Korešpondenčný seminár z programovania
XXXVI. ročník, 2018/19
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 1. kola letnej časti

Andrej (ajo@ksp.sk)

1. Párty

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

Našou úlohou je z daného počtu znakov `c` a `v` poskladať reťazec znakov, ktorý bude mať určitý počet prechodov(miesto, kde sú vedľa seba dva rozdielne znaky). Kedže máme počty `c` a `v` zadané, tak našou úlohou je tieto znaky len vhodne preusporiadať.

Riešenie hrubou silou

Kedže hľadáme iba preusporiadanie, ktoré spĺňa určité podmienky, môžeme samozrejme vyskúšať všetky preusporiadania. Pri každom preusporiadaní skontrolujeme, či má požadovaný počet prechodov. V prípade, že áno, ho vypíšeme a skončíme. Pokiaľ chceme skúšať všetky možnosti, zväčša sú na výber dve možné cesty.

Prvá, implementačne možno jednoduchšia, je založená na metóde `next_permutation`. Táto funkcia v `C++` vie vyrábať permutácie. Ako? Ako už názov napovedá, pokiaľ ju zavoláme na nejakú množinu prvkov vo vektore, vráti ďalšiu permutáciu. Ďalšiu v tomto prípade znamená v lexikografickom poradí. Ak teda túto funkciu zavoláme na ***vektor*** obsahujúci `0007`, funkcia vráti `true` a obsah sa zmení na `0070`. Pokiaľ však zavoláme túto funkciu na pole či vektor obsahujúci `7000`, čo je zjavne lexikograficky posledná permutácia, vráti nám `false` a nič nespraví. To znamená, že pokiaľ jej na začiatku dáme utriedený vektor a budeme túto funkciu volať, pokým nevráti `false`, vygeneruje nám každú permutáciu. Užitočný je fakt, že permutácie, ktoré sa líšia iba v preusporiadaní rovnakých prvkov nám vráti iba raz. Kód by mohol vyzerať takto: (Upozorňujeme na celkom neštandardnú konštrukciu `do-while`, ktorá sa často s `next_permutation` používa. Tá zapríčiní, že pred prvým volaním `next_permutation` sa ešte preverí úplne prvá permutácia.)

Listing programu (C++)

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

using namespace std;

int main()
{
    int a, b, x;
    cin >> a >> b >> x;

    vector<int> prac(a+b, 1);

    for(int i=0; i<a; ++i)
        prac[i] = 0;

    do
    {
        int pocet_inverz = 0;

        for(int i=0; i+1<a+b; ++i)
            if(prac[i] != prac[i+1])
                ++pocet_inverz;

        if(pocet_inverz == x)
            break;

    } while(next_permutation(prac.begin(), prac.end()));

    for(int i=0; i<a+b; ++i)
    {
        if(prac[i] == 1) cout << 'v';
        else cout << 'c';
    }
    cout << endl;

    return 0;
}
```

Aká je časová zložitosť tohto riešenia? Mohlo by sa zdať, že permutácií je $n!$. My však vieme, že niektoré permutácie sú totožné, pretože jednotlivé znaky `c` a `v` nerozlišujeme. Preto platí, že permutácií je $\frac{(a+b)!}{a!b!}$. Vďaka použitu funkcie `next_permutation` platí, že časová zložitosť je totožná počtu permutácií, teda $O(\frac{(a+b)!}{a!b!})$. Jedno

volanie `next_permutation` síce môže trvať rádovo až $O(n)$, potom však bude nasledovať viac rýchlych volaní. Toto sa volá amortizovaná časová zložitosť. Pamäťová zložitosť je $O(a+b)$, pretože si musíme pamätať preverovaný reťazec dĺžky práve $a+b$.

Druhá cesta, ktorou sa môžeme vydať, sú triky s bitmi. Pokiaľ chcete, môžete túto časť vzorového riešenia preskočiť, ide skôr o pekný bonus. Mohli by sme povedať, že je zbytočné hrať sa s bitmi, pokiaľ máme na permutovanie funkciu, ktorá je jednoduchá a funkčná. Fakt ale je, že bitové operácie sú na počítači rýchle a tento spôsob sa často dá použiť na výrazné urýchlenie bruteforcu. Predstavme si, že ideme riešiť túto úlohu znova generovaním všetkých preusporiadaní. Ukázali sme si, že máme nejakých n znakov. Čo je dôležité, tie sú iba dvoch typov, znak je teda buď `c`, alebo `v`. To je pri tejto technike veľmi dôležité. Veci, ktoré chceme odlíšiť, musia byť iba dvoch kategórií - to z dôvodu aby ich bolo možné reprezentovať ako `0` a `1`. Ako teraz prísť na všetky preusporiadania? Pokiaľ je vecí n, môžeme si túto situáciu predstaviť ako n-bitové číslo. Situáciu `10001` vieme teraz interpretovať ako `cvvvc`. Čo sme touto reprezentáciou získali? Odpoveď je, že teraz vieme jednoducho generovať všetky možnosti. Pokiaľ prejdeme čísla od 0 po 2^n-1, získame všetky možné n-tice `c` a `v` v každom možnom preusporiadaní, stačí sa nám teda len pozrieť na zápis týchto čísel v dvojkovej sústave. Vyskúšajme si to pre $n=3$, $c=2$, $v=1$. Pomocou troch bitov vieme získať čísla `000`, `001`, `010`, `011`, `100`, `101`, `110`, `111`. Iste, niektoré čísla neodpovedajú počtami bitov rozumnému počtu `c` a `v`. To nám však nevadí, pretože tento spôsob bude veľmi rýchly, nakoľko nám stačí iba inkrementovať jednú premennú. Otázka ešte je, ako z čísla x, vieme dostať hodnotu i-teho bitu... To už je miesto, kde prichádza k spomínanému triku. Na zistenie bitu nám stačí operácia `shift`, ktorá posunie bitovú reprezentáciu čísla doľava a nové miesto doplní nulou. Teda `1 << 2 = 4` pretože bitová reprezentácia 1 posunutá doľava a doplnená nulami nám dá `100`, čo je v desiatkovej sústave 4. Teraz už len potrebujeme operáciu `and`, ktorá vezme dve bitové reprezentácie čísel a urobí po bitoch logický `and`. Teda `121 and 55 = 1111001 and 0110111 = 0110001`. Teraz využijeme trik, ktorý spočíva v tom, že keď spravíme `x and (1<<i)`, dostaneme kladné číslo práve vtedy, keď i-ty bit x sprava bol 1 a nulu ak bol 0. Pýtaním sa na jednotlivé bity vieme zistiť ich jednotlivé hodnoty. Napriek tomu, že to môže vyzerať ako zdĺhavý proces. Pre $n=7$, teda 2^n možností urobíme pri pýtaní sa na jednotlivé bity iba 14 operácií, ktoré sú pre procesor zo súdka najjednoduchších a najrýchlejších. Program by mohol vyzerať nejak takto:

Listing programu (C++)

```
#include <iostream>

using namespace std;

int main()
{
    int c, v, p;
    cin >> c >> v >> p;

    int n = c+v;

    for(int i=0; i<(1<<n); ++i)
    {
        int inver = 0, nc = 0, nv = 0;

        for(int j=n-1; j>=0; --j)
        {
            if(i & (1<<j))
                ++nc;
            else
                ++nv;

            if(j>0 && (bool)(i & (1<<(j-1))) != (bool)(i & (1<<(j))))
                ++inver;
        }

        if(nc == c && nv == v && inver == p)
        {
            for(int j=n-1; j>=0; --j)
            {
                if(i & (1<<j)) cout << 'c';
                else cout << 'v';
            }
            cout << endl;
            return 0;
        }
    }

    return 0;
}
```

Ak si nie ste istý, či ste porozmueli tomuto triku, môžete si ho vyskúšať napr. na úlohe vojaci[1].

Aká je časová zložitosť tohto riešenia? Všetkých vyskúšaných možností je zjavne 2^{a+b}. Overenie každého preusporadania trvá $O(a+b)$. Dokopy to teda celé trvá $O(2^{a+b}\cdot(a+b))$.

[1] https://testovac.ksp.sk/tasks/ls14-vojaci1/

Obidve doterajšie riešenia fungujú na princípe, že si najprv vyberieme nejaké poradie a potom ho kontrolujeme. Nebolo by lepšie nájsť spôsob, akým vieme určiť správne usporiadanie bez toho, aby sme ho museli overovať?

Vzorové riešenie

Pokaždé budeme predpokladať, že počet `v` je väčší nanajvýš rovný počtu `c`. To samozrejme nemusí platiť. V tom prípade si na všetkých miestach, kde použijeme znak `c` môžeme predstaviť znak `v` a naopak. Predstavme si nachvíľu, že sme už všetky znaky `v` uložili, kedže na konci aj tak všetky skončia v koláči a všetky znaky `c` chceme nejako poukladať medzi nich. Na úlohu sa teraz môžeme teraz pozrieť ako na vkladanie `c` do niečoho tvaru `XvXvX...XvXvX`. Na každé miesto označené `X`(chlievik) môžeme dať nejaký, možno aj nulový počet `c`.

Prvá zaujímavá vec, ktorú by sme mohli zistiť je, aký najmenší a najväčší počet prechodov vieme dosiahnuť. Najmenší počet prechodov vieme zrejme dosiahnuť tak, že poukladáme jeden typ cesta za seba a potom ďalší. Dostaneme postupnosť tvaru `c...cv...v` s jedným prechodom. Rozoberme si ešte najväčší dosiahnuteľný počet prechodov. Uložením prvého `c` do chlievika, získame nejaký počet prechodov. Uložením ďalšieho `c` do chlievika, v ktorom sa už jedno `c` nachádza nové prechody nezískame. V krajných chlievikoch vložením získame 1 prechod. Uložením na ostatné `X` získame 2 prechody. Maximálny počet prechodov získame tak, že najskôr ukladáme `c` do vnútorných chlievikov a na koniec do dvoch krajných. Uvedomme si, že obidva krajné chlieviky nikdy nevyužijeme, nakoľko by to znamenalo, že počet `c` je väčší ako počet `v`(`cvcvc`). Rozlíšime teraz dva prípady. Pokiaľ je počet `c` rovný počtu `v` vieme získať $c + v - 1$ prechodov s tým, že využijeme práve jeden okrajový chlievik. Pokiaľ je počet c menší, vieme získať práve $2c$ prechodov. Kedže máme garantované, že riešenie existuje, vieme skonštruovať vzorové riešenie.

Najprv si označíme znak, ktorého je menej m a toho, ktorého je viac v. To, čo namiesto nich budeme vypisovať na výstup teda záleží na jednotlivých počtoch cesta. Uvedomme si, že pokiaľ je počet prechodov nepárny, vieme, že musíme využiť jeden z dvoch krajných chlievikov. Všetky ostatné nám totiž dajú párny počet prechodov. Dohodneme sa, že pokiaľ takéto niečo nastane, vždy umiestnime cesto farby m do prvého chlievika. Riešenie je teraz celkom priamočiare. Postupne plníme chlieviky zľava doprava, pokiaľ dosiahneme želaný počet prechodov, všetky ostávajúce cestá farby m vložíme do posledného použitého chlievika.

Listing programu (C++)

```
#include <iostream>

using namespace std;

int main()
{
    int a, b, x;
    cin >> a >> b >> x;

    char m = 'c', v = 'v';

    if(a > b)
    {
        swap(a, b);
        swap(m, v);
    }

    for(int i=0; i<b; ++i)
    {
        if(i == 0 && (x%2) == 1)
        {
            cout << m;
            --x;
            --a;
        }

        if(i != 0 && x != 0)
        {
            cout << m;
            x -= 2;
            --a;
        }

        if(x == 0) while(a-- > 0) cout << m;

        cout << v;
    }
    cout << endl;

    return 0;
}
```

Časová zložitosť tohto riešenia je $O(a + b)$, kedže pri každej našej operácii trvajúcej konštantný čas sa zníži počet `a` alebo `b`, ktoré zároveň aj vypisujeme. Pamäťová zložitosť je $O(1)$, keďže si výsledné poradie ciest nemusíme pamätať, lebo hneď vieme povedať, v akom poradí budú na torte.

2. Riadny strach

Hodobox (hodobox@ksp.sk)
(max. 12 b za popis, 8 b za program)

Čo vlastne hľadáme

Chceli by sme zistiť pre aké veľkosti skupiniek od 1 po n budú Krtko a Jerry zavolaní spolu. To vieme však vypočítať ako n mínus počet veľkostí skupiniek, pre ktoré budú Krtko a Jerry zavolaní oddelene, teda keď budú patriť do dvoch rôznych po sebe idúcich skupiniek.

A čo musí veľkosť skupinky spĺňať, aby Krtka a Jerry rozdelila? No čísla $r-1$ a r musia patriť do dvoch rôznych skupiniek, teda ak $r-1$ patrí do skupinky číslo k, tak r patrí do skupinky číslo $k+1$. Ak tieto skupinky majú veľkosť x, tak potom bolo v prvých k skupinkách dokopy $kx = r-1$ študentov. Aby toto nastalo, musí byť $r-1$ násobkom veľkosti skupinky x, inými slovami x musí byť deliteľom $r-1$. Keďže každý deliteľ $r-1$ označujúci veľkost skupinky dosiahne rozdelenie Krta a Jerry, celá úloha je vlastne len 'vypíš (n mínus počet deliteľov $r-1$) deleno n v základnom tvare'.

Všetky možnosti

Každý celočíselný deliteľ $r-1$ leží medzi 1 a $r-1$, vrátane – môžeme teda všetky vyskúšať.

Prejdeme for cyklom čísla od 1 po $r-1$, a ak je zvyšok po delení[2] rovný nule, pripočítame k odpovedi jedna.

Nakoniec chceme vypísať zlomok $\frac{n-odpoved}{n}$ v základnom tvare, teda tak aby $n-odpoved$ a n nemali spoločné delitele okrem 1. Môžeme jednoducho prejsť všetky čísla od 1 po $n-odpoved$, a všetkými ktoré delia aj $n-odpoved$ aj n ich predelíme dokým ich nedelia.

Keďže v cykle prechádzame čísla od 1 po $r-1$, čo je najviac n, a vždy vykonáme niekoľko konštantných operácií, toto nám zaberie $O(n)$ času. V druhom cykle tiež určite neprejdeme viac ako n čísel, čiže celková časová zložitosť je $O(n)$.

Stačí si nám pritom pamätať len konštantne veľa premenných, čiže pamäťová zložitosť je $O(1)$.

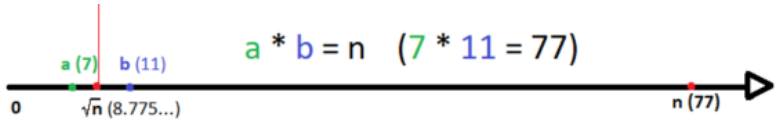
Listing programu (C++)

```
#include<iostream>

using namespace std;

int main()
{
        long long n, r;
        cin >> n >> r;

        long long counter = 0;

        for (long long i=1; i<=r-1; ++i)
        {
                if ((r-1)%i == 0) counter++;
        }

        long long citatel = n-counter;
        long long menovatel = n;

        for (long long i=2; i<=citatel;i++)
        {
                while(citatel%i == 0 && menovatel%i == 0)
                {
                        citatel = citatel/i;
                        menovatel = menovatel/i;
                }
        }
        cout << citatel << " " << menovatel << endl;
}
```

Vzorové riešenie

Ako vieme nájsť počet deliteľov efektívnejšie?

Použijeme pozorovanie ktoré sa často v úlohách o deliteľoch a prvočíslach používajú: ak $a \cdot b = x$, tak aspoň jedno z $a,\ b \leq \sqrt{n}$.



[2] Vo viacerých jazykoch je operátor na zvyšok po delení %

Každý deliteľ väčší-rovný $\sqrt{n}$ má svojho spoločníka ktorý je menší-rovný ako $\sqrt{n}$. Stačí nám teda hľadať delitele po odmocninu z $r-1$, a za každý nájdený pripočítať dva, ledaže by práve platilo $x^2 = r-1$, kde sa deliteľ opakuje a my ho chceme zarátať len raz.

Skrátime teda náš for cyklus a trochu ho prepíšeme, a máme riešenie v $O(\sqrt{n})$.

Na nájdenie zlomku v základnom tvare použijeme Euklidov algoritmus[3] na nájdenie najväčšieho spoločného deliteľa čitateľa a menovateľa, a predelíme ich ním. Euklidov algoritmus zbehne v $O(logn)$, číže celé riešenie nám pobeží v $O(\sqrt{n})$.

Pamäťová zložitosť sa nezmenila – $O(1)$.

Listing programu (C++)

```
#include <iostream>

using namespace std;

long long GCD(long long a, long long b)
{
        if (a>b) swap(a,b);
        if (a==0) return b;
        return GCD(b%a, a);
}

int main()
{
        long long n, r;
        cin >> n >> r;

        long long counter = 0;

        for (long long i = 1; i*i <= r-1; ++i)
        {
        if ((r-1)%i == 0) counter += 2;
        if (i*i == r-1) counter--;
        }

        long long cit = n-counter;
        cout << cit/GCD(cit, n)<< "␣" << n/GCD(cit, n)<< endl;

        return 0;
}
```

buj (bui@ksp.sk)
(max. 12 b za popis, 8 b za program)

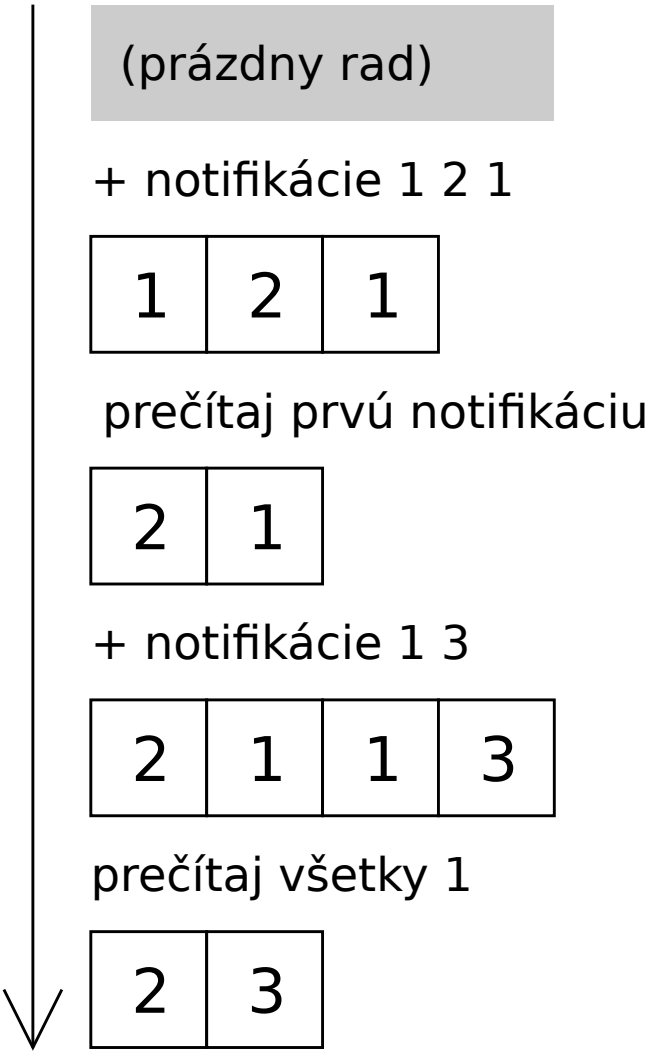
3. Otravné notifikácie

Jednoduchá simulácia

Neprečítané notifikácie si vieme predstaviť, že sú usporiadané do radu podľa toho, kedy prišli. Na začiatku radu sú najstaršie notifikácie, na konci najnovšie. Jednotlivé udalosti potom zodpovedajú nasledovným operáciám:

- Keď príde nová notifikácia, tak ju umiestnime na koniec radu.
- Keď prečítame všetky notifikácie od aplikácie x, z radu vyhodíme všetky také notifikácie. Na to si potrebujeme pre každú notifikáciu pamätať, od ktorej aplikácie pochádza.
- Keď prečítame prvých t notifikácii, z radu vyhodíme prvých t prvkov.

[3] https://sk.wikipedia.org/wiki/Euklidov_algoritmus

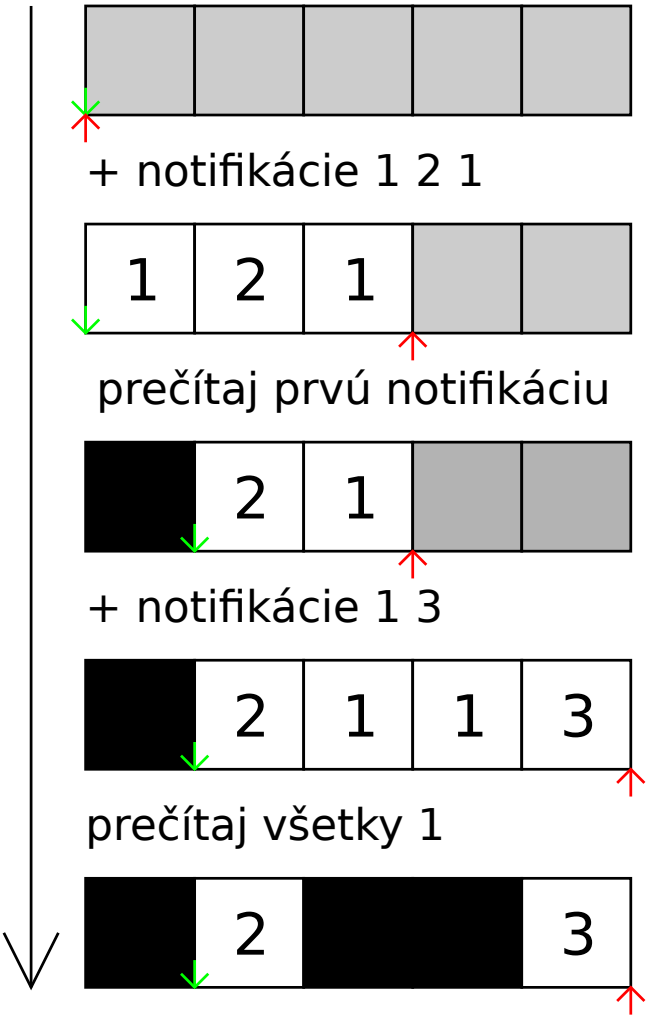


Náš rad je ale neštandardný, požadujeme totiž od neho nielen to, aby sme vedeli pridávať na koniec a mazať odpredu, ale aj to, aby sme vedeli mazať z vnútra radu. Ako to implementovať?

Aby sme vedeli pristupovať do vnútra radu, reprezentujeme si rad štandardne vo vnútri poľa.[4] Máme dva smerníky: jeden na začiatok radu a jeden na koniec radu. Obsah radu je tvorený prvkami, ktoré sa v poli nachádzajú za začiatkom, ale pred koncom, pričom "za" chápeme cyklicky (za posledným políčkom poľa nasleduje prvé). Takýto rad vie mať iba toľko prvkov, koľko je dĺžka poľa. To nám ale nevadí: za celý život nám príde nanajvýš q notifikácii, stačí teda dĺžka poľa q.

V takejto reprezentácii vieme pristupovať k prvkom radu. Čo ale keď chceme niektorý z nich zmazať? Stačí si pre jednotlivé prvky (t.j. políčka poľa) pamätať, či už boli zmazané alebo nie. Pri všetkých operáciách potom ignorujeme už zmazané prvky. - Ak chceme posunúť začiatok radu (napríklad lebo sme prečítali prvú 1 neprečítanú notifikáciu), tak preskočíme na prvý ešte-nezmazaný prvok. - Štandardne sa dĺžka radu dá vypočítať tak, že od seba odrátame pozície smerníkov na koniec a na začiatok radu, modulo dĺžka poľa. V tomto prípade ale nie všetky prvky medzi začiatkom a koncom skutočne sú v rade. Dĺžku radu si preto musíme explicitne počítať.

[4] Viac o rade a príbuzných štruktúrach nájdete tu: https://www.ksp.sk/kucharka/stack_queue_deque/



Listing programu (C++)

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


int main () {
  int n, q;
  cin >> n >> q;

  // implementacia radu
  int pole[q];
  int zac = 0, kon = 0;
  bool zmazane[q];
  for (int i = 0; i < q; i++) {
    zmazane[i] = false;
  }
  int poc = 0;

  for (int qi = 0; qi < q; qi++) {
    int t, x;
    cin >> t >> x;
    if (t == 1) { // pridame prvok na koniec
      pole[kon] = x;
      kon += 1;
      poc += 1;
    }
    else
    if (t == 2) { // precitame vsetko od aplikacie <x>
      for (int i = zac; i < kon; i++) {
        if (zmazane[i]) {
          continue;
        }
        if (pole[i] == x) {
          zmazane[i] = true;
          poc -= 1;
        }
      }
    }
    else { // precitame prvych <x> notifikacii
      for (int xi = 0; xi < x; xi++) {
        while (zmazane[zac]) {
```

```
          zac += 1;
        }
        zmazane[zac] = true;
        zac += 1;
      }
      poc -= x;
    }
    cout << poc << "\n";
  }

  return 0;
}
```

V skutočnosti sa dá zaobísť aj bez operácie "zmaž prvok vo vnútri". Pre záujemcov uvádzame zdrojový kód, skúste z neho sami vyčmuchať, že ako.

Listing programu (C++)

```
Zdrojové kódy budú zverejnené až po 17.6. 23:59:59. Dovtedy sa môžete pokúsiť úlohu naprogramovať sami
(a získať tým nejaké body navyše).
```

Implementácia s použitím modernejších dátových štruktúr, ako je pole —použijeme dynamické pole `std::vector`.

Listing programu (C++)

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

int main () {
  int n, q;
  cin >> n >> q;

  vector<int> notifikacie;
  for (int i = 0; i < q; i++) {
    int t, x;
    cin >> t >> x;
    if (t == 1) { // pridame prvok na koniec
      notifikacie.push_back(x);
    }
    else
    if (t == 2) { // precitame vsetko od aplikacie <x>
      for (int j = 0; j < (int)notifikacie.size(); j++) {
        if (notifikacie[j] == x) {
          notifikacie.erase(notifikacie.begin() + j);
          j--;
        }
      }
    }
    else { // precitame prvych <x> notifikacii
      notifikacie.erase(notifikacie.begin(), notifikacie.begin() + x);
    }
    cout << notifikacie.size() << "\n";
  }

  return 0;
}
```

Pamäťová zložitosť je $O(q)$. Časová zložitosť sa ale ukáže byť až $O(nq)$. Pozrime sa na zložitosti jednotlivých operácii nad radom.

Príchod novej notifikácie máme v čase $O(1)$. Prečítanie prvých t notifikácii trvá v najhoršom prípade toľko, koľko je prvkov medzi smerníkom na začiatok a na koniec, takže na prvý pohľad táto operácia trvá dlho. Ak sa ale pozrieme na "big picture", tak si uvedomíme, že pri tejto operácii sa nám smerník na začiatok vždy posunie v poli dopredu. Nikdy nepretečie, lebo notifikácii je len q a viac ich proste z radu nevyberieme. Pritom pole je dlhé tiež q, teda sa tam všetky notifikácie zmestia. Takže dokopy nám tieto operácie tiež zaberú len $O(q)$. [5]

Problém ale nastáva, keď chceme prečítať všetky notifikácie niektorej aplikácie. Na to potrebujeme z radu vyhodiť nejaké vnútorné prvky—my postupne prejdeme všetky prvky radu, a o každom rozhodneme, či ho chceme alebo nie. Tento postup môže trvať až $O(q)$, pričom niekedy za toľko veľa práce nemáme veľmi čo ukázať. Napríklad keď prečítame všetko od aplikácie 1, môže sa stať, že prejdeme celý rad dlhý $O(q)$ a zistíme, že žiadna notifikácia od aplikácie 1 v ňom nie je. Spravili sme teda veľa práce, ktorú sme si možno mohli ušetriť.

Vzorové riešenie

Vyššie uvedená analýza nám hovorí, ktorú operáciu potrebujeme urýchliť: čítanie všetkých notifikácii niektorej aplikácie. Ak by sme si vedeli nejakým spôsobom pre každú aplikáciu pamätať, v ktorých políčkach poľa sú jej notifikácie, vyhrali by sme.

[5] Odbornými slovami, táto operácia zaberá čas $O(1)$ amortizovane —niektoré jej volania síce budú trvať dlho, avšak dokopy to v priemere vyjde len $O(1)$.

No ale to vieme spraviť, nie? Iba si pre každú aplikáciu udržujeme zoznam políčok, na ktorých sa nachádzajú jej notifikácie. - Keď príde nová notifikácia, tak príslušnej aplikácii na koniec zoznamu pridáme pozíciu v poli, kam notifikácia prišla. - Keď prečítame všetky notifikácie aplikácie, tak iba prejdeme tie prvky poľa, ktoré sú v jej zozname, a označíme ich za zmazané. Zoznam potom vyprázdnime. - Čo ale, keď prečítame prvú neprečítanú notifikáciu? Celkom vhod nám príde, že je prvá—v zozname pre danú aplikáciu je teda určite na začiatku. Stačí je teda vymazať.

Od týchto zoznamov teda požadujeme len toľko, aby sme vedeli pridávať na koniec, prejsť cez všetky prvky, a mazať zo začiatku. Tieto požiadavky spĺňa (opäť) dátová štruktúra rad. Tentokrát na implementáciu ale nepoužijeme pole, nakoľko nemáme dobrý horný odhad na maximálnu veľkosť radu—určite v ňom bude nanajvýš q prvkov, avšak týchto zoznamov chceme n, a nq pamäte je už veľa. Namiesto toho tieto rady implementujeme ako spájané zoznamy.

Listing programu (C++)

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


// implementacia spajaneho zoznamu
struct Zoznam {
  int pozicia;
  Zoznam* dalsi;

  Zoznam () : dalsi(NULL) {}
  Zoznam (int poz0) : pozicia(poz0), dalsi(NULL) {}
};


int main () {
  int n, q;
  cin >> n >> q;

  // implementacia radu
  int pole[q];
  int zac = 0, kon = 0;
  bool zmazane[q];
  for (int i = 0; i < q; i++) {
    zmazane[i] = false;
  }
  int poc = 0;

  // zoznamy pre jednotlive aplikacie
  Zoznam* zaciatok[n+1];
  Zoznam* koniec[n+1];
  for (int i = 1; i <= n; i++) {
    zaciatok[i] = NULL;
    koniec[i] = NULL;
  }

  for (int qi = 0; qi < q; qi++) {
    int t, x;
    cin >> t >> x;
    if (t == 1) { // pridame prvok na koniec
      pole[kon] = x;
      { // upravime zoznam pre aplikaciu <x>
        if (koniec[x] != NULL) {
          koniec[x]->dalsi = new Zoznam(kon);
          koniec[x] = koniec[x]->dalsi;
        }
        else {
          zaciatok[x] = new Zoznam(kon);
          koniec[x] = zaciatok[x];
        }
      }
      kon += 1;
      poc += 1;
    }
    else
    if (t == 2) { // precitame vsetko od aplikacie <x>
      Zoznam* kde = zaciatok[x];
      while (kde != NULL) {
        zmazane[kde->pozicia] = true;
        Zoznam* dalsi = kde->dalsi;
        delete kde; // v C++ nie je garbage collection, pamat si manazujeme sami...
        kde = dalsi;
        poc -= 1;
      }
      zaciatok[x] = NULL;
      koniec[x] = NULL;
    }
    else { // precitame prvych <x> notifikacii
      for (int xi = 0; xi < x; xi++) {
        while (zmazane[zac]) {
          zac += 1;
        }
        int ap = pole[zac];
        Zoznam* dalsi = zaciatok[ap]->dalsi;
        delete zaciatok[ap]; // v C++ nie je garbage collection, pamat si manazujeme sami...
        zaciatok[ap] = dalsi; // zmazeme prvy prvok v zozname citanej aplikacie
        if (zaciatok[ap] == NULL) {
```

```
          koniec[ap] = NULL;
        }
        zmazane[zac] = true;
        zac += 1;
      }
      poc -= x;
    }
    cout << poc << "\n";
  }

  return 0;
}
```

Týmto sa nám podarilo zlepšiť časovú zložitosť druhej operácie. Opäť, jedna operácia môže trvať až $O(q)$, ale keď sa pozrieme na big picture, zo zoznamu nemôžeme vybrať viac prvkov, ako do neho za celý život vložíme. Každá notifikácia príde len do jedného zoznamu, a teda dokopy vyberieme $O(q)$ prvkov.[6]

Dostávame tak výslednú časovú zložitosť $O(n+q)$ (potrebujeme $O(n)$ na inicializáciu jednotlivých zoznamov).

Pre záujemcov zdrojový kód alternatívneho riešenia. Je celkom rovnaké, ako to predchádzajúce, až na to, že inak pristupuje k udalostiam tretieho typu. Skúste vyčmuchať, ako funguje a prečo má dobrú časovú zložitosť.

Listing programu (C++)

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


// implementacia spajaneho zoznamu
struct Zoznam {
  int pozicia;
  Zoznam* dalsi;

  Zoznam () : dalsi(NULL) {}
  Zoznam (int poz0) : pozicia(poz0), dalsi(NULL) {}
};


int main () {
  int n, q;
  cin >> n >> q;

  // implementacia radu
  int pole[q];
  int zac = 0, kon = 0;
  bool zmazane[q];
  for (int i = 0; i < q; i++) {
    zmazane[i] = false;
  }
  int poc = 0;

  // zoznamy pre jednotlive aplikacie
  Zoznam* zaciatok[n+1];
  Zoznam* koniec[n+1];
  for (int i = 1; i <= n; i++) {
    zaciatok[i] = NULL;
    koniec[i] = NULL;
  }

  for (int qi = 0; qi < q; qi++) {
    int t, x;
    cin >> t >> x;
    if (t == 1) { // pridame prvok na koniec
      pole[kon] = x;
      { // upravime zoznam pre aplikaciu <x>
        if (koniec[x] != NULL) {
          koniec[x]->dalsi = new Zoznam(kon);
          koniec[x] = koniec[x]->dalsi;
        }
        else {
          zaciatok[x] = new Zoznam(kon);
          koniec[x] = zaciatok[x];
        }
      }
      kon += 1;
      poc += 1;
    }
    else
    if (t == 2) { // precitame vsetko od aplikacie <x>
      Zoznam* kde = zaciatok[x];
      while (kde != NULL) {
        if (!zmazane[kde->pozicia]) {
          zmazane[kde->pozicia] = true;
          poc -= 1;
        }
        Zoznam* dalsi = kde->dalsi;
        delete kde; // v C++ nie je garbage collection, pamat si manazujeme sami...
        kde = dalsi;
      }
      zaciatok[x] = NULL;
      koniec[x] = NULL;
    }
```

[6] Amortizovane $O(1)$.

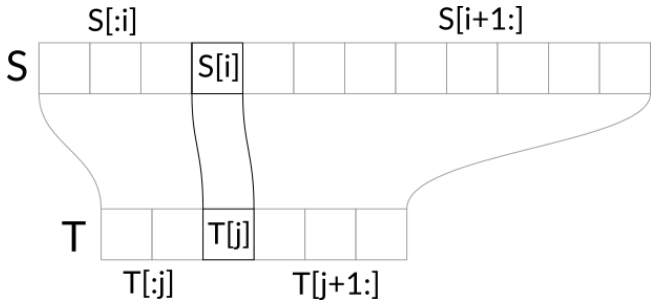
```
    else { // precitame prvych <x> notifikacii
      for (int xi = 0; xi < x; xi++) {
        while (zmazane[zac]) {
          zac += 1;
        }
        zmazane[zac] = true;
        zac += 1;
      }
      poc -= x;
    }
    cout << poc << "\n";
  }

  return 0;
}
```

Adam (kral@ksp.sk)
(max. 12 b za popis, 8 b za program)

4. Treba sa najesť

Predtým než sa ponoríme do riešenia, zaveďme si notácie zľahčujúce zápis niektorých vzťahov. `A[i:j]` je súvislý úsek reťazca `A`, ktorý začína na i-tom písmene a končí na j-tom písmene (použitý zápis teda zodpovedá polootvorenému intervalu). Tu treba zdôrazniť fakt, že ľavý index sa berie vrátane, ale pravý index sa neberie. Navyše, ak vynecháme ľavý index, znamená to začiatok slova a vynechanie pravého indexu znamená koniec slova. Teda `A[:]` je v tomto prípade celý reťazec `A`[7]. Okrem toho, ak povieme, že `A` *sa zmestí do* `B`, budeme tým myslieť to, že `A` je podreťazcom `B`. Posledné dôležité výrazy, ktoré možno už poznáte, sú prefix a suffix. Prefix označuje súvislú časť reťazca, ktorá začína na jeho začiatku, teda `A[:i]` a suffix označuje súvislú časť reťazca, ktorá končí na jeho konci, teda `A[i:]`.

Začnime si ujasnením toho, čo musí platiť pre dvojicu reťazcov `S` a `T`, pre ktoré je odpoveď `ano`. Vieme, že každý znak z `S` na pozícii i (pre $0 \leq i < |S|$) sa musí nachádzať v nejakom podreťazci tvoriacom `T`. To znamená, že musí existovať také j, že j-ty znak v `T` je rovnaký ako i-ty znak v `S` a zároveň sa `T[:j]` zmestí do `S[:i]` a `T[j+1:]` sa zmestí do `S[i+1:]`.



Ako prvé by sme si teda mohli rozmyslieť, ako overiť, že sa reťazec `A` zmestí do reťazca `B`. To je však pomerne jednoduché. Postupne budeme prechádzať reťazcom `B`, pričom si pamätáme index do `A`, ktorý nám hovorí, ako dlhý prefix `A` sme už v `B` videli. Počiatočná hodnota tohto indexu je, samozrejme, 0. Ak narazíme v `B` na znak, ktorý sa zhoduje s písmenom na tomto indexe v `A`, posunieme index o jedna ďalej. A na konci reťazca `B` iba overíme, že sme prešli celým `A`. Časová zložitosť tohto postupu je lineárne závislá od dĺžky `B`. To je však príliš pomalé, ak to chceme robiť pre každú dvojicu znakov v `S` a `T`.

My si však dokážeme takúto informáciu získať aj rýchlejšie, a to dynamickým programovaním. Musíme si totiž uvedomiť, že naše otázky sa netýkajú náhodných reťazcov. Zakaždým sa pýtame iba na to, či sa nejaký prefix `T` zmestí do prefixu `S` (a rovnako so sufixami). Pre každý prefix `S` by sme si mohli predpočítať, aký najdlhší prefix `T` sa doň zmestí. Jednak vďaka tomu vieme odpovedať na ľubovoľnú otázku, lebo dlhšie prefixy sa určite nezmestia a všetky kratšie sa zmestiť musia, a jednak vieme tieto hodnoty počítať v jednom prechode a to práve spomínaným dynamickým programovaním.

Ak je `T[:j]` podreťazcom `S[:i]`, potom určite bude `T[:j]` aj podreťazcom `S[:i+1]`, pretože sme len pridali písmeno. Ale naviac, ak sa `S[i]` rovnalo `T[j]`, tak vieme, že aj `T[:j+1]` je podreťazcom `S[:i+1]`. Rovnakým postupom ako predtým, pomocou jedného indexu do `T` si teda vieme predpočítať pole, ktoré nám pre ľubovoľný prefix `S` povie, aký najdlhší prefix `T` sa doň zmestí. Toto isté, akurát odzadu, vieme spraviť aj pre suffixy. Teraz už vieme ľahko zistiť, či sa `T[:j]` nachádza v `S[:i]`, stačí sa pozrieť, aký najdlhší prefix `T` sa zmestí do `S[:i]`, čo máme predpočítané v poli, a porovnať ho s j.

V tomto bode máme riešenie s časovou zložitosťou $O(|S| \cdot |T|)$. Pre každý index i v reťazci `S` vyskúšame každý možný index j v `T` a overíme, či sú tieto znaky rovnaké a či sa zvyšok reťazca `T` zmestí do príslušných častí `S`, čo pomocou predpočítaných hodnôt vieme robiť v konštantnom čase. Predpočítanie hodnôt však trvá

[7] Táto notácia zodpovedá tomu, ako sa s reťazcami pracuje v Pythone.

iba $O(|T|)$ a ich používanie konštantný čas, najpomalšia časť riešenia je teda tá, kde pre znak z `S` zisťujeme, ktorý znak z `T` mu priradiť. Túto časť by sme teda chceli urýchliť.

Majme i-ty znak z S a najdlhší možný prefix, ktorý sa zmestí do `S[:i]` je `T[:j]`. To znamená, že i-ty znak musí zodpovedať niektorému z prvých $j+1$ znakov T, inak by sa zvyšný prefix `T` do `S[:i]` nezmestil. Nech je teda i-ty znak napríklad `x`. Potom tento znak musí zodpovedať niektorému `x` z prvých $j+1$ znakov reťazca `T`. Uvedomme si však, že je vhodné vybrať čo najneskorší výskyt `x` v `T[:j+1]`. Prefix[8] totiž bude stále vyhovovať, a čím väčšiu hodnotu si vyberieme, tým kratší suffix stačí dať do `S[i+1:]`. To znamená, že ak sa i-ty znak z `S` nachádza v nejakej podpostupnosti tvoriacej `T`, tak to musí byť aj v prípade, keď je toto i-te písmeno najpravejší výskyt v `T[:j+1]`.

Chceli by sme preto vedieť rýchlo odpovedať na otázku: Ktorá najpravejšia pozícia `T[:j]` obsahuje písmeno x? Pre každé písmeno si vytvoríme pole dĺžky $|T|$, kde na pozícii k bude posledná pozícia daného písmena v prefixe `T` dĺžky k. To sa počíta ľahko, keďže buď je táto hodnota rovnaká ako znak predtým, alebo je to hodnota aktuálnej pozície, ak je na nej hľadaný znak.

Časová zložitosť predpočítania polí je lineárna od dĺžky reťazca `S` a časová zložitosť počítania výskytov písmen je lineárna od dĺžky reťazca `T` a dĺžky abecedy (v tomto prípade ju môžeme považovať za konštantu 26). Nakoniec netreba zabudnúť na počet vstupov n. Výsledná časová zložitosť je $O(n \cdot (|S|+|T|))$. Pamätať si potrebujeme iba dve polia dĺžky $|S|$ a 26 polí dĺžky $|T|$ a táto informácia nazáleží na počte vstupov, pamäťová zložitosť je teda $O(|S|+|T|)$.

Listing programu (C++)

```
#include <cstdio>
#include <iostream>
#include <string>
#include <algorithm>
#include <vector>
#include <cmath>
#include <queue>
#include <set>
#include <map>
#include <stack>
using namespace std;

#define For(i,n) for(int i=0; i<(n); i++)
#define mp(a,b) make_pair((a),(b))
typedef long long ll;
typedef pair<int,int> pii;

int main2() {
    char C[200047];
    scanf("␣%s",C);
    string s = C;
    scanf("␣%s",C);
    string t = C;
    vector<vector<int> > Pp(26), Ps(26);
    For(i, 26) {
        Pp[i].resize(t.length()+1);
        Pp[i][0] = -1;
        Ps[i].resize(t.length()+1);
        Ps[i][t.length()] = t.length();
    }
    For(i, t.length()) {
        For(j, 26) Pp[j][i+1] = Pp[j][i];
        Pp[t[i] - 'a'][i+1] = i;
    }
    for(int i = t.length()-1; i >=0; i--) {
        For(j, 26) Ps[j][i] = Ps[j][i+1];
        Ps[t[i] - 'a'][i] = i;
    }
    vector<int> P(s.length()),S(s.length());
    int index = 0;
    For(i, s.length()) {
        if(index != t.length() && s[i] == t[index]) index++;
        P[i] = index;
    }
    index = t.length()-1;
    for(int i = s.length()-1; i >= 0; i--) {
        if(index != -1 && s[i] == t[index]) index--;
        S[i] = index + 1;
    }
    bool res = true;
    For(i, s.length()) {
        int left = Pp[s[i] - 'a'][P[i]], right = Ps[s[i] - 'a'][S[i]];
        if(left < right) res = false;
    }
    if(res) printf("ano\n");
    else printf("nie\n");
}

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

[8] Prefix ako predpona, nie Prefix ako vedúci.

```
    For(i, t) main2();
}
```

Marek (marekc@ksp.sk)
(max. 12 b za popis, 8 b za program)

5. Ooo, Vlaky

Keďže zadanie úlohy je pomerne dlhé, zhrňme si najprv, o čo tu ide. Na vstupe máme popísaný graf (strom) s n vrcholmi a n bodov v rovine. Úlohou je položiť vrcholy grafu do bodov v rovine tak, aby sa žiadne hrany nepretínali. Pritom hrany grafu sa zobrazia na úsečky medzi bodmi v rovine. Ako to spraviť?

Riešenie hrubou silou

Prvou možnosťou by bolo niečo veľmi pomalé. Všetkým bodom v rovine priradíme nejaké vrcholy grafu a skontrolujeme, či nám kolidujú nejaké hrany. Ak áno, skúsime vrcholy rozmiestniť nejako inak. Inak povedané, permutácie usporiadanej množiny vrcholov skúšame zobraziť na usporiadanú množinu bodov, kým nenájdeme správne riešenie.

Hmmm. Jednak by to bolo trochu neefektívne, že aj keď zmeníme medzi jednotlivými permutáciami len malú časť vrcholov, zase musíme kontrolovať všetky dvojice úsečiek, a dvak, asi nechceme skúšať všetky permutácie, chceme tento problém vyriešiť nejako rozumne postupne.

Čo keby sme vrcholy ukladali postupne po jednom a pre každý pridaný vrchol by sme hneď zistili, či sa jeho položením niečo preťalo? Tým pádom by sme vedeli, či má zmysel pokračovať v takomto rozložení vrcholov. Takto by nám stačilo vždy pri uložení vrcholu skontrolovať len tie hrany, ktorými sme tento vrchol pripojili ku zvyšku grafu.

Úlohu si vyriešime rekurzívne tak, že v každom kroku máme x vrcholov už uložených a $n - x$, ktoré sú ešte voľné. Keďže chceme skúšať rôzne možnosti uloženia vrcholov, stačí nám meniť poradie vrcholov a body v rovine necháme napríklad v poradí zo vstupu. Preto majme usporiadanú množinu bodov v rovine. V i-tom kroku uložíme nejaký vrchol do bodu i.

Na začiatku $x = 0$. Vyberieme si nejaký vrchol, ten uložíme, skontrolujeme, či sa jeho hrany s niečím nepretínajú, a krok opakujeme s $n - x$ vrcholmi. Ak v nejakom kroku zistíme, že ešte máme neuložené vrcholy a nevieme už žiaden uložiť tak, aby nám nič nekolidovalo, vrátime sa o krok späť a skúsime namiesto naposledy uloženého vrcholu uložiť nejaký iný a pokračovať ďalej. Toto opakujeme, až kým nie sú všetky vrcholy uložené. Všimnime si, že keď sa zasekneme a pozmeníme nejako uloženie, nemusíme zase porovnávať všetky dvojice úsečiek, ale len tie, ktoré zahrňajú novo pridané hrany.

Takýto dynamický prístup síce vyzerá pekne, no jeho asymptotická časová zložitosť až tak pekne nevyzerá. Hrozí tu, že v najhoršom prípade potrebujeme preveriť $n!$ uložení grafu, pričom ale pre každú vetvu výpočtového stromu skontrolujeme n^2 dvojíc úsečiek. Z toho nám vychádza časová zložitosť $O(n! \cdot n^2)$.[9]

Zamyslime sa. Doteraz sme pracovali s myšlienkou, že budeme nejakým spôsobom skúšať napasovať zadaný graf do zadaných súradníc a testovať pri tom, či je daná možnosť vyhovujúca alebo niečo koliduje. Čo keby sme vedeli nejakým spôsobom vopred zaručiť, že vybrané uloženie je korektné, aj bez potreby overovania?

Vzorové riešenie

Každý strom si vieme rozdeliť na menšie podstromy a tie na menšie podstromy a tak ďalej, až dostaneme strom hĺbky 1. Strom hĺbky 1 môžeme inak nazvať hviezda. Takto nízky strom vieme uložiť do roviny ľubovoľne a aj tak sa nestane, že sa nejaké hrany budú pretínať (samozrejme počítame s tým, že žiadne 3 body neležia ne jedenej priamke).

Ak si zvolíme v zadanom strome nejaký koreň, potom si predstavme podstromy, ktorých koreňmi sú jeho synovia. Každý takýto podstrom bude okupovať nejakú časť roviny, ktorú vieme ohraničiť konvexným obalom jeho vrcholov. Ak by sme vedeli tieto pre každý podstrom vybrať množinu tak, aby sa ich obaly nepretínali, mali by sme už čiastočne vyhrané. Totiž ak sa obaly nepretínajú, nemôžu sa pretínať ani hrany. Ak by sme toto vedeli opakovať rekurzívne do hĺbky, mali by sme vyhrané úplne, lebo takto vieme uložiť všetky vrcholy grafu.

Jediné, čo už teraz potrebujeme vyriešiť je, ako rozdeliť body v rovine do jednotlivých podstromov tak, aby sa ich obaly nepretínali. Keď budú body každého podstromu "pri sebe", tak snáď nebudú zavadzať iným podstromom. Chceme ich teda "poskupinkovať" nejako lokálne.

Dobre. Z pohľadu ľubovoľného bodu koreňa vidíme každý iný bod pod nejakým uhlom. Zo zadania sme sa dočítali, že žiadne tri body neležia na rovnakej priamke. Preto vieme, že takýto uhol je pre každý bod unikátny. Hmmm. Ak sme v koreni a všetky dostupné body utriedime podľa tohto uhla, vieme si ich v tomto poradí "poskupinkovať" a prerozdeliť svojim synom. Tým pádom sa žiadna skupina bodov "nemieša" s nejakou inou,

[9] Dá sa spraviť šikovnejšia analýza časovej zložitosti, z ktorej vyjde $O(n \cdot n!)$. Nie že by sme si veľmi pomohli...

pretože body každej tejto množiny pozorujeme pod uhlom z nejakého intervalu, pričom body ďalšej sledujme pod uhlom z iného intervalu, pričom tieto intervaly majú prázdny prienik.

A funguje to vždy? Nie.

Ak si pre koreň vyberieme ľubovoľný bod, môže sa stať, že interval, pod ktorým vidíme istú množinu bodov bude väčší než π, čo by znamenalo, že plocha ohraničená konvexným obalom tejto množiny bodov by obsahovala aj koreň. Tým pádom, by sa mohlo stať, že nejaká hrana v tomto území by pretínala hranu medzi koreňom a iným synom. Tento problém vieme ale jednoducho vyriešiť tak, že ako koreň stromu zvolíme najľavejší bod. To nám zaručí, že všetky ostatné body sa budú nachádzať vpravo od neho, čo inak znamená aj to, že interval uhlov, pod ktorými budú viditeľné z koreňa, bude najviac od $\frac{\pi}{2}$ po $-\frac{\pi}{2}$ otvorený z aspoň jednej strany, keďže žiadne tri body neležia na rovnakej priamke.

Ten istý problém sa môže stať aj na ďalších úrovniach: možno sa podstrom zobraznený na svojej pridelenej podmnožine pretne s hranou, ktorá je medzi jeho koreňom a jeho otcom. To opäť vyriešime extrémnou voľbou koreňa podstromu: ak ho umiestnime do prvého bodu v poradí (podľa uhla). Keďže všetky ostatné body budú zvierať s jeho otcom menší uhol, nebude tak možné, aby sa ľubovoľná úsečka medzi týmito bodmi pretla s tou spomenutou.

Tým pádom všetky usporiadané podmnožiny bodov sú pridelené nejakým podstromom. To znamená, že túto úlohu vieme vyriešiť pekne rekurzívne.

Najprv si porebujeme prehľadaním do hĺbky spočítať veľkosti jednotlivých podstromov, aby sme neskôr vedeli, koľko bodov máme konkrétnemu podstromu vyčleniť. Potom už vieme rekurzívnym prehľadávaním do hĺbky ukladať najprv koreň celého stromu, potom jeho synov a tak ďalej, pričom vždy uložíme koreň, polárne utriedime zvyšné body práve vzhľadom na koreň daného podstromu a rozdelíme tieto body jeho synom, podľa veľkostí ich podstromov.

Zložitosti. To, že vrcholy ukladáme do roviny rekurziou do hĺbky, nás môže ľahko zlákať na myšlienku, že pre každú úroveň rekurzie si potrebujeme pamätať zase novú množinu bodov, lebo si chceme pôvodnú usporiadanú množinu pamätať pre prípad, že sa k nej z rekurzie vrátime a budeme s ňou ďalej pracovať. Najhorší prípad by potom nastal, ak by graf na vstupe bol cesta a zakorenili by sme si ho ma jej konci. Vtedy by sme si najprv pamätali n bodov, potom $n-1$, $n-2$, ... čo je asymptoticky n^2. Z toho dostávame teda zložitosť $O(n^2)$.

Uvedomme si ale, že ak časť (istý súvislý úsek usporiadanej množiny) bodov priradíme nejakému synovi a jeho podstromu, ostatní synovia s touto časťou už nepracujú. Preto nevadí, že v tejto podmnožine zmeníme poradie prvkov predtým, ako rozdelíme aj všetky ostatné prvky. To znamená, že nám úplne postačuje pamätať si množinu všetkých bodov len raz a potom pracovať s jej podmnožinami bez toho, aby sme museli zachovaávať pôvodné poradie. Pamäťovú zložitosť tohto riešenia tak vieme odhadnúť na $O(n)$.

A časová zložitosť? Spočítanie veľkostí podstromov trvá len $O(n)$. Triediť vieme v $O(n \log n)$, ale potrebujeme to robiť pre každý podstrom, no na druhej strane, pre každý podstrom sa nám mení n. V najhoršom prípade by zadaný graf bol cesta. Vtedy by sme vždy v i-tej úrovni rekurzie polárne triedili $n-i$ bodov a úrovní by bolo n. Z toho nám vychádza asymptotická časová zložitosť $O(n^2 \log n)$.

Listing programu (C++)

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

using namespace std;

struct Point {
        long x, y;
        int id;
};

// porovnavacia funkcia pre usporiadanie podla x-ovej suradnice
bool compare_x (Point a, Point b) {
        return a.x < b.x;
}

// chceme rozhodovat o poradi dvoch bodov, co zavisi od tretieho
// vieme to implementovat cez objekt toho tretieho bodu:
class CompareClass {
public:
        Point root;

        CompareClass (Point p) : root(p) {}

        // porovnavacia metoda pre polarne triedenie
        bool operator() (const Point a, const Point b) {
                return ((a.x - root.x) * (b.y - root.y) - (b.x - root.x) * (a.y - root.y)) > 0;
        }
};

vector <int> Tree_size;
vector <vector <int> > Node, Syn;
vector <Point> Points;
```

```
void dfs (int node, int parent) {
        Tree_size[node] = 1; // 1 bod zaberie uz koren podstromu

        for (int i = 0; i < Node[node].size(); i++) {

                int syn = Node[node][i];
                if (Node[node][i] != parent) { // ak to nie je moj parent
                        dfs(Node[node][i], node); // vypocitam velkost
                        Syn[node].push_back(syn); // a dam do zoznamu synov
                        Tree_size[node] += Tree_size[syn];
                }
        }
}

int solve (int node, int left, int right) {
        CompareClass compareClass(Points[left]); // koren je v prvom bode

        if (right - left > 1) // polarne triedenie
                sort(Points.begin() + left + 1, Points.begin() + right, compareClass);

        int tmp = 1; // 1 bod uz pre samotny koren
        for (int i = 0; i < Syn[node].size(); i++) {
                // zvysne rozdelime synom

                int left_border = left + tmp;
                tmp += Tree_size[Syn[node][i]];
                int right_border = left + tmp;

                // tunel ma existovat medzi aktualnym korenom a jeho synom
                printf("%d %d\n", compareClass.root.id, solve(Syn[node][i], left_border, right_border));
        }

        return compareClass.root.id;
}

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

        Tree_size.resize(n);
        Node.resize(n);
        Syn.resize(n);
        Points.resize(n);

        for (int i = 0; i < n - 1; i++) {
                int a, b;
                scanf("%d %d", &a, &b);
                Node[a].push_back(b);
                Node[b].push_back(a);
        }

        // spocitame velkosti podstromov
        dfs(0, -1);

        for (int i = 0; i < n; i++) {
                scanf("%ld %ld", &Points[i].x, &Points[i].y);
                Points[i].id = i;
        }

        // zoradime podla x-ovej suradnice
        sort(Points.begin(), Points.end(), compare_x);

        solve(0, 0, n);
}
```

Poznámka na záver: to, že týmto spôsobom nájdeme správne riešenie, neznamená, že nájdeme jediné správne. Môže existovať strom uložený v rovine aj keď neexistuje taký jeho koreň, z ktorého by sme videli množiny bodov podstromov jeho synov a vedeli si ich takto rozdeliť do konvexných množín. My sme chceli len nájsť ľubovoľné riešenie pre uloženie stromu, nie všetky.

Žaba (`zaba@ksp.sk`)
(max. 12 b za popis, 8 b za program)

6. Totálna pohroma

Úloha vyzerá už na prvý pohľad komplikovane. Máme počítať počet možností, niektoré časti sú už vopred určené a tie dopĺňať nemáme a popritom ešte máme kontrolovať, že v každom diagonálnom štvorci bude párny počet jednotiek. Keďže je toho veľa, treba sa skúsiť zamerať na nejakú menšiu časť, objaviť niekoľko základných pravidiel a odtiaľ sa posúvať k všeobecnému riešeniu. A samozrejme celý čas si kresliť, to pomáha najviac. Nuž a keď sme si nie istý, ktorým smerom sa vydať, pomôcť nám môžu návodné, ľahšie, sady.

Zadaná diagonála

Hlavná diagonála našej matice je dôležitá, keďže nás zaujímajú iba štvorce v jej okolí. V prvej sade navyše máme zaručené, že všetky prvky tejto diagonály sú vopred určené a nič naviac sme nedostali. Pokúsme sa teda postupne vypĺňať zvyšné políčka. Začať môžeme od najmenších možných štvorcov veľkosti 2×2. V tomto prípade máme určené dve políčka na diagonále a doplniť treba zvyšné dve. Vieme, že súčet týchto prvkov musí

byť párny (čo je len inak vyjadrená podmienka párneho počtu jednotiek), čísla z diagonály nám teda určujú či sú zvyšné dve čísla rovnaké alebo rozdielne.

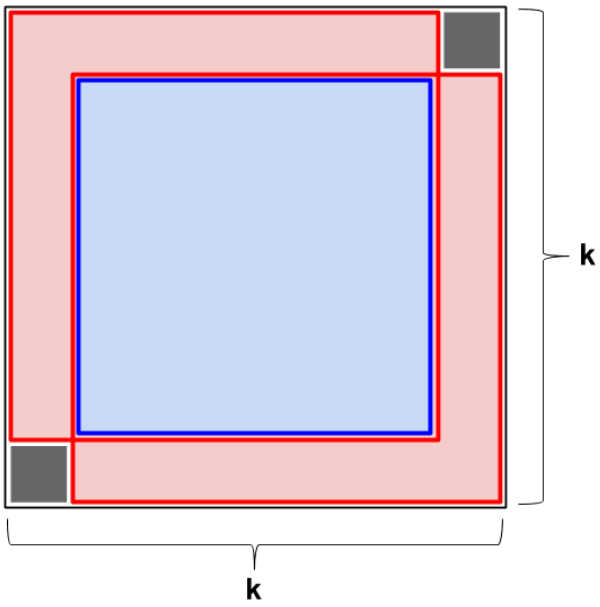
Predstavme si, že súčet dvoch zadaných čísel na diagonále je párny. Potom aj zvyšné dve čísla musia mať párny súčet – musia byť rovnaké. Naopak, ak čísla na diagonále dávajú nepárny súčet, sú rôzne, musia byť rôzne aj zvyšné dve čísla. V oboch prípadoch máme dve možnosti, ako tieto hodnoty doplniť. V párnom prípade to bude buď $0-0$ alebo $1-1$, v nepárnom buď $0-1$ alebo $1-0$ (na poradí záleží, keďže sú to rôzne políčka matice).

Našťastie, je jedno, ktorú možnosť si zvolíme, pri zvyšných štvorcoch to nemôže zavážiť. Môžete si totiž všimnúť, že ľubovoľný diagonálny štvorec, ktorý obsahuje jedno z týchto dvoch políčok musí nutne obsahovať aj druhé. No a ich súčet je určený diagonálnym štvorcom 2×2 a je jedno, ktorými hodnotami sa k tomuto súčtu dopracujeme. Do výsledku si teda môžeme poznačiť, že sme získali dve rôzne možnosti a pokračovať ďalej s ľubovoľnou z nich.

Po doplnení štvorcov 2×2 sa presunieme na štvorce 3×3. Opäť nám na doplnenie ostávajú už len dve hodnoty – rohy neležiace na hlavnej diagonále. Zvyšné políčka sú totiž zahrnuté v niektorom menšom diagonálnom štvorci. Zo súčtu už vyplnených políčok opäť zistíme, či má byť súčet týchto políčok párny alebo nepárny, teda či sú v nich čísla rovnaké alebo rôzne. Z toho opäť vyplývajú dve možné priradenia hodnôt, ktoré sú však ekvivalentné a je jedno, ktoré z nich si zvolíme, stačí že si zapamätáme, že sme mali dve možnosti na výber.

Vieme si však všimnúť zaujímavú vec. V skutočnosti je parita súčtu dvoch nedoplnených políčok rovnaká ako parita stredného políčka štvorca 3×3. Prečo? Všimnime si, že všetky už doplnené políčka sa dajú rozdeliť do dvoch diagonálnych štvorcov veľkosti 2×2, ktoré sa prekrývajú v strednom políčku, označme si ho x. Súčet prvkov všetkých týchto políčok sa rovná súčtu oboch štvorcov 2×2 mínus x, ktoré sme započítali dvakrát. Ale súčet prvkov v štvorci 2×2 musí byť párny. To znamená, že výsledný súčet musí mať rovnakú paritu ako hodnota x. Rohy štvorca 3×3 teda vieme doplniť iba na základe streného čísla, ktoré sa nachádza na hlavnej diagonále.

Pokúsme sa doplniť prvky v diagonálnych štvorcoch $k \times k$, pričom $k > 3$ a všetky menšie štvorce sú úspešne vyplnené. Opäť vidíme, že nevyplnené ostávajú iba dve políčka v rohoch tohto štvorca. Naviac, vyplnené políčka tvoria dva diagonálne štvorce veľkosti $(k-1) \times (k-1)$, ktoré sa prekrývajú v diagonálnom štvorci veľkosti $(k-2) \times (k-2)$. Súčet týchto prvkov vieme teda vypočítať aj tak, že sčítame súčet štvorcov s hranou $k-1$ a odčítame súčet štvorca s hranou $(k-2)$. Súčty všetkých týchto štvorcov musia byť však podľa zadania párne. To znamená, že aj tento výsledok bude párny. Z toho vyplýva, že hodnoty v nediagonálnych rohoch štvorca $k \times k$ **musia byť rovnaké** a navyše vôbec nezávisia od hodnôt na diagonále ani ničom inom.



Sivé políčka ešte nie sú vyplnené. Červené aj modré štvorce musia obsahovať párny počet 1. Z toho vyplýva, že počet 1 v celom štvorci s výnimkou sivých políčok je párny. Sivé políčka teda musia obsahovať rovnakú hodnotu.

Tento prístup nefunguje pre štvorce 3×3 a 2×2, pretože menšie štvorce, z ktorých sa skladajú, nie sú veľkosti aspoň 2×2, a tým pádom pre ne neplatí predpoklad o párnosti ich súčtu.

Z vyššie uvedených tvrdení vidíme, že v okamihu keď máme vyplnenú diagonálu, riešenie je jednoduché. Políčka symetrické cez diagonálu sú od seba navzájom závislé, v prípade, že sú od diagonály príliš vzidalené musia mať rovnakú hodnotu, inak sú hodnotami na diagonále mierne ovplyvnené. V každom prípade však dostaneme dve možnosti ich hodnôt, a preto výsledok musí byť $2^{\frac{n^2-n}{2}}$ – mimo diagonály je n^2-n políčok a ich hodnoty určujeme po dvojiciach.

Toto číslo vieme vypočítať metódou rýchleho umocňovania v čase $O(\log n)$. Myšlienka tohto algoritmu je rekurzívna. Namiesto toho aby sme mocninu počítali postupným násobením číslom 2, počítame hodnotu 2^a tak, že rekurzívne spočítame číslo $2^{a/2}$, ktoré potom umocníme na druhú. Namiesto postupného počítania všetkých hodnôt $2, 2^2, 2^3, 2^4 \dots 2^{a-2}, 2^{a-1}, 2^a$ budeme počítať iba hodnoty $2^a, 2^{a/2}, 2^{a/4} \dots 2^2, 2$, ktorých je výrazne menej, iba logaritmicky veľa.

Vopred určené hodnoty

V druhej sade sme okrem celej diagonály mali vyplnené aj nejaké ďalšie políčka. Ako to ovplyvní výsledok? Obmedzí to naše možnosti. Všetky možné doplnenia sa totiž vyskytovali v dvojiciach, napríklad sme mohli zistiť, že konkrétne dve políčka musia byť rovnaké, teda buď $0-0$ alebo $1-1$. Ak však máme určené, že jedno z týchto políčok má hodnotu 0, tak nám ostáva iba jedna možnosť. Navyše, ak by sme v takomto prípade zistili, že obe políčka sú predvyplnené rozdielnymi hodnotami, vieme, že maticu nevieme doplniť žiadnym spôsobom.

V tomto prípade musíme rozlišovať dva prípady podľa vzdialenosti zadaného políčka od hlavnej diagonály. Pre políčka, ktoré sú priďaleko, aspoň vo vzdialenosti 3 od diagonály nám stačí overiť, či nemáme zadané aj pre ne symetrické políčko s opačnom hodnotou. V takom prípade bude teda možných doplnený 0, inak započítame iba jednu, vstupom určenú, možnosť pre túto dvojicu políčok. Najjednoduchšie riešenie tohto problému je použitím binárneho vyhľadávacieho stromu (`set` v C++), ktoré ale vedia k riešeniu s časovou zložitosťou $O(m \log m)$. Toho logaritmu sa však vieme zbaviť, či už použitím hash mapy alebo nejakého rýchlejšieho triedenia, napr. radix sort. Pri programovaní však takýto malý rozdiel zvyčajne nepotrebujeme riešiť, keďže set je skutočne rýchly.

Zadané políčka v blízkosti diagonály sú o niečo dôležitejšie, sú totiž ovplyvnené hodnotami na diagonále a nevieme dopredu povedať, či majú byť symetrické políčka rozdielne alebo rovnaké. Potrebujeme ich teda prejsť samostatne. Navyše, s týmito hodnotami budeme musieť oveľa viac pracovať aj v prípade, že diagonála nie je určená, preto si ich treba vhodne zapamätať. Pomôcť nám môže buď hash mapa, ale takisto sa dá použiť klasické pole, keďže si potrebujeme pamätať iba 5 diagonál – hlavnú a po dvoch susedných na oboch strán.

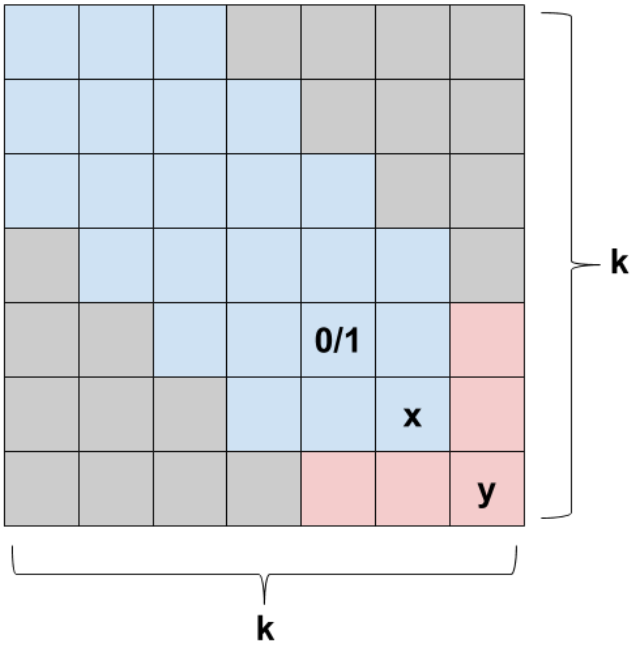
Určenie hodnôt na diagonále

Zatiaľ sme riešili iba problém, v ktorom boli hodnoty na diagonále vopred určené. Toto však nie je všeobecný prípad. Ako si s ním teda poradíme?

Nuž, budeme si ju musieť určiť. Samozrejme, mohli by sme ju určovať celú naraz, v takom prípade by sme museli prejsť postupne každú z 2^n možností. Uvedomme si však, že na diagonále závisia iba štvorce veľkosti 2×2 a 3×3. To znamená, že hodnoty na spodku diagonály nijak neurčujú, ako majú vyzerať hodnoty na jej vrchu. Presnejšie, políčka na diagonále, ktoré sú od seba vo vzdialenosti viac ako 3 sa už vôbec neovplyvňujú.

Na riešenie teda použijeme dynamické programovanie. Väčšinou keď počítate počet možností, riešením bude nejaká verzia dynamického programovania. Stav nášho dynamického programovania bude $D_{k,x,y}$ – koľkými možnosťami vieme vyplniť diagonálne štvorce veľkosti 2×2 a 3×3 v štvorci začínajúcom na políčku $(0,0)$ a končiacom na políčku (k,k) ak posledné dve hodnoty na diagonále budú x a y?

Výpočet tejto hodnoty bude závisieť na tom, ako sme vyplnili o jedno menší štvorec, teda od hodnôt $D_{k-1,0,x}$ a $D_{k-1,1,x}$. V tomto prípade nás zaujíma iba to, koľkými možnosťami vieme doplniť políčka $(k-1,k)$, $(k-2,k)$, $(k,k-1)$ a $(k,k-2)$, keďže vyplnenie zvyšných políčok bolo určené hodnotami $D_{k-1,z,x}$.



Sivé políčka nás nezaujímajú, pretože sú od diagonály priďaleko. Aktuálne doplňané políčka sú červené. Modré boli vyplnené v predchádzajúcich iteráciách dynamického programovania a poznáme pre ne počet rôznych možností. Na diagonále sme si určili hodnoty x a y, ktoré ovplyvňujú aktuálne počítané políčka. Pri tretej hodnote vyskúšame obe možnosti, aj 0 aj 1.

Toto overenie je navyše jednoduché, stačí sa pozrieť, či sú určené hodnoty kompatibilné s hodnotami, ktoré máme zadané na vstupe, hodnoty niektorých z týchto políčok totiž už môžu byť doplnené, a vrátiť jednu z hodnôt 0, 1 alebo 2.

Hodnotu jedného stavu dynamického programovania vieme vypočítať v konštantnom čase a všetkých stavov je $4n$. Celková časová zložitosť tohto kroku je preto $O(n)$. Celková časová zložitosť tohto riešenia je $O(n+m)$, pamäťová $O(n+m)$. Pred programovaním je ešte treba poriadne si rozmyslieť okrajové prípady, začiatok dynamického programovania a rozobrať krok samotnej dynamiky. Potom je to však pomerne priamočiare.

Listing programu (C++)

```
#include <cstdio>
#include <iostream>
#include <string>
#include <algorithm>
#include <vector>
#include <cmath>
#include <queue>
#include <set>
#include <map>
#include <stack>
using namespace std;

#define For(i,n) for(int i=0; i<(n); i++)
#define mp(a,b) make_pair((a),(b))
typedef long long ll;
typedef pair<int,int> pii;

ll MOD = 100000000911;
map<pii, int> M;

int vratM(int x, int y) {
    if (M.count({x, y}) == 0)
        return -1;
    return M[{x, y}];
}

ll moznosti_2x2(int k, int x, int y) {
    // hodnoty na diagonale su ine
    if ((vratM(k, k) != -1 && vratM(k, k) != y)
            || (vratM(k-1, k-1) != -1 && vratM(k-1, k-1) != x)) return 0;
    // rohove policka nie su zadane
    if (vratM(k, k-1) == -1 && vratM(k-1, k) == -1) return 2;
    // prave jedno rohove policko je zadane
    if (vratM(k, k-1) == -1 || vratM(k-1, k) == -1) return 1;
    // obe rohove policka su zadane
    return 1 - (x + y + vratM(k, k-1) + vratM(k-1, k))%2;
}

ll moznosti_3x3(int k, int x) {
```

```
    // rohove policka nie su zadane
    if (vratM(k, k-2) == -1 && vratM(k-2, k) == -1) return 2;
    // prave jedno rohove policko je zadane
    if (vratM(k, k-2) == -1 || vratM(k-2, k) == -1) return 1;
    // obe rohove policka su zadane
    return 1 - (x + vratM(k, k-2) + vratM(k-2, k))%2;
}

// vrat hodnotu a^b
ll umocni(ll a, ll b) {
    if (b == 0) return 1;
    if (b%2 == 1) return (a * umocni(a, b-1)) % MOD;
    ll p = umocni(a, b/2);
    return (p * p) % MOD;
}

ll D[100047][2][2];

int main() {
    int n, m;
    scanf("%d%d", &n, &m);
    For(i, m) {
        int x, y, w;
        scanf("%d%d%d", &x, &y, &w);
        M[{x-1, y-1}] = w;
    }
    if (n == 1) {
        if (vratM(0, 0) == -1) printf("2\n");
        else printf("1\n");
        return 0;
    }
    // dynamicke programovanie
    For(x, 2) For(y, 2) D[1][x][y] = moznosti_2x2(1, x, y);
    for (int k = 2; k < n; k++) {
        For(x, 2) For(y, 2) {
            D[k][x][y] = ((D[k-1][0][x] + D[k-1][1][x]) * moznosti_2x2(k, x, y) * moznosti_3x3(k, x)) % MOD;
        }
    }
    // pocet policok mimo 3x3 stvorcov
    ll pocet = (ll)n * n - 5*(n-2) - 4;
    ll jed = 0, dvoj = 0;
    for (auto it = M.begin(); it != M.end(); it++) {
        int x = it->first.first, y = it->first.second, w = it->second;
        // v niektorom 3x3 stvorci
        if (abs(x - y) < 3) continue;
        if (vratM(y, x) == -1) jed++;
        else {
            dvoj++;
            if (w != vratM(y, x)) {
                printf("0\n");
                return 0;
            }
        }
    }
    // spocitanie vysledku
    pocet -= dvoj + 2*jed;
    pocet /= 2;
    ll vysledok = 0;
    For(x, 2) For(y, 2) vysledok += D[n-1][x][y];
    vysledok *= umocni(2, pocet);
    vysledok %= MOD;
    printf("%lld\n", vysledok);
}
```

Mišof (misof@ksp.sk)
(max. 12 b za popis, 8 b za program)

7. Ye ti zima?

Táto úloha mala až prekvapivo jednoduché riešenie. Netreba žiadne logaritmické dátové štruktúry, vystačíme si s obyčajnými prefixovými súčtami a každú otázku zodpovieme v konštantnom čase.

Skôr, než sa do riešenia pustíme, dohodneme sa, že všetky intervaly, ktoré budeme používať, budú polootvorené: ľavý koniec do nich patrí, pravý už nie. Napr. $[0, 3)$ je interval obsahujúci čísla 0, 1 a 2, zatiaľ čo $[4, 5)$ obsahuje len číslo 4.

Základná myšlienka riešenia bude nasledovná: keď máme zistiť počet spôsobov, ako vybrať reťazec 'ksp' zo znakov, ktorých indexy ležia v intervale $[a, b)$, zoberieme počet spôsobov, ako vybrať 'ksp' zo znakov s indexmi v intervale $[0, b)$ a od nich potom odpočítame tie zlé. Zlé spôsoby sú troch typov: buď leží v intervale $[0, a)$ len znak 'k', alebo znaky 'ks', alebo tam leží celé 'ksp'. Nižšie si detailne popíšeme, ako všetky tieto počty v konštantnom čase vypočítať z obyčajných prefixových súčtov.

Počet spôsobov, ako vybrať reťazec r spomedzi prvých i písmen reťazca S si označíme $P_r[i]$.

Pre jednopísmenové reťazce je tento počet veľmi ľahké spočítať: je to jednoducho počet výskytov dotyčného znaku v dotyčnom úseku reťazca S. Napríklad teda platí $P_k[0] = 0$ a $\forall i : P_k[i+1] = P_k[i] + (1 \text{ ak } S[i]=\text{'k'})$. Analogicky spočítame hodnoty P_s a P_p.

Pozrime sa teraz na hodnoty P_{ks}. Zjavne aj tu platí $P_{ks}[0] = 0$, lebo keď nemáme žiadne znaky, nevyberieme žiadne ks. Predstavme si teraz, že postupne zväčšujeme dĺžku prefixu S, z ktorého môžeme vyberať. Keď pribudne nové písmeno, sú dve možnosti. Ak to nie je 's', tak sa počet spôsobov výberu 'ks' nijak nezmení – toto nové písmeno nemáme ako použiť. A ak pribudne 's', zväčší sa počet 'ks' o toľko, koľko rôznych 'k' sme už videli. Dokopy teda dostávame toto: $\forall i : P_{ks}[i+1] = P_{ks}[i] + (P_k[i]$ ak $S[i]$='s').

Rovnakým spôsobom vieme spočítať aj hodnoty P_{sp}. No a úplne na záver celej prípravy si spočítame hodnoty P_{ksp}, pričom znova opakujeme tú istú úvahu: ak pre nejaké i vidíme, že $S[i]$='p', pribudlo nám do $P_{ksp}[i+1]$ oproti $P_{ksp}[i]$ práve toľko nových možností výberu reťazca 'ksp', koľko máme možností na výber 'ks' zo znakov s indexmi menšími ako i.

Ako teraz zodpovieme na otázku, koľkými spôsobmi vieme 'ksp' vybrať z daného úseku $[a, b)$ reťazca S? Hľadanú odpoveď môžeme zapísať v tvare $\alpha - \beta - \gamma - \delta$, kde:

- α je počet výskytov 'ksp' v intervale $[0, b)$.
- β je počet výskytov 'ksp' v intervale $[0, a)$.
- γ je počet výskytov 'ksp' takých, že 'k' leží v intervale $[0, a)$ a 'sp' v intervale $[a, b)$.
- δ je počet výskytov 'ksp' takých, že 'ks' leží v intervale $[0, a)$ a 'p' v intervale $[a, b)$.

Skoro všetky tieto hodnoty už vieme:

- $\alpha = P_{ksp}[b]$
- $\beta = P_{ksp}[a]$
- $\delta = P_{ks}[a] \cdot (P_p[b] - P_p[a])$. Rozmyslite si, prečo je $P_p[b] - P_p[a]$ rovné počtu výskytov 'p' v $[a, b)$.

Jediné, s čím ešte bude trocha práce, je γ. Ale tej práce bude skutočne len trocha, keďže hodnotu γ môžeme tiež určiť podobnou úvahou. Máme $P_k[a]$ možností, ako vybrať 'k' ležiace v intervale $[0, a)$, ostáva nám už len určiť počet spôsobov, ako vybrať 'sp' z intervalu $[a, b)$. No a toto vieme zistiť ako $\zeta - \eta - \theta$, kde:

- ζ je počet výskytov 'sp' v intervale $[0, b)$, teda $\zeta = P_{sp}[b]$
- η je počet výskytov 'sp' v intervale $[0, a)$, teda $\eta = P_{sp}[a]$.
- θ je počet výskytov 'sp' takých, že 's' leží v $[0, a)$ a 'p' leží v $[a, b)$, čiže $\theta = P_s[a] \cdot (P_p[b] - P_p[a])$.

A to už je všetko. V lineárnom čase od dĺžky S sme si predpočítali hodnoty P a následne v konštantnom čase vieme odpovedať na ľubovoľnú otázku. Celková časová zložitosť je teda $O(n + q)$.

Listing programu (C++)

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

int main() {
    int N;
    cin >> N;
    string S;
    cin >> S;

    vector<long long> prefixK(N+1,0), prefixS(N+1,0), prefixP(N+1,0);
    for (int n=0; n<N; ++n) prefixK[n+1] = prefixK[n] + (S[n] == 'k');
    for (int n=0; n<N; ++n) prefixS[n+1] = prefixS[n] + (S[n] == 's');
    for (int n=0; n<N; ++n) prefixP[n+1] = prefixP[n] + (S[n] == 'p');

    vector<long long> prefixKS(N+1,0), prefixSP(N+1,0);
    for (int n=0; n<N; ++n) prefixKS[n+1] = prefixKS[n] + (S[n] == 's')*prefixK[n];
    for (int n=0; n<N; ++n) prefixSP[n+1] = prefixSP[n] + (S[n] == 'p')*prefixS[n];

    vector<long long> prefixKSP(N+1,0);
    for (int n=0; n<N; ++n) prefixKSP[n+1] = prefixKSP[n] + (S[n] == 'p')*prefixKS[n];

    int Q;
    cin >> Q;
    vector<long long> C(Q,0);
    for (int q=0; q<Q; ++q) {
        int x, y;
        cin >> x >> y;
        if (q > 0) { x = (x+C[q-1])%N; y = (y+C[q-1])%N; }
        int a = min(x,y), b = max(x,y)+1;

        long long SP_in_ab = prefixSP[b];
        SP_in_ab -= prefixSP[a];
        SP_in_ab -= prefixS[a] * (prefixP[b] - prefixP[a]);

        C[q] = prefixKSP[b];
        C[q] -= prefixKSP[a];
        C[q] -= prefixKS[a] * (prefixP[b] - prefixP[a]);
        C[q] -= prefixK[a] * SP_in_ab;

        cout << C[q] << "\n";
    }
}
```

8. Pochúťka

ralbo (peto@ksp.sk)
(max. 12 b za popis, 8 b za program)

Prvé pozorovanie, ktoré môže skúseným riešiteľom napadnúť je, že pri riešení úloh, ktoré sa odohrávajú v tabuľke si vieme situáciu zakresliť ako bipartitný graf. V jednej časti (partícii) vrcholy reprezentujú riadky a v druhej časti vrcholy reprezentujú jednotlivé stĺpce. Hrany medzi nimi reprezentujú políčka tabuľky. Týmto spôsobom si môžeme úlohu pretransformovať na nasledovnú: Máme bipartitný graf v ktorom môže byť medzi niektorými dvojicami vrcholov viac hrán (počet hrán reprezentuje počet koláčov v danom políčku). Koľkými spôsobom vieme vybrať hrany (ozdobiť koláče/vyfarbiť hrany) tak aby podgraf tvorený všetkými vrcholmi a vybranými (vyfarbenými) hranami spĺňal podmienku, že každý vrchol má nepárny stupeň?

Ďalšie pozorovanie je, že v rôznych komponentoch môžeme hrany vyfarbovať nezávisle od seba. To znamená, že celkový počet možností akým vyfarbíme hrany je súčinom počtu možností ako vyfarbiť hrany v jednotlivých komponentoch.

Poďme si na chvíľu predstaviť, že nejaký z komponentov je strom. To znamená že neobsahuje cykly. Kedy ho vieme vyfarbiť a ako? Ak je vrchol list tak potrebuje aby z neho išiel nepárny počet vyfarbených hrán. Keďže list má iba jednu hranu tak táto hrana musí byť určite vyfarbená. A toto nám teda jednoznačne určuje, že všetky hrany k listom budú vyfarbené. Čo teda ostatné hrany? Uvažujme nasledovný rekurzívny algoritmus na vyfarbenie hrán v strome: Zoberme si vrchol rekurzívne ofarbime všetky hrany v jeho podstorme. Ak sme vyfarbili párny počet hrán vedúcich z vrcholu musíme vyfarbiť ešte aj hranu k otcovi, aby sme pre tento vrchol dostali nepárny počet zafarbených hrán. Keď sa lepšie pozriete na tento algoritmus zistíte že vďaka invariantom v rekurzií vždy je jednoznačne určené, ktoré hrany v strome budú zafarbené. Môže sa však stať, že zistíme, že koreň má zafarbený iba párny počet hrán a s tým nevieme nič spraviť. (Nevedie totiž od neho hrana k otcovi.) Vďaka tomuto algoritmu však vieme, že v každom strome vieme zafarbiť hrany najviac jedným povoleným spôsobom a aj to, že môžu existovať stromy v ktorých sa to nedá.

Prichádza čas na otázku, kedy sa strom nedá zafarbiť. Každý vrchol v storme má nejakú hĺbku. Zoberme si vrcholy v párnych hĺbkach. (koreň je BUNV v hĺbke 0). Z týchto vrcholov všetky hrany idú do vrcholov v nepárnych hĺbkach. Počet zafarbených hrán z vrcholov v párnych hĺbkach sa preto musí rovnať počtu zafarbených hrán z vrcholov v nepárnych hĺbkach. To znamená že musia mať aj rovnakú paritu. Keďže z každého vrchola vedie nepárny počet zafarbených hrán, tak parita počtu vrcholov v párnej hĺbke musí byť rovná parite počtu vrcholov v nepárnej hĺbke na to aby strom išiel zafarbiť. Ak sú rôzne tak práve vtedy sa stane ten prípad že po jednoznačnom zafarbení hrán v strome zistíme že z koreňu vychádza párny počet zafarbených hrán. Ak sú však parity rovnaké práve tento prípad nastať nemôže.

Tento princíp vieme využiť aj na nestromové bipartitné grafy. Teda ak máme ľubovolný komponent tak sa bude dať zafarbiť práve vtedy keď má počet vrcholov na jednej jeho strane a na druhej strane rovnakú paritu. Toto sa dá ľahko dokázať. Ak má počet vrcholov na jednotlivých stranách rôznu paritu a z každého vrcholu vychádza nepárny počet zafarbených hrán potom nám nesedí celkový počet zafarbených hrán z týchto strán(partícii) grafu. Zároveň ak má počet vrcholov rovnakú paritu tak si môžeme z komponentu zobrať jeho kostru. Podľa predchádzajúceho odseku vieme, že táto kostra (strom) sa dá zafarbiť jednoznačným spôsobom a preto existuje aspoň jedno zafarbenie tohoto bipartitného grafu, tak aby boli splnené podmienky zo zadania.

Myšlienka s kostrou je kľúčom k najdôležitejšiemu pozorovaniu úlohy. Uvažujme komponent pre ktorý existuje zafarbenie. Vyberme si z neho ľubovolnú kostru. Všetky ostatné hrany môžeme zafarbiť úplne ľubovolne a stále bude existovať zafarbenie kostry, také aby nám sedela parita v každom vrchole. Ako toto dokážeme? Konštrukciou. Nech všetko v komponente okrem kostry je zafarbené ľubovolne. Teraz poďme zafarbiť hrany v kostre. Opätovne postupujme rekurzívnym prehľadávaním do hĺbky. Keď sa dostaneme k nejakému vrcholu tak najprv zistíme, ako majú byť zafarbené hrany v jeho podstrome. Potom zafarbíme poslednú hranu, ktorá sa ho týka (tú ktorá vedie k jeho otcovi) a to iba v prípade, že pred jej zafarbením by mal vrchol párny počet zafarbených hrán. Takto vieme zafarbiť celý strom okrem koreňu jednoznačným spôsobom. To, že bude mať správnu paritu počtu zafarbených hrán ukážeme nasledovne: Vieme, že počet vrcholov v komponente na jednotlivých stranách (partíciach) má rovnakú paritu. Preto je ich súčet párny. Predstavme si, že by zo všetkých vrcholov vychádzal nepárny počet hrán okrem koreňu z ktorého vychádza párny počet hrán. Potom z vrcholov okrem koreňu celkovo vychádza nepárny počet hrán a z koreňa párny. Celkový súčet počtu vychádzajúcich hrán z vrcholov je nepárny. To je ale v spore s tým že každá hrana vychádza dva krát - raz z každého vrcholu. Preto aj z koreňu musí vychádzať nepárny počet zafarbených hrán.

Ako zistíme počet zafarbení nejakého komponentu? Je to 0, keď je počet vrcholov v komponente nepárny. (Ak je párny tak počet vrcholov na jednotlivých stranách má rovnakú paritu). A je to 2^k keď je počet vrcholov v komponente párny, kde k je počet hrán ktoré nie sú v kostre. Každú z nich totiž vieme zafarbiť jedným z dvoch spôsobov. Aký je teda celkový počet možných zafarbení? Je to súčin počtu zafarbení jednotlivých komponentov. Ak je teda nejaký komponent nezafarbiteľný tak je výsledok nula. Inak je to súčin mocnín dvojky, teda súčet exponentov. Takže by sme mohli povedať, že je to 2^{n+m-l}, kde $n+m$ je počet hrán a l

je počet hrán v kostrách jednotlivých komponentov. Všimnime si, že tento počet hrán môžeme zase vypočítať nasledovne. Nech u je počet komponentov. Ak si spojíme kostry, pričom použijeme $u-1$ hrán, tak dostaneme jednu megakostru ktorá bude mať o jednu hranu menej ako je počet vrcholov. [10] Preto je teda celkový počet zafarbení $2^{n+m-((r+s-1)-(u-1))} = 2^{n+m+u-r-s}$. Vypočítať zvyšok tohoto čísla po delení iným (malým číslom) vieme jednoducho cez známy algoritmus v logaritmickom čase od veľkosti tohoto čísla.

Zostala posledná časť úlohy: Zistiť počet komponentov. O čo by to bolo jednoduchšie keby nám v zadaní nenadiktovali dodatočné hrany...

Zoberme si teda ľubovoľný takýto graf. Bez ujmy na všeobecnosti nech $r > s$. Potom doňho pridáme aspoň $n >= r+s$ hrán. Pozrime sa na tieto hrany podrobnejšie. Prvých r hrán nám spojí každý riadok so stĺpcom, ktorý ma číslo rovnaké, ako číslo riadku modulo počet stĺpcov. Týmto nám v grafe zostane už iba s komponentov. Teraz pridáme ešte s hrán. Každá spoji riadok s číslom i (ktorý je už spojený so stĺpcom číslo i) s riadkom s číslom $i+r \pmod r$. Nech sú všetky komponenty definované číslom stĺpca ktorý sa v nich nachádza po prvých r hranách. Potom týchto s ďalších hrán nám spojí pre všetky i stĺpec i so stĺpcom $i+r \pmod s$. Poďme sa pozrieť na situáciu po pridaní $r+s$ hrán.

Máme s stĺpcov (riadky už môžeme ignorovať lebo ku každému máme už jednoznačne priradený stĺpec, ktorý ho zastupuje. (Riadok i zastupuje stlpec $(i \pmod s)$). Vieme nasledovné: Stĺpec i je spojený so stĺpcom $i+r \pmod s$, žiadne ďalšie dodatočné hrany momentálne nemáme. Teda predstavme si to ako situáciu, kde mame nasledovný graf: Každý stĺpec predstavuje jeden vrchol. Stĺpec i je spojený so stĺpcom $i+r \pmod s$ a so stĺpcom $i-r \mod s$.

Tvrdím nasledovné: Stĺpec a je v jednom komponente so stĺpcom b práve vtedy, keď $\gcd(r,s)$ [11] delí $a-b$.

Dôkaz: Ak existuje cesta z a do b tak prechádza nejakými vrcholmi. Všimnime si, že dvojice susediacich vrcholov majú rovnaký zvyšok po delení $\gcd(r,s)$. Preto počas celej cesty prechádzame vrcholmi s rovnakým zvyškom po delení $\gcd(r,s)$. Ak teda existuje cesta z a do b tak vtedy musia mať a aj b rovnaký zvyšok po delení $\gcd(r,s)$.

Podobne môžeme dokázať, ze ak $a-b$ delí $\gcd(r,s)$ tak existuje cesta z a do b: Z a sa vieme dostať na najviac $\frac{s}{\gcd(r,s)}$ vrcholov vrátane a. To je preto, že toľko vrcholov má rovnaký zvyšok po delení $\gcd(r,s)$ ako a. Vieme sa odtiaľ dostať napríklad na vrcholy $(a \mod s)$, $(a+r \mod s)$, $(a+2r \mod s)$, $(a+3r \mod s)$... Všimnite si, ze napriek tomu že táto postupnosť je nekonečne dlhá, môže obsahovať len konečne veľa čísel. Preto sa niekedy stane že bude obsahovať dva krát to isté číslo a teda sa nutne niekde zacyklí. Zoberme si teda jej najkratší cyklus a zistime aký je dlhý. Pre nejaké k a j nutne platí že $a+kr \equiv a+jr \pmod s$. Potom ale $(k-j)r \equiv 0 \pmod s$. Počet riadkov r si vieme rozpísať ako súčin $r = x \cdot \gcd(r,s)$, kde x je nesúdeliteľné s s. Potom dostaneme $(k-j)r \equiv (k-j) \cdot x \gcd(r,s) \equiv 0 \pmod s$. Keďže x je nesúdeliteľné nijako nám neovplyvňuje deliteľnosť s. Preto $(k-j)\gcd(r,s) \equiv 0 \pmod s$, no nato aby bolo nejaké číslo deliteľné s tak musí byť buď 0 alebo aspoň s. Preto $(k-j)\gcd(r,s) \geq s$, teda $k-j >= s/\gcd(r,s)$ a tým pádom má najkratší cyklus dĺžku aspoň $s/\gcd(r,s)$. No lenže všimnite si, že už vieme, že v komponente s a bude najviac $s/\gcd(r,s)$ vrcholov. Preto v komponente s a musia byť práve tie vrcholy ktoré majú rovnaký zvyšok po delení $s/\gcd(r,s)$.

Pozrime sa ešte na to, čo sa stane ak je $n > r+s$. Všimnite si, že sa vzniknú hrany medzi $(i \pmod r \pmod s)$ a $i \pmod s$. Tieto čísla však zaručene majú rovnaký zvyšok po delení $\gcd(r,s)$. Preto sa tým už žiadne ďalšie komponenty nespoja.

Tým pádom po pridaní n hrán, budú dva stĺpce v jednom komponente práve vtedy keď $\gcd(r,s)$ delí ich vzdialenosť.

Všimnite si že takto dostaneme $gcd(r,s)$ komponentov a každý z nich je rovnako veľký, takže si ľahko spočítame ich veľkosť a z toho paritu.

Teraz už zostáva len posledná otázka – ako si zistiť koľko komponentov vlastne máme, keď ich dodatočne pospájame m hranami. Na tento účel vieme použiť Union-Find, no nebude úplne jednoduché keďže si možno nebudeme vedieť zapamätať všetky vrcholy. Preto spravíme pár drobných zmien. V prvom rade si budeme hlavné vrcholy každého komponentu pamätať v mape namiesto poľa. (Tým pádom si nemusíme pamätať všetky vrcholy, stačia iba tie kde nastala zmena.) Taktiež si budeme pamätať celkový počet komponentov a počet komponentov s nepárnym počtom vrcholov. Okrem toho si zapamätáme o každom komponente či ma párny alebo nepárny počet hrán. Pomocou Union Findu budeme vždy vedieť ľahko zistiť, či hrana spája dva komponenty, ktoré ešte spojené neboli a ak áno, tak budeme vedieť, určiť ktoré to sú. Tým pádom budeme vedieť aktualizovať paritu počtu ich vrcholov a tiež počet komponentov s nepárnym počtov vrcholov. Tým pádom nakoniec budeme vedieť či je komponent s nepárnym počtom vrcholov a koľko komponentov je celkovo v grafe, z čoho budeme vedieť vyrátať odpoveď.

Union find bude potrebovať $O(mlog(r+s))$ krokov vzhľadom k tomu že bude uložený v mape. Umocňovanie

[10] Toto vieme z vlastností stromov

[11] *Greatest Common Divisor*, po Slovensky najväčší spoločný deliteľ

spravíme na rádovo $\log(n+m)$ krokov. Najväčší spoločný deliteľ vieme pomocou Euklidovho algoritmu zrátať radovo v čase $O(\log(r+s))$. Celková časová zložitosť je teda $O(mlog^2(r+s)+log(n+m))$.

Listing programu (C++)

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

#define MOD 1000000009

int exp2 (int e) {
  /** 2 to the power of <e>, modulo MOD. */
  if (e == 0) {
    return 1;
  }
  long long sub = exp2(e/2);
  int res = ((sub*sub) * (e%2 ? 2ll : 1ll)) % MOD;
  return res;
}

int gcd (int a, int b) {
  /** Greatest common divisor of <a> and <b>. */
  if (a < b) {
    swap(a, b);
  }
  while (b != 0) {
    a %= b;
    swap(a, b);
  }
  return a;
}

struct DFU {
  /** Disjoint find union structure for arbitrarily large graphs. */
  map<int, int> parent, rank;
  map<int, bool> parity;

  int root_of (int v) {
    /** Returns the ultimate ancestor of <v>. Also compresses the path
     * from <v> to that ancestor. */
    if (parent.count(v) == 0) {
      return v;
    }
    int par = parent[v];
    int res = root_of(par);
    parent[v] = res;
    return res;
  }
  bool same_root (int u, int v) {
    /** Returns true if <u> and <v> are in the same component, false
     * otherwise. */
    return root_of(u) == root_of(v);
  }
  bool parity_of (int v) {
    /** Returns the parity of <u> (the parity of the size of the component
     * of <u>). */
    v = root_of(v);
    if (parity.count(v)) {
      return parity[v];
    }
    return 1;
  }

  int rank_of (int v) {
    /** Returns the rank of vertex <v>. */
    if (rank.count(v) == 0) {
      return 0;
    }
    return rank[v];
  }
  void inc_rank (int v) {
    /** Increments rank of <v> by 1. */
    if (rank.count(v) == 0) {
      rank[v] = 0;
    }
    rank[v] += 1;
  }

  void join (int u, int v) {
    /** Joins two vertices <u> and <v>. */
    u = root_of(u);
    v = root_of(v);
    if (rank_of(u) == rank_of(v)) {
      inc_rank(v);
    }
    else
    if (rank_of(u) > rank_of(v)) {
      swap(u, v);
    }
    int temp = (parity_of(u) + parity_of(v)) % 2; // for WTF_C++?? reasons, one cannot put the right hand size immediately into t
    parity[v] = temp;
    parent[u] = v;
  }
};
```

```
struct Grid {
  int r, s;

  // stuff related to the dimensionality of the vector space
  int dim, dim2;
  int num_comps, num_taken;
  DFU comps;

  void load () {
    /** Loads all input data: grid dimensions, number of default cakes
     * and number of additional cakes. Then loads all the bonus cakes. */
    int n, m;
    cin >> r >> s >> n >> m;
    bool flipped = (r < s);
    if (flipped) {
      swap(r, s);
    }
    num_comps = gcd(r, s);
    num_taken = s - num_comps;
    dim = r + num_taken;
    dim2 = n - dim;

    // load all the bonus cakes
    vector<int> rows(m), cols(m);
    for (int i = 0; i < m; i++) {
      cin >> rows[i];
    }
    for (int i = 0; i < m; i++) {
      cin >> cols[i];
    }
    if (flipped) {
      swap(rows, cols);
    }
    // process them
    for (int i = 0; i < m; i++) {
      int ri = rows[i];
      int si = cols[i];
      int c_ri = ri % num_comps;
      int c_si = si % num_comps;
      if (comps.same_root(c_ri, c_si)) {
        dim2 += 1;
      }
      else {
        dim += 1;
        comps.join(c_ri, c_si);
      }
    }
  }
  bool can_generate_ones () {
    /** Returns true if we can generate the configuration with all rows
     * and columns containing odd number of decorated cakes, false otherwise. */
    if (r%2 != s%2) {
      return false;
    }
    if (dim == r+s-1) {
      return true;
    }
    int comp_size = (r+s) / num_comps;
    if (comp_size % 2 == 0) {
      return true;
    }
    for (int i = 0; i < num_comps; i++) {
      if (comps.parity_of(i)) {
        return false;
      }
    }
    return true;
  }
  void solve () {
    if (can_generate_ones()) {
      cout << exp2(dim2) << "\n";
      return;
    }
    cout << "0\n";
  }
};

int main () {
  Grid G;
  G.load();
  G.solve();
  return 0;
}
```