Korešpondenčný seminár z programovania
XXXIX. ročník, 2020/21
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

# Vzorové riešenia 1. kola zimnej časti

Dano a Naďka

## 1. Pokazená tlačiareň

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

V tejto úlohe sme mali zistiť, koľko obrázkov vedia Mišof a Hodobox vyfarbiť, keď majú dané koľko ktorých voskoviek majú k dispozícii a koľko ktorých voskoviek je potrebných na vyfarbenie každého typu obrázku.

### Vzorové riešenie

Nech v optimálnom riešení máme $x$ obrázkov typu 1 a $y$ obrázkov typu 2. Našou úlohou je zistiť čísla $x, y$ tak, aby $x + y$ bolo maximálne možné.

Môžeme postupne skúšať všetky dosiahnuteľné hodnoty $x$. Ku každej z nich si ľahko vieme dorátať, koľko ktorých voskoviek nám ostane po vyfarbení $x$ obrázkov typu 1. Všetky tieto zvyšné voskovky môžeme použiť na vyfarbenie čo najviac obrázkov druhého typu. Jednoducho si teda dorátame číslo $y$.

Skúsme sa bližšie pozrieť na to, ako dopočítať číslo $y$. Ak sme na $x$ obrázkov prvého typu použili $x_b = x \cdot b_1$ voskoviek béžovej, $x_r = x \cdot r_1$ ružovej a $x_m = x \cdot m_1$ voskoviek modrej farby, tak na obrázky typu 2 nám ostalo $y_b = b - x_b$ voskoviek béžovej, $y_r = r - x_r$ ružovej a $y_m = m - x_m$ modrej farby. Ako zistíme, koľko obrázkov druhého typu vieme týmito voskovkami zafarbiť? Na vyfarbenie jedného obrázku typu 2 potrebujeme $b_2, r_2, m_2$ voskoviek príslušných farieb. Jedna z týchto farieb voskoviek sa nám minie ako prvá. Obrázkov typu 2 teda vieme vyfarbiť $y = \min\{\frac{y_b}{b_2}, \frac{y_r}{r_2}, \frac{y_m}{m_2}\}$.

Pre nejaké nami zvolené $x$ sme teda dopočítali maximálne možné $y$. Na doriešenie úlohy nám teda stačí postupne skúsiť všetky možné $x$, ku každému dopočítať $y$ a vypísať tú možnosť, kde bolo $x + y$ najväčšie.

Prečo to funguje? Pretože skúšame všetky možnosti. Žiadna nám teda neunikne.

### Zložitosť

Koľko rôznych $x$ potrebujeme skúšať? Na každý obrázok spotrebujeme aspoň jednu voskovku každého druhu. Stačí nám teda vyskúšať $\min\{b, r, m\}$ možností pre $x$. Ku každému vieme potom dopočítať $y$ v konštantnom čase. Časová zložitosť teda bude $O(\min\{b, r, m\})$.

Pamätať si nám stačí jednotlivé počty voskoviek, ktoré máme k dispozícii a doteraz najlepšie nájdené $x, y$. Stačí nám teda konštantná pamäť a pamäťová zložitosť bude $O(1)$.

### Listing programu (C++)

```
#include <bits/stdc++.h>

using namespace std;

int main() {
    vector<int> mame(3);
    for(int i = 0; i < 3; i++) {
        cin >> mame[i];
    }
    vector<vector<int>> obrazky(2, vector<int>(3));
    int max_x = INT_MAX;
    for(int typ = 0; typ < 2; typ++) {
        for(int farba = 0; farba < 3; farba++) {
            cin >> obrazky[typ][farba];
            if(typ == 0) {
                max_x = min(max_x, mame[farba] / obrazky[typ][farba]);
            }
        }
```

```
    }
    int odpoved = 0;
    // Skusime postupne vsetky mozne x
    for(int x = 0; x <= max_x; x++) {
        int y = INT_MAX; // Kolko obrazkov typu 2 vieme este vyfarbit zvysnymi
            ↪ voskovkami
        for(int farba = 0; farba < 3; farba++) {
            int ostalo_farby = mame[farba] - x * obrazky[0][farba];
            y = min(y, ostalo_farby / obrazky[1][farba]);
        }
        odpoved = max(odpoved, x + y);
    }
    cout << odpoved << "\n";

    return 0;
}
```

**Listing programu (Python)**

```
def nacitaj():
    return [int(t) for t in input().split()]

mame, prvy, druhy = [nacitaj() for _ in range(3)]

odpoved = 0
for x in range(mame[0] + 1):
    ostalo = [mame[farba] - x * prvy[farba] for farba in range(3)]
    if any([farba < 0 for farba in ostalo]):
        break
    y = min([ostalo[j] // druhy[j] for j in range(3)])
    odpoved = max(odpoved, x + y)

print(odpoved)
```

## 2. Obmenené zátvorky

12 b za popis, 8 b za program

Skúsme si najprv povedať, v akých situáciách budeme vedieť zo zápisu zátvoriek povedať, že táto postupnosť nemohla nastať. Sú to tieto situácie:

- v zápise je, že by sme mali zo stola zobrať nejakú voskovku, ktorá už nie je na stole, alebo ani nikdy na stole nebola
- v zápise je, že by sme mali vrátiť na stôl nejakú voskovku, ktorú sme si nikdy nezobrali

Ako takéto veci kontrolovať? Jednoducho postupne prechádzame postupnosť zátvoriek a pamätáme si počet voskoviek jednotlivých druhov, ktoré aktuálne sú na stole. Vždy, keď nájdeme nejakú zátvorku, tak si správnym spôsobom zmeníme aktuálny počet voskoviek na stole. Ak sa hocikedy v priebehu stane, že by počet nejakých voskoviek klesol pod 0 alebo stúpol nad počet, koľko voskoviek toho druhu máme v škôlke, tak vypíšeme `NIE`. Netreba zabudnúť skontrolovať, že na konci dňa (na konci vstupu) musia byť na stole všetky voskovky. V prípade ak nie sú, tak tiež vypíšeme `NIE`. V ostatných prípadoch vypíšeme `ANO`.

Časová aj pamäťová zložitosť je lineárna, priamo úmerná dĺžke postupnosti zátvoriek.

**Listing programu (C++)**

```
#include<iostream>

using namespace std;
```

```
int main(){
    int max_gulate, max_kucerave, max_spicate, max_hranate;
    int gulate=0, kucerave=0, spicate=0, hranate=0;
    cin>>max_gulate>>max_kucerave>>max_spicate>>max_hranate;
    string vstup;
    getline(cin, vstup); //nacitame znak konca riadku
    getline(cin, vstup);

    for(int i=0;i<vstup.size();i++){
        if(vstup[i]=='(')
            gulate++;
        if(vstup[i]==')')
            gulate--;
        if(vstup[i]=='[')
            hranate++;
        if(vstup[i]==']')
            hranate--;
        if(vstup[i]=='{')
            kucerave++;
        if(vstup[i]=='}')
            kucerave--;
        if(vstup[i]=='<')
            spicate++;
        if(vstup[i]=='>')
            spicate--;

        if(gulate<0 || hranate<0 || kucerave<0 || spicate<0 ||
           gulate>max_gulate || hranate>max_hranate ||
           kucerave>max_kucerave || spicate>max_spicate){
            cout<<"NIE"<<endl;
            return 0;
        }
    }

    if(gulate==0&&hranate==0&&kucerave==0&&spicate==0){
        cout<<"ANO"<<endl;
    }else{
        cout<<"NIE"<<endl;
    }

    return 0;
}
```

**Listing programu (Python)**

```
def spravny(voskovky, k_dispozicii):
    d = {"(":(0,1),"{":(1, 1),"<":(2, 1),"[":(3, 1),")":(0,-1),"}":(1, -1),">"
        ↪ :(2, -1),"]":(3, -1)}
    '''
    v d je kluc dany znak voskovky a hodnota je dvojica,
    kde prve cislo je index do pola poctov a
    druhe cislo je o kolko ho zmenime (plus alebo minus 1)
    '''
    pocty_voskoviek = [0,0,0,0] # kolko voskoviek aktualne mame
    for voskovka in voskovky:
        kde, o_kolko = d[voskovka]
        pocty_voskoviek[kde] += o_kolko
        for i in range(len(pocty_voskoviek)):
```

```
            # ak niekedy mame voskoviek menej ako 0 alebo viac ako ich mame k
              ↪ dispozicii
            if pocty_voskoviek[i] < 0 or pocty_voskoviek[i] > k_dispozicii[i]:
                return False

    for i in pocty_voskoviek:
        # ak na konci dna nie su vsetky vratene
        if i != 0:
            return False

    return True

k_dispozicii = list(map(int, input().split()))
voskovky = input()
if spravny(voskovky, k_dispozicii):
    print("ANO")
else:
    print("NIE")
```

Sabinka

## 3. Strašná kopa papierov

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

### Priamočiare riešenie

Ako priamočiare riešenie nám môže napadnúť, že odsimulujeme, čo sa stane pre každý papier každej kôpky. Teda si budeme pamätať aktuálny kúsok voskovky ktorý nám ostáva a postupne pre každý papier každej kôpky sa spýtame, či je kus voskovky ktorý nám aktuálne ostáva dostatočne dlhý na jeho celé zafarbenie. Ak nie, vieme že bude viacfarebný a zapamätáme si to do výsledného počtu. Nesmieme pri tom zabudnúť "odobrať" aj časť ďalšej voskovky (ktorá sa použije na vyfarbenie), prípadne niekoľko celých a časť ďalšej, prípadne len niekoľko celých (podľa toho, koľko ešte potrebujeme na dovyfarbenie papiera). Kód by mohol vyzerať takto:

### Listing programu (C++)

```
#include<iostream>

using namespace std;

int main(){
    long long n,m;
    cin>>n;
    long long d[n];
    for(int i=0;i<n;i++){
        cin>>d[i];
    }
    cin>>m;
    long long c[m], k[m];
    for(int i=0;i<m;i++)
        cin>>k[i]>>c[i];

    long long index_pastelka=0, roznofarebne=0;

    for(long long i=0;i<m;i++){
        for(long long j=0;j<k[i];j++){
            long long nafarbit=c[i];
            bool roznofarebny_papier=false;
            while(nafarbit!=0){
                if(d[index_pastelka]>=nafarbit){
```

```
                    d[index_pastelka]-=nafarbit;
                    nafarbit=0;
                }else{
                    roznofarebny_papier=true;
                    nafarbit-=d[index_pastelka];
                    d[index_pastelka]=0;
                    index_pastelka++;
                }
                if(d[index_pastelka]==0)
                    index_pastelka++;
            }
            if(roznofarebny_papier)
                roznofarebne++;
        }
    }

    cout<<roznofarebne<<endl;
}
```

**Listing programu (Python)**

```
n = int(input())
dlzky = []
for i in range(n):
    dlzky.append(int(input()))
m = int(input())
skupinky = []
for i in range(m):
    skupinky.append(list(map(int, input().split())))


index_voskovka = 0
ostava_z_voskovky = dlzky[index_voskovka]
roznofarebne = 0

for i in skupinky:
    pocet, treba_nafarbit = i[0], i[1]
    for obr in range(pocet):
        treba_nafarbit =  i[1]

        if ostava_z_voskovky == 0:
            index_voskovka += 1
            ostava_z_voskovky = dlzky[index_voskovka]

        if ostava_z_voskovky >= treba_nafarbit:
            ostava_z_voskovky -= treba_nafarbit

        else:
            roznofarebne += 1
            while treba_nafarbit > ostava_z_voskovky:
                index_voskovka += 1
                ostava_z_voskovky += dlzky[index_voskovka]
            ostava_z_voskovky -= treba_nafarbit


print(roznofarebne)
```

Problémom ale je, že takéto riešenie nebude dostatočne efektívne, keďže potrebujeme $O(m \cdot k)$ operácií a v zadaní vidíme, že $m$ a $k$ môžu byť každé až 200 000. Teda potrebujeme rozhodovať o zhruba 4 000 000 000 papierov. To je priveľa a náš program to nestihne.

### Ako to zlepšiť ?

Tu si treba všimnúť, že ak by sme namiesto každého papiera každej kôpky prechádzali voskovkami, budeme potrebovať oveľa menej operácií, lebo podľa obmedzení, voskoviek bude najviac milión. Potom náš algoritmus bude fungovať tak, že si pamätá, koľko ostáva z aktuálnej voskovky a až kým sa neminie, berie celé kôpky a pýta sa:

- dokážem z tejto voskovky zafarbiť celú kôpku ?
- dokážem z tejto voskovky zafarbiť niekoľko celých papierov kôpky ? Toto sa dá jednoducho urobiť pomocou zvyšku po delení – modulo.
- inak určite vznikne viacfarebný papier

Pri každom prípade treba dopočítať, koľko z aktuálnej voskovky ostane, prípadne ak potrebujeme viac ako len aktuálnu voskovku, tak si vypočítať, koľko ostane z poslednej ktorú použijeme.

### Časová zložitosť :

Riešenie bude mať zložitosť $O(m+n)$, pretože ako si môžeme všimnúť, s každou voskovkou a každou kôpkou pracujeme iba raz. Teda ak si napríklad označíme index voskovky s ktorou práve pracujeme ako $i$, tak toto $i$ vždy rastie, nikdy neklesá (k vypísaným voskovkám sa nevraciame). Rovnako ak si označíme index kôpky ktorú práve zafarbujeme ako $j$, tak aj toto $j$ vždy rastie, nikdy neklesá (k zafarbeným kôpkam sa nevraciame). Keďže pri zafarbovaní kôpky robíme len konštantné operácie - vetvenie a delenie, tak nič iné nám zložitosť neovplyvňuje.

### Pamäťová zložitosť :

Aj pamäťová zložitosť bude $O(m + n)$, lebo si musíme pamätať $n$ dĺžok voskoviek, $m$ veľkostí kôpok a $m$ hodnôt opisujúcich dĺžku voskovky na zafarbenie 1 papiera danej kôpky. Teda pamäťová zložitosť bude $O(2m + n)$, teda $O(m + n)$.

### Listing programu (C++)

```
#include<iostream>

using namespace std;

int main(){
    long long n,m;
    cin>>n;
    long long d[n];
    for(int i=0;i<n;i++)
        cin>>d[i];
    cin>>m;
    long long c[m], k[m];
    for(int i=0;i<m;i++)
        cin>>k[i]>>c[i];

    long long index_kopky=0, index_obrazok=0, nafarbene=0, roznofarebne=0;

    for(long long i=0;i<n;i++){ //iterujeme cez pastelky
        while(index_kopky<m && d[i]>0){
                        // vieme nafarbit celu kopku resp. zvysok kopky
            if(d[i]>=(c[index_kopky]*(k[index_kopky]-index_obrazok)-nafarbene)){
                d[i]-=c[index_kopky]*(k[index_kopky]-index_obrazok)-nafarbene;
                index_kopky++;
                index_obrazok=0;
```

```
                nafarbene=0;
            }else{
                if(d[i]<c[index_kopky]-nafarbene){ // neviem zafarbit cely
                    ↪ obrazok
                    if(nafarbene==0){
                        roznofarebne++;
                    }
                    nafarbene+=d[i];
                    d[i]=0;
                }else{ // viem zafarbit (aspon jeden) cely obrazok
                    index_obrazok++;
                    d[i]-=(c[index_kopky]-nafarbene);

                    index_obrazok+=d[i]/c[index_kopky];
                    nafarbene=d[i]%c[index_kopky];
                    d[i]-=(d[i]/c[index_kopky])*c[index_kopky];
                    d[i]-=nafarbene;
                    if(nafarbene){
                        roznofarebne++;
                    }
                }
            }
        }
    }
    cout<<roznofarebne<<endl;
}
```

**Listing programu (Python)**

```
n = int(input())
dlzky = []
for i in range(n):
    dlzky.append(int(input()))

m = int(input())
skupinky = []
for i in range(m):
    skupinky.append(list(map(int, input().split())))

index_voskovky = 0
ostava = dlzky[index_voskovky]
roznofarebne = 0
i = 0
pocet_na_kopke = 0
treba_nafarbit = 0

while i < len(skupinky) or (i == len(skupinky) and pocet_na_kopke > 0):

    if pocet_na_kopke == 0:
        pocet_na_kopke, treba_nafarbit = skupinky[i][0], skupinky[i][1]
        i += 1

    if ostava == 0:
        index_voskovky += 1
        ostava += dlzky[index_voskovky]

    if ostava > pocet_na_kopke*treba_nafarbit: # zafarbim celu kopku
        ostava -= pocet_na_kopke*treba_nafarbit
```

```
        pocet_na_kopke = 0

    elif ostava == pocet_na_kopke*treba_nafarbit: # zafarbim celu kopku a miniem
        ↪ voskovku
        ostava -= pocet_na_kopke*treba_nafarbit
        pocet_na_kopke = 0

    else: #zafarbim len cast kopky

        if ostava % treba_nafarbit == 0: # zafarbim cele obrazky
            pocet_na_kopke -= ostava//treba_nafarbit
            ostava = 0

        elif ostava > treba_nafarbit: # zafarbim nejake cele
            pocet_na_kopke -= ostava//treba_nafarbit
            ostava -= (ostava//treba_nafarbit) * treba_nafarbit

            # naberiem farby pre viacfarebny
            while ostava < treba_nafarbit:
                index_voskovky += 1
                ostava += dlzky[index_voskovky]

            # zafarbim viacfarebny
            roznofarebne += 1
            ostava -= treba_nafarbit
            pocet_na_kopke -= 1

        else: # zafarbim viacfarebne jeden

            # naberiem farby pre viacfarebny
            while ostava < treba_nafarbit:
                index_voskovky += 1
                ostava += dlzky[index_voskovky]

            # zafarbim viacfarebny
            roznofarebne += 1
            ostava -= treba_nafarbit
            pocet_na_kopke -= 1

print(roznofarebne)
```

Michal Staník

## 4. Kruhová knižnica

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

### Pomalé riešenie

Predstavme si, že by sme vedeli, do ktorého stĺpca (na ktorú pozíciu) chceme presunúť všetky červené knihy. Označme túto pozíciu $c$. Vieme nejako vypočítať vzdialenosť, o ktorú treba posunúť knihu, ktorá sa práve nachádza na pozícii $p$?

Mohli by sme ju presunúť bez využitia cyklickosti políc. Takto sa kniha posunie o vzdialenosť $|p - c|$, kde $|x|$ značí absolútnu hodnotu čísla $x$ (teda číslo po odobratí znamienka). Ak by sme ju chceli posunúť opačným smerom, teda cez hranicu medzi pozíciami 0 a $s - 1$, táto vzdialenosť by bola doplnkom vyššie uvedenej vzdialenosti do celého kruhu (čiže do $s$). V tomto prípade teda dostávame vzdialenosť $s - |p - c|$. Z týchto dvoch vzdialeností vieme vybrať tú menšiu a toto urobiť a posčítať pre všetky police.

Keďže cieľovú pozíciu nepoznáme, musíme vyskúšať všetkých $s$ možných. Pre každú z nich prejdeme všetky knihy a vyberieme najlepšiu pozíciu. Máme tak riešenie s časovou zložitosťou $O(ns)$, ktoré zvládne vyriešiť prvé 2 sady.

### Listing programu (C++)

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

typedef long long int ll;
using namespace std;

ll bruteForce(int n, int columns, vector<int> positions){
    ll best = 1LL*n*n;
    for (int c=0; c<columns; c++){
        ll current = 0;
        for (int pos: positions)
            current += min(abs(c-pos), columns - abs(c-pos));
        best = min(best, current);
    }
    return best;
}

int main() {
    int n, columns;
    cin >> n >> columns;
    vector<int> positions(n);
    ll best = n*n;
    for (int i=0; i<n; i++) {
        cin >> positions[i];
    }
    cout << bruteForce(n, columns, positions) << '\n';
    return 0;
}
```

### Efektívnejšie riešenie jednoduchšej úlohy

Pozrime sa teraz na jednoduchšiu úlohu, v ktorej police nie sú cyklické, ale majú ľavý a pravý koniec. V takomto prípade je odpoveď jednoduchá – je ňou medián čísel na vstupe. Medián postupnosti je jej stredná hodnota, teda číslo, ktoré po usporiadaní postupnosti skončí v strede. Ak je čísel nepárne veľa, je táto hodnota jednoznačne určená. Ak je čísel párne veľa, medián je definovaný ako priemer dvoch prostredných, ale ako riešenie úlohy je možné vziať ľubovoľnú hodnotu medzi dvomi prostrednými (alebo rovno jednu z nich) a bude to rovnako dobré optimálne riešenie.

Prečo je riešením vždy medián? Začnime s tým, že cieľovú pozíciu, kam chceme knihy premiestniť, dáme úplne naľavo, takže každá kniha bude musieť byť presunutá o určitú vzdialenosť doľava. Keď teraz začneme našu pozíciu posúvať doprava, knihám, ktoré sú od nej napravo, sa potrebná vzdialenosť bude znižovať, zatiaľ čo knihám, ktoré sú od nej naľavo, sa bude znižovať.

Ak je napravo od cieľovej pozície $j$ kníh a naľavo $i$, posunom o 1 doprava sa celkový súčet vzdialeností zmenší o $j$ a zvýši o $i$, pričom nejaké knihy sa môžu začať nachádzať na opačnej strane od tejto pozície.

Na začiatku sú všetky knihy napravo a približujú sa, na konci sú všetky naľavo a vzďaľujú sa. Ideálnym riešením je stav, kedy je naľavo aj napravo rovnako veľa kníh (vtedy zvyšovanie začne predbiehať znižovanie), čo je práve vtedy, keď sme v mediáne. (Pre prostredný prvok platí, že naľavo aj napravo od neho je rovnako veľa prvkov.)

### Vzorové riešenie

Túto myšlienku vieme zovšeobecniť na riešenie našej úlohy. Začnime s cieľovou pozíciou 0 a poďme ju posúvať (po kruhu) doprava. Počítajme si pritom, koľko pozícií sa približuje a koľko sa vzďaľuje. Keď poznáme aktuálny celkový súčet vzdialeností a pohneme cieľovou pozíciou, na základe týchto hodnôt vieme celkový súčet aktualizovať.

### Ktoré knihy sa k nej približujú a ktoré sa vzďaľujú?

Ak je cieľová pozícia 0, približujú sa tie, ktoré sú na pozícii najviac $s/2$ (najkratšia cesta pre ne vedie priamo k menším pozíciám) a vzďaľujú sa tie, ktoré sú na pozícii väčšej ako $s/2$ (pre ne je optimálne ísť k 0 doprava cez vyššie pozície).

### Kedy sa mení, či sa kniha približuje alebo vzďaľuje

Keď s cieľovou pozíciou prídeme na pozíciu nejakej knihy, táto sa doteraz približovala a odteraz sa začne vzďaľovať. Pri našich kruhových pozíciách môže nastať ešte jeden typ situácie, a to ten, že sa kniha prestane vzďaľovať a začne približovať. Deje sa to vtedy, keď sme na kruhu presne oproti nej. Cieľová pozícia sa od nej vzdialila natoľko, že už je pre knihu lepšie k nej ísť opačným smerom a v tomto smere sa približuje.

Keď sa však pustíme do implementácie, uvedomíme si, že to nie je také jednoduché. Pre párne $s$ sa kniha v istom momente prestane vzďaľovať a hneď sa začne približovať, napríklad pre $s = 100$ a knihu na pozícii 10 sa toto stane, keď cieľová pozícia je 60.

Avšak pre nepárne $s$ sa to stane na neceločíselnej pozícii, napríklad pre $s = 101$ a knihu na pozícii 10 sa to stane, keď cieľová pozícia je 60.5. My však pracujeme iba s celočíselnými pozíciami. Pre knihu ostáva vzdialenosť rovnaká pre cieľovú pozíciu 60 aj 61.

Elegantný spôsob riešenia tohto problému je taký, že si nebudeme pamätať len počet kníh, ktoré sa približujú a vzďaľujú, ale aj počet takých, ktoré svoju vzdialenosť nemenia. V tomto stave vie byť každá kniha len po dobu presunu cieľovej pozície o 1 (a po dobu 0 pri párnom $s$ – nechceme mať zvlášť implementáciu pre párne a pre nepárne $s$).

### Dajme to všetko dokopy

Vzorové riešenie teda bude vyzerať nasledovne: najskôr si vypočítame súčet vzdialeností, keby sme všetky knihy chceli presunúť na pozíciu 0. Tiež si napočítame počet približujúcich sa a vzďaľujúcich sa kníh (a tých, čo svoju vzdialenosť pri posune z 0 na 1 nezmenia, to sú tie na pozícii $\frac{s+1}{2}$, ak to je celé číslo).

Cieľovú pozíciu posúvame doprava a udržiavame si aktuálny súčet vzdialeností a najlepší doteraz nájdený. Aby sme nemuseli prechádzať $10^9$ pozícii v 4. sade, stačí si uvedomiť, že zaujímavé pozície sú len tie, kde sa niečo mení. Pre každú knihu je to jej pozícia a jedna alebo dve pozície presne naproti nej (podľa parity $s$). Tieto udalosti si vieme predpočítať dopredu, usporiadať podľa súradnice a postupne ich všetky spracovať.

Časová zložitosť riešenia je $O(n \log n)$, pretože výpočet hodnôt pred začatím posúvania cieľovej pozície spravíme v $O(n)$, potom utriedime $3n$ udalostí v $O(n \log n)$ a následne každú udalosť spracujeme v konštantnom čase. Každá udalosť je totiž iba zmena stavu jednej knihy (približuje sa/vzďaľuje sa/nemení sa). Pamäťová zložitosť je $O(n)$.

### Listing programu (C++)

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

typedef long long int ll;
using namespace std;

#define ONE 1
#define OPPOSITE1 2
#define OPPOSITE2 3

ll solve(int n, int columns, vector<int> positions){
    ll best = 1LL*n*columns, goingLeft = 0, goingRight = 0, stagnant = 0;
    vector<pair<int, int>> events;
    for (int i=0; i<n; i++) {
        if (columns % 2 == 1 && positions[i] == columns / 2)
            stagnant++;
        else if (positions[i] <= (columns-1) / 2)
            goingLeft++;
        else
            goingRight++;
        events.push_back({positions[i], ONE});
```

```
        events.push_back({(positions[i] + columns/2) % columns, OPPOSITE1});
        events.push_back({(positions[i] + (columns+1)/2) % columns, OPPOSITE2});
    }

    int lastpos = 0;
    ll current = 0;
    for (int pos: positions)
        current += min(pos, columns - pos);
    sort(events.begin(), events.end());

    for (pair<int, int> ev: events){
        int curpos = ev.first;
        current = current + goingRight * (curpos - lastpos) - goingLeft * (
            ↪ curpos - lastpos);
        best = min(best, current);
        if (ev.second == ONE){
            goingLeft--;
            goingRight++;
        }
        else if (ev.second == OPPOSITE1){
            goingRight--;
            stagnant++;
        }
        else if (ev.second == OPPOSITE2){
            stagnant--;
            goingLeft++;
        }
        lastpos = curpos;
    }
    return best;
}

int main() {
    int n, columns;
    cin >> n >> columns;
    vector<int> positions(n);
    for (int i=0; i<n; i++) {
        cin >> positions[i];
    }
    cout << solve(n, columns, positions) << '\n';
    return 0;
}
```

mišof

## 5. Okno do študentovej duše

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

Pripomeňme si, akú úlohu riešime. Máme predpísané, aké skóre majú jednociferné čísla. Každé väčšie číslo má skóre o jedno väčšie ako je skóre jeho ciferného súčinu. Našou úlohou je pre dané $c$ nájsť najmenšie číslo $n$, ktorého skóre je $c$.

### Pointa celej úlohy

Môže byť správnou odpoveďou číslo $n = 3\,141\,592$?

Nemôže. Prečo? Lebo číslo $1\,123\,459$ má presne tie isté cifry (a teda ten istý ciferný súčin, a teda aj to isté skóre) a je od $n$ menšie.

A môže byť číslo $1\,123\,459$ správnou odpoveďou?

Tiež nie. Prečo? Lebo číslo $23\,459$ má tiež ten istý ciferný súčin a je ešte menšie.

Ale poďme sa na to celé pozrieť od začiatku a pekne pomaly.

### Každé číslo má skóre

Keď sa chvíľu pohráme s ciernými súčinmi, pomerne rýchlo si všimneme, že pre ľubovoľné $n \geq 10$ nám jeho ciferný súčin $s(n)$ vyjde menší ako $n$. Je toto naozaj vždy pravda?

Keď už si to všimneme, dokázať to nie je ťažké. Pointa je v tom, že dopísanie ďalšej cifry na koniec čísla zväčší samotné číslo aspoň desaťkrát, zatiaľ čo jeho ciferný súčin len nanajvýš deväťkrát.

To isté ešte raz a poriadnejšie: Majme nejaké $x$-ciferné číslo $n$ ktorého prvá cifra je $y$. Čo o ňom vieme povedať?

- Hodnota $n$ je aspoň $y \times 10^{x-1}$.
- Každá z $x-1$ cifier nasledujúcich za úvodným $y$ je nanajvýš 9, takže hodnota $s(n)$ je nanajvýš $y \times 9^{x-1}$.
- A teda pre $x \geq 2$ máme $n > s(n)$.

No a z toho už by malo byť zjavné, že úplne každé nezáporné celé číslo má korektne definované skóre. (Indukciou podľa $n$. Ak všetky čísla menšie ako $n$ majú korektne definované skóre, $s(n)$ je jedným z nich, a keďže $s(n)$ má skóre, aj $n$ má skóre.)

### Malé skóre vieme efektívne počítať

Pripomeňme si, že nás zaujímajú len čísla, ktorých skóre je nanajvýš 11. Skóre ľubovoľného čísla $n$ preto vieme ľahko a efektívne vyhodnotiť: postupne počítame $s(n)$, $s(s(n))$, atď., až kým sa buď nedostaneme k jednocifernému číslu (kedy vieme povedať, aké skóre má $n$) alebo nespravíme 12 krokov (kedy vieme povedať, že $n$, z ktorého sme začínali, bude mať určite priveľké skóre).

Neskôr si ukážeme, že ani to počítanie krokov do 12 nie je potrebné robiť, lebo úplne každé číslo do $10^{18}$ má vždy maličké skóre. Ak ste teda v programe len priamo implementovali počítanie skóre podľa definície, bolo to v praxi rovnako dobré.

### Ako vyzerá hľadané riešenie?

Zamyslime sa teraz, čo vieme povedať o správnej odpovedi.

Začneme tým, že skontrolujeme čísla od 0 po 19. Čísla od 0 po 9 majú skóre dané na vstupe a hociktoré z nich môže byť tým, ktoré hľadáme. Čísla od 10 po 19 sú najmenšie viacciferné čísla, ktorých ciferný súčin je od 0 po 9. Aj každé z nich môže byť správnou odpoveďou. Ak sme medzi nimi správnu odpoveď nenašli, vieme, že má hodnotu aspoň 20. Uvažujme teraz o situáciách, kedy je správna odpoveď aspoň 20:

- Správna odpoveď nemôže obsahovať cifru 0. Ak by ju obsahovala, mala by ciferný súčin 0, a teda rovnaké skóre ako číslo 10, ktoré je od nej menšie.
- Ako už vieme, správna odpoveď musí mať cifry usporiadané od najmenšej po najväčšiu. (Keďže už vieme, že nemáme cifru 0, nemôže nám pri usporadúvaní cifier vzniknúť problém s nulou na začiatku čísla.)
- Správna odpoveď následne nemôže obsahovať ani cifru 1. Takéto číslo musí byť aspoň trojciferné (dvojciferné začínajúce 1 sme už pozerali) a keď mu úvodnú cifru 1 zmažeme, tak sa samotné číslo zmenší a jeho ciferný súčin sa nezmení.

### Už sme vyhrali?

Áno. Totiž po úvahe, ktorú sme práve spravili, nám ostane tak malé množstvo kandidátov na správnu odpoveď, že si môžeme dovoliť všetkých vygenerovať a skontrolovať. Dokopy je medzi nanajvýš 18-cifernými číslami len približne 1.5 milióna takých, ktorých cifry sú 2-9 a sú usporiadané od najmenšej po najväčšiu.

### Implementácia

Samotná implementácia je už pomerne priamočiara. Všetkých $k$-ciferných kandidátov na riešenie vieme vygenerovať tak, že vygenerujeme všetkých $(k-1)$-ciferných a každému na koniec pridáme všetky možnosti pre ďalšiu cifru. Takýmto postupom vieme dokonca všetkých kandidátov generovať v usporiadanom poradí, takže akonáhle narazíme na nejakého so správnym skóre, môžeme si byť istí, že sme práve našli riešenie.

### Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;

const long long INF = 1LL << 62;
```

```
vector<int> base;
vector<long long> answers;
int answers_found;

long long product(long n) {
    if (n == 0) return 0;
    long answer = 1;
    while (n > 0) { answer *= n%10; n /= 10; }
    return answer;
}

int f(long long n) {
    if (n < 10) return base[n];
    return f(product(n))+1;
}

void record_answer(long long number, int score) {
    if (score > int(answers.size())) return;
    if (answers[score] == INF) {
        answers[score] = number;
        ++answers_found;
    } else {
        assert( number > answers[score] );
    }
}

void go(long long prefix, int add_digits) {
    if (add_digits == 0) {
        record_answer( prefix, f(prefix) );
    } else {
        int last2 = prefix % 100;
        if (last2 == 22 || last2 == 23 || last2 == 33) return;
        int last = prefix % 10;
        for (int d=last; d<=9; ++d) {
            go(10*prefix+d, add_digits-1);
            if (answers_found == int(answers.size())) return;
        }
    }
}

int main() {
    base.resize(10);
    for (int i=0; i<10; ++i) cin >> base[i];
    int goal;
    cin >> goal;

    answers.clear();
    answers.resize(12, INF);
    answers_found = 0;

    for (int n=0; n<100; ++n) record_answer(n, f(n));
    for (int l=3; l<=18; ++l) for (int d=2; d<=9; ++d) go(d,l-1);
    cout << answers[goal] << endl;
}
```

### Efektívnejšie hľadanie

Množinu kandidátov, ktorých treba prezrieť, vieme ešte ďalej celkom výrazne prečistiť. Napríklad takto:

- Číslo $n$ nebude obsahovať cifry 22 (lepšie je mať jednu štvorku), 23 (šestku), 24 (osmičku) ani 33 (deviatku).
- Číslo $n$ nebude obsahovať cifry 34, lebo lepšie je mať cifry 26 – číslo bude menšie a ciferný súčin rovnaký. Podobne vieme, že $n$ nebude obsahovať cifry 36 (lepšie je 29), 44 (lepšie je 28), 46 (lepšie je 38) ani 66 (lepšie je 49).
- Ak máme v $n$ cifru 5 tak nemôžeme mať žiadnu párnu cifru – lebo $s(n)$ bude končiť nulou a potom $s(s(n))$ bude nula. Medzi zakázané dvojice cifier teda môžeme pridať aj 25, 45, 56 a 58.

Ak do generovania kandidátov pridáme tieto obmedzenia, zredukujeme ich počet z vyššie spomínaného približne 1.5 milióna na iba približne 9 000.

### Listing programu (Python)

```
def je_validne(cislo):
    cifry = [0]*10
    while cislo:
        cifry[cislo%10] += 1
        cislo //= 10

    if any( cifry[x] >= 2 for x in [2,3,4,6] ):
        return False

    forbidden_pairs = [ (2,3), (2,4), (2,5), (3,4), (3,6), (4,5), (4,6), (5,6),
    ↪ (5,8) ]
    if any( cifry[x] >= 1 and cifry[y] >= 1 for x, y in forbidden_pairs ):
        return False

    return True

def pridaj_dalsiu_cifru(zoznam):
    novy_zoznam = []
    for cislo in zoznam:
        posledna = cislo % 10
        for nova in range(posledna, 10):
            if je_validne( cislo*10 + nova ):
                novy_zoznam.append( cislo*10 + nova )
    return novy_zoznam

def generuj_vsetkych_kandidatov():
    for n in range(100):
        yield n
    kandidati = list(range(2,10))
    kandidati = pridaj_dalsiu_cifru(kandidati)
    for d in range(3,19):
        kandidati = pridaj_dalsiu_cifru(kandidati)
        for k in kandidati:
            yield k

def ciferny_sucin(n):
    odpoved = 1
    while n: odpoved, n = odpoved * (n%10), n // 10
    return odpoved

def skore(n, Z):
```

```
    odpoved = 0
    while n >= 10:
        odpoved += 1
        n = ciferny_sucin(n)
    return odpoved + Z[n]

Z = [ int(_) for _ in input().split() ]
c = int( input() )
for kandidat in generuj_vsetkych_kandidatov():
    if skore(kandidat, Z) == c:
        print(kandidat)
        break
```

### Checkpoint

Niekde tu by sa dalo vzorové riešenie tejto úlohy ukončiť. My v ňom však ešte nejakú tú chvíľu budeme pokračovať a doplníme ešte nejaké detaily a zaujímavostí navyše.

### Počítanie kandidátov

Vyššie sme uvádzali nejaké číselné odhady počtov kandidátov. Tie samozrejme vieme získať tak, že ich naozaj vygenerujeme a spočítame. Vieme ich ale veľmi ľahko spočítať aj pomocou kombinatoriky.

Pozrime sa napríklad na všetky $d$-ciferné čísla, ktorých cifry sú od 2 po 9 a sú usporiadané od najmenšej po najväčšiu. Koľko ich je?

Predstavme si, že máme premennú, v ktorej je na začiatku cifra 2, a dve rôzne inštrukcie: inštrukcia `+` zväčší hodnotu v premennej a inštrukcia `p` ju vypíše.

Pomocou takýchto inštrukcií vieme vyrábať čísla, ktorých cifry sú usporiadané. Napr. postupnosť inštrukcií `++p+pp+++p+` vyrobí číslo 4558.

Pozrime sa na postupnosti inštrukcií, ktoré obsahujú práve sedem inštrukcií `+` a práve $d$ inštrukcií `p`. Každá takáto postupnosť vyrobí nejaké $d$-ciferné číslo, ktorého cifry sú od 2 do 9 a postupne rastú. A naopak, keď zoberieme ľubovoľné takéto číslo, vieme nájsť práve jednu takúto postupnosť inštrukcií, ktorá ho vyrobí: ideme zľava doprava a pri každej cifre najskôr stlačíme správne veľa `+` aby sme premennú nastavili na jej hodnotu a potom stlačíme `p`. (Po poslednej cifre ešte doplníme prípadné chýbajúce `+`.)

Hľadaných čísel je teda presne toľko isto ako reťazcov dĺžky $d + 7$, ktoré obsahujú $d$ péčok a 7 plusov. No a takýchto reťazcov je zjavne $\binom{d+7}{7}$: vyberieme, na ktorých siedmich pozíciách sú plusy a tým je jednoznačne určené, kde sú péčka.

### Asymptotické odhady

Môžeme rozpísať, že $\binom{d+7}{7} = \frac{(d+7)(d+6)\cdots(d+1)}{7!}$. Z toho je zjavné, že počet čísel, ktoré majú usporiadané cifry od 2 po 9, závisí od ich dĺžky $d$ len polynomiálne: je ich $\Theta(d^7)$.

Keď sme prečistili množinu kandidátov, ostali nám v nej len čísla, ktoré mali nanajvýš po jednej z cifier 2, 3, 4, 6. Navyše buď mali len nepárne cifry (vtedy sú premenlivé len počty cifier 5, 7 a 9) alebo neobsahovali cifru 5 (a vtedy sú premenlivé zase len počty cifier 7, 8 a 9). Podobnou kombinatorickou úvahou dostávame, že čísel dĺžky $d$ jedného aj druhého typu je len $\Theta(d^2)$.

### Otvorený problém na záver

Najjednoduchšou možnou formou zadania našej úlohy je verzia v ktorej všetky jednociferné čísla majú skóre nula. Pre takýto vstup platí, že skóre čísla udáva jednoducho počet opakovaní operácie "nahraď číslo jeho ciferným súčinom" po ktorom dostaneme jednociferné číslo. Matematici v teórii čísel toto občas nazývajú nie skóre, ale *multiplikatívna perzistentnosť* čísla.

Ak ste si skúšali svoj program spustiť pre najväčšie platné vstupy, určite ste v jeho výstupoch často narazili na číslo 277 777 788 888 899. Toto je najmenšie číslo, ktoré má multiplikatívnu perzistentnosť 11.

Prečo sme ako najväčšie povolené $c$ v zadaní zvolili práve 11? O koľko väčšia je vlastne správna odpoveď pre $c = 12$? Nevieme podobným postupom nájsť aj tú?

Nuž… nielen že nevieme, my vlastne dodnes ani nevieme, či vôbec nejaké takéto číslo existuje. Jeho existencia je otvoreným problémom, ktorý už nejaký ten rok odoláva snahám matematikov. Zatiaľ vieme dokázať, že *ak* takéto číslo existuje, *tak* určite musí mať výrazne viac ako 20 000 cifier. Ale skôr si myslíme, že neexistuje.

A dokonca *pravdepodobne* platí ešte podivuhodnejšie tvrdenie: je možné, že už poznáme *úplne všetkých* kandidátov, ktorí majú multiplikatívnu perzistentnosť väčšiu ako 2. Vyzerá to totiž tak, že keď budeme postupne prezerať väčších a väčších kandidátov, tak nielen že nenájdeme žiadneho s multiplikatívnou perzistentnosťou 12, ale naopak sa veľmi rýchlo minú všetci s multiplikatívnou perzistentnosťou väčšou ako 2.

Prečo je to tak? Intuitívne sa to dá priblížiť nasledovne. Majme nejakého kandidáta s veľmi veľkým počtom cifier. Toto je nejaké pekné systematické číslo, ktorého cifry sú od 2 od 9. Jeho ciferným súčinom je nejaké iné, tiež ešte obrovské číslo. Toho cifry sú však už všelijaké chaotické a skoro vždy niekde medzi nimi vybehne nejaká tá nula. Takže to vyzerá tak, že pre ľubovoľného veľkého kandidáta $n$ skoro určite platí $s(s(n)) = 0$.

Posledná známa výnimka má 140 cifier. Presnejšie, 140-ciferné číslo $v = 2^{25} \times 3^{227} \times 7^{28}$ neobsahuje žiadnu nulu a jeho multiplikatívna perzistentnosť je presne 2. Toto číslo $v$ je rovné $s(n)$ pre niekoľko podobne veľkých kandidátov $n$ ktorí potom majú multiplikatívnu perzistentnosť až 3. Ale od 140 cifier ďalej až po vyše 20 000 cifier už sme prezreli všetky možnosti a sme si úplne istí, že úplne všetky možné $s(n)$ obsahujú v sebe aspoň jednu nulu.

Zrejme to bude platiť aj ďalej pre ešte väčšie čísla. Hja, ale ako takéto niečo exaktne dokázať?

Paulinka

## 6. Výtvarný Ateliér

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

Ako si mohol skúsenejší riešiteľ isto všimnúť, úloha sa rieši *dynamickým programovaním*.

Zjednodušene, spočítame si hodnotu výsledku pre menšie časti zadania a skombinujeme aby sme dostali výsledok.

### Pomalé riešenie

Predstavme si, že by sme pre pozície $l$, $j$ vedeli nasledovné:

Koľko najviac spokojnosti vieme dostať, ak by sme mali len prvých $l$ ôk zo Samovej reťaze, a len prvých $j$ ôk z Adamovej reťaze a:

- Samo vymazal nejaký neprázdny koncový úsek svojich ôk. Označme toto číslo ako $D[l][j]$
- Samo na koniec svojej reťaze pridal nejaký neprázdny úsek ôk. Označme toto číslo ako $I[l][j]$
- Samo buď posledné oko nezmenil (ak sú koncové[1] oká rovnaké), alebo prefarbil (ak sú rôzne). Označme toto číslo ako $M[l][j]$

Navyše, označme si najväčšiu spokojnosť akú vie Samo dostať pre takéto začiatky reťazí ako $S[l][j]$.

Všimnime si, že $S[l][j] = \max(D[l][j], I[l][j], M[l][j])$

Predstavme si, že máme spočítanú hodnotu $S$ pre všetky skoršie začiatky (vrátane $S[l][j-1]$ a $S[l-1][j]$). Ako z toho získať hodnotu $S[l][j]$?

Keďže $S[l][j]$ je minimum z $D[l][j], I[l][j]$ a $M[l][j]$, chceme spočítať tie.

Všimnime si, že

$$M[l][j] = \begin{cases} S[l-1][j-1] + m & \text{ak sú } i\text{-te Samovo oko a } j\text{-te Adamovo oko rovnakej farby} \\ S[l-1][j-1] - r & \text{ak sú rôznej farby} \end{cases}$$

Pre $I$ a $D$ vieme vyskúšať všetky možné dĺžky pridaného, resp. odstráneného úseku, teda $I[l][j]$ je maximum z $S[l][j-k] - i$, pre všetky $1 \leq k \leq j$. A nápodobne pre $D[i][j]$.

Takto vieme postupne prejsť všetky začiatky a napokon nájsť aj riešenie pre celé reťazce v čase $O(|S||A|(|A|+|S|))$, kde $|S|$, $|A|$ sú postupne dĺžky Samovho a Adamovho reťazca, a v pamäti $O(|S||A|)$.

### Listing programu (C++)

```
#include<bits/stdc++.h>
#define ii pair<int, int>
#define FOR(i,n)        for(int i=0;i<(int)n;i++)

using namespace std;

const int inff = 1000000999;
```

[1] koncové v tomto prípade znamená $l$-te Samovo a $j$-te Adamovo oko

```
int compute_score(int ma, int d, int ins, int r, string S, string A) {
    int n = S.size();
    int m = A.size();

    vector<vector<int> > V_gap(n + 1, vector<int>(m + 1, -inff)),
                         F(n + 1, vector<int>(m + 1, -inff)),
                         E(n + 1, vector<int>(m + 1, -inff));

    V_gap[0][0] = 0;

    FOR(i, n + 1) {
        F[i][0] = -d;
        V_gap[i][0] = max(V_gap[i][0], -d);
    }

    FOR(j, m + 1) {
        E[0][j] = -ins;
        V_gap[0][j] = max(V_gap[0][j], -ins);
    }

    for (int i = 1; i <= n; i ++) {
        for (int j = 1; j <= m; j ++) {
            FOR(k, j) {
                E[i][j] = max(V_gap[i][k] - ins, E[i][j]);
            }
            FOR(k, i) F[i][j] = max(F[i][j], V_gap[k][j] - d);
            V_gap[i][j] = max(max(F[i][j], E[i][j]),
                              V_gap[i - 1][j - 1] + (S[i - 1] == A[j - 1] ? ma :
                                  ↪ -r));
        }
    }
    return V_gap[n][m];
}

int main() {
    int m, d, ins, r;
    cin >> m >> d >> ins >> r;
    string samo, adam;
    cin >> samo >> adam;
    cout << compute_score(m, d, ins, r, samo, adam) << endl;
}
```

### Ako rýchlejšie?

Chceli by sme zrýchliť naše riešenie. Čo robíme navyše?

Predstavme si, že okrem hodnôt $S$ si pamätáme aj hodnoty $I$, $D$ pre predchádzajúce začiatky. Potom si všimnime, že akonáhle je Samovi lepšie vložiť väčší úsek ako 1, potom $I[l][j] = I[l][j-1]$. Inak je ideálne vložiť jediné oko, teda $I[l][j] = \max(I[l][j-1], S[l][j-1] - i)$. Nápodobný vzorec získame pre hodnoty $D$.

Takto nám stačí pre výpočet $S[l][j]$ spraviť konštantne veľa operácií, takže dostávame časovú zložitosť $O(|S||A|)$ s rovnakou pamäťovou zložitosťou ako predtým.

### Ako to ešte zlepšiť?

Čas už nezlepšíme, ale máme tu ešte pamäť. Všimnime si, že pri počítaní $S[l][j]$ sa pozeráme len na hodnoty s indexami najmenej $[l-1][j-1]$. Teda nám stačí si pamätať posledný riadok a stĺpec výpočtu, teda iba lineárne veľa údajov.

Takže dostávame pamäťovú zložitosť $O(|S| + |A|)$.

**Listing programu (C++)**

```
#include<bits/stdc++.h>

using namespace std;

const int inff = 1000000999;

int compute_score(int me, int d, int ins, int r,
                          string U, string V) {
    int n = U.size();
    int m = V.size();

    vector<int> V_gap(m + 1, 0), F(m + 1, -inff);

    for (int j = 1; j <= m; j ++) {
        V_gap[j] = -ins;
    }

    for (int i = 1; i <= n; i ++) {
        vector<int> new_V_gap(m + 1), new_F(m + 1);
        new_V_gap[0] = -d;
        new_F[0] = -d;
        int E = -inff;

        for (int j = 1; j <= m; j ++) {
            int G = (V_gap[j - 1] +
                        (U[i - 1] == V[j - 1] ? me : -r));
            E = max(E, new_V_gap[j - 1] - ins);
            new_F[j] = max(F[j], V_gap[j] - d);
            new_V_gap[j] = max(G, max(E, new_F[j]));
        }

        F = new_F;
        V_gap = new_V_gap;
    }

    return V_gap[m];
}

int main() {
    int me, d, ins, r;
    cin >> me >> d >> ins >> r;
    string samo, adam;
    cin >> samo >> adam;
    cout << compute_score(me, d, ins, r, samo, adam) << endl;
}
```

**Máme to naozaj vyriešené?**

A naozaj, v riešení hore sme sa nezmienili čo robiť so $S[l][j]$, keď je jedno z indexov 0. Schválne, čo jediné môže Samo spraviť ak má on alebo Adam prázdny string?

Jano

## 7. Kreslenie stromov

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

To, čo sme v zadaní volali *rovnakosť* sa v informatike zvykne nazývať *izomorfizmus*. Ďalej teda budeme hovoriť, že dva zakorenené stromy sú ***izomorfné***, ak existuje bijektívne zobrazenie $f$ vrcholov jedného stromu

na vrcholy druhého stromu, také že pre všetky $x, y$ platí, že $x$ je otec $y$ práve vtedy, keď $f(x)$ je otec $f(y)$. (Toto je rovnaká definícia ako v zadaní, len na pripomenutie.)

Intuitívna predstava izomorfizmu je, že zodpovedajúce vrcholy sa môžu líšiť len v poradí synov. Toto je dôležité mať na pamäti.

Mimochodom, keď máme zakorenený strom za koreň $v$, tak stromy, ktorých korene sú synovia $v$, budeme volať podstromy tohto stromu, alebo podstromy koreňa, alebo podobne. Občas budeme zamieňať pojmy podstrom a vrchol, väčšinou by malo byť jasné, že ide o tú istú vec. Ak napríklad označíme vrchol $u$ nejakým číslom, tak toto číslo obvykle hovorí niečo o celom podstrome.

### Obsah

Riešenie si rozdelíme na niekoľko častí:

V prvej časti sa pozrieme na to, ako algoritmicky spočítať, či ľubovoľné dva stromy sú izomorfné. Ukážeme si tri spôsoby, jeden z nich ilustruje podstatu zvyšných dvoch spôsobov, avšak je pomalý. Zvyšné dva implementujú túto podstatu efektívne, či už pomocou hashovania alebo pomocou iných techník. Obe rýchlejšie riešenia budú vedieť porovnať dva stromy v čase $O(n)$, hoci v jednom prípade si podrobne rozoberieme iba jednoduchšiu $O(n \log n)$ variantu.

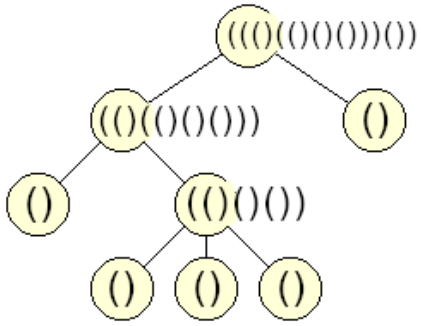
V druhej časti sa pozrieme, ako zovšeobecniť algoritmus z dvoch stromov na izomorfizmus $n$ stromov, a následne ako pomocou toho vyriešiť samotnú úlohu v celkovom čase $O(n)$ resp. $O(n \log n)$.

Na plný počet bodov za popis úlohy stačí časová zložitosť $O(n \log n)$ a zároveň žiaden riešiteľ neodovzdal popis deterministického riešenia so zložitosťou $O(n)$, takže nebudú ani bonusové body ;)

### Izomorfizmus dvoch stromov – naivné riešenie

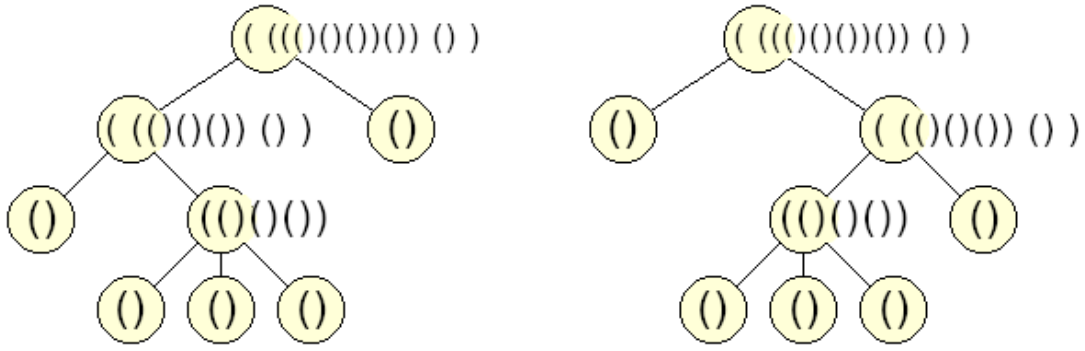
Stromy porovnávať zatiaľ nevieme, ale napríklad také reťazce znakov vieme porovnávať celkom dobre. Čo keby sme každý strom zapísali ako reťazec znakov? Celkom štandardný zápis zakoreneného stromu je: *ľavá zátvorka, zreťazené zápisy všetkých podstromov koreňa, pravá zátvorka.*

Napríklad jednovrcholový strom, má zápis "()", trojvrcholový binárny strom má zápis "(()())", reťaz vysoká 3 má zápis "((()))". Na obrázku môžete vidieť pri každom vrchole zápis zodpovedajúceho podstromu.




Žiaľbohu a niekedy aj chvalabohu, izomorfné stromy môžu mať rôzny zápis, lebo v tomto zápise záleží na poradí podstromov. Tento problém však ľahko vyriešime, nasledovnou zmenou. Nový zápis zakoreneného stromu bude: *ľavá zátvorka, zretazené* ***abecedne zoradené*** *zápisy všetkých podstromov koreňa, pravá zátvorka.*

Dôkaz, že s touto definíciou, dva stromy majú rovnaký zápis práve vtedy, keď sú izomorfné, necháme ako cvičenie pre čitateľa. Na obrázku môžeme vidieť príklad pre dva izomorfné stromy.




Skonštruovať zápis celého stromu nám môže v najhoršom prípade (dlhá reťaz) trvať až $O(n^2)$. Porovnať dva zápisy a tým overiť izomorfizmus stromov potom vieme už v lineárnom čase.

### Izomorfizmus dvoch stromov – hashovanie

Celý problém predošlého algoritmu je, že zápis stromu je potenciálne veľmi dlhý reťazec. Čo keby sme dokázali celý strom zapísať ako jedno číslo? Na základe predošlej úvahy by toto číslo malo byť rekurzívne definované cez podstromy a zároveň nezávislé od poradia podstromov.

Ako riešenie sa nám ponúka nasledovná hashovacia funkcia. Hash stromu $T$ bude

$$H_T = chaos(S \mod M) = chaos\left(\left(\sum_{t \in P(T)} H_t\right) \mod M\right)$$

kde $P(T)$ je zoznam podstromov stromu $T$, resp. $S$ je súčet hashov podstromov stromu $T$. $M$ je nejaké prvočíslo, ktorým modulujeme, aby nám udržala rozumná veľkosť čísel. *chaos* je funkcia, ktorá pre každý vstup z $\{0..M\}$ vráti iný náhodný výstup z $\{0..M\}$.

Čo prosím? (Sa právom pýtate). V prvom rade súčet hashov podstromov je jednoduchý spôsob, ako zabezpečiť, aby nezáležalo na poradí podstromov. Potom ale potrebujeme, aby sa hashovacia funkcia nesprávala lineárne (t.j. zväčšenie nejakého zo vstupov o $x$ zväčší výstup o $kx$), lebo to by sa nám ľahko stalo, že presun jedného vrchola z jedného podstromu do iného by nemuselo zmeniť hash (na jednom mieste $+kx$, na inom $-kx$, výsledok rovnaký), hoci by zmenilo izomorfizmus. A čo sa správa menej lineárne ako random?

Funkciu chaos môžeme implementovať lazy spôsobom – keď nám príde nový vstup, aký sme ešte nevideli, vygenerujeme si nový náhodný výstup a zapamätáme si ho. Ak nám príde vstup, aký už sme niekedy videli, vrátime zapamätanú hodnotu. Buď si môžeme dávať explicitne pozor, aby sme vždy vygenerovali iný výstup, alebo pokiaľ $M >> n^2$, je pomerne malá šanca, že by nastala kolízia.

S touto hashovacou funkciou, dva izomorfné stromy majú vždy rovnaký hash, a dva neizomorfné stromy majú s veľkou pravdepodobnosťou rôzny hash. Stále môže nastať neželaná kolízia, ale tak to pri hashovacích algoritmoch často býva a tak v princípe iba nastavíme dostatočne veľké $M$ a dúfame, že to bude fungovať.

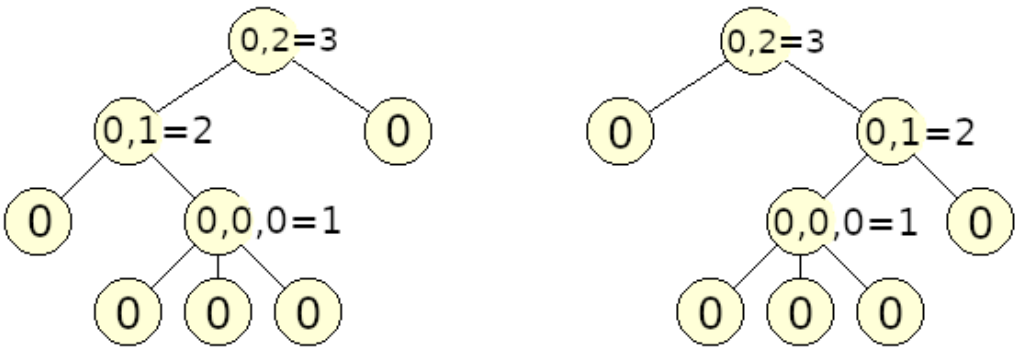
Spočítať hash celého podstromu dokážeme rekurzívne v čase $O(n)$

### Izomorfizmus dvoch stromov – labely a AHU

Ak sa nechceme spoliehať na náhodu, môžeme skúsiť nájsť iný spôsob, ako každému stromu priradiť jedno číslo, nazvime ho label (čítaj lejbl).

Pre spočítanie labelu stromu najprv rekurzívne spočítame labely všetkých podstromov, a zoberieme utriedený zoznam týchto labelov. Napríklad, pre list stromu (ktorý nemá žiadnych potomkov) dostaneme prázdny zoznam []. Ak má koreň stromu 5 synov s labelami 2,11,4,2,4, dostaneme zoznam [2,2,4,4,11].

Vždy, keď dostaneme takto utriedený zoznam, buď vidíme takýto zoznam prvýkrát a pridelíme mu najmenšie nepoužité prirodzené číslo ako label, alebo sme už zoznam videli a tak použijeme rovnaký label ako naposledy. Tu je príklad ako by vyzerali labely pre ukážkový strom a jeho podstromy. (Labely číslujeme od nuly).




Ako vieme efektívne priraďovať, labely zoznamom? Napríklad pomocou mapy (vyvažovaný binárny strom), kde zoznam bude kľúč a label bude hodnota. Na utriedenie zoznamu labelov podstromov a tiež vyhľadanie zoznamu v mape potrebujeme čas $O(p \log p)$, kde $p$ je dĺžka zoznamu, resp. počet potomkov vrchola. Keďže súčet počtu potomkov pre všetky vrcholy stromu je $n-1$, celkový čas je $O(n \log n)$.

Veľmi podobne funguje AHU algoritmus (Aho, Hopcroft and Ullman), čo je štandardný a asi najznámejší algoritmus na overenie izomorfizmu dvoch stromov. Tento algoritmus používa fakt, že dva stromy môžu byť izomorfné, iba ak majú rovnakú hĺbku (môžete si nechať chvíľku na zamyslenie, prečo je to tak). Potom izomorfizmus overujeme iba pre stromy s rovnakou hĺbkou, nech je to $h$. Najprv spočítame labely pre vrcholy v hĺbke $h$, potom pre vrcholy v hĺbke $h-1$, atď. až po koreň. Na jednej úrovni vždy najprv spočítame všetky zoznamy a potom im naraz priradíme labely postupne od nuly. V AHU algoritme, môžeme použiť v rôznych hĺbkach ten istý label.

Vďaka tomu, že labely počítame po vrstvách, môžeme naraz skonštruovať v čase $O(p)$, kde $p$ je počet vrcholov vo vrstve. Celý algoritmus potom beží v čase $O(n)$. Detaily sú nad rámec tohto vzoráku ale aspoň stručné zhrnutie je, že lexikograficky triedime samotné utriedené sekvencie labelov pomocou zovšeobecného radix-sortu pre rôzne dĺžky sekvencií a potom priradíme nové labely podľa tohto poradia. Jednotlivé sekvencie nemusíme triediť, lebo ich už konštruujeme v utriedenom poradí, a to tak, že spracúvame synov vrcholov na vyššej vrstve od najmenších labelov po najväčšie.

Ak vás to zaujíma a viete po anglicky, celý algoritmus na overenie izomorfizmu dvoch stromov je popísaný v knižke The Design and Analysis of Computer Algorithms z roku 1974[2], strana 84-85, example 3.2.

### Izomorfizmus $n$ stromov.

Ak máme $n$ zakorenených stromov a každý z nich má $n$ vrcholov, môžeme pomocou predošlých algoritmov spočítať zápis, hash, alebo label každého z nich, a potom overiť, koľko rôznych zápisov, hashov, resp. labelov vidíme. Toto vieme spraviť v čase $O(n^3)$ resp. $O(n^2)$, pretože zápis, hash, resp. label každého so stromov vieme spočítať v čase $O(n^2)$ resp. $O(n)$.

Riešenie, založené na tomto princípe, pokiaľ ako základ algoritmu použijeme hashovanie by mohlo vyzerať takto:

### Listing programu (Python)

```
import random
import sys
sys.setrecursionlimit(1000000)
random.seed(47)
MOD = 10 ** 9 + 7
n = int(input())
E = [[] for _ in range(n)]

for i in range(n - 1):
    a, b = [int(x) - 1 for x in input().split()]
    E[a].append(b)
    E[b].append(a)

_chaos = {}
def chaos(x):
    if x not in _chaos:
        _chaos[x] = random.randint(1, MOD - 1)
    return _chaos[x]

def dfs(v, parent):
    s = 0
    for e in E[v]:
        if e != parent:
            s += dfs(e, v)
    return chaos(s % MOD)

print(len(set(dfs(i, -1) for i in range(n))))
```

Posledné dve časti vzoráku sa budú zaoberať tým, ako to spraviť rýchlejšie.

### Vzorové riešenie – hashovaie

Máme teda nezakorenený strom a chceme vedieť, aký hash by mali všetky jeho zakorenené verzie. Označme si hash vrcholu $u$ v strome zakorenenom za vrchol $v$ ako $H_v(u)$. Označme si tiež súčet hashov potomkov vrchola $u$ v strome zakorenenom za vrchol $v$ ako $S_v(u)$. Ak si ešte pamätáme definíciu našej hash funkcie, tak

$$H_v(u) = chaos(S_v(u))$$

[2] https://doc.lagout.org/science/0_Computer%20Science/2_Algorithms/The%20Design%20and%20Analysis%20of%20Computer%20Algorithms%20%5BAho,%20Hopcroft%20&%20Ullman%201974-01-11%5D.pdf

Zakoreňme si náš strom najprv za vrchol 1 a spočítajme všetky hashe s týmto koreňom, čiže $H_1(v)$ pre všetky $v$.

Následne pre vyriešenie úlohy nám stačí vedieť $H_v(v)$ pre všetky $v$, a to vieme spočítať pomerne jednoducho. Väčsinu informácie potrebnú na spočítanie už máme. Pokiaľ $u$ je predok $v$ (v strome zakorenenom za vrchol 1) a $w_1..w_k$ sú potomkovia vrchola $u$ okrem $v$, tak platí, že

$$H_1(w_i) = H_v(w_i)$$

pre všetky $i \in 1..k$. Je to kvôli tomu, že podstromy pod $w_i$ neobsahujú ani vrchol $v$ ani vrchol 1, a teda je ich podoba rovnaká pri zakorenení či už za $v$ alebo za 1.

Zároveň z definície našej hash funkcie vieme, že:

$$H_1(u) = chaos(S_1(u)) = chaos\left(H_1(v) + \sum_{i=1}^{k} H_1(w_i)\right)$$

$$H_v(u) = chaos(S_v(u)) = chaos\left(\sum_{i=1}^{k} H_1(w_i)\right) = chaos(S_1(u) - H_1(v))$$

$$H_v(v) = chaos(S_1(v) + H_v(u)) = chaos\left(S_1(v) + chaos(S_1(u) - H_1(v))\right)$$

Na ľavej strane je to, čo chceme vedieť a na pravej sú informácie spočítané v strome zakorenenom za prvý vrchol. Implementácia sú dve po sebe spustené rekurzívne prehľadania stromu (`dfs_1` a `dfs_2`). Časová zložitosť je $O(n)$, pamäťová tiež $O(n)$

**Listing programu (Python)**

```
import random
import sys

sys.setrecursionlimit(1000000)
random.seed(47)
MOD = 10 ** 16 + 99  # if this prime doesn't work, pick a larger one
n = int(input())
E = [[] for _ in range(n)]
S1, S2 = [0 for _ in range(n)], [0 for _ in range(n)]
for i in range(n - 1):
    a, b = [int(x) - 1 for x in input().split()]
    E[a].append(b)
    E[b].append(a)

_chaos = {}
def chaos(x):
    if x not in _chaos:
        _chaos[x] = random.randint(1, MOD - 1)
    return _chaos[x]

def dfs_1(v, parent):
    for e in E[v]:
        if e != parent:
            S1[v] += dfs_1(e, v)
    S1[v] = S1[v] % MOD
    return chaos(S1[v])

def dfs_2(v, parent):
    if parent < 0:
        S2[v] = S1[v] % MOD
    else:
        sp = (S2[parent] - chaos(S1[v])) % MOD
        S2[v] = (chaos(sp) + S1[v]) % MOD
```

```
    for e in E[v]:
        if e != parent:
            dfs_2(e, v)

dfs_1(0, -1)
dfs_2(0, -1)

print(len(set(S2)))
```

### Vzorové riešenie – labely a AHU

Našim cieľom je efektívne všetkým $n$ zakoreneným stromom, a všetkým vrcholom (tieto zodpovedajú menším stromom pod vrcholmi) v nich priradiť labely (čísla) také, že dva stromy majú rovnaké číslo práve vtedy, keď sú izomorfné.

Podobne ako v predošlej časti, label vrcholu $u$, pokiaľ je celý strom zakorenený za $v$, budeme volať $L_v(u)$.

Zdalo by sa, že takto musíme skonštruovať potenciálne $n^2$ labelov, ale v skutočnosti môže existovať najviac $3n$ rôznych labelov, resp. stromov. Jeden label má každý vrchol ak je koreňom a najviac jeden ďalší label za každého suseda (ak by daný sused bol otcom vrchola). Dokopy majú všetky vrcholy v strome $2n-2$ susedov. Vôbec nás ale nezaujíma, aké labely by dostali nejaké úplne iné stromy, mimo týchto $3n$ stromov, a to nám dá trocha slobody.

Vo vzorovom riešení s hashovaním, sme v podstate riešili problém, že na spočítanie labelu vrcholu $v$ potrebujeme spočítať label vrcholu $u$ a naopak, ale obišli sme to dvoma dfs prechodmi a odčítavaním hashov. Iný spôsob je skúsiť nájsť nejaký jedinečný vrchol $v$ taký, že strom zakorenený vo $v$, nie je izomorfný so žiadnym iným stromom. Takýmto jedinečným vrcholom je centrum stomu, ktoré si hneď vysvetlíme.

Priemer stromu je vzdialenosť najvzdialenejších vrcholov v strome, resp. dĺžka najdlhšej cesty v strome. Na chvíľu ignorujme stromy s priemerom nepárnej dĺžky, neskôr sa k nim vrátime. Ak je priemer stromu párny, tak existuje takzvané centrum, čo je stred najdlhšej cesty. Aj keď najdlhších ciest môže byť viac, centrum je vždy jednoznačné (zamyslite sa prečo). Označme si centrum nášho stromu $c$.

Ak zakoreníme strom za centrum, dostaneme najplytšie možné zakorenenie a navyše strom zakorenený za vrchol $v$ má hĺbku rovnú vzdialenosti $v$ od $c$ plus hĺbka pri zakorenení za $c$. Z tohto vyplýva zaujímavá vec – ak majú byť dve zakorenenia stromu izomorfné, musia byť ich korene rovnako vzdialené od centra.

Druhá, dôležitejšia, vec je, že strom zakorenený za centrum nie je izomorfný so žiadnym iným stromom. $L_c(c) \neq L_v(u)$ pre žiadne $(v,u) \neq (c,c)$. Pre $v = u$ nesedí hĺbka a pre ostatné nesedí počet vrcholov. Dokonca, ak aj strom zakoreníme za hociaký vrchol, tak $L_v(c) \neq L_v(u)$ pre $c \neq u$. Opäť dôvod je jednoduchý – všetky ostatné stromy pod $v$ musia mať inú hĺbku a teda nemôžu byť izomorfné.

S týmito pozorovaniami by sme mali vedieť zvoliť takú definíciu $L_v(u)$, ktorá sa dá ľahko spočítať a zároveň dostatočne spĺňa izomorfickú podmienku (až na jednu výjnimku, viď nižšie).

Definujme si $L_v(c) = -1$ pre všetky $v$, inak povedané, dáme centru label, ktorý nepoužijeme nikde inde. Pre všetky ostatné stromy použijeme na definíciu labelu postup, ktorý poznáme zo sekcie *Izomorfizmus dvoch stromov - labely a AHU*.

$$L_v(u) = m(S_v(u))$$

kde $S_v(u)$ je utriedený zoznam labelov synov $w$ vrchola $u$ za predpokladu, že $v$ je koreň. $m$ je funkcia, ktorá na vstupe dostane zoznam celých čísel a na výstupe vráti jedno prirodzené číslo. Funkcia $m$ vracia rovnaké čísla pre rovnaké zoznamy, a rôzne čísla pre rôzne zoznamy (inak povedané $m$ je prostá funkcia). Implementácia takéhoto niečoho je jednoduchá pomocou mapy alebo hash-mapy (python vie dobre hashovať tuples).

Výstupom algoritmu je počet rôznych hodnôt $L_v(v)$, ktoré spočítame podobne ako v predošlej sekcii, zakoreníme strom za $c$ a dvakrát ho prejdeme pomocou `dfs`.

No počkať, ale strom pod $c$ môže vyzerať inak pre rôzne korene, prečo môžeme mať všade rovnakú hodnotu $L_v(c) = -1$? Všimnime si, že stromy zodpovedajúce $L_v(c)$ a $L_u(c)$ sú izomorfné práve vtedy, keď sú izomorfné stromy zodpovedajúce $L_c(v)$ a $L_c(u)$ (cvičenie pre čitateľa). Vďaka tomu, že pre výpočet $L_v(v)$ používame aj $S_c(v)$, tak nám jedna hodnota $L_v(c)$ nikdy nespraví problém.

A čo ak strom na vstupe nemá centrum? (T.j. priemer stromu má párnu dĺžku). V takom prípade je v strede jedinej najdlhšej cesty hrana. Do stredu tejto hrany pridáme nový vrchol, ktorý bude centrum. Vďaka tomu, že toto nové centrum je opäť unikátne (z hľadiska izomorfizmu), ostane všetko v poriadku. Ak v pôvodnom strome boli nejaké dva zakorenené stromy izomorfné, tak aj v strome s pridaným centrom budú izomorfné. Nesmieme však zabudnúť z výslednej odpovede odpočítať jednotku, lebo sme pridali jeden nový koreň, iný od všetkých ostatných.

Časová zložitosť riešenia je $O(n \log n)$, ale podobne ako pri AHU algoritme, môžeme počítať labely po vrstvách, viď. popis AHU algoritmu vyššie, a dokážeme v lineárnom čase od počtu vrcholov na o jedna hlbšej vrstve spočítať všetky labely pre danú vrstvu. Takéto riešenie má časovú aj pamäťovú zložitosť $O(n)$.

**Listing programu (Python)**

```
import random
import sys
sys.setrecursionlimit(1000000)

# read input
n = int(input())
E = [[] for _ in range(n)]
far_p = [0 for _ in range(n)]

for i in range(n - 1):
    a, b = [int(x) - 1 for x in input().split()]
    E[a].append(b)
    E[b].append(a)

# find center
def farthest(v, parent):
    far_p[v] = parent
    f = (0, v)
    for e in E[v]:
        if e != parent:
            ef = farthest(e, v)
            f = max(f, (ef[0] + 1, ef[1]))
    return f

perimeter, center = farthest(farthest(0, -1)[1], -1)
for i in range(perimeter // 2):
    center = far_p[center]

# add virtual center if needed
if perimeter % 2:
    a, b = center, far_p[center]
    E[a].remove(b)
    E[b].remove(a)
    E.append([a, b])
    E[a].append(n)
    E[b].append(n)
    center = n
    n += 1

# solve
label_map = {}
def label(x):
    x = tuple(x)
    if x not in label_map:
        label_map[x] = len(label_map)
    return label_map[x]

S = [[] for _ in range(n)]
L = [0 for _ in range(n)]

def dfs_1(v, parent):
    S[v] = [dfs_1(e, v) for e in E[v] if e != parent]
```

```
    return label(sorted(S[v]))

def dfs_2(v, parent):
    L[v] = -1 if parent < 0 else label(sorted(S[v] + [L[parent]]))
    _ = [dfs_2(e, v) for e in E[v] if e != parent]

dfs_1(center, -1)
dfs_2(center, -1)
print(len(set(L)) - (perimeter % 2))
```

Pri doprogramovaní si dávajte pozor, že vzorové riešenie v Pythone sa nemusí zmestiť do časového limitu, obzvlášť ak máte $O(n \log n)$ časovú zložitosť. Vzorák v C++ zbehne s prehľadom a naše hashovacie riešenie v Pythone tiež v pohode prejde.

Peter Ralbovský

## 8. Absurdistanské diaľnice

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

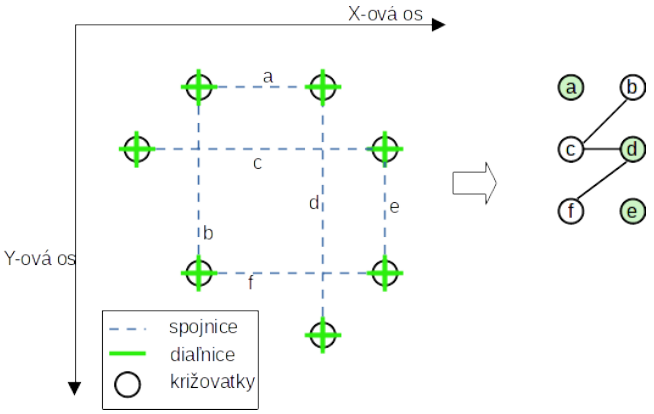
Na vstupe máme niekoľko diaľničných križovatiek. Naším cieľom je navrhnúť severojužné a západovýchodné diaľnice tak, aby:

1. každá križovatka bola križovatkou dvoch diaľníc,
2. aby sa žiadne dve diaľnice nekrižovali mimo križovatie,
3. a aby bolo diaľníc čo najmenej.

Skúsme najprv ignorovať poslednú podmienku a pozrieť sa na riešenie iteratívne: Na začiatku máme veľa križovatiek a na každej z nich sa križujú dve samostatné diaľnice. Ak máme $n$ križovatiek tak máme $2n$ diaľníc. Chceme aby diaľníc bolo čo najmenej, preto chceme tieto diaľnice pospájať. Križovatku môžeme napojiť iba na tú ktorá je k nej najbližšia v každom zo štyroch smeroch. Každé takéto spojenie nám zníži počet diaľníc o jedna. Keby sme ignorovali druhú podmienku tak v podstate každú križovatku môžeme spojiť s najbližšími križovatkami vo všetkých štyroch smeroch a tým pádom by nám pre každú x-ovú súradnicu zostala práve jedna zvislá diaľnica a pre každú y-novú súradnicu zostala práve jedna vodorovná diaľnica.

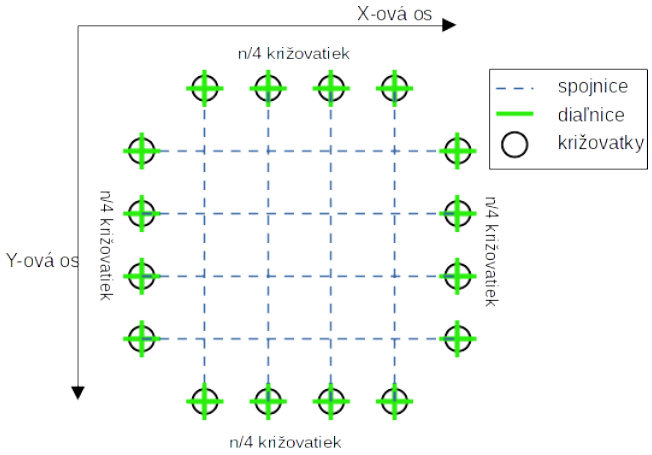
Druhá podmienka však situáciu komplikuje. V praxi pre nás znamená že na každej križovatka spojníc diaľníc ktorá nie je na vstupe si musíme vybrať ktorú spojnicu ponecháme a ktorú nie. Toto nám však úlohu transformuje na vcelku štandartný grafový problém: Pre každú spojnicu majme jeden vrchol v grafe. Pre každú križovatku dvoch spojníc ktorá nie je na vstupe majme hranu ktorá spojí dva vrcholy reprezentujúce tieto dve spojnice. Teraz z grafu chceme vybrať čo najviac vrcholov (spojníc), tak aby žiadne dva vybrané vrcholy neboli spojené hranou. Chceme teda vybrať najväčšiu nezávislú množinu (*maximum independent set*).

Tento postup si môžme ozrejmiť aj na nasledovnom príkladnom obrázku. Na obrázku máme jednotlivé križovatky. Na začiatku sa v každej z nich stretávajú dve diaľnice. Keďže chceme počet diaľníc minimalizovať, chceme vybudovať ich spojnice, kde každá spojnica nám zníži počet jedinečných diaľníc o jedna. V ideálnom prípade chceme využiť všetky spojnice, ale v tomto prípade to nie je možné, keďže diaľnice sa nesmú križovať mimo križovatiek. Túto úlohu si pretransformujeme do bipartitného grafu, kde vrcholy reprezentujú spojnice. Hrany sú medzi vrcholmi práve vtedy keď sa dve spojnice križujú a teda nemôžeme postaviť obidve. Chceme teda vybrať čo najviac vrcholov tak aby medzi žiadnou dvojicou vybraných vrcholov nebola hrana.




Problém najväčšej nezávislej množiny v grafe je vo všeobecnosti v kategorí NP-úplných problémov a teda je pravdepodobné, že sa nám naň nikdy nepodarí nájsť riešenie v polynomiálnom čase. Náš graf má však užitočnú

vlastnosť – je bipartitný. To znamená, že vrcholy v ňom vieme rozdeliť do dvoch kategórií: severojužné spojnice a východozápadné spojnice, pričom hrany sú iba medzi vrcholmi z rôznych kategórií. Tento problém už má známe riešenie, ktoré sme sa pokúsili zhrnúť tu[3]. Časová zložitosť pre riešenie tohto problému je $O(E\sqrt{V})$, pre bipartitný graf s $E$ hranami a $V$ vrcholmi ak hľadáme maximálne párovanie pomocou Edmonds–Karp. V našom prípade však absolútne postačí pomalší algoritmus na hľadanie párenia s časovou zložitosťou $O(EV)$. Keď máme križovatiek $n$, tak môže byť medzi nimi $4n \in O(n)$ spojníc. Tieto spojnice sa však môžu križiť pomerne veľa krát. Napríklad na obrázku nižšie vidíme situáciu v ktorej máme až $(n/4)^2 \in O(n^2)$ križovatiek spojníc. Viac ich však nemôže byť, keďže $4n$ spojníc sa môže križovať najviac v $16n^2 \in O(n^2)$ miestach. Tým pádom je časová zložitosť výsledného riešenia $O(E\sqrt{V}) = O(n^2\sqrt{n}) = O(n^{2.5})$. Pamäťová zložitosť je $O(n^2)$, keďže potrebujeme vyrobiť graf všetkých hrán (križení spojníc).




**Listing programu (C++)**

```
#include <iostream>
#include <vector>
#include <tuple>
#include <map>
#include <algorithm>
#include <cstdio>
#include <set>

using namespace std;

#define FOR(i,n) for(int i = 0; i<(n); ++i)
bool dfs(int a, int vrstva, vector<vector<int>>& G, vector<int>& z_b_do_a,
    ↪ vector<int>& A, vector<int>& B) {
     if (A[a] != vrstva) return 0;
     A[a] = -1;
     for (int b : G[a]) if (B[b] == vrstva + 1) {
             B[b] = 0;
             if (z_b_do_a[b] == -1 || dfs(z_b_do_a[b], vrstva + 1, G, z_b_do_a, A
                 ↪ , B))
                 return z_b_do_a[b] = a, 1;
         }
     return 0;
}

int hopcroftKarp(vector<vector<int>>& G, vector<int>& btoa) {
     int res = 0;
     vector<int> A(G.size()), B(btoa.size()), terajsia_vrstva, dalsia_vrstva;
     while (true) {
```

[3] https://www.ksp.sk/kucharka/parenie/

```
        //polia A a B budu obsahovat vzdialenost po striedavych cestach z
            ↪ nesparovanych vrcholov v A
        fill(A.begin(), A.end(), 0);
        fill(B.begin(), B.end(), 0);
        /// Najdenie nesparovanych vrcholov v A
        terajsia_vrstva.clear();
        // -1 su sparovane vrcholy v A
        for (int a : btoa) if(a != -1) A[a] = -1;
        FOR(a, G.size()) if(A[a] == 0) terajsia_vrstva.push_back(a);
        //
        for (int vrstva = 1; true; vrstva++) {
            bool posledna = false;
            dalsia_vrstva.clear();
            for (int a : terajsia_vrstva) for (int b : G[a]) {
                    if (btoa[b] == -1) {
                        //nasli sme zlepsujucu cestu
                        B[b] = vrstva;
                        posledna = true;
                    }
                    else if (btoa[b] != a && !B[b]) {
                        //tento vrchol je uz sparovany a teda haladame
                            ↪ alternujucu cestu cez jeho sparovanu hranu
                        B[b] = vrstva;
                        dalsia_vrstva.push_back(btoa[b]);
                    }
                }
            if (posledna) break;
            //nenasli sme zlepsujucu cestu ale nie je kam pokracovat teda mame
                ↪ maximalne parenie
            if (dalsia_vrstva.empty()) return res;

            for (int a : dalsia_vrstva) A[a] = vrstva;
            terajsia_vrstva.swap(dalsia_vrstva);
        }
        /// Use DFS to scan for augmenting paths.
        FOR(a, G.size())
            res += dfs(a, 0, G, btoa, A, B);
    }
}

//najde vrcholove pokrytie, vyuziva minimalne parenie
vector<int> vrcholove_pokrytie(vector<vector<int>>& g, int A, int B) {
    vector<int> z_b_do_a(B, -1);
    int res = hopcroftKarp(g, z_b_do_a);
    vector<bool> X(A, true), Y(B);

    //oznaci vrcholy, ktore su sparene, ako tie, ktore zatial nie su v X, usetky
        ↪  ostatne su v X0
    for (int b : z_b_do_a) if (b != -1) X[b] = false;

    // kedze nam nezalezi na poradi vrcholov pocas vyhladavania (chceme iba
        ↪ najst vrcholy v X), tak mozeme namiesto fronty pouzit vector ako stack
    vector<int> q, C;
    // na zaciatku do stacku prehladavanych vrcholov pridame vrcholy z X_0
    for(int i=0; i<A; i++) if (X[i]) q.push_back(i);


    while (!q.empty()) {
```

```
        int a = q.back(); q.pop_back();
        // pridame vrchol a do X
        X[a] = true;
        for (int b : g[a]) if (!Y[b] && z_b_do_a[b] != -1) {
                // ak sme vrchol b doteraz nevideli a je spareny, tak ho pridame
                   ↪ do Y
                // ak je spareny, tak sa uz nechceme vracat po hrane, v ktorej
                   ↪ sme uz boli
                Y[b] = true;
                q.push_back(z_b_do_a[b]);
            }
    }
    // na zaver precislujeme vrcholy napravo a vratime vector vrcholov v
       ↪ najmensom vrcholovom pokryti
    for(int i = 0; i < A; i++) if (!X[i]) C.push_back(i);
    for(int i = 0; i < B; i++) if (Y[i]) C.push_back(A + i);
    return C;
}



long long getn() {
    long long x;
    scanf("%lld",&x);
    return x;
}
typedef tuple<int,int, bool> dialnicny_usek;
int main() {
    map<int, map<int, int>> YX;
    map<int, map<int, int>> XY;
    map<int, map<int, int>> krdole;
    map<int, map<int, int>> krdoprava;
    int n = getn();
    FOR(i,n) {
        int x=getn();
        int y=getn();
        YX[y][x]=i;
        XY[x][y]=i;
    }
    //zlava doprava


    vector<vector<dialnicny_usek> > nechcene_krizenia;
    map<dialnicny_usek, int> mapa_zvislych_usekov, mapa_vodorovnych_usekov;
    for (const auto &it1: YX)
    {
        int y = it1.first;
        for (const auto &it2: XY) {
            int x = it2.first;
            auto it = it2.second.lower_bound(y);
            auto it_pred_x = it1.second.lower_bound(x);
            if(it->first != y && it!=it2.second.begin() && it != it2.second.end
               ↪ ()
            && it_pred_x->first != x && it_pred_x != it1.second.begin() &&
               ↪ it_pred_x != it1.second.end())
            {
                //x y zvisly?
                auto it3= it1.second.lower_bound(x);
```

```
            nechcene_krizenia.push_back({{x, it->first, true}, {it3->first,
                ↪ y, false}});
            mapa_zvislych_usekov[{x, it->first, true}] = 0;
            mapa_vodorovnych_usekov[{it3->first, y, false}] = 0;
        }
    }
}
vector<dialnicny_usek> pole_zvislych_usekov;
for (auto &usek : mapa_zvislych_usekov) {
    usek.second = pole_zvislych_usekov.size();
    pole_zvislych_usekov.push_back(usek.first);
}

vector<dialnicny_usek> pole_vodorovnych_usekov;
for (auto &usek : mapa_vodorovnych_usekov) {
    usek.second = pole_vodorovnych_usekov.size();
    pole_vodorovnych_usekov.push_back(usek.first);
}

vector<vector<int> > nechcene_krizenia_graf(pole_zvislych_usekov.size());
for (auto &krizenie : nechcene_krizenia) {
    nechcene_krizenia_graf[mapa_zvislych_usekov[krizenie[0]]].push_back(
        ↪ mapa_vodorovnych_usekov[krizenie[1]]);
}

vector<int> cov = vrcholove_pokrytie(
        nechcene_krizenia_graf,
        nechcene_krizenia_graf.size(),
        pole_vodorovnych_usekov.size());

set<dialnicny_usek> nepostavime;
for (int i : cov) {
    dialnicny_usek d;
    if (i < mapa_zvislych_usekov.size()) {
        d = pole_zvislych_usekov[i];
    } else {
        d = pole_vodorovnych_usekov[i - mapa_zvislych_usekov.size()];
    }
    nepostavime.insert(d);
}

vector<vector<int> > vodorovne_dialnice, zvisle_dialnice;

for (const auto &it1: YX)
{
    //hladame po riadkoch vodorovne dialnice
    int y = it1.first;
    auto it_next = it1.second.begin();
    int x_zac = -1;
    for (auto it2 = it_next++; ; it2 = it_next++) {
        if (x_zac == -1) {
            x_zac = it2->first;
        }

        // ak uz dalej ziadna krizovatka nie je tak dialnicu skoncime
        bool skonci = it_next == it1.second.end();

        //skoncime ju aj ked dalsi usek sa nam neoplati postavit
```

```
            if(!skonci)
            {
                dialnicny_usek usek = {it_next->first, y, false};
                skonci = nepostavime.count(usek);
            }

            //ked skoncime tak sme nasli koniec a ten si zapiseme
            if (skonci)
            {
                vodorovne_dialnice.push_back({x_zac, y, it2->first, y});
                x_zac = -1;
            }
            if (it_next == it1.second.end()) break;
        }
    }

    for (const auto &it1: XY)
    {
        //hladame po stlpcoch zvisle dialnice
        int x = it1.first;
        auto it_next = it1.second.begin();
        int y_zac = -1;
        for (auto it2 = it_next++; ; it2 = it_next++) {
            if (y_zac == -1) {
                y_zac = it2->first;
            }

            // ak uz dalej ziadna krizovatka nie je tak dialnicu skoncime
            bool skonci = it_next == it1.second.end();

            //skoncime ju aj ked dalsi usek sa nam neoplati postavit
            if(!skonci)
            {
                dialnicny_usek t = {x, it_next->first, true};
                skonci =  nepostavime.count(t);
            }

            //ked skoncime tak sme nasli koniec a ten si zapiseme
            if (skonci)
            {
                zvisle_dialnice.push_back({x, y_zac, x, it2->first});
                y_zac = -1;
            }
            if (it_next == it1.second.end()) break;
        }
    }

    printf("%d\n", vodorovne_dialnice.size());
    for(auto x: vodorovne_dialnice)
        printf("%d %d %d %d\n", x[0], x[1], x[2], x[3]);
    printf("%d\n", zvisle_dialnice.size());
    for(auto x: zvisle_dialnice)
        printf("%d %d %d %d\n", x[0], x[1], x[2], x[3]);

}
```