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

Vzorové riešenia 2. kola zimnej časti

1. Terms and conditions

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

Riešenie tejto úlohy rozdelíme na dve časti.

Načítavanie vstupu

Prvou komplikovanejšou úlohou je správne načítať vstup. Ak používate C++, pravdepodobne na načítavanie vstupu používate `cin`. Ten však načíta iba časť textu po najbližší tzv. 'whitespace' znak, teda medzeru. Preto sa na načítanie vstupu doporučuje použiť príkaz `getline(cin, string)`. Tento načíta všetky znaky až po prvý znak nového riadku (`'\n'`). Ešte si treba dať pozor, že keď zo vstupu načítame počet riadkov n ako číslo, tak za ním ostane znak konca riadku. Tento znak príkaz `getline` zachytí, a teda načíta prázdny string. Treba teda `getline` zavolať ešte raz po načítaní n.

Ak používate napríklad Python, tak príkaz `input()`, ktorý bežne používate na načítanie vstupu, načíta celý riadok, takže tu problém nie je. Rovnako by nemal byť problém v ostatných jazykoch, ktoré testovač podporuje.

Hľadanie políčka

Teraz, keď máme text v pamäti, môžeme začať hľadať správne políčko. Hľadanie políčka môžeme vždy začínať od jeho ľavého horného rohu, stačí skontrolovať všetkých 9 znakov, či sa zhodujú – buď manúalne overíme každý znak, alebo si vytvoríme vlastnú kópiu políčka, ktoré hľadáme, a overíme či sa všetky znaky zhodujú v štvorčeku $3x3$ pomocou dvoch cyklov.

Treba si pritom dávať pozor na to, aby sme neskúsili pozrieť na neexistujúci index niektorého reťazca, keďže susedné riadky môžu byť rôzne dlhé.

Čo všetko si potrebujeme pamätať?

Keď sa zamyslíme nad tým ako hľadáme políčka, uvedomíme si, že keď skontrolujeme trojicu riadkov, údaje z prvého z nich už nikdy nebudeme potrebovať. Preto nám stačí pamätať si iba tri riadky, v ktorých aktuálne hľadáme políčko, teda pamäťová zložitosť tohto riešenia je $O(m)$, kde m označuje počet znakov v jednom riadku.

Zhrnutie

Postupne načítame riadky. Pre každý načítaný znak skontrolujeme či sa "pod ním" nenachádza políčko. Ak sa nachádza políčko zaškrtneme, inak hľadáme ďalej. Takýmto postupom spravíme pre každý znak iba konštantný počet operácií. Teda časová zložitosť je lineárna od veľkosti vstupu $O(n \cdot m)$

Listing programu (C++)

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

using namespace std;

string empty_line;
string line[3];
string look_at[] = {"+-+", "|␣|", "+-+"};

bool success = false, failure = false;

int n;

//funkcia kontrolujuca ci sa na policku i vobec moze nachadzat policko
bool make_sense(int i){
        if(i+2 < line[0].size() && i+2 < line[1].size() && i+2 < line[2].size()) return true;
        return false;
}

int main(){
        //najprv nacitame pocet riadkov a odstranime medzeru co ostala za tymto cislom
        cin >> n;
        getline(cin, empty_line);
```

```
        //nacitame prve dva riadky
        getline(cin, line[0]);
        getline(cin, line[1]);
        //rovno vypyseme prvy riadok
        cout << line[0] << '\n';

        for (int i = 0; i < n-2; i++) {
                //nacitame postupne vsetky riadky
                getline(cin, line[2]);
                //skontrolujeme ci sa na i-tej pozicii na riadku nenachadza policko
                for (int i = 0; make_sense(i) && !success; i++) {
                        failure = false;
                        for (int j = 0; j < 3 && !failure; j++) {
                                for (int k = 0; k < 3 && !failure; k++) {
                                        if(line[j][i+k] != look_at[j][k]) {
                                                failure = true;
                                        }
                                }
                        }
                        //ak najdeme policko zaskrtneme ho
                        if(!failure) {
                                line[1][i+1] = 'X';
                                success = true;
                        }
                }

                //vypiseme riadok v ktorom uz urcite nenastane zmena
                cout << line[1] << '\n';
                //presunieme si este relevantne riadky
                swap(line[0], line[1]);
                swap(line[1], line[2]);
        }

        //vypiseme posledny riadok
        cout << line[1] << '\n';
}
```

Marek

2. Iregulárna strava

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

Zadanie úlohy hovorí, že na vstupe máme dva reťazce a našou úlohou je zistiť, či spĺňajú zadané podmienky. Podmienky vieme zhrnúť tak, že každým rovnakým (malým) znakom prvého slova majú zodpovedať rovnaké (veľké) znaky – obrazy – v druhom slove. Odlišné znaky prvého slova majú mať odlišné obrazy v druhom.

Funkčné riešenie

V najjednoduchšom riešení stačí, ak prejdeme cez všetky dvojice znakov pôvodného slova a overíme, či sú splnené podmienky zo zadania.

V prvom rade skontrolujeme, či sú slová rovnakej dĺžky. Ak nie sú, vieme, že nejaký znak prvého slova nemá žiaden obraz v druhom slove, alebo naopak, a tak vieme hneď povedať, že odpoveď je "nie".

Teraz prichádza na rad hlavná časť riešenia. Povedzme, že oba reťazce A a B dĺžky n máme načítané v pamäti a prechádzame obe slová po znakoch. Cieľom je overiť, či všetky znaky spĺňajú určené podmienky zo zadania. Pre každú i-tu dvojicu znakov A_i a B_i prechádzame ďalších $n-i$ dvojíc A_j, B_j. Pri tomto prechádzaní môžu nastať 4 situácie:

- A_j, B_j sa zhodujú s A_i, B_i – vtedy je všetko v poriadku. $A_i = A_j$ a preto sú rovnaké aj ich obrazy $B_i = B_j$.
- $A_i \neq A_j$ a $B_i \neq B_j$ – vtedy je tiež všetko v poriadku. Pôvodné písmená sú rozdielne a tak aj ich obrazy sú rozdielne.
- $A_i = A_j$ a $B_i \neq B_j$ (alebo opačne $A_i \neq A_j$ a $B_i = B_j$) znamená, že, buď máme dve rovnaké písmena v pôvodnom slove, ktoré majú odlišné obrazy, alebo, v druhom prípade, sú to rôzne písmena v pôvodnom slove a majú rovnaké obrazy. Oba tieto prípady znamenajú, že sme našli znak, ktorý nespĺňa podmienku zo zadania a odpoveď je tým pádom "nie".

Ak týmto spôsobom prejdeme celé slovo a nenájdeme žiadnu chybu, podmienky sú splnené pre všetky znaky a teda odpoveď je "áno".

Časová zložitosť tohto riešenia je $O(n^2)$, keďže pre každú z n pozícií i musíme ešte prejsť rádovo n pozícií j.

Pamäťová zložitosť je $O(n)$, lebo si pamätáme dve slová s n písmenami.

Na vstupe ale možeme dostať slovo dĺžky až 2 000 000, čo znamená, že pri vyššie spomenutej časovej zložitosti by riešenie bežalo pomerne dlho...

Substitučná šifra

Podmienky zo zadania vieme ale zjednodušiť a preformulovať tak, že chceme, aby sa každé písmeno malej abecedy (z prvého slova) zašifrovalo na písmeno veľkej abecedy (z druhého slova). Tiež musí platiť, že žiadne dve malé písmená sa nemôžu zašifrovať na to isté veľké.

Uvedomme si, že takéto šifrovanie sa dá celé definovať pomocou **šifrovacej abecedy**, čo je reťazec, v ktorom sa na i-tom mieste nachádza šifrovaný znak (obraz) i-teho písmena klasickej abecedy. Napríklad, použitím šifrovacej abecedy `ANCDEFGHIJKLMBOPQRSTUVWXYZ` vznikne zo slova "abba" slovo "ANNA". Písmeno `a` sa pri tom mení na `A`, `b` na `N`.

Takéto šifrovanie sa bežne nazýva **substitučná šifra**. Našou úlohou je teda zistiť, či šifrovaný text mohol vzniknúť z pôvodného pomocou takejto šifry.

Lepšie riešenie

A práve pomocou šifrovacej abecedy vieme navrhnúť rýchlejšie riešenie! Ak si budeme pamätať šifrovaciu abecedu, tak nemusíme pre každý znak prechádzať celý zvyšok slova, ale stačí nám len overiť, či sú všetky nové znaky zašifrované podľa tej istej abecedy ako predošlé znaky.

Budeme si pamätať dve šifrovacie abecedy, vďaka ktorým budeme vedieť realizovať šifrovanie a dešifrovanie. Jedna abeceda bude šifrovať znaky z pôvodného slova do šifry, a druhá bude inverzná k nej – ak sa podľa šifrovacej abecedy mení `a` na `B`, tak podľa dešifrovacej abecedy sa mení `B` na `a`.

Pre každý i-ty znak z pôvodného slova zistíme, či už máme k nemu priradený nejaký obraz v abecede pre šifrovanie. Ak tam žiaden obraz nie je, znamená to, že takéto písmeno sme objavili prvýkrát. V takomto prípade znaku pridelíme obraz podľa jeho šifry, čiže i-ty znak v druhom slove.

To isté zisťujeme aj pre znaky v druhom, šifrovanom, slove, a vytvárame si dešifrovaciu abecedu.

Ak už i-ty znak má v šifrovacej abecede priradený obraz, overíme, či sa tento obraz zhoduje s i-tym znakom v druhom slove. Ak sa nezhodujú, našli sme chybu v šifrovaní – dva rovnaké znaky sú zašifrované na odlišné obrazy. Tiež musíme overiť, či sa už náhodou nejaký odlišný znak nezašifruje na obraz i-teho. To overíme pomocou doteraz vybudovanej dešifrovacej abecedy.

Pamäťová zložitosť je v podstate rovnaká ako predtým, aj keď si teraz musíme pamätať aj dve abecedy – 2×26 znakov. To znamená $O(2 \times n + 2 \times 26)$, no konštanty môžeme pri odhade zložitostí zanedbať, keďže nás zaujíma iba rádový odhad množstva použitej pamäte. Tým pádom vieme určiť výslednú pamäťovú zložitosť na $O(n)$.

Pri časovej zložitosti získavame výrazné zlepšenie a dostávame sa na lineárnu zložitosť, keďže pre každé písmeno vykonáme iba konštantný počet operácií. Takže $O(n)$.

Listing programu (C++)

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

using namespace std;

const int NEDEFINOVANE = 0;

int main() {
        int t;
        cin >> t;

        for (int tt = 0; tt < t; tt++) {
                // v poliach si pamatame, ktore znaky sa sifruju na ktore
                char sifrovy_predpis[26] = {NEDEFINOVANE}, spatny_predpis[26] = {NEDEFINOVANE};

                string text, sifrovany_text;
                cin >> text;
                cin >> sifrovany_text;

                if (text.size() != sifrovany_text.size()) {
                        cout << "nie" << endl;
                        continue;       // slova maju rozdielne dlzky, ideme na dlasiu dvojicu
                }

                bool korektna_sifra = true;
                for (int i = 0; i < text.size(); i++) {
                        // prvy vyskyt znaku v povodnom slove, ulozime si prislusny obraz
                        if (sifrovy_predpis[ text[i] - 'a' ] == NEDEFINOVANE)
                                sifrovy_predpis[ text[i] - 'a' ] = sifrovany_text[i];

                        // prvy vyskyt sifroveho znaku v sifre, k sifrovemu znaku si ulozime povodny
                        if (spatny_predpis[ sifrovany_text[i] - 'A' ] == NEDEFINOVANE)
                                spatny_predpis[ sifrovany_text[i] - 'A' ] = text[i];

                        // ak i-ty znaku textu nie je zasifrovany podla "sifroveho predpisu"
                        // alebo i-ty znak sifroveho textu nie je obrazom znaku textu, sifra nie je korektna
                        if (sifrovy_predpis[ text[i] - 'a' ] != sifrovany_text[i] || spatny_predpis[ sifrovany_text[i] - 'A' ] !=
                                korektna_sifra = false;
                                break;
```

```
                }
        }

        if (korektna_sifra)
                cout << "ano" << endl;
        else
                cout << "nie" << endl;
    }
}
```

matusf

3. Dohrajte hru

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

Prvá sada (jedna kôpka)

Na riešenie prvej sady, kde bola iba jedna kôpka, stačilo striedavo simulovať Baklažánove a Bujove ťahy. Takéto riešenie pobehá v lineárnom čase (neskôr ukážeme, že vieme mať dokonca konštantnú pamäť).

Riešenie pre viacero kôpok

Tu je situácia podstatne ťažšia. Je veľmi veľa možností ako vie hra prebiehať a nestíhame preskúšať každú. Prvá vec, čo si treba uvedomiť je, že sa nestačí pozerať len na prvú/poslednú kartu na kôpke. Takáto karta môže mať veľmi malú hodnotu, ale môže odkryť karty, ktoré sa veľmi oplatia zobrať.

Ďalšia vec, ktorú si vieme všimnúť je, že porebujeme rozlišovať medzi 2 typmi kôpok – *párnymi* a *nepárnymi*. Zamyslime sa nad párnymi kôpkami.

Kôpky s párnym počtom kariet

Ako tu vyzerá optimálna hra? Všimnime si, že žiaden hráč nemôže získať kartu, ktorá je vo vzdialenejšej polke kôpky. Prečo?

1. Bujovi stačí ťahať z tej istej kôpky, z ktorej bral Baklažán.
2. Baklažánova stratégia je podobná. Ak Buj potiahne kartu z inej kôpky ako on, Baklažán sa prispôsobí a zoberie z tej kôpky, čo Buj.

Čo sme týmto zistili? Predstavme si, že existuje optimálne riešenie pre jedného z hráčov, s tým, že v nejakom momente zoberie kartu zo vzdialenej polky kôpky. My sme ale dokázali, že obaja hráči vedia takémuto prípadu zabrániť. To znamená, že v optimálnom riešení Baklažán získa vrchnú polku kôpky a Buj spodnú.

Kôpky s nepárnym počtom kariet

Teraz sa pozrime na nepárne kôpky. Zjavne aj tu platí tá istá stratégia, akurát, čo s kartou v prostriedku kôpky? Keď popremýšľame, vieme ukázať, že Buj nevie zabrániť Baklažánovi získať najlepšiu strednú kartu spomedzi nepárnych kôpok. Podobne, ak Baklažán túto kartu zoberie, nevie zabrániť Bujovi zobrať druhú najdrahšiu. A toto vieme zovšeobecniť – Baklažán a Buj si postupne poberú karty v nepárnych kôpkach.

Celé riešenie dokopy

Na začiatku načítame vstup, pripočítame Baklažánovi skóre za vrchné polky kôpok, Bujovi za spodné. Potom, zoberieme prostredné karty v nepárnych kôpkach, utriedime ich a striedavo rozdelíme medzi Baklažána a Buja. Časová zložitosť takéhoto riešenia je $O(n \cdot s + n \cdot \log(n))$, pamäťová $O(n \cdot s)$, kde n je počet kôpok a s je maximálny počet kariet v kôpke.

Ide to aj rýchlejšie a úspornejšie

V skutočnosti si vôbec nemusíme pamätať všetky kôpky. Stačí, keď ich budeme spracuvávať jednu po druhej. Párnu rovno spracujeme a z nepárnych si uložíme prostredný prvok. Tie na konci usporiadame a rozdelíme chalanom. Týmto sme zlepšili pamäťovú zložitosť z $O(n \cdot s)$ na $O(n + s)$ lebo máme s kariet v kôpke a môžeme dostať n nepárnych kôp.

Časovú zložitosť vieme zlepšiť usporiadaním prostredných kariet pomocou counting sortu[1], keďže hodnoty jednotlivých kariet sú do 1000. Pretože counting sort triedi v linárnom čase od počtu triedených prvkov, teda v $O(n)$, výsledná časová zložitosť bude $O(n \cdot s)$. Prípadne vieme spraviť tesnejší odhad – zložitosť bude lineárna od počtu všetkých kariet na vstupe. Keďže musíme vždy prečítať celý vstup, takéto riešenie má dokonca najlepšiu možnú asymptotickú zložitosť spomedzi všetkých programov riešiacich túto úlohu.

[1] https://sk.wikipedia.org/wiki/Counting_sort

Ešte jedna finta nakoniec. Vďaka tomu, že vieme koľko kariet má každá kôpka, kôpky si nemusíme pamatať. Vieme ich spracovať postupne po kartách a pamätať si len Bujove a Baklažánove skóre. Pamäťová zložitosť bude potom $O(n)$.

Listing programu (Python)

```
#!/usr/bin/env python3

odd, even = [], []
player1_turn = True
player1 = player2 = 0
pile_number = int(input())

for _ in range(pile_number):
    n, *pile = tuple(map(int, input().split()))
    if n % 2 == 0:
        even.append(pile)
    else:
        odd.append(pile)

for pile in even:
    n = len(pile)
    player1 += sum(pile[:n//2])
    player2 += sum(pile[n//2:])

for pile in sorted(odd, reverse=True, key=lambda x: x[len(x)//2]):
    n = len(pile)
    top, middle, bottom = pile[:n//2], pile[n//2], pile[n//2+1:]
    player1 += sum(top)
    player2 += sum(bottom)
    if player1_turn:
        player1 += middle
        player1_turn = not player1_turn
    else:
        player2 += middle
        player1_turn = not player1_turn

print(player1, player2)
```

4. Energetické pole

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

Hrubá Sila

Na začiatok si preformulujme zadanie úlohy pomocou všeobecnejších pojmov, a potom si ukážeme prvé jednoduché riešenie tejto úlohy.

Máme tabuľku, ktorá bola ofarbená postupnosťou ofarbení jednotlivých stĺpcov a riadkov v nejakom poradí. Platí pritom, že sme vždy ofarbili celý daný stĺpec/riadok tou istou farbou. Našou úlohou je určiť, koľko najmenej takýchto ofarbení potrebujeme použiť na dosiahnutie tabuľky zo vstupu. Všimnime si, že ak jeden riadok či stĺpec ofarbíme druhýkrát, všetka informácia o prvom ofarbení sa stratí. Vďaka tomu môžeme predpokladať, že vo výslednom ofarbení bol každý riadok aj každý stĺpec zafarbený najviac raz.

Tu sa naskytá nápad na jednoduché riešenie: Keďže každý riadok aj stĺpec sme zafarbili najviac raz, ak si zoberieme všetky možné zoradenia stĺpcov a riadkov, tak môžeme v danom poradí skúšať odfarbovať stĺpce a riadky tabuľky. Riadok či stĺpec budeme môcť odfarbiť iba vtedy, ak sú všetky zafarbené políčka v ňom jednej farby. V prípade, kedy sa nám podarí celú tabuľku vyčistiť najskôr, dostaneme optimálne riešenie. V tomto riešení teda hľadamé postupnosť ofarbení riadkov a stĺpcov tak, že pre každú postupnosť odzadu (odfarbovaním) overíme, či vstupná tabuľka mohla vzniknúť touto postupnosťou.

Odfarbenie riadka/stĺpca sa dá jednoducho implementovať tak, že odfarbený riadok/stĺpec z tabuľky odstránime (budeme ho ďalej ignorovať), pretože nám nezáleží na tom, aké farby boli "pod posledným náterom". Namiesto skúšania všetkých permutácií riadkov a stĺpcov by sme tiež mohli myšlienku riešenia implementovať aj mierne efektívnejšie – pomocou backtracku (rekurzívne prehľadávanie s návratom).

Takéto priame a myšlienkovo jednoduché riešenie musí vyskúšať všetky permutácie riadkov a stĺpcov[2], ktorých je $(r+s)!$ a pre každú permutáciu musí overiť, či vedie k tabuľke zo vstupu, napríklad v čase $O(rs)$. Celé riešenie tak bude mať časovú zložitosť $O((r+s)! \cdot rs)$, vďaka čomu na bežnom počítači vyrieši vstupy s $r+s<10$.

Greedy riešenie

Pri opravovaní vašich riešení sme sa dozvedeli, že vďaka tomu, že sa na vstupe nachádzali len korektné zadania (každá ofarbená tabuľka bola výsledkom farbenia riadkov a stĺpcov), sa táto úloha sa dala vyriešiť aj pažravým (greedy) prístupom.

[2] Ak neveríte, porozmýšľajte nad vstupom, kde sú všetky políčka tabuľky jednej farby.

Hlavná myšlienka pažravého riešenia je, že sa pokúsime tabuľku postupne od konca odfarbovať tak, ako v riešení hrubou silou. Ak ale máme na výber z viacero možností čo spraviť (môžeme odfarbiť viacero stĺpcov alebo môžeme odfarbiť riadky aj stĺpce), nebudeme sa rekurzívne rozvetvovať a skúšať všetky možné poradia, ale jednoducho odfrabíme všetky jednofarebné stĺpce alebo riadky jednej farby naraz, skoro vždy v ľubovoľnom poradí.

Jediné špeciálne prípady nastanú vtedy, ak nám zostáva tabuľka, kde sú všetky riadky jednofarebné, všetky stĺpce jednofarebné alebo ak sa po odstránení jednofarebných stĺpcov/riadkov dostaneme do takejto situácie. Vtedy sa nám môže oplatiť odfarbiť všetky stĺpce okrem stĺpcov jednej farby a zvyšné stĺpce odfarbiť pomocou odfarbenia riadkov, ak je riadkov menej ako stĺpcov.

Takéto riešenie sa pomerne ľahko vymyslí a naprogramuje, no tá najťažšia časť pri ňom je dokázať, že vždy vedie k optimálnemu riešeniu. Dôkazom si tiež overíte, že ste pri vymýšľaní postupu nezabudli na žiadne špeciálne prípady. Mnohí z vás stratili body na nepremyslení si špeciálnych prípadov alebo na nedôslednom dôkaze (keďže sme greedy riešeniu spočiatku neverili, bolo nás nutné presviedčať). Rozhodli sme sa teda napísať časť dôkazu, ktorý zachytáva hlavné myšlienky toho, prečo pažravé riešenia môžu fungovať a otázky nad ktorými sa bolo potrebné sa zamyslieť.

Zadefinujme si poriadne všetky situácie, ktoré môžu nastať. Počas celého riešenia (odfarbovania) musí platiť, že sa v tabuľke vždy nachádza aspoň jeden celý riadok alebo stĺpec. Vzhľadom na to, že riadky sa stanú stĺpcami keď tabuľku otočíme o 90 stupňov, budeme hovoriť len o prípadoch, kedy máme aspoň jeden jednofarebný riadok.

1. V tabuľke sú **iba jednofarebné riadky**. Rôzne riadky môžu byť rôznych farieb. Takáto situácia je veľmi blízko k odfarbeniu celej tabuľky – už nám stačí odfarbiť len všetky riadky, prípadne možno riadky poslednej farby odfarbiť po stĺpoch.
2. V tabuľke **je aspoň jeden nejednofarebný riadok**. Tabuľka obsahuje jeden alebo niekoľko jednofarebných riadkov, ktoré sú všetky tej istej farby a neobsahuje žiaden jednofarebný stĺpec. V tomto prípade existuje len jediná cesta ako pokračovať ďalej v riešení – odstránime všetky tieto riadky.
3. V tabuľke je aspoň jeden nejednofarebný riadok. Tabuľka obsahuje jeden alebo niekoľko jednofarebných riadkov, ktoré sú **aspoň dvoch rôznych farieb** a neobsahuje žiaden jednofarebný stĺpec. V tomto prípade máme na výber – riadky ktorej farby odstránime najskôr?
4. V tabuľke je aspoň jeden nejednofarebný riadok. Tabuľka obsahuje aspoň jeden jednofarebný riadok a **aspoň jeden jednofarebný stĺpec**, ktoré sú (musia byť) všetky tej istej farby. Tu máme opäť na výber – odstránime najprv riadky alebo stĺpce?

Môžete si rozmyslieť, že iné situácie (až na triviálne – všetky políčka jednej farby a prázdna tabuľka) už nikdy nenastanú.

Čo robiť v situácii 4? Ak sa odobraním riadka dostaneme do situácie 1 s jednofarebnými stĺpcami, môže sa nám oplatiť teraz neodstrániť stĺpec tejto farby, lebo výhodnejšie môže byť jeho odstránenie na konci, v podobe riadkov. Existuje ešte nejaká iná situácia, kedy neodstrániť aj riadok aj stĺpec (prípadne všetky jednofarebné riadky aj stĺpce)? Nie. Ak po odstránení riadkov nie sme ešte v situácii 1, znamená to, že na doriešenie úlohy je potrebné odoberať ešte stĺpce a potom ešte riadky inej farby. To ale znamená, že naše jednofarebné stĺpce budeme musiť tiež odobrať, a teda ich môžeme odobrať hneď teraz! V skoro všetkých prípadoch je teda jedno, či odstránime najprv riadky, najprv stĺpce alebo všetko naraz.

Čo robiť v situácii 3? Na to, aby sme pohli v riešení ďalej (do inej situácie) musíme buď odobrať všetky jednofarebné riadky (dostanem sa do 1, 2 alebo 3, kde dostaneme jednofarebné stĺpce) alebo odoberieme všetky riadky okrem riadkov jednej farby (dostaneme sa do 2 alebo 4). Úvahu si môžeme zjednodušiť tak, že nikdy neodstránime všetky riadky ale vždy zachováme riadky jednej farby a potom sa budeme rozhodovať ako keby sme boli v situácii 4 (alebo 2). Ak sa potom znova rozhodneme odobrať riadok, dosiahneme rovnaký výsledok ako keby sme rovno odstránili všetky riadky. Poslednou otázkou teda zostáva to, či závisí na tom, ktoré riadky jednej farby ponecháme ako posledné. Na tomto ale záleží len vtedy, ak by sme sa z následnej situácie 4 dostali do 1. Inak v situácii 4 tieto riadky aj tak ostránime a znova teda vo väčšine prípadov nezáleží na tom, ktorú farbu ponecháme!

Konkrétne riešenia špeciálnych prípadov sa dali vyriešiť najjednoduchšie tak, že v každom kroku odsimulujeme pár krokov algoritmu hrubou silou. Dali sa tiež navrhovať konkrétne postupy riešenia jednotlivých špeciálnych prípadov, no tie by si vyžiadali ešte viac dokazovania.

Ďalej nasleduje naše pôvodné vzorové riešenie, ktoré je možno náročnejšie (alebo trikovejšie) na vymyslenie, no jednoduchšie vidno, že je správne.

Optimálna postupnosť farbení a náčrt lepšieho riešenia

Pozrime sa znovu na nejakú postupnosť farieb, ktorá je optimálnym riešením tejto úlohy. Keďže každé políčko tabuľky je nejako zafarbené, tak musel byť zafarbený buď každý stĺpec alebo každý riadok. Predpokladajme,

preto, že počas procesu bol každý riadok aj každý stĺpec zafarbený práve raz. Potom si vieme všetky riadky a stĺpce zoradiť do jednej postupnosti, podľa toho, v akom poradí boli ofarbené. Všimnime si, že prvý stĺpec alebo prvý riadok, ktorý sme ofarbili, na výsledné ofarbenie tabuľky vôbec nevplýva, keďže všetky riadky resp. stĺpce boli neskôr prefarbené. Preto ho v optimálnom riešení nezafarbíme, keďže aj tak nič nemení. Povedzme, že to bol i-ty riadok. Keďže sme ho nezafarbili ani raz, tak podľa neho vieme zistiť, akými farbami boli zafarbené všetky stĺpce a podľa týchto stĺpcov vieme zistiť ako (resp. či) boli zafarbené zvyšné riadky. Riešením úlohy potom bude počet stĺpcov + počet riadkov, ktoré sme zafarbili.

Ideálne riešenie

Keďže vieme, že jeden z riadkov alebo stĺpcov nebol v optimálnom riešení zafarbený, môžeme ich všetky prejsť a pre každý zistiť, či to mohol byť daný riadok/stĺpec. Najprv zistíme, či sú farby riadkov a stĺpcov, ktoré sú implikované týmto riadkom/stĺpcom, konzistenté s tabuľkou vo vstupe. Ak áno, tak vieme, koľko zafarbení by sme museli použiť. Ak je tento počet menší, ako počet prefarbení v doteraz najlpšom riešení (ktoré sme dostali tak, že sme predpokladali, že nezafarbený bol iný riadok/stĺpec), tak sme našli lepšie riešenie, ktoré si uložíme.

Náš program teda bude vyzerať nasledovne. Budeme mať funkciu, ktorej určíme jeden riadok, o ktorom budeme predpokladať, že nebol zafarbený. Táto funkcia nám vráti počet ofarbení riadkov a stĺpcov potrebných na ofarbenie tabuľky. Podľa farieb v určenom riadku určíme farby, ktorými boli zafarbené jednotlivé stĺpce. Potom prejdeme všetky políčka v tabuľke, pričom ak dané políčko nie je farby príslušného stĺpca, priradíme jeho farbu k danému riadku. Ak mal tento riadok už priradenú nejakú farbu, ktorá je odlišná od farby políčka, tak nebude existovať žiadne riešenie, pretože každé políčko môže mať farbu len od stĺpca alebo od riadka. V takomto prípade môžeme vrátiť $r + s$, keďže optimálne riešenie je od $r + s$ ostro menšie. Ak nenájdeme žiaden spor, zrátame počet riadkov, ktorým sme priradili farbu, pripočítame ho ku počtu stĺpcov a toto číslo vrátime.

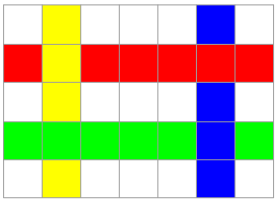
Túto funkciu spustíme postupne na všetkých riadkoch tabuľky, pričom si uložíme minimum čo nám vrátila. Keďže sme leniví a nechce sa nám robiť samostatnú funkciu pre stĺpce, tabuľku následne transponujeme (vymeníme riadky za stĺpce a naopak) a zopakujeme to isté, čo sme spravili predtým.

Časová zložitosť takéhoto riešenia bude $O((r+s)\cdot rs)$ keďže pre každý potenciálne nezafarbený riadok/stĺpec prechádzame celú tabuľku. Jeho pamäťová zložitosť bude $O(rs)$, keďže najväčšie, čo si kedy pamätáme, je samotná tabuľka a tú si potrebujeme pamätať celú.

Nájde náš algoritmus skutočne správne ofarbenie tabuľky?

Tu by bolo na mieste poznamenať, že predpokladáme, že riešenie, ktoré sme našli je naozaj riešením, a že takýmto počtom ofarbení naozaj vieme získať tabuľku zo vstupu. Overili sme to, či sme niektoré riadky neofarbili dvoma farbami. Podobne aj pre stĺpce. No neoverili sme to, či sa tabuľka dá naozaj škontruovať, a či by pri takejto konštrukcii nemohli vzniknúť cykly. Pod cyklom myslíme prípady, kedy by sme museli riadok i zafarbiť skôr ako stĺpec j, lenže zároveň by sme museli stĺpec j zafarbiť skôr ako riadok i, viď. obrázok, kde sa dané zafarbenie nedá skonštruovať:

Žltý stĺpec bol zafarbený pred zeleným riadkom. Ten bol zafarbený pred modrým stĺpcom. Modrý stĺpec musel byť zafarbený pred červeným riadkom, takže sme červený riadok museli zafarbiť po žltom stĺpci. No vidíme, že žltý stĺpec sme museli zafarbiť po červenom riadku, a obe tieto podmienky sa nedajú splniť naraz.



Jeden spôsob, akým by sme tento problém mohli vyriešiť, je detekcia cyklov. Mohli by sme si skonštruovať orientovaný bipartitný graf, v ktorom vedie hrana z riadku do stĺpca vtedy, keď sme riadok ofarbili neskôr a hrana zo stĺpca do riadku, keď sme ofarbili neskôr stĺpec. Následne by nám stačilo overiť, že sa v grafe nevyskytuje cyklus.

Namiesto toho si ale môžeme dokázať, že za predpokladu, že vstupná tabuľka je ofarbiteľná, sa takýto cyklus nebude v žiadnom nájdenom riešení vyskytovať. Rozoberme si dva prípady: prvý, keď sú všetky hrany tohoto cyklu (čiže políčka, ktoré ležia na prienikoch riadkov a stĺpcov cyklu) mimo určeného nezafarbeného riadku; a druhý, keď sa nejaká hrana (políčko) nachádza v tomto riadku.

V prvom prípade sa tento cyklus v tabuľke nachádzal aj predtým, ako sme sa rozhodli, že náš riadok bude vo výslednom riešení ten, ktorý nebol ofarbený. Toto sa ale nemohlo stať, keďže zadanie úlohy garantuje, že pre tabuľku na vstupe existuje nejaké riešenie.

Druhý prípad, kedy sa náš zvolený riadok nachádza v cykle, sa rovnako nemôže stať. O tomto riadku predpokladáme, že sme ho nezafarbili, a teda nebol zafarbený pred ani po žiadnom stĺpci (mohli by sme povedať, že bol zafarbený pred všetkými stĺpcami, tým by sme ale vylúčili akýkoľvek cyklus).

Pri riešení úlohy by ste si mali byť istí, že váš algoritmus produkuje správne riešenia. V tejto úlohe ste teda mali na výber buď implementovať hľadanie cyklov, alebo dokázať, že aj riešenie bez ich hľadania bude vždy fungovať.

Listing programu (Python)

```
def zisti_pocet_ofarbeni(tabulka, nezafarbeny_riadok):
    r = len(tabulka)
    s = len(tabulka[0])

    farby_stlpcov = [tabulka[nezafarbeny_riadok][j] for j in range(s)]
    farby_riadkov = [-1 for i in range(r)]

    for y in range(r):
        for x in range(s):
            if tabulka[y][x] != farby_stlpcov[x]:
                if (farby_riadkov[y] != -1 and farby_riadkov[y] != tabulka[y][x]):
                    return r + s
                farby_riadkov[y] = tabulka[y][x]

    pocet_ofarbeni = s
    for y in range(r):
        if farby_riadkov[y] != -1:
            pocet_ofarbeni += 1

    return pocet_ofarbeni


r, s = map(int, input().split())
tabulka = []
for i in range(r):
    tabulka.append(list(map(int, input().split())))

odpoved = r + s - 1
for i in range(r):
    odpoved = min(odpoved, zisti_pocet_ofarbeni(tabulka, i))

tabulka = [*zip(*tabulka)]  # sikovna implementacia transponovania tabulky
for i in range(s):
    odpoved = min(odpoved, zisti_pocet_ofarbeni(tabulka, i))

print(odpoved)
```

Andrej

5. Prestavba bytu

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

Zadanie tejto úlohy bolo pomerne priamočiare. Máme zadanú štvorcovú mriežku nejakých špeciálnych rozmerov a máme ju celú okrem jedného daného políčka vykachličkovať veľa kachličkami tvaru L.

Riešenie hrubou silou

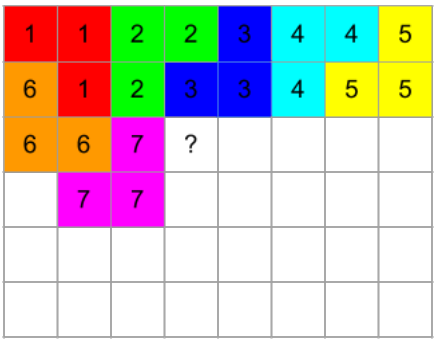
Prvé riešenie, ktoré môžeme vyskúšať pokiaľ nevieme s ničím lepším prísť, je riešenie hrubou silou, teda skúšanie všetkých možností. V tomto prípade ale nie je úplne jasné, ako takéto skúšanie pekne a dobre implementovať. Ako teda v tomto prípade rozumne postupovať?

Začneme tým, že asi najprirodzenejšou reprezentáciou podlahy kúpeľne je klasické dvojrozmerné pole. To chceme vyplniť číslami kachličiek a následne ho vypísať na výstup. Ďalej vieme už vopred povedať, koľko kachličiek použijeme: stačí si vypočítať počet políčok, odrátať jedno za odtok a vydeliť tromi. Ostáva teda už len zistiť, ako majú byť jednotlivé kachličky otočené a rozmiestnené.

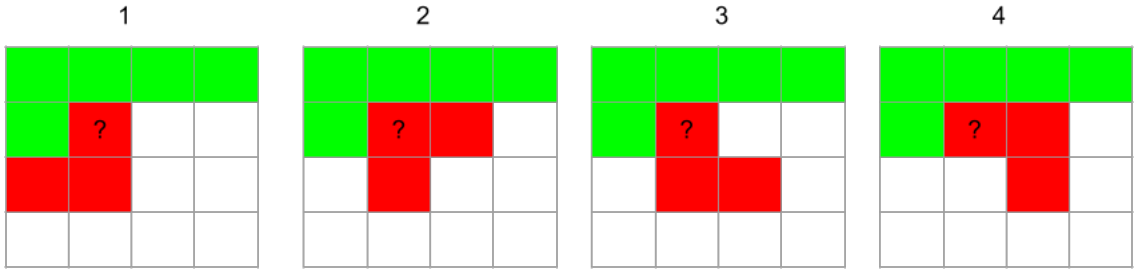
Chceli by sme teraz vymyslieť nejaký systematický postup skúšania všetkých možností. Vezmime si nejaké políčko, ktoré ešte nie je pokryté kachličkou (a ani odtokom). Toto políčko musíme nejako pokryť. Kedže máme iba malý (a hlavne konečný) počet možností, ako to spraviť, môžeme postupne vyskúšať všetky. Presnejšie, budeme postupovať nasledovne: vždy si vyberieme nejakú možnosť, ako toto políčko pokryť, tú si zaznačíme do poľa a rekurzívne sa zavoláme ("vyskúšaj všetky možnosti ako dokončiť toto aktuálne kachličkovanie"). Ak sa z rekurzie vrátime s tým, že žiadne riešenie sme nenašli, odstránime z poľa zaznačený spôsob pokrytia aktuálneho políčka a prejdeme na ďalšiu možnosť, ako ho pokryť.

Naše konkrétne riešenie bude prechádzať pole systematicky: zhora dole a v rámci riadku zľava doprava. Zakaždým, keď nájdeme prázdne políčko, spravíme preň vyššie popísaný postup, pričom pri rekurzívnom volaní

si odovzdáme súradnice, kde sme boli, aby sme vedeli, odkiaľ ďalej stačí hľadať nasledujúce voľné políčko. Jeden možný stav počas behu tohto riešenia je znázornený na nasledujúcom obrázku:



Akými možnými spôsobmi ide vo všeobecnosti takéto políčko pokryť? Pri našom systematickom postupe máme zaručené, že všetky políčka v skorších riadkoch aj všetky políčka v aktuálnom riadku naľavo od práve pokrývaného sú už pokryté. Stačí nám teda vyskúšať nanajvýš štyri možnosti, a to tieto:



Na implementáciu tohto riešenia nám stačí jedna funkcia, ktorá ako parametre dostane súradnice prvého políčka (v nami zvolenom poradí), ktoré sme ešte neskontrolovali. Pri každom volaní skontrolujeme jedno políčko. Ak už je pokryté, alebo je to odtok, len sa zavoláme na nasledujúce. Ak nie, tak vyskúšame všetky štyri možnosti, ako ho pokryť, a pre každú, ktorá naozaj pasuje, sa rekurzívne zavoláme na nasledujúce políčko. No a akonáhle sa dostaneme k tomu, že úspešne skontrolujeme, že posledné políčko v poslednom riadku je pokryté, môžeme vyhlásiť, že sme našli riešenie.

Táto funkcia tiež musí mať prístup k poľu, do ktorého zostrojujeme popis kachličkovania. Toto sa dá riešiť rôznymi spôsobmi, napríklad tak, že si referenciu na pole budeme odovzdávať ako ďalší parameter našej rekurzívnej funkcie. (Treba si dať pozor na to, aby sme nekopírovali celé pole pri každom rekurzívnom volaní.)

Technika pre skúšanie všetkých možností, ktorú sme si práve popísali, je známa pod menom backtracking. Pamäťová zložitosť tohto riešenia je zjavne $O(n^2)$, kedže nám stačí pamätať si iba aktuálne pokrytie mriežky, ktoré sa práve pokúšame doplniť. Časová zložitosť môže byť v najhoršom prípade rádovo až $O(4^{(n^2/3)})$ kedže pri prikladaní každej kachličky máme na výber nanajvýš 4 možnosti.

Listing programu (C++)

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

using namespace std;

vector<pair<int, int> > otoc1 = { {0, 0}, {1, 0}, {1, -1} };
vector<pair<int, int> > otoc2 = { {0, 0}, {1, 0}, {1, 1} };
vector<pair<int, int> > otoc3 = { {0, 0}, {0, 1}, {1, 0} };
vector<pair<int, int> > otoc4 = { {0, 0}, {0, 1}, {1, 1} };

vector<vector<pair<int, int> > > otocenia = {otoc1, otoc2, otoc3, otoc4};

////////////////////////////////////////

bool hotovo = false;
int n;
vector<vector<int> > kupel;

////////////////////////////////////////

bool preoznac(int i, int j, int k, int oznacenie)
// preoznaci L-ko typu k na policku (i, j)
// oznacujem vyjadruje ci oznacujem alebo premazavam policka
{
    bool oznacujem = true;
    if(kupel[i][j] != 0) oznacujem = false;

    for(int a=0;a<3;++a)
    // skontrolujeme, ci vieme preoznacit vsetky 3 policka
```

```
    {
        int di, dj;
        di = i + otocenia[k][a].first;
        dj = j + otocenia[k][a].second;

        if(di < 0 || dj < 0 || di >= n || dj >= n) return false;

        if(oznacujem && kupel[di][dj] != 0) return false;
        else if(!oznacujem && kupel[di][dj] != oznacenie) return false;
    }

    for(int a=0;a<3;++a)
    {
        int di, dj;
        di = i + otocenia[k][a].first;
        dj = j + otocenia[k][a].second;

        if(oznacujem) kupel[di][dj] = oznacenie;
        else kupel[di][dj] = 0;
    }

    return true;
}

bool f(int i, int j, int ozn)
{
    if(hotovo) return true;

    if(j == n)
    // pokazila sa nam indexacia v riadku, presuvame sa na dalsi alebo koncime
    {
        if(i == n-1)
        {
            hotovo = true;
            return true;
        }
        else return f(i+1, 0, ozn);
    }

    if(kupel[i][j] != 0)
    // uz je zaplnene, posuvame sa
    {
        return f(i, j+1, ozn);
    }

    for(int k=0;k<4;++k) if(!hotovo)
    {
        if( !preoznac(i, j, k, ozn) ) continue;

        if( f(i, j+1, ozn+1) ) return true;

        preoznac(i, j, k, ozn);
    }

    return false;
}

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

    kupel.resize(n, vector<int>(n, 0));
    kupel[a-1][b-1] = -1;

    f(0, 0, 1);

    for(int i=0;i<n;++i)
    {
        for(int j=0;j<n;++j)
        {
            if(j != 0) cout << " ";

            if(kupel[i][j] == -1) cout << 'X';
            else cout << kupel[i][j];
        }
        cout << "\n";
    }

    return 0;
}
```

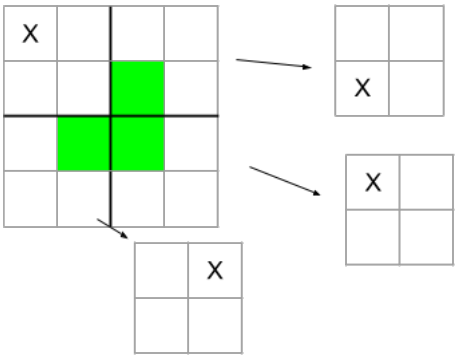
Niektoré implementácie tohto riešenia však fungujú v skutočnosti prekvapivo dobre. Riešenie totiž nielen že vždy existuje, je ich dokonca tak veľa, že sa často stane, že už prvá možnosť priloženia kachličky, ktorú vyskúšame, vedie k riešeniu – a tak skutočné skúšanie možností budeme robiť až pri okrajoch, kde už niektoré umiestnenia kachličiek budú robiť problémy.

Vzorové riešenie

Ako by sme mohli začať hľadať nejaké riešenie, ktoré bude zaručene efektívne? Dobrým začiatkom je skúsiť ručne vyriešiť nejaké mriežky menších rozmerov a získať tak trochu obraz o tom, ako riešenia vyzerajú a pre ktoré kombinácie (veľkosť mriežky, poloha odtoku) riešenie neexistuje.

Ak sa pozrieme na mriežky veľkosti 2×2 či 4×4, zistíme, že nech je odtok kdekoľvek vždy nájdeme nejaké riešenie. To by nám mohlo napovedať, že by to mohlo ísť aj pre úplne všetky väčšie mriežky – len budeme musieť náš postup nejak zovšeobecniť.

Skúsme sa pozrieť napríklad na to, ako vieme vykachličkovať mriežku veľkosti 4×4 s odtokom v ľavom hornom rohu. Ľahko zistíme, že vieme doložiť jednu kachličku tak, aby sme doplnili celý ľavý horný štvorec veľkosti 2×2. Ako teraz pokryť ostatné tri štvorce rozmerov 2×2? Každý z nich by sme vedeli pokryť samostatne, ak by obsahoval odtok. Inými slovami, každý z nich vieme pokryť celý okrem jedného jeho políčka. A už sme skoro vyhrali. Všimnite si kachličku, ktorá je na nasledujúcom obrázku zelenou farbou. Jej priložením sme pokryli práve jedno políčko v každom štvorci 2×2, ktorý neobsahoval odtok. A teda nám ostali štyri štvorce 2×2, z ktorých každý má práve jedno pokryté a tri nepokryté políčka.



Teraz uvažujme mriežku 4×4, ktorá má odtok niekde inde. Zjavne aj tú vieme celú pokryť, a to presne rovnakým postupom. Jediný rozdiel bude v tom, že bude inak otočená kachlička v tom štvorci 2×2, ktorý obsahuje odtok.

Ako je to s väčšími mriežkami? Mriežku 8×8 si môžeme rozdeliť na štyri mriežky rozmerov 4×4. Je jasné, že iba v jednom z týchto blokov je práve jedno políčko zabrané odtokom. Vhodným uložením jednej "zelenej" kachličky ku stredu mriežky teraz vieme zabrať po jednom políčku aj v každom zo zvyšných troch blokov. No a každý z týchto blokov vieme vyriešiť vyššie popísaným spôsobom.

No a tento postup už ľahko zovšeobecníme: ľubovoľnú mriežku veľkosti $2^k \times 2^k$ vieme rozdeliť na štyri mriežky veľkosti $2^{k-1} \times 2^{k-1}$ a následne vieme vhodne priložiť jednu kachličku tak, aby každá mriežka menších rozmerov mala jedno políčko, ktoré už netreba pokryť. Potom sa na každú z menších mriežok rekurzívne zavoláme, čiže pre každú z nich znova zopakujeme túto istú úvahu.

Vhodnou implementáciou tohto riešenia je jedna rekurzívna funkcia, ktorá si bude pamätať, akú časť mriežky vypĺňa a kde v nej sa nachádza "odtok", teda to jedno políčko, ktoré je už pokryté. Pamäťová zložitosť tohto riešenia je $O(n^2)$, kedže si pamätáme iba mriežku, do ktorej zároveň hneď dopĺňame kachličky.

Časová zložitosť je tiež $O(n^2)$, čiže priamo úmerná veľkosti mriežky. Prečo? Pri každom zavolaní našej funkcie totiž spravíme konštantne veľa výpočtov a priložíme jednu kachličku. No a keďže kachličiek je $(n^2 - 1)/3$, toľko bude aj volaní našej funkcie.

Listing programu (C++)

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

using namespace std;

// mame 4 rozne natocenia kachlicky
// X1 2X 33 44
// 11 22 X3 4X

const vector<int> dx = {0, 0, -1, -1};
const vector<int> dy = {0, -1, 0, -1};

vector<pair<int, int> > d2 = { {0, 0}, {-1, 0}, {0, 1} };
vector<pair<int, int> > d4 = { {0, 0}, {0, 1}, {1, 0} };
vector<pair<int, int> > d3 = { {0, 0}, {1, 0}, {0, -1} };
vector<pair<int, int> > d1 = { {0, 0}, {0, -1}, {-1, 0} };

vector<vector<pair<int, int> > > delta = {d1, d2, d3, d4};

int kvadrant(const int i, const int j, const int pol_size, const int a, const int b)
// v ktorom kvadrante sa (a;b) nachadza :
// 1 2
// 3 4
{
    int counter = 1;

    for(int k=0;k<2;++k)
        for(int l=0;l<2;++l)
        {
```

```
            int roh_i = i + k*pol_size;
            int roh_j = j + l*pol_size;

            if(a >= roh_i && a < roh_i + pol_size && b >= roh_j && b < roh_j + pol_size) return counter;

            ++counter;
        }

    return -1;
}

int ozn = 1;

void zapis(vector<vector<int> >& vypln, const int i, const int j, int typ)
// vypln kachlicku so stredom v i;j otocenia typ
{
    for(int k=0;k<3; ++k) vypln[i+delta[typ-1][k].first][j+delta[typ-1][k].second] = ozn;
    ++ozn;
}

void f(vector<vector<int> >& vypln, const int a, const int b, const int i, const int j, const int size)
// rekurzivna funkcia vlozi do 3 spravnych kvadrantov jednu kachlicku
{
    if(size == 1) return;

    int size_new = (size)/2;
    int kde = kvadrant(i, j, size_new, a, b);

    zapis(vypln, i+size_new+dx[kde-1], j+size_new+dy[kde-1], kde);

    int counter = 1;
    for(int k=0;k<2;++k)
            for(int l=0;l<2;++l, ++counter)
            {
                if(kde == counter)
                {
                    f(vypln, a, b, i+k*size_new, j+l*size_new, size_new);
                    continue;
                }

                f(vypln, i+size_new-1+k*1, j+size_new-1+l*1, i+k*size_new, j+l*size_new, size_new);
            }

    return;
}

int main()
{
    int n, a, b;
    cin >> n >> a >> b;
    --a, --b;

    vector<vector<int> > vypln(n, vector<int>(n, 0));
    vypln[a][b] = -1;

    f(vypln, a, b, 0, 0, n);

    for(int i=0;i<n;++i)
    {
        for(int j=0;j<n;++j)
        {
            if(j != 0) cout << "␣";

            if(vypln[i][j] == -1) cout << 'X';
            else cout << vypln[i][j];
        }
        cout << "\n";
    }

    return 0;
}
```

Peťo R

6. Optimalizácia písania na klávesnici

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

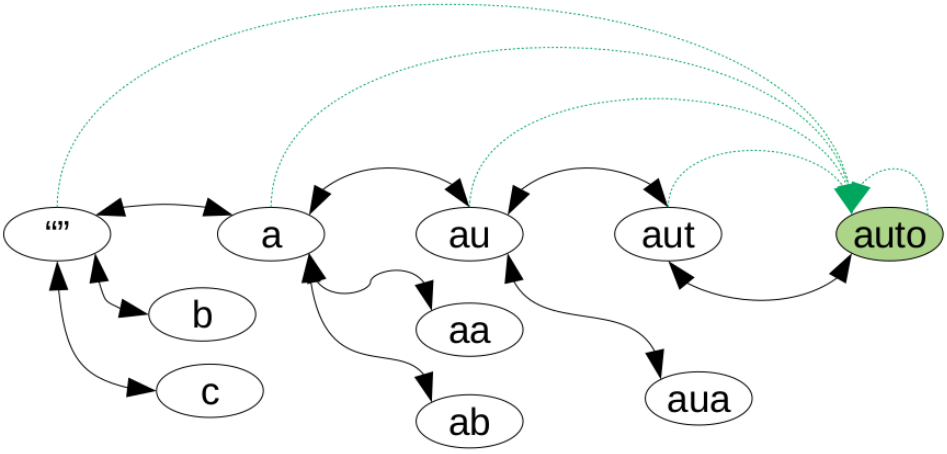
Pred tým než sa pustíme do vzoráku si ujasnime niekoľko pojmov:

- Reťazec (angl. *string*) je postupnosť znakov. Môže to byť slovo, jeho časť ale aj celá veta.
- Prefix je začiatočný úsek reťazca. Formálnejšie, reťazec a nazveme prefixom reťazca b, ak za a vieme dopísať niekoľko (možno aj nula) znakov tak, aby sme dostali reťazec b. Napríklad reťazec `auto` má päť prefixov – „" (prázdny reťazec), `a`, `au`, `aut` a `auto`.

Pri písaní zadaného slova nás zaujíma iba to, čo máme aktuálne napísané. Tento reťazec reprezentuje nejaký stav, v ktorom sa nachádzame. Stlačením klávesy sa náš reťazec zmení a my sa presunieme do iného stavu. Napríklad ak sú v slovníku slová `autobus` a `autostrada`, tak ak sa nachádzame v stave `au`, môžeme sa jedným stlačením presunúť do nasledovných stavov:

- `aut`, tým, že stlačíme `t` (za `t` si môžeme dosadiť ľubovoľné písmeno, stlačením `l` by sme sa dostali do stavu `aul` a podobne)
- `autobus`, tým že stlačíme `TAB`, automaticky sa nám doplní lexikograficky najmenšie slovo zo slovníka, ktoré začína na `au`
- `a`, tým že stlačíme `Backspace` zmažeme posledný znak

V okamihu ako nám v úlohe vystupujú stavy, medzi ktorými sa presúvame, môžeme sa na ne pozrieť ako na graf. Stavy reprezentujú jednotlivé vrcholy a presun zo stavu do stavu orientovanú hranu. To zásadne zmení aj úlohu, ktorú riešime. V pôvodnom zadaní sme chceli vedieť, ako čo najrýchlejšie dostať z prázdneho reťazca reťazec výsledný. Prázdnemu aj výslednému reťazcu však teraz zodpovedajú nejaké vrcholy a nás zaujíma najkratšia cesta medzi nimi.



Na obrázku môžeme vidieť, ako by vyzerala časť grafu so slovom `auto`. Skutočný graf je totiž nekonečný a obsahuje všetky možné reťazce. Všimnite si, že čierne šipky sú obojsmerné. To značí možnosť dopísania písmena na koniec a jeho vymazania stlačením `Backspace`. Tiež si všimnite zelené šípky, ktoré predstavujú stláčanie klávesy `TAB`.

Povedali sme si, že písanie slova je ekvivalentné hľadaniu najkratšej cesty medzi zadanými vrcholmi. Takže môžeme využiť dobre známy algoritmus prehľadávania do šírky (BFS). Problémom ale je, že náš graf je nekonečný, keďže na klávesnici vieme napísať čokoľvek. Pokúsme sa teda zmenšiť počet stavov, ktorými sa musíme zaoberať.

Kľúčovým pozorovaním je, že si nepotrebujeme pamätať stavy, ktoré nie sú prefixom žiadného slova v slovníku.

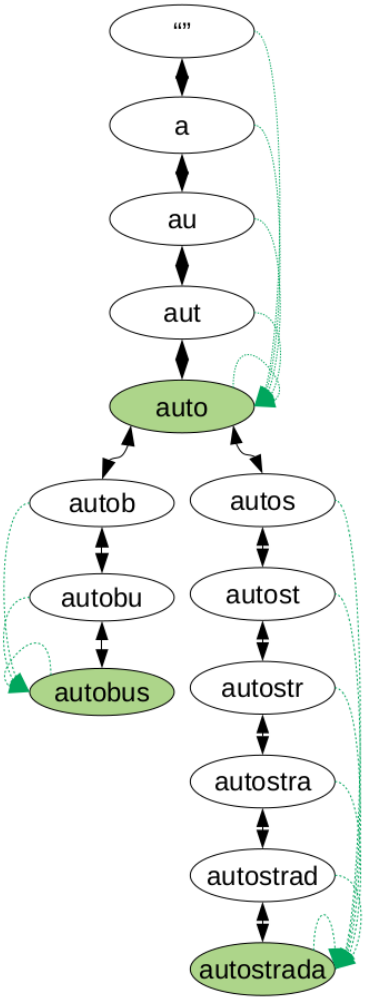
Prečo je tomu tak? Keby sme na písanie mohli používať iba písmená, úloha by bola jednoduchá, lebo cestu by sme si nevedeli nijak skracovať a museli by sme proste napísať zadané slovo písmeno po písmene. Nám však pribudli aj klávesy `TAB` a `Backspace`. A klávesa `TAB` dopĺňa slová zo slovníku, ktoré majú rovnaký prefix ako reťazec, ktorý sme doposiaľ napísali. Ak sme teda v stave, ktorý nie je prefixom žiadneho slova v slovníku, klávesa `TAB` nič nerobí.

Klávesa `Backspace` má tiež len obmedzené využitie. Uvedomme si, že v optimálnom riešením nikdy nepoužijeme `Backspace` tesne po tom, čo napíšeme nejaké písmeno. Keby sme to totiž spravili, reťazec by sa vlastne nezmenil a my by sme tieto dve stlačenia mohli radšej vynechať. `Backspace` teda môžeme stlačiť iba po nejakom `TAB` alebo `Backspace`. To znamená, že vymazávanie znakov využijeme iba v prípade, že stlačíme `TAB`, doplnené slovo však bude pridlhé a niekoľko posledných znakov budeme chcieť vymazať. Pri tom všetkom sme sa však neustále pohybovali po reťazcoch, ktoré sú prefixami slov v slovníku.

Z vyššie uvedených pozorovaní tiež vyplýva, že v okamihu ako sa rozhodneme opustiť prefixy slov v slovníku, tak klávesy `TAB` a `Backspace` už nemá zmysel použiť a nám ostáva zvyšný text dopísať písmeno po písmene.

Keď si teda predstavíme ako funguje naše riešenie, ak chceme napísať slovo s, tak najskôr sa budeme pohybovať po prefixoch slova s, ktoré sú zároveň prefixami niektorých slov v slovníku a keď tieto prefixy opustíme, zvyšok slova s jednoducho dopíšeme.

Na obrázku vidíme, ako vyzerá graf všetkých prefixov slov zo slovníka `auto`, `autobus` a `autostrada`.



Skúsený riešiteľ si isto všimne, že graf ktorý nám ostane pripomína písmenkový strom (po anglicky aj *trie*). Písmenkový strom je dátová štruktúra, v ktorej si pamätáme informácie o slovách v slovníku a ich prefixoch. Každý vrchol reprezentuje konkrétny prefix slova v nej – presne ako v grafe ktorý vytvárame. Z každého vrcholu vedie najviac 26 hrán do synov – vrcholov, ktoré reprezentujú stavy do ktorých sa dostaneme pridaním jedného písmenka na koniec reťazca. My si však do tohto písmenkového stromu ešte musíme pridať hrany, ktoré reprezentujú stláčanie kláves `TAB` a `Backspace`.

Písmenkový strom sa tradične konštruuje tak, že začíname s prázdnym stromom, do ktorého postupne pridávame slová. Slovo pridávame po písmenách začnúc v koreni. Postupne sa snažíme posúvať po hranách, ktoré reprezentujú jednotlivé písmená slova. V okamihu, keď sa chceme posunúť po hrane, ktorá ešte z aktuálneho vrcholu nevedie, vytvoríme nový vrchol a do neho vedúcu hranu. Pri vytváraní vrcholu môžeme veľmi jednoducho spraviť hranu aj pre `Backspace` – stačí si zapamätať otca vrcholu.

Hrany pre autocomplete sú najzložitejšie, vieme ich však vygenerovať rekurzívne. Ak vrchol reprezentuje slovo v slovníku tak jeho autocomplete vedie do neho samého. Ak nie, tak pre každého syna najprv rekurzívne zistíme jeho autocomplete a potom vyberieme autocomplete zo syna, do ktorého vedie hrana s lexikograficky najmenším písmenom v abecede.

Po skompletizovaní grafu na ňom spustíme prehľadávanie do šírky, ktoré nám pre každý vrchol vypočíta, ako najrýchlejšie sa do neho vieme dostať zo začiatku, ktorý reprezentuje prázdny reťazec, a tieto hodnoty si zapamätáme.

Ostáva nám už len odpovedať na otázku, ako najrýchlejšie vieme napísať slovo s. Na to nájdeme najdlhší prefix tohto slova, ktorý sa nachádza aj v našom písmenkovom strome. Z predpočítanej informácie vieme, ako najrýchlejšie napísať tento prefix a keďže je to prefix najdlhší, zvyšok už aj tak musíme napísať písmeno po písmene.

Časová zložitosť konštruovania je lineárna od súčtu dĺžok slov v slovníku. BFS tiež spravíme v lineárnom čase od počtu vrcholov v strome. A nájsť najdlhší prefix tiež zvládneme v čase lineárnom od dĺžky hľadaného slova. Celková časová zložitosť je teda lineárna od dĺžky vstupu. Pamäťová zložitosť je lineárna od počtu vrcholov v písmenkovom strome. Naviac, táto zložitosť je určite optimálna, keďže rovnako veľa času nám zaberie načítanie

celého vstupu.

Listing programu (C++)

```
#include<cstdio>
#include<vector>
#include<string>
#include<queue>
#include<cstring> //strlens
#include<algorithm> // min
#include<utility> //pair
#include<iostream> //cin
#include<cassert> //assert
using namespace std;
int charToInt(char c) //zmeni znak na cislo
{
    return c-'a';
}
struct pismenkac
{
    vector<pismenkac *> V;
    int vzdOdKorena = -1;
    pismenkac *TAB, *rodic;

    //konstruktor -> skonstruuj novy vrchol, a pridaj do neho slovo c
    pismenkac(const char * c, pismenkac * _rodic)
    {
        V.resize(26,nullptr);
        TAB = nullptr;
        rodic = _rodic;

        if(*c == '\0') {TAB=this; return;}
        V[charToInt(*c)]=new pismenkac(c+1, this);
    }

    pismenkac()
    {
        V.resize(26,nullptr);
        TAB = nullptr;
        rodic = nullptr;
    }

    //pridaj slovo
    void pridajSlovo(const char *c)
    {
        if(*c == '\0')
        {
            //ak sa v tomto vrchole konci slovo
            TAB=this;
        }
        else if(V[charToInt(*c)]) // treba pridat zvysok slova
            V[charToInt(*c)]->pridajSlovo(c+1);
        else //treba vytvorit vrchol do ktoreho pridame zvysok slova
            V[charToInt(*c)] = new pismenkac(c+1, this);
    }

    void doplnTABlinky()
    {
        for(auto &x: V)
            if(x)
                x->doplnTABlinky();
        if(!TAB)
            for(auto &x: V)
            {
                if(x)
                {
                    TAB = x->TAB;
                    return;
                }
            }
    }

    //zisti ako daleko je slovo od korena, c je zostavajuci suffix - este nespracovane pismena zo stromu
    int najdiSlovo(const char *c){
        if(*c=='\0') //ak sme nasli slovo
            return vzdOdKorena;
        if(V[charToInt(*c)] == nullptr) //ak zvysok slova nie je v grafe
            return strlen(c)+vzdOdKorena; //scitame vzdialenost tohto vrchola od korenu a dlzku zvysku slova
        return V[charToInt(*c)]->najdiSlovo(c+1); //pokracuj v hladani hladaneho slova/prefixu u syna
    }
};

//vyrataj BFS
void BFS(pismenkac * begin)
{
  queue<pismenkac *> Q;
  Q.push(begin); //v Q budu iba vrcholy ktore este neboli spracovane a budu spracovane v najblizsej iteracii
  begin->vzdOdKorena = 0;
  while(Q.size())
  {
      pismenkac * kde = Q.front(); // vrchol

      int vzd = kde->vzdOdKorena;
      Q.pop();

      if(kde->TAB->vzdOdKorena == -1) //chod po hrane TAB linky
      {
```

```
            kde->TAB->vzdOdKorena = vzd+1;
            Q.push(kde->TAB);
        }
        if(kde->rodic && kde->rodic->vzdOdKorena == -1) //chod po hrane Backspace k rodicovi
        {
            kde->rodic->vzdOdKorena = vzd+1;
            Q.push(kde->rodic);
        }
        for(auto &p : kde->V) //chod po hrane pismenka
        {
            if(p && p->vzdOdKorena == -1)
              {
                  p->vzdOdKorena = vzd+1;
                  Q.push(p);
              }
        }
    }
}

int main()
{
    string slovo;
    pismenkac p;
    int n, q;
    cin>>n>>q;
    while(n--)
    {
        cin>>slovo;
        p.pridajSlovo(slovo.c_str());
    }
    p.doplnTABlinky();
    BFS(&p);
    while(q--)
    {
        cin>>slovo;
        cout<<p.najdiSlovo(slovo.c_str())<<endl;
    }
}
```

Mišof

7. Daj het

(max. 4 b za popis, 16 b za program)

Inšpiráciou pre túto úlohu bola logická hra Rullo. Zadania, ktoré dostali vaše programy, len boli o čosi väčšie, ako tie, ktoré riešia ľudia.

Riešenie hrubou silou

Ako dať z mriežky preč tie správne čísla? Zamyslíme sa nad niečím šikovným, nič nevymyslíme, a tak sa rozhodneme vyskúšať hrubú silu.

Aj pri hrubej sile je ale veľa možností. Všetkých možností, ako z tabuľky rozmerov $n \times n$ niektoré čísla vyhodiť a niektoré nechať je až 2^{n^2}, a to už od $n = 6$ začína byť nezvládnuteľne veľa možností.

Omnoho lepšie sú riešenia založené na tzv. backtrackingu (prehľadávaní s návratom). Pri takýchto riešeniach sa rozhodujeme postupne, políčko po políčku. Výhodou je, že po každom rozhodnutí môžeme spraviť rôzne kontroly a ak zistíme, že už určite nevieme vyrobiť žiadne platné riešenie, môžeme sa z aktuálnej "vetvy" prehľadávania rovno vrátiť a tak ušetriť veľa ďalšieho skúšania možností.

Pri našej úlohe si napríklad pre každý riadok aj stĺpec môžeme zvlášť pamätať súčet čísel, ktoré sme sa v ňom už rozhodli nechať, a zvlášť súčet čísel, o ktorých sme sa ešte nerozhodli. Potom vždy, keď spravíme nejaké rozhodnutie o ďalšom políčku, upravíme si tieto súčty pre jeho riadok a stĺpec, a následne pomocou nich zistíme, či ešte želané súčty pre ne ležia v zostrojiteľnom rozsahu.

Ďalšie zefektívnenie takéhoto typu riešení môže spočívať v šikovnejšej voľbe poradia, v ktorom sa o jednotlivých políčkach rozhodujeme. V princípe platí, že čím skôr o nejakej vetve zistíme, že k riešeniu nevedie, tým viac času ušetríme.

Kombinácia hrubej sily a dedukcie

Keby túto úlohu riešil človek, často by používal dedukciu: "Toto číslo nemôžem použiť, lebo by mi nesedela posledná cifra, tamto zas použiť musím, lebo by inak bol súčet primalý, a hento tiež, lebo by mi v stĺpci nesedela parita." A vždy, keď nám z takejto dedukcie niečo vypadne, tak nám to môže pomôcť. Ak napríklad z pohľadu na riadok zistíme, že nejaké jeho políčko treba odstrániť, môžeme sa potom pozrieť na stĺpec a v ňom zrazu máme tiež o políčko menej.

Našou druhou cestou k riešeniu bude snaha spraviť program, ktorý bude vedieť robiť takéto dedukcie. Vopred síce nevieme, či nám pomôžu, ale za pokus to stojí – ak sa ukáže, že áno, máme lepšie riešenie, ak sa ukáže, že nie, aspoň sme sa naučili niečo nové o vlastnostiach tejto úlohy.

Hja, ale ako naučiť program robiť takéto dedukcie? K plnohodnotnej AI máme ďaleko a implementácia nejakých jednoduchých pravidiel nám asi žiadne ohromujúce výsledky nedá.

Existuje ale pekný systematický spôsob, ktorý môžeme použiť a ktorý zahrnie všetky možné veci, ktoré sa dajú vydedukovať len z toho jedného riadku či stĺpca. Trik je jednoduchý: Pozrieme sa na všetkých 2^n možností, čo spraviť s dotyčným riadkom (respektíve menej, ak už sme o niektorých jeho políčkach rozhodnutí). Spomedzi všetkých možností si vyberieme všetky tie, ktoré vyrobia správny súčet. No a teraz sa pozrieme na to, na ktorých políčkach sa všetky tieto možnosti zhodujú. Takto sa dozvieme aj to, ktoré políčka určite musíme nechať, aj to, ktoré určite zahodiť.

(Elegantná implementácia je pomocou bitových vzoriek. Na čísla od 0 po $2^n - 1$ sa môžeme dívať ako na všetky možné n-tice bitov, teda ako na všetky možné postupnosti hovoriace, ktoré políčka zahodiť a ktoré zobrať. Keď si potom zoberieme všetky čísla, ktoré zodpovedali platným riešeniam pre dotyčný riadok, z ich bitového ANDu vidíme, kde mali všetky 1, a z ich bitového ORu zase to, kde mali všetky 0.)

Vyššie popísanú úvahu vieme postupne opakovať pre všetky riadky a stĺpce, až kým buď celú tabuľku nevyriešime, alebo sa nedostaneme do situácie, kedy už v žiadnom riadku ani stĺpci nevieme nič nové odvodiť. No a čo spravíme v tomto druhom prípade? Použijeme predchádzajúce riešenie – teda vyberieme si nejaké nerozhodnuté políčko a postupne rekurzívne vyskúšame obe možnosti preň.

Pri implementácii si potom ešte treba dať pozor na to, že vždy, keď sa ideme z rekurzie vrátiť, treba zabudnúť nie len na stav políčka, o ktorom sme sa priamo rozhodli, ale tiež na stav všetkých políčok, o ktorých sme po tom rozhodnutí vyššie popísanou dedukciou zistili, že zrazu už sú jednoznačne určené.

Listing programu (C++)

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

int N;
vector< vector<int> > original_table;
vector<int> desired_row_sums, desired_col_sums;

vector< vector<int> > state;        // o kazdom policku: -1 este neviem, 0 zahodit, 1 nechat
vector< vector<int> > change_depth; // pomocne pole kde si pamatame, na ktorej urovni rekurzie sme zmenili stav pre toto policko

int min_row_sum(int r) { int answer=0; for (int c=0; c<N; ++c) if (state[r][c]==1) answer += original_table[r][c]; return answer;
int max_row_sum(int r) { int answer=0; for (int c=0; c<N; ++c) if (state[r][c]!=0) answer += original_table[r][c]; return answer;
int min_col_sum(int c) { int answer=0; for (int r=0; r<N; ++r) if (state[r][c]==1) answer += original_table[r][c]; return answer;
int max_col_sum(int c) { int answer=0; for (int r=0; r<N; ++r) if (state[r][c]!=0) answer += original_table[r][c]; return answer;

bool is_solved() {
    for (int r=0; r<N; ++r) for (int c=0; c<N; ++c) if (state[r][c] == -1) return false;
    for (int r=0; r<N; ++r) if (min_row_sum(r) != desired_row_sums[r]) return false;
    for (int c=0; c<N; ++c) if (min_col_sum(c) != desired_col_sums[c]) return false;
    return true;
}

void deduce_row(int r, int depth, bool &was_change, bool &was_contradiction) {
    vector<int> indices;
    for (int c=0; c<N; ++c) if (state[r][c] == -1) indices.push_back(c);
    vector<int> good_subsets;
    int S = indices.size();
    for (int subset=0; subset<(1<<S); ++subset) {
        for (int s=0; s<S; ++s) state[r][ indices[s] ] = (subset >> s) & 1;
        if (min_row_sum(r) == desired_row_sums[r]) good_subsets.push_back(subset);
    }
    for (int s=0; s<S; ++s) state[r][ indices[s] ] = -1;
    if (good_subsets.empty()) { was_contradiction = true; return; }

    int bitwise_and = (1<<S) - 1;
    for (int subset : good_subsets) bitwise_and &= subset;
    for (int s=0; s<S; ++s) if (bitwise_and & 1<<s) {
        state[r][ indices[s] ] = 1;
        change_depth[r][ indices[s] ] = depth;
        was_change = true;
    }

    int bitwise_or = 0;
    for (int subset : good_subsets) bitwise_or |= subset;
    for (int s=0; s<S; ++s) if (!(bitwise_or & 1<<s)) {
        state[r][ indices[s] ] = 0;
        change_depth[r][ indices[s] ] = depth;
        was_change = true;
    }
}

void deduce_col(int c, int depth, bool &was_change, bool &was_contradiction) {
    vector<int> indices;
    for (int r=0; r<N; ++r) if (state[r][c] == -1) indices.push_back(r);
    vector<int> good_subsets;
    int S = indices.size();
    for (int subset=0; subset<(1<<S); ++subset) {
        for (int s=0; s<S; ++s) state[ indices[s] ][c] = (subset >> s) & 1;
        if (min_col_sum(c) == desired_col_sums[c]) good_subsets.push_back(subset);
    }
    for (int s=0; s<S; ++s) state[ indices[s] ][c] = -1;
    if (good_subsets.empty()) { was_contradiction = true; return; }

    int bitwise_and = (1<<S) - 1;
    for (int subset : good_subsets) bitwise_and &= subset;
    for (int s=0; s<S; ++s) if (bitwise_and & 1<<s) {
```

```
        state[ indices[s] ][c] = 1;
        change_depth[ indices[s] ][c] = depth;
        was_change = true;
    }

    int bitwise_or = 0;
    for (int subset : good_subsets) bitwise_or |= subset;
    for (int s=0; s<S; ++s) if (!(bitwise_or & 1<<s)) {
        state[ indices[s] ][c] = 0;
        change_depth[ indices[s] ][c] = depth;
        was_change = true;
    }
}

void print() {
    for (int pr=0; pr<N; ++pr) {
        for (int pc=0; pc<N; ++pc) {
            if (state[pr][pc] == 1) cout << original_table[pr][pc];
            if (state[pr][pc] == 0) cout << 0;
            if (state[pr][pc] == -1) cout << original_table[pr][pc] << "?";
            cout << (pc==N-1 ? "\n" : "␣");
        }
    }
}

bool go(int depth) {
    // zisti, ci nahodou nie je zjavne, ze aktualna vetva nevedie k rieseniu
    for (int r=0; r<N; ++r) if (min_row_sum(r) > desired_row_sums[r]) return false;
    for (int r=0; r<N; ++r) if (max_row_sum(r) < desired_row_sums[r]) return false;
    for (int c=0; c<N; ++c) if (min_col_sum(c) > desired_col_sums[c]) return false;
    for (int c=0; c<N; ++c) if (max_col_sum(c) < desired_col_sums[c]) return false;
    // dokola prechadzaj riadky a stlpce a rob pre ne dedukcie
    while (true) {
        bool was_change = false, was_contradiction = false;
        for (int r=0; r<N; ++r) deduce_row(r,depth,was_change,was_contradiction);
        for (int c=0; c<N; ++c) deduce_col(c,depth,was_change,was_contradiction);
        if (was_contradiction) {
            for (int r=0; r<N; ++r) for (int c=0; c<N; ++c) if (change_depth[r][c] == depth) state[r][c] = -1;
            return false;
        }
        if (!was_change) break;
    }
    // ak uz si vsetko vyriesil, vypis riesenie, inak rekurzivne skusaj dalsie policko
    if (is_solved()) {
        print();
        return true;
    }
    int nr=0, nc=0;
    while (state[nr][nc] != -1) { ++nc; if (nc==N) { nc=0; ++nr; } if (nr==N) break; }
    change_depth[nr][nc] = depth;
    state[nr][nc] = 1;
    if (go(depth+1)) return true;
    state[nr][nc] = 0;
    if (go(depth+1)) return true;
    for (int r=0; r<N; ++r) for (int c=0; c<N; ++c) if (change_depth[r][c] == depth) state[r][c] = -1;
    return false;
}

int main() {
    int T; cin >> T;
    while (T--) {
        cin >> N;
        desired_row_sums.clear(); desired_row_sums.resize(N);
        desired_col_sums.clear(); desired_col_sums.resize(N);
        original_table.clear();   original_table.resize(N, vector<int>(N));
        state.clear();            state.resize(N, vector<int>(N,-1));
        change_depth.clear();     change_depth.resize(N, vector<int>(N,-1));
        for (int n=0; n<N; ++n) cin >> desired_row_sums[n];
        for (int n=0; n<N; ++n) cin >> desired_col_sums[n];
        for (int r=0; r<N; ++r) for (int c=0; c<N; ++c) cin >> original_table[r][c];
        go(0);
    }
}
```

Nechám oboje na niekoho iného

"Robenie dedukcií" a "skúšanie možností" sú natoľko bežné súčasti riešenia problémov, že už zrejme niekto prišiel na to, ako spraviť špecializované nástroje, ktoré sú fakt dobré v jednom aj druhom.

Jedným zo základných typov takýchto nástrojov sú tzv. SAT solvery. Ich použitie funguje nasledovne: Zoberieme problém, ktorý chceme vyriešiť, zakódujeme jeho "pravidlá" do jedného veľkého logického výrazu, a potom poprosíme SAT solver, aby nám našiel, pre aké ohodnotenie premenných je celý tento výraz splnený. Ak to nedáva zmysel, nebojte sa, o chvíľu začne.

Ako môžeme našu úlohu zapísať ako logický výraz? Keď máme tabuľku $n \times n$, začneme tým, že budeme mať n^2 boolovských premenných: pre každé políčko jednu premennú, ktorá nám hovorí, či je súčasťou riešenia.

Logický výraz, ktorý sme pred chvíľou spomínali, bude veľkou konjunkciou (t.j. logickým ANDom) veľa podmienok, ktoré musia tieto premenné spĺňať, aby sme dokopy dostali platné riešenie. Presnejšie, vstup pre bežný SAT solver musí byť v špecifickom tvare, tzv. konjuktívnej normálnej forme. To znamená, že musí byť vyššie popísaného tvaru a navyše môžeme používať len veľmi jednoduché podmienky. Každá podmienka musí

byť tvaru "spomedzi týchto premenných a týchto ich negácií musí byť aspoň jedna pravdivá". Teda napríklad môžeme mať v našej úlohe podmienku tvaru "zober políčko (3,2) alebo zober políčko (3,7) alebo zahoď políčko (3,1)".

Logický výraz, ktorý zostrojíme, bude ANDom $2n$ sád podmienok: jedna sada pre každý riadok aj stĺpec. Každá sada podmienok bude hovoriť "táto sada premenných musí byť taká, aby sedel súčet tohto riadku / stĺpca". Ako toto dosiahnuť? Jednoducho tak, že si hrubou silou vygenerujeme všetky zlé kombinácie premenných a každú z nich jednou podmienkou zakážeme. (Ak nechceme, aby nastala situácia "zahoď (3,2) a zahoď (3,7) a zober (3,1)", pridáme podmienku, ktorá je jej negáciou – čiže podmienku, ktorú sme uviedli ako príklad v minulom odseku.) Pre každý riadok a stĺpec takto vygenerujeme najviac 2^n zlých kombinácií, každú s najviac n premennými a celkovo teda bude mať logický výraz dĺžku $O(n^2 \cdot 2^n)$. Logický výraz – vstup pre SAT solver – musíme, samozrejme, skonštruovať pomocou vlastnoručne napísaného programu.

Viac o tejto téme sa dozviete v zadaní a riešení staršej úlohy KSP: úloha "Ono to rieši samo!" (31. ročník KSP, úloha 5 v prvom kole zimnej časti).

Za záver

Riešenie už síce skončilo, my si ale povieme ešte niečo navyše. Prečo sme sa uspokojili len s riešeniami používajúcimi hrubú silu a nehľadali sme napríklad lepšie riešenie, ktoré by úlohu vždy vyriešilo v čase $O(n^2)$? Preto, že vieme dokázať, že táto úloha je ťažká.

Jednou zo známych algoritmických úloh, o ktorých sa vie, že sú ťažké (natoľko, že sa všeobecne verí, že nemajú žiadne polynomiálne riešenie) je úloha SubsetSum: "Tu máš sadu prirodzených čísel $a_0, \ldots, a_{n-1}$ a cieľovú hodnotu s, existuje nejaká podmnožina daných čísel, ktorá má súčet presne s?" No a riešiť našu úlohu je aspoň tak ťažké, ako riešiť SubsetSum. Totiž ľubovoľné zadanie SubsetSum vieme prerobiť na zadanie našej úlohy. Stačí si spraviť tabuľku, kde v riadku i a stĺpci j máme číslo $a_{(i+j) \bmod n}$, a predpísať, že v každom riadku aj stĺpci chceme dostať súčet s.

Ak by teda existoval efektívny algoritmus pre našu úlohu, dostali by sme aj efektívny algoritmus pre SubsetSum. A keďže ten zrejme neexistuje, nemá zmysel sa snažiť ani pre našu úlohu.

(To, čo sme si práve predviedli, bola redukcia, ktorá dokazuje, že náš problém je rovnako ako SubsetSum jedným z tzv. NP-ťažkých rozhodovacích problémov.)

buj

8. Svinstvo na koberci

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

Táto úloha bola fakt ťažká. Dĺžka vzoráku to celkom odzrkadľuje. Nebojte sa teda vzorák spracovávať po kúskoch, kľudne aj vo viacerých sedeniach.

Na získanie 5 bodov z programu si stačilo len uvedomiť, čo v skutočnosti znamenajú niektoré geometrické pojmy a ako ich vyjadriť v programe. V geometrii totiž častokrát "vidíme", že niečo platí, ale nevieme to poriadne zdôvodniť/vyjadriť. Napríklad, čo znamená, že bod A je vo vnútri nejakej oblasti? Čo to vlastne je oblasť, ako môže vyzerať? Nielen týmito otázkami sa budeme zaoberať v nasledujúcej časti.

Geometria

Základy

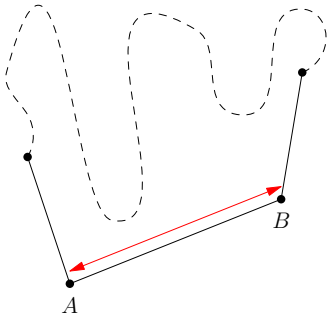
Ak ste sa s výpočtovou geometriou ešte nestretli, odporúčame vám najprv prečítať si v našej kuchárke Úvod do výpočtovej geometrie[3].

Vo vzorovom riešení sa často využívajú skalárny súčin, a obzvlášť vektorový súčin. Ak vám niekde vo vzoráku nebude jasné, ako je niektorý zo súčinov použitý, môžete hľadať odpovede v odpovedajúcom článku v kuchárke: Skalárny a vektorový súčin[4]

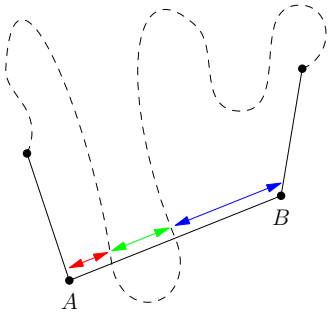
Čím je určená oblasť?

Najprv si rozmyslime jednu vec. Naľavo od každej hrany je vždy práve jedna oblasť, a napravo od hrany je vždy práve jedna oblasť. Pritom to môže byť tá istá oblasť, ako naľavo, a obe oblasti môžu byť nekonečné. Je zrejmé, že v oboch smeroch hrana susedí s aspoň jednou oblasť; zároveň vieme prejsť "tesne" popri hrane od jedného konca hrany k druhému, takže oblasť môže byť len jedna.

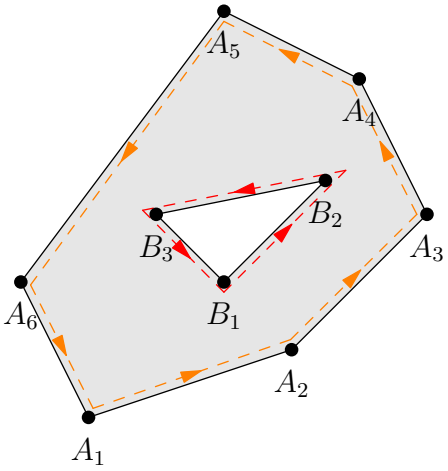
[3] https://www.ksp.sk/kucharka/uvod_do_vypoctovej_geometrie/
[4] https://www.ksp.sk/kucharka/skalarny_a_vektorovy_sucin/



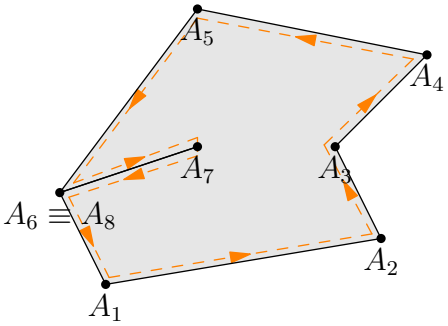
Tento prechod je umožnený tým, že sa hrany grafu nekrižujú, a že na žiadnej hrane neleží tretí bod. Ak by jedna z týchto podmienok bola porušená, mohlo by na jednej strane hrany byť viacero oblastí. Ilustrujeme na nasledujúcom obrázku:



Každá oblasť má teda svoju hranicu, ktorá obsahuje celé hrany. Pretože zadaný graf je súvislý, hranica oblasti musí byť súvislá: ak sú teda nejaké dve hrany na hranici oblasti, vieme sa od jednej hrany dostať k druhej tak, že pôjdeme po okraji. Toto v nesúvislých grafoch nemusí platiť, ako vidno na nasledujúcom obrázku:



Predstavme si teraz, že ideme po okraji oblasti proti smeru hodinových ručičiek. To samozrejme neznamená, že vždy zatáčame doľava, nakoľko oblasť nemusí byť konvexná. Ako je potom ale definovaný smer "proti smeru hodinových ručičiek"? Je to taký smer, v ktorom je vnútro oblasti vždy na ľavej strane hrany. Môžete si to predstaviť tak, že hranám na okraji oblasti priradíme takú orientáciu, aby bolo vnútro oblasti vždy naľavo od hrany. Samozrejme, sú prípady, kedy je vnútro aj na pravej strane hrany:



V takom prípade pôjdeme pri našom prechode raz v jednom smere pozdĺž takejto hrany, a raz v druhom

smere. Takže tento sled, ktorý "obtiahne" okraj oblasti, môže niektoré hrany obsahovať dvakrát: raz v jednej orientácii a raz v druhej orientácii.

Orientované hrany budeme v ďalšom texte volať *šípky*. Budeme hovoriť, že šípka patrí tej oblasti, ktorá je naľavo od šípky.

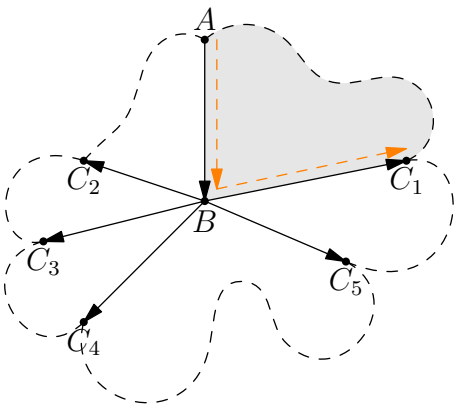
Listing programu (C++)

```
// geometry.cpp

#pragma once

#include <cmath>
using namespace std;

#define INF 1023456789
#define NONE -1
#define EPSILON 0.000000001

typedef long double ld;

bool ld_equals (ld a, ld b) {
  /** Robime s long doublami, treba byt benevolentny pri porovnavani. */
  return abs(a - b) < EPSILON;
}


struct Point {
  /** Bod/vektor v 2D rovine, so standardnymi vektorovymi operaciami. */
  ld x, y;

  Point (ld x0, ld y0) : x(x0), y(y0) {}

  // Klasicke operacie na vektoroch: sucet, rozdiel, a skalarny nasobok.
  Point operator+ (const Point p) const {
    return Point(x + p.x, y + p.y);
  }
  Point operator- (const Point p) const {
    return Point(x - p.x, y - p.y);
  }
  Point operator* (const ld k) const {
    return Point(k*x, k*y);
  }

  // Porovnavanie dvoch vektorov: mensi, vacsi, rovnost, mensi rovny, vacsi rovny.
  bool operator< (const Point p) const {
    /** Porovnavame primarne podla x-ovej, sekundarne podla y-ovej suradnice. */
    return x < p.x || (x == p.x && y < p.y);
  }
  bool operator> (const Point p) const {
    return p < (*this);
  }
  bool operator== (const Point p) const {
    return x == p.x && y == p.y;
  }
  bool operator<= (const Point p) const {
    return !((*this) > p);
  }
  bool operator>= (const Point p) const {
    return !((*this) < p);
  }

  ld sp (const Point p) const {
    /** Skalarny sucin tohto vektora s vektorom <p>. */
    return x*p.x + y*p.y;
  }
  ld vp (const Point p) const {
    /** Vektorovy sucin tohto vektora s vektorom <p> (v tomto poradi). */
    return x*p.y - y*p.x;
  }
};

Point operator* (const ld k, const Point p) {
  return p*k;
}


ld comp_time = -INF;    // v ktorom case (x-ovej pozicii) porovnavame?
bool comp_epsilon = 0;  // blizime sa zlava (0) alebo ideme doprava (1)?

struct Edge {
  /** Obsahuje informacie o tom, medzi ktorymi dvomi bodmi ide hrana
   * a aka stena je na lavej a pravej strane. */
  Point from, to;
  int lface, rface;

  Edge (Point p, bool epsilon) : from(p), to(p.x + 1, p.y + (epsilon ? INF : -INF)), lface(NONE), rface(NONE)
  {
    /** Ked chceme bod porovnat s hranou, musime to spravit pomocou
     * pomocnej hrany. Na to pouzijeme tento konstruktor, kde <epsilon>
     * hovori, ci sa k suradnici blizime zlava (0), alebo sa od nej
     * vzdalujeme doprava (1). */
  }
  Edge (Point from0, Point to0, int lface0, int rface0)
    : from(from0), to(to0), lface(lface0), rface(rface0) {}
```

```
  Point to_vector () const {
    return to - from;
  }

  bool is_vertical () const {
    return from.x == to.x;
  }
  bool contains (Point p) const {
    /** Lezi bod <p> na tejto hrane? */
    return (p - from).vp(p - to) == 0 && (p - from).sp(p - to) <= 0;
  }

  // Porovnavacie operatory.
  bool operator== (const Edge e) const {
    return (from == e.from) && (to == e.to);
  }
  bool operator!= (const Edge e) const {
    return !(*this == e);
  }

  ld at (ld x) const {
    /** Vrati y-suradnicu, ktoru hrana nadobuda na danej x-suradnici.
     * Ak je hrana vertikalna, spravi nieco nedefinovane. */
    ld p_from = x - from.x;
    ld p_to = to.x - x;
    ld p_total = to.x - from.x;
    return p_from/p_total * to.y + p_to/p_total * from.y;
  }
};

bool comp (const Edge a, const Edge b) {
  /** Porovna hrany ako vektory ("Je <a> mensi (napravo od) <b>?"). */
  return a.to_vector().vp(b.to_vector()) > 0; // ak je <b> nalavo od <a>
}

bool operator< (const Edge a, const Edge b) {
  /** Porovnanie dvoch hran, na x-ovej suradnici <comp_time>, pricom sa
   * blizime/vzdalujeme podla <comp_epsilon>. */
  ld ay = a.at(comp_time);
  ld by = b.at(comp_time);
  return (ld_equals(ay, by) ? (comp_epsilon ? comp(a, b) : comp(b, a)) : ay < by);
}

bool operator> (const Edge a, const Edge b) {
  return b < a;
}

bool operator<= (const Edge a, const Edge b) {
  return !(b < a);
}

bool operator>= (const Edge a, const Edge b) {
  return b <= a;
}


struct Arrow {
  /** Hrana z pohladu jedneho z jej koncovych bodov. */
  Edge* edge;
  bool reversed;

  Arrow (Edge* edge0, bool r0) : edge(edge0), reversed(r0) {}

  // Ktory region sa nachadza nalavo/napravo?
  int lface () const {
    return (reversed ? edge->rface : edge->lface);
  }
  int rface () const {
    return (reversed ? edge->lface : edge->rface);
  }

  Point from () const {
    return (reversed ? edge->to : edge->from);
  }
  Point to () const {
    return (reversed ? edge->from : edge->to);
  }
  Point to_vector () const {
    return to() - from();
  }

  bool operator< (const Arrow other) const {
    return to_vector().vp(other.to_vector()) > 0; // ak som nalavo od other
  }
  bool is_vertical () const {
    return edge->is_vertical();
  }
};
```

Ktoré šípky sú na okraji?

Predstavme si, že pri obťahovaní okraju sme sa zastavili na nejakej šípke $A \to B$. Vieme povedať, ktorá šípka bude nasledovať? Zrejme to bude niektorá šípka vychádzajúca z vrcholu B. Ak si nakreslíme všetky šípky vychádzajúce z B, nie je ťažké vidieť, že hľadaná šípka bude $B \to C$ taká, pri ktorej zatočíme čo najviac doľava:



Prečo je toto tá správna šípka? Totiž, vždy vieme ísť tesne popri šípke $A \to B$ a potom popri $B \to C$. Pri tomto prechode neprekrížime žiadnu inú šípku, a teda určite patria tieto šipky do tej istej oblasti. Ak by sme skúšali prejsť podobným spôsobom popri iných šípkach vychádzajúcich z vrcholu B, križovali by sme niektorú inú šípku.

Takýmto spôsobom vieme pre každú šípku zistiť celú hranicu oblasti, do ktorej patrí. Pretože každá oblasť má na okraji aspoň jednu šípku, vieme týmto spôsobom získať hranice všetkých oblastí. A pretože šípka je len hrana s priradenou orientáciou, získame týmto pre každú hranu informáciu o tom, ktorá oblasť je na jednej strane a ktorá na druhej strane hrany.

Na celý tento proces sa ale dá pozerať aj iným, prirodzenejším, spôsobom. Pre každú šípku máme dve premenné: číslo oblasti, ktorá je na ľavej strane hrany a číslo oblasti, ktorá je na pravej strane. Pritom pre dvojicu navzájom opačných šípok sú tieto dve premenné iba vymenené – to, čo je naľavo od jednej šípky, je napravo od druhej, a naopak.

Zoberme si teraz všetky šípky vychádzajúce z nejakého bodu A a usporiadajme si ich polárne (podľa uhlu), proti smeru hodinových ručičiek. Potom pre každú dvojicu po sebe idúcich šípok vieme povedať, že oblasť naľavo od prvej z nich je tá istá, ako oblasť napravo od druhej z nich – teda že sa príslušné premenné rovnajú.

Toto vieme spraviť pre všetky vrcholy, a dostaneme tak sadu rovností nad $2m$ premennými. Premenné, ktoré sa rovnajú, potom reprezentujú tú istú oblasť. Premenné, ktoré sa nerovnajú, reprezentujú rôzne oblasti. Ak si premenné predstavíme ako vrcholy nejakého grafu a rovnosti ako hrany grafu, tak oblasti zodpovedajú súvislým komponentom tohto grafu.

Dostávame tak jednoduchý algoritmus na výpočet okrajov oblastí, bežiaci v čase $O(m \log m)$, logaritmus kvôli polárnemu usporiadaniu. Aby sme vedeli šípky polárne usporiadať, potrebujeme vedieť pre dva vektory povedať, ktorý je naľavo od ktorého. To vieme určiť podľa znamienka vektorového súčinu.

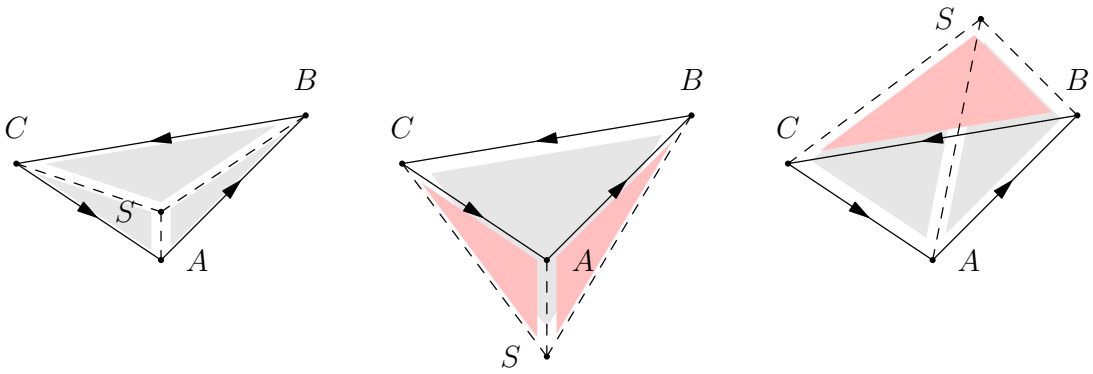
Ako vypočítať obsah oblasti?

Ak poznáme hranicu oblasti, vieme aj vypočítať obsah oblasti, nasledovne. Zoberme si ľubovoľný bod v rovine S, napríklad $S = (0, 0)$. Hranica oblasti zodpovedá nejakému sledu $A_1 \to A_2 \to \ldots \to A_n \to A_1$, proti smeru hodinových ručičiek. Jednoducho teraz vezmeme orientované obsahy trojuholníkov určených dvojicami vektorov

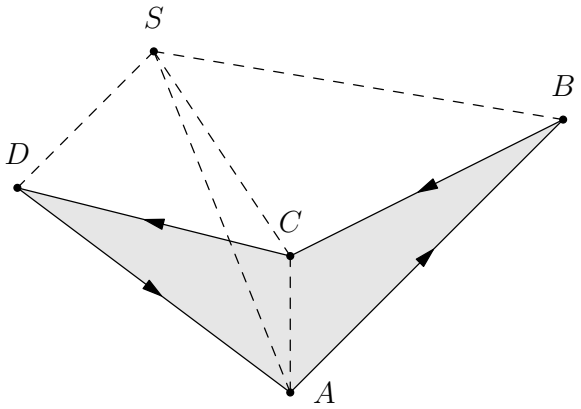
$$(\overrightarrow{SA_1}, \overrightarrow{SA_2}), (\overrightarrow{SA_2}, \overrightarrow{SA_3}), \ldots, (\overrightarrow{SA_n}, \overrightarrow{SA_1})$$

a sčítame ich. Tieto orientované obsahy vieme dostať pomocou vektorového súčinu.

Prečo to ale funguje? Zamyslime sa nad najjednoduchším prípadom, keď máme trojuholník ABC. Potom náš výraz je súčtom orientovaných obsahov troch trojuholníkov, jeden pre každú stranu $\triangle ABC$. Ak je bod S vo vnútri $\triangle ABC$, tak jednotlivé podtrojuholníky budú mať všetky kladný obsah, nakoľko $\overrightarrow{SA_{i+1}}$ je vždy naľavo od $\overrightarrow{SA_i}$. Ich súčtom dostaneme celý trojuholník. Ak je bod S mimo $\triangle ABC$, tak jeden alebo dva podtrojuholníky budú záporné, a všetko sa to vyruší tak, že zostane iba $\triangle ABC$.



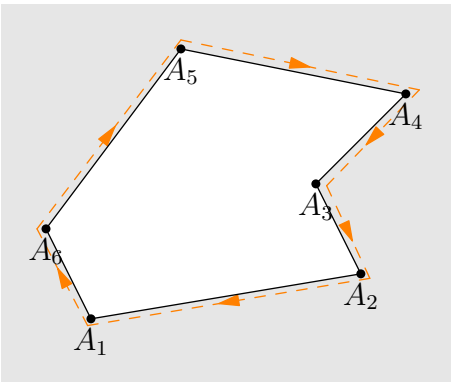
Čo v prípade, keď nemáme trojuholník? Každý (aj nekonvexný) mnohouholník sa dá rozdeliť na niekoľko dizjunktných trojuholníkov (tzv. *triangulácia*). Na každý z týchto trojuholníkov vieme aplikovať vyššie uvedené tvrdenie, a dostaneme tak obsah nášho mnohouholníka vyjadreného ako súčet niekoľkých orientovaných trojuholníkov. Každý z týchto trojuholníkov má ako jeden z vrcholov S, a ostatné dva vrcholy patria mnohouholníku – teda každý trojuholník zodpovedá buď niektorej hrane v triangulácii (ktorá oddeľuje dva z trojuholníkov), alebo hrane na obvode mnohouholníka.



$$\square ABCD = \triangle ABC + \triangle CDA$$
$$\triangle ABC = \triangle SAB + \triangle SBC + \triangle SCA$$
$$\triangle CDA = \triangle SCD + \triangle SDA + \triangle SAC$$

My chceme ukázať, že v skutočnosti tam budú vystupovať iba hrany z obvodu mnohouholníka. To je ale zrejmé: každá vnútorná hrana susedí s dvomi podtrojuholníkmi: v jednom z nich bude vnútro trojuholníka naľavo od hrany (a zarátame ho s kladným znamienkom), v druhom z nich bude vnútro trojuholníka napravo od hrany (záporné znamienko).

Pozorný čitateľ si isto všimne jeden zvláštny prípad. Čo sa stane, ak tá oblasť, po okraji ktorej ideme, je nekonečná? Zrejme vypočítame obsah útvaru určeného týmto okrajom (tj. obsah toho, čo je "skutočne vnútri", nie vonku). Aké znamienko ale bude mať takto vypočítaný obsah? Definovali sme "proti smeru hodinových ručičiek" tak, že vnútro oblasti je naľavo od šípky. Nie je ťažké rozmyslieť si, že pre (jedinú) nekonečnú oblasť je takto definovaný sled v skutočnosti **v smere** hodinových ručičiek:



Dostaneme teda záporný alebo nulový obsah. Pomocou znamienka teda vieme určiť, či je bod vo vnútri nekonečnej oblasti, alebo v konečnej.

Časová zložitosť výpočtu obsahu oblasti je $O(k)$, kde k je počet vrcholov/hrán na okraji oblasti. Nakoľko každá šípka patrí iba do jednej oblasti, a všetkých šípok je $2m$, celková časová zložitosť tohto predrátania je $O(m)$.

Listing programu (C++)

```
// graph.cpp

#pragma once

#include <vector>
using namespace std;


#define NONE -1

struct Graph {
  /** Vieme pridavat do grafu hrany a vieme potom spocitat jednotlive
   * komponenty grafu (tj. do ktoreho komponentu ktory vrchol patri). */
  int n = 0, num_comps = 0;
  vector<vector<int> > adj;
  vector<int> comps;

  void resize (int n1) {
    /** Zvacsi graf, aby v nom bolo <n1> vrcholov. */
    if (n1 > n) {
      adj.resize(n1);
      n = n1;
    }
  }
  void add_edge (int a, int b) {
    resize(1 + max(a, b));
    adj[a].push_back(b);
    adj[b].push_back(a);
  }
  int component_of (int v) {
    return comps[v];
  }

  void calc_comps () {
    /** Spocita pre kazdy vrchol cislo komponentu, do ktoreho patri.
     * Malo by byt zavolane len raz po tom, co je graf finalny. */
    comps.resize(n, NONE);
    vector<int> stack;
    for (int i = 0; i < n; i++) {
      if (comps[i] != NONE) {
        continue;
      }
      comps[i] = num_comps;
      stack.push_back(i);
      while (!stack.empty()) {
        int u = stack.back();
        stack.pop_back();
        for (int v : adj[u]) {
          if (comps[v] != NONE) {
            continue;
          }
          comps[v] = num_comps;
          stack.push_back(v);
        }
      }
      num_comps++;
    }
  }
};

#undef NONE
```

Listing programu (C++)

```
// regions.cpp

#pragma once

#include <vector>
using namespace std;

#include "geometry.cpp"
#include "graph.cpp"

void polar_sort (vector<Arrow>& arrows) {
  /** Pre kazdy vrchol polarne usporiadame sipky z neho vychadzajuce. */
  vector<Arrow> left, right;
  for (Arrow a : arrows) {
    if (a.to_vector() < Point(0, 0)) {
      left.push_back(a);
    }
    else {
      right.push_back(a);
    }
```

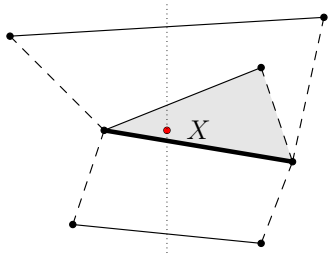
```
  }
  sort(left.begin(), left.end());
  sort(right.begin(), right.end());
  arrows.clear();
  for (Arrow a : left) {
    arrows.push_back(a);
  }
  for (Arrow a : right) {
    arrows.push_back(a);
  }
}

void calc_regions (vector<vector<Arrow> >& adj, vector<Edge*>& E,
  vector<vector<Arrow> >& borders, vector<ld>& areas)
{
  /** Pre kazdu hranu zistime, ktory region je napravo/nalavo. Dalej
   * ktore sipky tvoria hranicu jednotlivych regionov. Podla toho
   * spocitame obsahy regionov.
   *    <adj>: pre kazdy vrchol dostaneme sipky veduce z neho
   *    <E>: zoznam vsetkych hran (nie sipok!)
   *    <borders>: tu ulozime pre kazdy region sipky na jeho hranici
   *    <areas>: tu ulozime pre kazdy region jeho obsah
   * */
  int n = adj.size();

  // Zistime pre kazdu hranu, ktory region je nalavo/napravo.
  for (auto& edges : adj) {
    polar_sort(edges);
  }
  Graph G;
  for (int i = 0; i < n; i++) {
    int num_adj = adj[i].size();
    for (int j = 0; j < num_adj; j++) {
      int nj = (j+1) % num_adj;
      int side1 = adj[i][j].lface();
      int side2 = adj[i][nj].rface();
      G.add_edge(side1, side2);
    }
  }
  G.calc_comps();
  for (Edge* e : E) {
    e->lface = G.component_of(e->lface);
    e->rface = G.component_of(e->rface);
  }

  // Pre kazdy region zistime, ktore sipky tvoria jeho hranicu (v nejakom
  // poradi, pricom vnutro je vzdy nalavo od sipky). Na zaklade toho
  // potom spocitame obsahy regionov.
  borders.resize(G.num_comps);
  for (Edge* e : E) {
    borders[e->lface].push_back(Arrow(e, 0));
    borders[e->rface].push_back(Arrow(e, 1));
  }
  areas.resize(G.num_comps);
  for (int i = 0; i < G.num_comps; i++) {
    for (Arrow a : borders[i]) {
      areas[i] += a.from().vp(a.to());
    }
  }
}
```

Prvé riešenie

Už teda vieme, ako nájsť hranice všetkých oblastí, a tiež ako podľa týchto hraníc vypočítať obsah oblasti. Aby sme ale vedeli odpovedať na otázky, musíme ešte vedieť pre ľubovoľný bod X povedať, v ktorej oblasti leží, prípadne, že leží na niektorej hrane.

Zistiť, či bod leží na hrane, vieme v čase $O(m)$: iba postupne prejdeme všetky hrany, a pre každú hranou overíme, či na nej X leží alebo nie. Ako to ale overíme? Pomocou vektorového súčinu vieme zistiť, či X leží na priamke určenej touto hranou. Pomocou skalárneho súčinu zistíme, či je X "vo vnútri" úsečky alebo mimo nej.

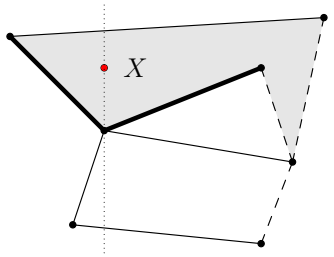
Ak neleží na hrane, ako zistíme, do ktorej oblasti patrí? Vieme sa pozrieť na najbližšiu hranu, ktorá je "pod" naším bodom. Hľadaná oblasť je tá, ktorá je "hore" od tejto hrany. Ak žiadna hrana pod bodom X nie je, tak bod zrejme neleží v žiadnej konečnej oblasti.



Čo to ale znamená hore? Ak každú (nezvislú) hranu orientujeme tak, že ide zľava doprava (tj. od menších x-ových súradníc k väčším), tak "hore" znamená "vľavo". Budeme hovoriť, že ľavý koniec hrany je jej *začiatok*

a pravý koniec hrany je jej *koniec*.

Potrebujeme teda zistiť, ktoré hrany majú spoločnú x-ovú súradnicu s naším bodom X. (Ak by sme nakreslili zvislú čiaru cez X, tak sú to tie hrany, ktoré majú s touto čiarou nejaký spoločný bod.) Spomedzi týchto hrán chceme nájsť tú, ktorá pretína zvislú čiaru cez X čo najvyššie, ale stále pod bodom X. Čo ale v prípade, že takýchto najvyšších hrán je viac? Takýto prípad môže nastať jedine vtedy, ak je bod X priamo nad niektorým vrcholom. Ako ukazuje nasledujúci obrázok, nie je jedno, ktorú hranu vyhlásime za najvyššiu:



V takom prípade chceme vybrať takú hranu, ktorá v tom vrchole začína (končí) a ktorá ide čo najrýchlejšie hore (dole). Inak povedané, spomedzi všetkých hrán začínajúcich (končiacich) v tomto vrchole chceme vybrať tú najľavejšiu (najpravejšiu). Porovnať dve hrany, že ktorá je vľavo (vpravo) od ktorej, vieme jednoducho pomocou vektorového súčinu.

Listing programu (C++)

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

// V riadnych projektoch prosim neincludujte .cpp, ale len header (.hpp)
// subory. Tu pre jednoduchost includujeme .cpp subory.
#include "geometry.cpp"
#include "regions.cpp"
#include "bst.cpp"


#define NONE -1


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

  vector<Point> P;
  P.reserve(n);
  for (int i = 0; i < n; i++) {
    int x, y;
    cin >> x >> y;
    P.push_back(Point(x, y));
  }

  vector<Edge*> E;
  E.reserve(m);
  vector<vector<Arrow> > adj(n);  // adj[i] := sipky vychadzajuce z vrcholu i
  for (int i = 0; i < m; i++) {
    int a, b;
    cin >> a >> b;
    a--; b--;
    if (P[b] < P[a]) {  // hrana vzdy ide od mensieho bodu do vacsieho
      swap(a, b);
    }
    Edge* e = new Edge(P[a], P[b], 2*i, 2*i+1);
    E.push_back(e);
    adj[a].push_back(Arrow(e, 0));
    adj[b].push_back(Arrow(e, 1));
  }

  // spocitame si regiony
  vector<vector<Arrow> > borders;
  vector<ld> areas;
  calc_regions(adj, E, borders, areas);

  // Zodpovieme otazky!
  for (int quest = 0; quest < q; quest++) {
    int x, y;
    cin >> x >> y;
    Point p(x, y);

    // Prejdeme vsetky hrany a overime, ci nas bod lezi na niektorej.
    // Ak ano, odpoved je -1.
    bool contained = false;
    for (Edge* e : E) {
      if (e->contains(p)) {
        contained = true;
        break;
      }
    }
```

```
    if (contained) {
      cout << "-1" << endl;
      continue;
    }

    // Najdeme najblizsiu hranu ktora je pod nasim bodom, a podla nej
    // cislo regionu nad nou. Podla toho zodpovieme otazku.
    comp_time = x;
    comp_epsilon = 1;
    Edge temp(p, 1);
    Edge* lb = NULL;
    for (Edge* e : E) {
      if (e->to.x <= x || e->from.x > x) { // toto automaticky eliminuje zvisle hrany
        continue;
      }
      if (*e <= temp) {
        if (lb == NULL || *e > *lb) {
          lb = e;
        }
      }
    }
    if (lb == NULL) {
      cout << "-1" << endl;
      continue;
    }
    int face = lb->lface;
    if (areas[face] <= 0) {
      cout << "-1" << endl;
      continue;
    }
    cout << (long long)(areas[face] / 2) << ((long long)areas[face] % 2 ? ".5" : "") << endl;
  }

  return 0;
}
```

Máme tak algoritmus s časovou zložitosťou $O(m)$ na jednu otázku, celková časová zložitosť je teda $O(m \log m + mq)$. Toto riešenie si vyslúžilo 5 bodov z testovania.

Cesta k lepšiemu riešeniu

Potrebujeme urýchliť dve veci: vedieť rýchlo zistiť, či náš bod X leží na niektorej hrane; a vedieť rýchlo nájsť najbližšiu hranu pod naším bodom.

Rozdeľme si prácu trochu inak: ak by sme za hrany pod naším bodom považovali aj tie, ktoré cez nehoprechádzajú, stačilo by nám v prvej časti overovať iba zvislé hrany. Ak totiž X leží na nezvislej hrane, bude to najvyššia hrana pod X. Nezvislé hrany teda vieme vybaviť tak, že pre najvyššiu hranu pod X overíme, či na nej X leží alebo nie.

Ako zistiť, či náš bod leží na nejakej zvislej hrane? Jednoducho: spomedzi vrcholov s rovnakou x-ovou súradnicou nájdeme ten vrchol A, ktorý je najvyššie, ale stále nižšie, ako X. Ak náš bod leží na zvislej hrane, musí to byť zvislá hrana, ktorá vychádza z A a ide "hore". Stačí si teda usporiadať všetky vrcholy primárne podľa x-ovej, sekundárne podľa y-ovej súradnice, a pre každý vrchol si zapamätať (najviac jednu) šípku hore, ktorá z neho vychádza. V tomto zozname potom binárne vyhľadáme najväčší bod menší rovný X a overíme, či X leží na šípke hore alebo nie.

Ako ale rýchlo nájsť tú hranu, ktorá je pod X? Ak by sme mohli riešiť úlohu offline, teda ak by sme vedeli dopredu všetky otázky, vedeli by sme použiť prístup zvaný *zametanie*.

Offline zametanie

Ak by sme mali k dispozícii všetky otázky dopredu, mohli by sme sa rozhodnúť ich spracovať v nejakom konkrétnom poradí. Napríklad si môžeme všimnúť, že keď bod X iba trochu posunieme doprava alebo doľava, množina hrán, ktoré sú pod alebo nad X, sa zmení iba trochu. V probléme je istá *lokálnosť*. Dobré poradie, v ktorom spracovať otázky, je potom napríklad zľava doprava.

Predstavme si teda, že nekonečne vľavo máme zvislú priamku, tzv. *zametaciu čiaru*, ktorá sa posúva doprava. Táto čiara nám postupne bude "skenovať" všetky hrany a všetky otázky. Keď čiara narazí na začiatok alebo koniec hrany, upraví svoj stav (tj. ktoré hrany ju pretínajú), a keď čiara narazí na otázku, na základe svojho stavu ju zodpovie.

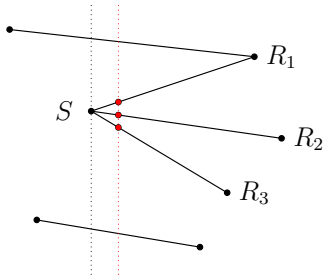
Konkrétnejšie, keď čiara narazí na ľavý koniec hrany, túto hranu pridá do množiny hrán, s ktorými máme aktuálne priesečník. Keď čiara narazí na pravý koniec hrany, danú hranu z množiny vyhodí. A nakoniec, keď narazí na bod X reprezentujúci otázku, nájde spomedzi všetkých aktuálnych hrán tú, ktorá má na tejto x-ovej súradnici priesečník čo najvyššie, ale stále nižšie alebo rovnako vysoko, ako je X.

Zametacia čiara nám teda musí byť schopná povedať, ktorá hrana je na aktuálnej x-ovej súradnici "tesne pod" bodom X. Tiež musíme vedieť do nej pridávať a odoberať hrany. Vhodnou dátovou štruktúrou na toto je vyvažovaný vyhľadávací strom, v ktorom budeme mať hrany usporiadané podľa ich aktuálnej y-ovej súradnice (tj. v akej výške pretínajú zametaciu čiaru).

Môžeme si ho ale dovoliť použiť? V našom prípade je totiž výsledok porovnania dvoch hrán závislý od toho, aká je x-ová pozícia zametacej čiary. Nemôže sa počas posúvania čiary stať, že sa výsledky niektorých porovnaní zmenia, a štruktúra stromu už nebude korektná? Dve hrany zmenia svoje relatívne poradie jedine vtedy, keď sa krížia. To sa ale v našom planárnom grafe nikdy nestane: pred tým, než by k tomu došlo, by jedna alebo obe hrany skončili v niektorom (možno v tom istom) vrchole.

Môže sa ale stať, že dve hrany skončia v jednom a tom istom vrchole; alebo že jedna hrana skončí vo vrchole, v ktorom druhá hrana začne; alebo že obe hrany začnú v tom istom vrchole. V tom momente ich nie sme schopní porovnať, lebo ich priesečníky so zametacou čiarou budú ten istý bod. Pritom ale nie je jedno, ktorú z tých hrán prehlásime za väčšiu a ktorú za menšiu.

Ak totiž dve hrany začínajú v tom istom vrchole, v tom momente ich síce nevieme porovnať, ale hneď v nasledujúcom momente (tj. keď sa zametacia čiara posunie o máličko doprava) ich budeme vedieť porovnať: tá hrana, ktorá rastie rýchlejšie (je viac vľavo), je tá väčšia. Teda pri pridávaní nových hrán ich musíme do stromu dať podľa toho.



$SR_1 > SR_2 > SR_3$

Naopak, ak dve hrany končia v tom istom vrchole, v tom momente ich síce nevieme porovnať, ale v predchádzajúcom momente sme ich vedeli porovnať: tá hrana, ktorá klesá rýchlejšie (je viac vpravo), je väčšia. Ak by sme ale toto pri mazaní hrany odignorovali, mohli by sme omylom zmazať tú nesprávnu hranu.

A čo v treťom prípade, keď jedna hrana začína v tom vrchole, v ktorom iná hrana končí? Našťastie sa tomuto prípadu vieme vyhnúť tým, že najprv vybavíme všetky hrany končiace vo vrchole, a až potom všetky hrany začínajúce vo vrchole. Týmto zaručíme, že nikdy nebudeme mať v našom vyhľadávacom strome hranu, ktorá v nejakom vrchole začína spolu s hranou, ktorá v tomto vrchole končí.

Takže naše porovnávanie bude závisieť nielen od toho, aká je momentálna x-ová súradnica zametacej čiary, ale aj od toho, či sa k nej blížime zľava (vybavujeme konce hrán) alebo sa od nej vzďaľujeme doprava (vybavujeme začiatky hrán). Týmto sme vybavili všetky možné prípady.

Drobný technický detail: vo vyhľadávacom strome vieme porovnávať iba dve hrany, nevieme porovnať bod a hranu. Nebudeme teda porovnávať priamo s bodom X, ale s akousi pomocnou hranou, ktorá bude prechádzať cez X. Aký bude jej sklon? Nemôže byť ľubovoľný: chceme totiž, aby všetky hrany prechádzajúce cez X boli menšie, aby sme vedeli detekovať, že X leží na hrane. Takže podľa toho, či sa zľava blížime do x, alebo sa od neho sprava vzďaľujeme, nastavíme sklon hrany na "takmer zvislo hore" alebo "takmer zvislo dole".

Na začiatku musíme všetky začiatky, konce hrán a otázky utriediť, čo trvá $O((m+q)\log(m+q))$. Samotné zametanie trvá rovnako dlho, nakoľko v každom kroku robíme jednu operáciu s vyvažovaným vyhľadávacím stromom (`insert`, `delete` alebo `lower_bound`), ktorá trvá $O(\log(m+q))$. Celková časová zložitosť je teda $O((m+q)\log(m+q))$.

Toto riešenie si samozrejme vyslúžilo 0 bodov, so správou `TLE` – čakalo totiž na všetky otázky, ktoré mu testovač nebol ochotný dať...

Perzistentný stav čiary

Ak by sme sa vedeli pozrieť na stav zametacej čiary v ľubovoľnom momente, nemuseli by sme všetky otázky zodpovedať od najľavejšej po najpravejšiu. Stačilo by sa pre každú otázku (x, y) pozrieť na stav zametacej čiary v momente x, a podľa neho odpovedať.

Potrebujeme si teda zapamätať celú históriu zametacej čiary a vedieť v nej vyhľadať posledný stav, ktorý predchádzal otázke X. Takýmto "historickým" dátovým štruktúram (ktoré si vedia pamätať nielen ich aktuálny stav, ale aj všetky predchádzajúce) sa hovorí *perzistentné dátové štruktúry.*

Jedna možnosť je zakaždým, keď chceme zmeniť stav čiary, celý stav skopírovať a vykonať zmenu v kópii. Tieto stavy si potom uložíme do poľa, a pre každý stav si budeme pamätať, v akom časovom momente začal. V tomto poli potom budeme pri otázke (x, y) vedieť binárne vyhľadať poslednú verziu pred (alebo zarovno s) x.

Toto je ale pomalé: pri každej udalosti musíme skopírovať celý strom, čo trvá až $O(m)$. Udalostí je tiež $O(m)$, predpočítanie teda bude trvať až $O(m^2)$, a pamäťová zložitosť bude rovnaká.

My si ukážeme dve prístupy, ktoré dosahujú lepšiu časovú zložitosť: *fat nodes* a *path copying*.

Fat nodes

Pri programovaní používame rôzne objekty, a každý objekt si vieme predstaviť ako zoskupenie niekoľkých údajov, nie nutne rovnakých typov. Napríklad vrchol vo vyhľadávacom strome môže obsahovať tieto údaje: ukazovateľ na ľavého syna, pravého syna, a čo je vo vrchole.

Štandardne, keď zmeníme jeden z týchto údajov na iný, stratíme pôvodnú hodnotu. Nevieme ale zmeny údajov spracovať iným spôsobom, tak, aby sa nám zachovala aj pôvodná informácia? Vieme. Budeme si pamätať všetky doterajšie verzie údaju, a pre každú verziu aj čas, kedy táto verzia vznikla (tj. kedy nahradila starú verziu). Keď teraz chceme pristúpiť k údaju v (diskrétnom) čase t, iba binárne vyhľadáme, akú hodnotu v tom čase mal, a vrátime ju. Toto spôsobí iba mierne spomalenie: konkrétne, všetko bude $O(\log t)$-krát pomalšie, kde t je počet verzii údaju. Pamäť bude ale až $O(t)$ namiesto $O(1)$.

Ak náš vyhľadávací strom implementujeme týmto spôsobom, dostaneme perzistentný vyhľadávací strom. Samozrejme, potrebujeme vyvažovaný vyhľadávací strom, napríklad treap[5].

Otázka je, o koľko pomalší bude oproti pôvodnému stromu. Udalostí (začiatok alebo koniec hrany) je $O(m)$. Pri každej udalosti sa zmení nanajvýš $O(\log m)$ vrcholov stromu, a "narazíme" pri binárnom vyhľadávaní nanajvýš na všetky doterajšie zmeny. Pred i-tou udalosťou bolo týchto zmien $O(i \log m)$, na jeden vrchol teda pripadá spomalenie $O(\log \frac{i \log m}{\log m}) = O(\log i)$.

Spomalenie jednej operácie je teda $O(\log m)$ a časová zložitosť predpočítania teda bude $O(m \log^2 m)$. Podobne, zodpovedanie jednej otázky bude o $O(\log m)$ pomalšie, a teda dokopy $O(q \log^2 m)$. Pamäťová zložitosť bude $O(m \log m)$, nakoľko každá zmena prispeje 1.

Path copying

Predstavme si, že objekty, ktoré už existujú, nevieme meniť. Jediné, čo vieme robiť, je vytvárať zo starších objektov nové. Takémuto prístupu k programovaniu sa hovorí aj *funkcionálne programovanie*.

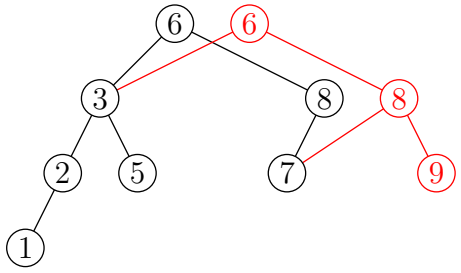
Načo také niečo je? Dá sa takto vôbec programovať? Ako to súvisí s perzistentnými dátovými štruktúrami? Hneď uvidíme.

Predstavme si, že chceme funkciu, ktorá zmení stav nejakého objektu. Napríklad vezme ako vstup zoznam čísel a na jeho koniec pridá číslo 4. Nevieme to spraviť priamo, nakoľko nevieme zmeniť stav vstupného zoznamu. Vieme ale spraviť funkciu, ktorá nám vráti nový zoznam, ktorý bude reprezentovať nový stav vstupného zoznamu. Napríklad ak vstupom je $a = [1, 2, 3]$, výstupom bude nejaké $b = [1, 2, 3, 4]$. Zoznam a ale stále obsahuje pôvodné údaje.

No a tu si môžeme uvedomiť, že čo sa v podstate stalo je, že sme síce dostali novú verziu objektu, ale máme aj starú verziu objektu. Ak by sme týmto prístupom implementovali náš vyhľadávací strom, dostali by sme perzistentný vyhľadávací strom.

Presnejšie, každý vrchol stromu reprezentuje vyhľadávací storm zodpovedajúci podstromu tohto vrcholu. Každá operácia, ktorá mení stav podstromu (`insert`, `delete` a rotácie) nám strom nezmení, ale vráti nový vrchol reprezentujúci nový stav (pod)stromu. Tieto operácie sú z veľkej časti definované pomocou operácii na synoch.

Ilustrujeme operáciu `insert` na nevyvažovanom vyhľadávacom strome nad číslami. Keď chceme vložiť číslo 5 do koreňa, chceme ho vložiť do ľavého syna, lebo 5 je menšie ako číslo v koreni. Takže výsledkom insertu bude niečo ako `strom(lavy_syn->insert(5), hodnota_v_koreni, pravy_syn)`, kde `strom` je konštruktor (vytvorí nový strom s daným ľavým synom, danou hodnotou v koreni, a s daným pravým synom). To isté sa potom udeje aj v ľavom synovi, ..., až kým nedôjdeme do listu. Dostaneme tak $O(h)$ nových vrcholov, kde h je hĺbka stromu.



Prístup sa volá *path copying* podľa toho, že v podstate pri každej zmene skopírujeme celú cestu od listu až ku koreňu, a kópiu upravíme. Samozrejme, pre efektívnosť potrebujeme strom vyvažovať, my sme (opäť) zvolili

[5] https://www.ksp.sk/kucharka/treap/

treap[6].

Listing programu (C++)

```
// random.cpp

#pragma once

#include <random>
using namespace std;

// random_device rd;
mt19937 mt(4247);


int randint (int l, int r) {
  uniform_int_distribution<int> distrib(l, r);
  return distrib(mt);
}

double randfrom (double l, double r) {
  uniform_real_distribution<double> distrib(l, r);
  return distrib(mt);
}

double randprob () {
  return randfrom(0., 1.);
}
```

Listing programu (C++)

```
// bst.cpp

#pragma once

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

#include "random.cpp"


#define NONE -1

template<class T>
struct Treap {
  /** Vrchol BSTcka (treap), obsahuje hodnoty typu T. Vrchol s hodnotou
   * NULL je specialny: reprezentuje prazdny strom. Na prazdnom strome ma
   * zmysel volat iba tie metody, ktore sa na strukturu pozeraju ako na
   * strom, nie ako na vrchol (napr. dir_to, is_leaf, ... davaju undefined
   * behaviour). */
  T* value;
  double priority;
  Treap<T>* child[2] = {NULL, NULL};

  Treap (T* x) : value(x), priority(randprob()) {}
  Treap () : Treap(NULL) {}
  Treap (Treap<T>* src, bool dir, Treap<T>* ch) {
    /** Spravime kopiu <src>, az na to, ze v smere <dir> bude zaveseny <ch>. */
    value = src->value;
    priority = src->priority;
    child[!dir] = src->child[!dir];
    child[dir] = ch;
  }

  bool empty () const {
    return value == NULL;
  }
  int size () const {
    /** Vrati velkost podstromu. */
    if (value == NULL) {
      return 0;
    }
    int res = 1;
    for (int dir = 0; dir <= 1; dir++) {
      res += (child[dir] == NULL ? 0 : child[dir]->size());
    }
    return res;
  }

  int dir_to (const Treap<T>* ch) const {
    /** Vrati smer, v ktorom je syn <ch>. Ak to nie je nas syn, vratime NONE. */
    for (int i = 0; i < 2; i++) {
      if (child[i] == ch) {
        return i;
      }
    }
    return NONE;
  }

  bool is_leaf () const {
    return (child[0] == NULL) && (child[1] == NULL);
```

[6] https://www.ksp.sk/kucharka/treap/

```
  }
  bool has_left () const {
    return child[0] != NULL;
  }
  bool has_right () const {
    return child[1] != NULL;
  }

  Treap<T>* rotate (int dir) {
    /** Spravi rotaciu v smere <dir>. Vrati vysledny podstrom, alebo
     * NULL ak rotaciu nie je mozne vykonat (lebo nema prislusneho syna). */
    Treap<T>* x = child[!dir];
    if (x == NULL) {
      return NULL;
    }
    Treap<T>* xy = x->child[dir];
    Treap<T>* new_me = new Treap<T>(this, !dir, xy);
    return new Treap<T>(x, dir, new_me);
  }

  Treap<T>* lower_bound (T* x) {
    /** Najde najvacsi mensi vrchol v tomto podstrome. Ak taky neexistuje
     * (vsetky vrcholy su vacsie ako <x>), vrati NULL. Na porovnavanie
     * prvkov pouzije <comp> (hrajuci rolu '<'). */
    if (empty()) {
      return NULL;
    }
    if (*x < *value) {
      return (has_left() ? child[0]->lower_bound(x) : NULL);
    }
    Treap<T>* r_lb = (has_right() ? child[1]->lower_bound(x) : NULL);
    return (r_lb == NULL ? this : r_lb);
  }

  Treap<T>* insert (T* x) {
    /** Vytvori novy vrchol s hodnotou <x> a vlozi ho do podstromu.
     * Vrati vysledny podstrom. */
    if (empty()) {
      return new Treap<T>(x);
    }
    int dir = (*value < *x);
    Treap<T>* ch = (child[dir] == NULL ? new Treap<T>(x) : child[dir]->insert(x));
    Treap<T>* res = new Treap<T>(this, dir, ch);
    return (priority > ch->priority ? res : res->rotate(!dir));
  }

  Treap<T>* remove (T* x) {
    /** Zmaze vrchol s hodnotou <x> z podstromu ak existuje. Prerotuje,
     * aby bol treap spokojny. Vrati vysledny podstrom. */
    if (empty()) {
      return this;
    }
    if (*value != *x) {
      int dir = (*value < *x);
      Treap<T>* ch = (child[dir] == NULL ? NULL : child[dir]->remove(x));
      ch = (ch->empty() ? NULL : ch);
      Treap<T>* res = new Treap<T>(this, dir, ch);
      return res;
    }
    // ak odstranovany vrchol je listom, easy
    if (is_leaf()) {
      return new Treap<T>();
    }
    // inak prepasujeme vrchol nadol, aby sa stal listom
    double max_p = -1.;
    int dir = -1;
    for (int i = 0; i < 2; i++) {
      if (child[i] == NULL) {
        continue;
      }
      if (child[i]->priority > max_p) {
        max_p = child[i]->priority;
        dir = i;
      }
    }
    return rotate(!dir)->remove(x);
  }
};

#undef NONE
```

Aká je časová a pamäťová zložitosť? Vždy, keď sa v najaktuálnejšej verzii stromu zmení nejaký vrchol, musíme zmeniť (skopírovať) aj všetkých jeho predkov. Pri každej operácii v štandardnom treape sa zmení vždy nanajvýš $O(\log m)$ vrcholov, z tohto by plynul (časový aj pamäťový) odhad $O(\log^2 m)$ na jednu operáciu. Avšak ukazuje sa, že všetky tieto zmeny v treape sú "zarovnané", tj. že väčšinou sa mení iba jeden vrchol a jeho predkovia. Z toho vyplynie lepší odhad $O(\log m)$.

Časová zložitosť predpočítania teda bude $O(m \log m)$, pamäťová zložitosť bude rovnaká. Zodpovedanie otázok bude trvať $O(q \log m)$. Toto je lepšie, ako v prístupe fat nodes.

Listing programu (C++)

```
#include <iostream>
```

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

// V riadnych projektoch prosim neincludujte .cpp, ale len header (.hpp)
// subory. Tu pre jednoduchost includujeme .cpp subory.
#include "geometry.cpp"
#include "regions.cpp"
#include "bst.cpp"


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

  vector<Point> P;
  P.reserve(n);
  for (int i = 0; i < n; i++) {
    int x, y;
    cin >> x >> y;
    P.push_back(Point(x, y));
  }

  vector<Edge*> E;
  E.reserve(m);
  vector<vector<Arrow> > adj(n);  // adj[i] := sipky vychadzajuce z vrcholu i
  vector<Edge*> vertical(n);      // vertical[i] := hrana vychadzajuca z i iduca hore,
                                  //                alebo NULL ak taka nie je
  for (int i = 0; i < m; i++) {
    int a, b;
    cin >> a >> b;
    a--; b--;
    if (P[b] < P[a]) {  // hrana vzdy ide od mensieho bodu do vacsieho
      swap(a, b);
    }
    Edge* e = new Edge(P[a], P[b], 2*i, 2*i+1);
    E.push_back(e);
    adj[a].push_back(Arrow(e, 0));
    adj[b].push_back(Arrow(e, 1));
    if (P[a].x == P[b].x) { // nastavime, ze <e> je vertikalna hrana z <a> hore
      vertical[a] = e;
    }
  }

  // spocitame si regiony
  vector<vector<Arrow> > borders;
  vector<ld> areas;
  calc_regions(adj, E, borders, areas);

  // Usporiadame si vrcholy od najmensieho po najvacsi, aby sme
  // potom vedeli binarne vyhladat najvacsi mensi alebo rovny bod.
  vector<int> pOrder(n);                  // pOrder[i] := ktory bod je i-ty?
  vector<pair<Point, int> > sortedP;
  sortedP.reserve(n);
  for (int i = 0; i < n; i++) {
    sortedP.push_back({P[i], i});
  }
  sort(sortedP.begin(), sortedP.end());
  for (int i = 0; i < n; i++) {
    pOrder[i] = sortedP[i].second;
  }

  // Zametanie, s perzistentnym stavom.
  vector<Treap<Edge>*> H;
  Treap<Edge>* curr = new Treap<Edge>();
  H.reserve(n+1);
  H.push_back(curr);
  for (int i : pOrder) {
    comp_time = P[i].x;
    comp_epsilon = 0;
    for (Arrow a : adj[i]) {
      if (a.is_vertical() || !a.reversed) { // zvisle hrany a hrany iduce doprava ignorujeme
        continue;
      }
      curr = curr->remove(a.edge);
    }
    comp_epsilon = 1;
    for (Arrow a : adj[i]) { // zvisle hrany a hrany vchadzajuce zlava ignorujeme
      if (a.is_vertical() || a.reversed) {
        continue;
      }
      curr = curr->insert(a.edge);
    }
    H.push_back(curr);
  }

  // Zodpovieme vsetky otazky!
  for (int quest = 0; quest < q; quest++) {
    int x, y;
    cin >> x >> y;
    Point p(x, y);

    { // Najprv zistime, ci bod lezi na niektorej zvislej hrane.
      // Na to najprv vyhladame najvacsi mensi alebo rovny bod.
      int l = -1, r = n;
      while (r-l > 1) {
        int s = (l+r) / 2;
        Point midp = P[pOrder[s]];
```

```
        if (p >= midp) {
          l = s;
        }
        else {
          r = s;
        }
      }
      // Ak existuje mensi alebo rovny bod: ak je to rovny bod, rovno
      // balime. Inak sa pozrieme sa na hranu iducu z neho hore. Lezi
      // nas bod na tejto hrane?
      if (l >= 0) {
        int i = pOrder[l];
        Point start = P[i];
        Edge* up = vertical[i];
        if (start == p || (up != NULL && up->contains(p))) {
          cout << "-1" << endl;
          continue;
        }
      }
    }

    // Vyhladame spravnu verziu treapu (poslednu, co bola pred alebo
    // zarovno s nasim bodom).
    int l = -1, r = n;
    while (r-l > 1) {
      int s = (l+r) / 2;
      Point midp = P[pOrder[s]];
      if (p >= midp) {
        l = s;
      }
      else {
        r = s;
      }
    }

    // V tejto verzii treapu najdeme lower_bound (najvacsiu hranu,
    // ktora je mensia alebo rovna ako nas bod). Podla tejto hrany
    // zodpovieme otazku.
    comp_time = x;
    comp_epsilon = 1;
    Edge temp(p, 1);
    Treap<Edge>* node = H[r]->lower_bound(&temp);
    if (node == NULL || node->value->contains(p)) {
      cout << "-1" << endl;
      continue;
    }
    int face = node->value->lface;
    if (areas[face] <= 0) {
      cout << "-1" << endl;
      continue;
    }
    cout << (long long)(areas[face] / 2) << ((long long)areas[face] % 2 ? ".5" : "") << endl;
  }

  return 0;
}
```

Záverečné poznámky

Dátovým štruktúram, ktorých stav sa nemení (ang. *immutable*) a vedia iba vracať nové objekty, sa hovorí *funkcionálne dátové štruktúry*. Ich výhodou oproti iným implementáciám perzistencie je, že sme schopní "pokračovať" nielen v aktuálnej verzii objektu, ale aj v starých verziách. Časová os teda nemusí byť os, môže to byť aj strom.

Naproti tomu ale existuje implementácia perzistentného treapu, ktorá potrebuje menej pamäte: iba $O(1)$ amortizovane na každú zmenu, oproti $O(\log m)$ v prípade funkcionálneho treapu. Pre záujemcov odporučím tento článok[7] od legiend Sleatora a Tarjana, strany 93 až 97.

[7] http://www.cs.cmu.edu/~sleator/papers/another-persistence.pdf