tu pozíciu v rade by potom patril ten prvok, od ktorého má byť práve $i-1$ prvkov naľavo. Takéto riešenie by však malo časovú zložitosť až $O(n^2)$.

Vo vzorovom riešení namiesto toho využijeme triedenie. Vezmime ľubovoľnú permutáciu čísel $1, 2, \ldots, n$ a zotrieďme ju, pričom ako porovnávaciu funkciu použijeme rozhodovanie o vzájomnej polohe dvojice detí. Dostaneme tak π (čísla budú usporiadané od „najľavejšieho v π“ po „najpravejšie v π“). Ak na triedenie použijeme niektorý z optimálnych algoritmov, získame riešenie s časovou zložitosťou $O(n \log n)$.

Listing programu (C++)

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

vector<vector<int> > A;  // A[i][j] = pozícia prvku j v i-tej permutácii

bool compare(int u, int v) {  // vráti true, ak má byť prvok u naľavo od v
    int score = 0;
    for (int i = 0; i < 5; ++i)
        score += A[i][u] < A[i][v];
    return score >= 3;
}

int main() {
    int n;
    scanf("%d", &n);
    A.resize(5, vector<int>(n));
    for (int i = 0; i < 5; ++i)
        for (int j = 0; j < n; ++j) {
            int x;
            scanf("%d", &x);
            A[i][x - 1] = j;
        }

    vector<int> P(n);  // začneme s identickou permutáciou
```

```
    for (int i = 0; i < n; ++i)
        P[i] = i;
    sort(P.begin(), P.end(), compare);

    for (int i = 0; i < n; ++i)
        printf("%d%c", P[i] + 1, (i == n - 1) ? '\n' : ' ');
}
```

vzorák napísal Usámec

8. Otravná blbosť

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

Predstavme si najprv, že čísla a, b sú trochu menšie (maximálne milión).

V tomto prípade vieme úlohu riešiť dvoma pomerne priamočiarymi prístupmi. Prvý z nich nad číslami skonštruuje orientovaný graf. Vrcholy grafu sú jednotlivé čísla a z vrcholu x do y vedie hrana práve vtedy, keď sa vieme na jednou operáciou dostať z čísla x číslo y. Keď máme takýto graf, tak môžeme pustiť z vrcholu a prehľadávanie do hĺbky a zistiť najmenší počet operácii, aby sme dostali číslo b.

Druhý prístup je trochu drzejší. Zoberme si naše číslo a. A použime takú operáciu, ktorá z neho vyrobí najmenšie možné číslo, ktoré nie je menšie ako b.

V tomto prípade potrebujeme dôkaz toho, že takýto postup je zaručene optimálny. Označme si ako $K(x)$ minimálny počet operácií nutných na zmenenie čísla x na číslo b. Dokážeme nasledovné tvrdenie: Ak $y < z$, potom $K(y) \leq K(z)$. Zoberme si nejakú optimálnu postupnosť operácii na zmenenie čísla z. Tieto operácie nám vyrobia postupnosť čísel $z = z_0, z_1, \ldots, z_{K(z)} = b$. Zoberme z nich také z_i, že platí $z_i < y$ a $z_{i-1} \geq y$. Potom rovnakou operáciou akou zo z_{i-1} vyrobíme z_i, vieme aj z y vyrobiť z_i.

Z dokázeného tvrdenia nám vyplýva, že ak greedy spôsobom zvolíme v danom bode operáciu, ktorá vyrobí najmenšie číslo ($\geq b$), tak určite nič nepokazíme, lebo sa dostaneme najbližšie k b ako to len ide.

Teraz ešte potrebujeme náš algoritmus rozšíriť na väčší rozsah čísel a, b. Zoberme si číslo $x = nsn(2, 3, \ldots, k)$. Nech teraz $a > x$ a $b < x$. Všimnime si, že nech je postupnosť operácii akákoľvek, tak vždy budeme musieť prejsť cez číslo x. Toto sa dá rozšíriť na tvrdenie typu: Nech $a \geq nx$ a $b \leq mx$, potom každá postupnosť operácii prejde cez čísla $nx, (n-1)x, \ldots, mx$. Navyše si môžeme všimnúť, že na optimálny prechod medzi ix a $(i-1)x$ nám stačia rovnaké operácie ako na prechod medzi $2x$ a x. Z toho vyplýva výsledný algoritmus:

- Nájdi najväčšie n také, že $a > nx$, najmenšie m také, že $b \leq mx$.
- Ak $m \leq n$, tak priamo spočítaj počet operácii na prechod z a do b.
- Spočítaj optimálny počet operácii na prechod z a do nx, z $2x$ do x a z mx do b. Označ ich postupne p, q, r. Výsledok bude $p + (n-m)q + r$.

Ešte zhodnoťme časovú zložitosť. Najviac času zaberá zisťovanie počtu operácii na prechod z čísla c do čísla d, na to potrebujeme $k(c-d)$ času. Celkovo to vieme odhadnúť na čas $O(kx)$. Pamäťová zložitosť bude konštantná, pokiaľ použijeme greedy prístup.

Listing programu (C++)

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

using namespace std;

int nsn(int a, int b) {
  return a*b/__gcd(a,b);
}

long long GetDist(long long a, long long b, int k) {
  long long dist = 0;
  while (a != b) {
    long long mi = a - 1;
    for (int i = 2; i <= k; i++) {
      if (a - (a % i) < b) continue;
      mi = min(mi, a - (a % i));
    }
    a = mi;
    dist++;
  }
  return dist;
}

int main() {
  long long a, b, k;
  cin >> a >> b >> k;
  int x = 1;
```

```
  for (int i = 2; i <= k; i++) {
    x = nsn(x, i);
  }
  long long n = a / x;
  long long m = (b+x-1) / x;
  if (n < m) {
    cout << GetDist(a, b, k) << endl;
    return 0;
  }
  cout << GetDist(a, n*x, k) + (n-m)*GetDist(2*x, x, k) +
          GetDist(m*x, b, k) << endl;
}
```


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