Korešpondenčný seminár z programovania
XXXV. ročník, 2017/18
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 1. kola zimnej časti

Zajo

1. Kopa kopania

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

Na začiatok si uvedomme nasledovnú myšlienku: *Kým bude na stavebnej ploche aspoň jedna jama a zároveň aspoň jeden kopec, určite budeme vedieť niečo ešte zarovnať.*

Inými slovami, ak je na pozemku aspoň jedna jama a aspoň jeden kopec, tak je tam aj taká jama a taký kopec, medzi ktorými nie je žiadna iná nerovnosť.

Aj keď sa takéto tvrdenie môže zdať zrejmé, skúsme ho poriadne dokázať. Predpokladajme teda, že na pozemku je ešte aspoň jedna jama a aspoň jeden kopec. Zoberme si najľavejší kopec. Ak je bezprostredne naľavo od neho jama, tak sme práve našli hľadanú dvojicu. Ak naľavo od neho nie je jama, tak to znamená, že tento kopec je prvá nerovnosť na pozemku (lebo je to najľavejší kopec). Ak teraz budeme prechádzať postupne po pozemku smerom doprava, stretávame najprv kopce (možno len ten jeden najľavejší, možno aj ďalšie), až kým nestretneme jamu (na pozemku sa aspoň jedna musí nachádzať). Táto jama je však vedľa posledného kopca, ktorý sme videli, teda aj v tomto prípade sme našli susediacu dvojicu kopec-jama.

Zistili sme, že stavebnú plochu budeme vedieť zarovnávať, až kým nám neostanú samé kopce alebo samé jamy (prípadne ani jedno). Zároveň vieme, že pri každom zarovnaní zmizne jedna jama a jeden kopec. Počet nerovností, ktorý ostane na konci, bude preto rovný rozdielu medzi počtom kopcov a počtom jám na začiatku.

Lepší výsledok sa zjavne dosiahnuť nedá – ak máme priveľa jám, môžeme ich zasypať iba toľko, koľko máme kopcov. Podobne, ak máme priveľa kopcov, môžeme sa zbaviť iba toľkých, koľko máme jám.

Na vyriešenie úlohy nám teda stačí spočítať, koľko je na pozemku jám a koľko kopcov (koľko je vo vstupnom reťazci núl a koľko jednotiek), odčítať od väčšieho čísla menšie a výsledok vypísať.

Priamočiara implementácia popísaného algoritmu v Pythone:

Listing programu (Python)

```
n = int(input())
retazec = input()

# rozdiel poctu kopcov a jam
rozdiel = 0

for i in range(n):
    # za kazdu jamu jednotku odpocitame
    if retazec[i] == '0':
        rozdiel -= 1
    # za kazdy kopec jednotku pripocitame
    else:
        rozdiel += 1

# ak bolo viac jam ako kopcov, bude rozdiel zaporny,
# preto vypiseme jeho absolutnu hodnotu
print(abs(rozdiel))
```

Už na načítanie vstupu je potrebná časová zložitosť $O(n)$ a v našom riešení len v jednom cykle prejdeme pole so vstupom, preto aj celé riešenie má zložitosť $O(n)$. Pamäťová zložitosť je, naopak, niečo, s čím pohnúť vieme. Ak načítame celý reťazec naraz, naša pamäťová zložitosť bude $O(n)$. Reťazec si ale nepotrebujeme pamätať celý naraz. Ak budeme vstup načítavať po jednom znaku, zlepšíme pamäťovú zložitosť na $O(1)$.

Tu uvádzame riešenie v jazyku C++ s konštantnou pamäťovou zložitosťou:

Listing programu (C++)

```
#include <iostream>
#include <cmath> // na absolutnu hodnotu

using namespace std;

int main() {
  // dlzka retazca a rozdiel poctu kopcov a jam
  int n, rozdiel = 0;
  cin >> n;

  for (int i = 0; i < n; i++) {
```

```
    char znak;
    cin >> znak;
    if (znak == '0') {
        // za kazdu jamu jednotku odpocitame
        rozdiel -= 1;
    } else {
        // za kazdy kopec jednotku pripocitame
        rozdiel += 1;
    }
  }
  // ak bolo viac jam ako kopcov, bude rozdiel zaporny,
  // preto vypiseme jeho absolutnu hodnotu
  cout << abs(rozdiel) << endl;
}
```

Denis a Kubo

2. Obávaná Skratka

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

Ešte pred tým, ako sa pustíme do riešenia, si zopakujeme parametre zo vstupu. Zadanú máme dĺžku účinku čarovného nápoja k, dĺžky jednotlivých záhrad $l_1, \ldots, l_n$ a počty sekúnd $t_1, \ldots, t_n$, koľko sa v jednotlivých záhradách môže Emanix zdržať.

Najskôr sa pozrieme na to, ako by sa táto úloha dala vyriešiť, ak by Emanix prechádzal len cez jednu záhradu. Následne si ukážeme, ako takéto riešenie rozšíriť na viacej záhrad.

Riešenie pre jednu záhradu

Predpokladajme, že Emanix chce prejsť cez záhradku dlhú l a môže sa v nej zdržať maximálne t sekúnd. Môžu nastať 3 prípady:

1. Emanix stihne ujsť pred diviakom bez použitia elixíru. Keďže jeho rýchlosť je $0.5\,\mathrm{m/s}$, čas ktorý sa Emanix v záhrade zdrží zrátame ako $2 \cdot l$. Tento prípad teda nastane práve vtedy, keď $2 \cdot l \leq t$.

2. Emanix nevie utiecť pred diviakom ani s použitím elixíru. Jeho rýchlosť s elixírom je $1\,\mathrm{m/s}$, takže utiecť nestihne ak $l > t$.

3. Emanix nestihne prejsť bez elixíra, ale s elixírom to stihne. Toto nastane ak $l \leq t < 2 \cdot l$

V treťom prípade nás zaujíma, koľko elixírov potrebuje Emanix vypiť. Tu nám pomôže to, že za s sekúnd s elixírom prejde Emanix rovnakú vzdialenosť ako za $2 \cdot s$ sekúnd bez elixíru. Na toto sa dá pozerať aj tak, že ak chceme, aby nejakú vzdialenosť prešiel o s sekúnd rýchlejšie, stačí aby bol s sekúnd pod vplyvom elixíru. My si teda zrátame o koľko sekúnd by diviakovi nestihol ujsť ak by nepoužil elixír ($s = 2 \cdot l - t$) a následne Emanix vypije aspoň toľko elixíru ($\lceil \frac{s}{k} \rceil$).

Viacero záhrad

Rozšírenie pre viacero záhradiek by bolo veľmi jednoduché, keby záhradky neboli nalepené jedna na druhú, lebo by sme ich mohli riešiť samostatne. Keďže však nasledujú tesne za sebou, tak elixír použitý v jednej záhradke môže posobiť aj v niekoľkých nasledujúcich, čo nám vie ušetriť niekoľko napití elixíru.

Ak pri niektorej záhradke nastane 2. prípad, Emanix nedokáže cez záhradky prejsť, teda môžeme rovno vypísať `-1`. Zamerajme sa ďalej už len na možnosť, že cez všetky záhradky Emanix stíha (možno len za pomoci elixíru) prejsť, teda vždy nastáva prípad 1. alebo prípad 3..

Budeme postupne riešiť jednotlivé záhradky od prvej až po poslednú. Vezmime si riešenie pre prvú záhradku. Ak nastane prípad 1., nič zaujímavé sa nám na probléme nezmení.

Rozoberme si teda prípad 3, kde Emanix potrebuje byť pod vplyvom elixíru s sekúnd. Keďže elixíry vieme piť len po k-sekundových dávkach, tak sa nám mohlo stať, že musíme vypiť $a > s$ elixíru. Keďže v prvej záhradke potrebuje byť Emanix pod vplyvom elixíru iba s sekúnd, môže si jeho vypitie načasovať tak, aby mu zvyšných $a - s$ sekúnd zostalo do ďalšej záhradky (inými slovami, vypiť elixír čo najneskôr ako môže). Toto sa zjavne oplatí, keďže riešenie prvej záhradky tým nijako nepokazíme a vieme len získať lepšie riešenie nasledujúcich záhradiek.

Pri riešení každej ďalšej záhradky už budeme predpokladať, že na začiatku e sekúnd už Emanix má vypitý elixír. Ak celú záhradu prejde na e elixíru, tak od zostávajúceho elixíru odrátame čas prejdenia tejto záhradky a presunieme sa na ďalšiu s novou hodnotou $e := e - l_i$ (kde l_i je dĺžka aktuálnej záhradky).

Elixír sa nám však môže minúť aj uprostred. V tomto prípade je najjednoduchšie „odrátať" od dĺžky záhrady a času diviaka časť, keď bol Emanix pod vpluvom zvyšku elixíru a ďalej záhradu riešiť ako záhradu bez počiatočného elixíru: $l_i := l_i - e;\ t_i := t_i - e$.

Pre každú záhradu teda zistíme, koľko elixírov na nej vypijeme. Už nám to len stačí sčítať (alebo počítať priebežne) a vypísať.

Zložitosť

Pre každú záhradu robíme len niekoľko operácií v konštantnom čase. Náš algoritmus teda bude mať časovú zložitosť lineárnu od počtu záhrad n, čo označujeme ako $O(n)$. Ďalej si musíme pamatať dĺžku každej záhrady a čas, ktorý v nej môžme stráviť. No a keďže si pre každú záhradku potrebujeme pamätať konštantne veľa informácie, pamäťová zložitosť bude tiež $O(n)$.

Listing programu (C++)

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

using namespace std;

int main(int argc, const char * argv[]) {
    long long n, r;
    cin >> n >> r;
    vector<long long> dlzka_zahrady(n), cas(n);

    for (int i = 0; i < n; i++)
        cin >> dlzka_zahrady[i];
    for (int i = 0; i < n; i++)
        cin >> cas[i];
    for (int i = 0; i < n; i++) {
        if (dlzka_zahrady[i] > cas[i]) {
            cout << "-1\n";
            return 0;
        }
    }

    long long timer = 0, zostavajuci_ucinok_elixiru = 0, celkovy_pocet_pouzitych_elixirov = 0;

    for (int i = 0; i < n; i++) {
        if (zostavajuci_ucinok_elixiru > 0) {
            long long zrychleny = min(zostavajuci_ucinok_elixiru,
                                      dlzka_zahrady[i]);

            zostavajuci_ucinok_elixiru -= zrychleny;
            dlzka_zahrady[i] -= zrychleny;
            cas[i] -= zrychleny;
            timer += zrychleny;
        }

        if (dlzka_zahrady[i] == 0 || 2 * dlzka_zahrady[i] <= cas[i])
            continue;

        long long m = 2 * dlzka_zahrady[i] - cas[i];
        long long pocet_pozitych_elixirov = m / r + (m % r > 0);
        celkovy_pocet_pouzitych_elixirov += pocet_pozitych_elixirov;
        zostavajuci_ucinok_elixiru = pocet_pozitych_elixirov * r - m;
    }

    cout << celkovy_pocet_pouzitych_elixirov << "\n";

    return 0;
}
```

MikX

3. Liečivá alpa

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

Čo vám napadne ako prvé po prečítaní úlohy? Pravdepodobne priamočiaro vyskúšať všetky možnosti. V tomto príklade to znamená vyskúšať liečiť všetky možné podúseky bojovníkov a pre každý overiť, či po ich vyliečení nezostanú v nejakej ampulke ešte nejaké kvapky. Pre každý možný začiatok a koniec úseku spočítame, koľko kvapiek alpy na tomto úseku minieme. Ak je tento počet deliteľný veľkosťou jednej alpulky a, potom sa daný úsek dá vyliečiť bez plytvania alpou. Zapamätáme si a vypíšeme dĺžku najdlhšieho vyliečiteľného úseku. Easy!

Asi tušíte, že toto nebude vzorové riešenie. Hlavne kvôli svojej časovej zložitosti $O(n^3)$ - máme totiž $O(n^2)$ možných podúsekov a pre každý z nich potrebujeme sčítať všetky čísla v ňom (ktorých je $O(n)$).

Šikovnou implementáciou (alebo aj bezduchým použitím prefixových súm) sa dá časová zložitosť tohto algoritmu zlepšiť na $O(n^2)$. To je však stále pomalé. Pre počet bojovníkov 100 000 už počítač jednoducho nebude stíhať.

Prefixové súčty

Predstavme si, že by sme si nad zraneniami bojovníkov spočítali prefixové súčty[1]. Súčet čísel medzi i-tym a j-tym bojovníkom (vrátane) potom vieme vypočítať ako rozdiel j-teho a $i-1$-vého prefixového súčtu.

Ak je počet zranení na nejakom úseku deliteľný veľkosťou ampulky a, potom musia mať príslušné dva prefixové súčty rovnaký zvyšok po delení a (lebo ich rozdiel je deliteľný a). Toto platí aj obrátene: ak majú dva prefixové súčty rovnaký zvyšok po delení a, potom sa úsek bojovníkov medzi nimi dá vyliečiť bez plytvania alpou.

Aby sme teda našli najdlhší vyliečiteľný úsek bojovníkov, stačí nám nájsť dva najvzdialenejšie prefixové súčty, ktoré dávajú rovnaký zvyšok po delení a.

Už to len nakódiť

Ak si všetky prefixové súčty vymodulujeme[2] číslom a, bude nám stačiť nájsť dve rovnaké čísla, ktoré sú čo najďalej od seba.

Toto sa dá implementovať napríklad tak, že budeme prechádzať pole s vymodulovanými prefixovými súčtami zľava doprava a v ďalšom poli o veľkosti a (pretože všetky čísla budú po upravení z tohto rozsahu!), si budeme pamätať prvý výskyt každého čísla. Akonáhle narazíme na číslo, ktoré už poznamenané máme, vieme, že sme objavili podúsek dlhý od prvého výskytu až po aktuálny index. Takto vieme vypočítať správny výsledok v jednom prechode a teda v časovej zložitosti $O(n)$. Pamäťová zložitosť je $O(a)$, teda závislá od veľkosti alpy.

Listing programu (Python)

```
INFINITY = 10 ** 9 + 47

n, MOD = [int(x) for x in input().split()]
A = [int(input()) for x in range(n)]

P = [INFINITY for x in range(MOD)]
P[0] = -1

s, mx = 0, 0
for i in range(n):
        s += A[i]
        s %= MOD

        if P[s] == INFINITY:
                P[s] = i
        else:
                mx = max(mx, i - P[s])

print(mx)
```

Poznámka na záver - mega veľké alpy

Dalo by sa to riešiť aj keby a bolo veľké?[3] Áno ide to, jediná zmena by bola v pamätaní prvých výskytov. Nepoužili by sme pole, ale napr. hashmapu. Vďaka nej by čas stále ostal $O(n)$ a pamäť by bola závislá už iba od počtu bojovníkov, tiež $O(n)$.

4. Ovce v koloseu ?!

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

Predslov o priamkach

Ešte predtým, ako sa pustíme do samotných algoritmických riešení, musíme vymyslieť ako vlastne budeme reprezentovať priamku a ako potom overíme, či na nej niektorá ovca je alebo nie je.

Priamku si vieme jednoznačne určiť pomocou dvojice bodov, ktoré na nej ležia – v tejto úlohe teda všetky priamky, ktoré nás môžu zaujímať, budú definované dvoma rôznymi ovcami.

V programoch vzorových riešení sme použili nasledovný spôsob: keď máme dve ovce na súradniciach (x_1, y_1) a (x_2, y_2), priamku si vyjadríme ako prvý bod a smernicu od tohto bodu $(x_2 - x_1, y_2 - y_1) = (d_x, d_y)$. Voľne prerozprávané, smernica nám hovorí že za každý krok dĺžky d_x v kladnom smere x-ovej osi máme spraviť krok dĺžky d_y v kladnom smere y-ovej osi.

Túto smernicu ešte upravíme do nami definovaného kanonického tvaru – vydelíme d_x a d_y ich najväčším spoločným deliteľom, ak je d_x záporné vynásobíme ju -1 (teda našu smernicu chceme mať v 'kladnom smere'), a ak je $d_x = 0$ tak chceme d_y kladné.

[1] https://www.ksp.sk/kucharka/prefixove_sumy/
[2] operácia *modulo* znamená zvyšok po delení
[3] Na Orave som naozaj videl v potravinách predávať litrovú alpu ;).

Teraz bod patrí na túto priamku, ak je jeho smernica od prvého bodu rovnaká ako tá, ktorou sme si vyjadrili priamku. Teda napríklad pre ovce $(10, 15)$ a $(12, 25)$ by sme dostali smernicu $(2, 10)$, ktorú upravíme do základného tvaru $(1, 5)$ – čo voľne prerozprávané znamená, že aby niektorá ovca ležala na tejto priamke, musíme sa k nej vedieť dostať tak že začneme na súradniciach prvej ovce a pohybujeme sa tak, že za každý posun v kladnom smere x o 1 sa posunieme o 5 v kladnom smere y (posun v zápornom smere je analogický). Teraz napríklad bod $(6, -5)$ na túto priamku patrí – jeho smernica k prvému bodu je $(6 - 10, -5 - 15) = (-4, -20)$ čo sa po upravení do kanonického tvaru $(\rightarrow (-1, -5) \rightarrow (1, 5))$ rovná smernici ktorú sme prvotne vyrátali. Bod $(8, 25)$ na túto priamku nepatrí, keďže jeho smernica je $(8 - 10, 25 - 15) = (-2, 10) \rightarrow (-1, 5) \rightarrow (1, -5)$, čo sa našej smernici nerovná.

Inou možnosťou, ktorá vyžaduje trochu viac znalostí stredoškolskej geometrie, je využiť vektorový súčin – o ňom sa môžete dočítať v kuchárke KSP[4].

Riešenie hrubou silou

Už teda vieme priamku aj reprezentovať, aj overovať, či na nej leží niektorá ovca. Čo nám chýba k riešeniu? Rozmyslieť si, ktoré priamky musíme overiť.

Prvým, najprimitívnejším riešením je jednoducho vyskúšať všetky možné riešenia (dvojice priamok) – zoberieme postupne všetky štvorice oviec na vstupe, určíme prvú priamku pomocou prvých dvoch oviec vo štvorici a druhú priamku pomocou tretej a štvrtej ovce. Následne spôsobom, ktorý nám je milší, vždy overíme, či všetky ovce zo vstupu patria aspoň na jednu z týchto priamok. Ak sa nám to raz podarí, odpovieme „áno”, ak nám to ani raz nevyjde, tak také dve priamky neexistujú – vyskúšali sme predsa všetky možnosti čo pripadali do úvahy.

Štvoríc oviec je rádovo $O(n^4)$ a overiť pre každú z n oviec, či leží na aspoň jednej z nich, nám zaberie $O(n)$, teda dokopy nám to zaberie $O(n^5)$ času. Pritom nám netreba pamätať si nič viac ako súradnice všetkých oviec, čiže pamätová zložitosť je $O(n)$.

Listing programu (C++)

```
#include <iostream>
#include <algorithm> // najvacsi spolocny delitel __gcd
#include <vector> // pole ktoremu mozeme za behu menit velkost
using namespace std;

struct ovca{
    int x,y;
};

ovca zakladny_tvar(ovca vektor){          // upravi smernicu priamky do kanonickeho tvaru
    int gcd = __gcd(vektor.x,vektor.y);   // vydelime dx,dy najvacsim spolocnym delitelom
    vektor.x /= gcd;
    vektor.y /= gcd;

    if(vektor.x < 0){                     // v kanonickom tvare chceme dx kladne
        vektor.x *= -1;
        vektor.y *= -1;
    }
    else if (vektor.x == 0)               // ak je dx 0, chceme dy kladne
        vektor.y = abs(vektor.y);

    return vektor;
}


int n;
vector<ovca> stado;

int main(){
    cin >> n;
    if(n<=4){
        cout << "ANO\n";
        return 0;
    }
    stado.resize(n);

    for(int i=0;i<n;++i)
        cin >> stado[i].x >> stado[i].y;

    for(int i=0;i<n;++i){                          // budeme skusat priamku urcenu ovcami 0 a i.
        for(int j=i+1;j<n;++j){
            for(int k=j+1;k<n;++k){
                for(int l=k+1;l<n;++l){
                    ovca prva = {stado[j].x-stado[i].x,stado[j].y-stado[i].y};
                    prva = zakladny_tvar(prva);

                    ovca druha = {stado[l].x-stado[k].x,stado[l].y-stado[k].y};
                    druha = zakladny_tvar(druha);

                    for(int m=0;m<n;++m){
```

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

```
                        if(m==i || m==k) continue;

                        ovca vektor_k_prvej = {stado[m].x-stado[i].x,stado[m].y-stado[i].y};
                        vektor_k_prvej = zakladny_tvar(vektor_k_prvej);

                        ovca vektor_k_druhej = {stado[m].x-stado[k].x,stado[m].y-stado[k].y};
                        vektor_k_druhej = zakladny_tvar(vektor_k_druhej);

                        // ak sa ovca cislo m nenachadza ani na jednej z priamok, prestaneme overovat
                        //a zacneme skusat dalsiu priamku
                        if((vektor_k_prvej.x != prva.x || vektor_k_prvej.y != prva.y) &&
                           (vektor_k_druhej.x != druha.x || vektor_k_druhej.y != druha.y))
                            break;

                        if(m==n-1){   // vsetky ovce lezia na aspon jednej priamke
                            cout << "ANO\n";
                            return 0;
                        }
                    }
                }
            }
        }
    }

    cout << "NIE\n"; // ani raz sme nenasli riesenie, povieme nie
    return 0;
}
```

Riešenie polohrubou silou

Horeuvedené riešenie vieme jednoduchým pozorovaním vylepšiť. Zoberme si ľubovoľnú ovcu – pre jednoduchosť pokojne prvú zo vstupu. Ak všetky ovce sú na najviac dvoch priamkach, tak aj táto ovca je na jednej z nich. A na akej priamke môže byť? No na priamke určenej ňou a druhou ovcou, alebo ňou a treťou ovcou, ..., alebo ňou a poslednou ovcou. Postupne pre každú ovcu okrem prvej si teda určíme priamku (označme ju p) pomocou nej a prvej ovce a overíme ktoré ďalšie ovce na nej stoja.

No a teraz si stačí uvedomiť, že všetky ovce, ktoré neležia na priamke p, musia byť na jednej inej priamke q – tá bude teda určená ľubovoľnými dvomi z nich (napríklad prvými dvomi, o ktorých pri overovaní zistíme, že nepatria na priamku p). Zoberieme si teda priamku q a opäť prejdeme všetky ovce (prípadne len tie, o ktorých sme zistili, že neležia na priamke p, ak sme si ich niekam ukladali) – ak potom všetky ovce patria na nejakú priamku (p alebo q), zahlásime Denisiov úspech. Inak opäť od začiatku skúšame novú priamku p určenú prvou a ďalšiou ovcou. Ak bez úspechu vyskúšame všetky možné priamky p, odpovieme nakoniec „nie".

O koľko sme naše riešenie urýchlili? V prípade, že riešenie neexistuje, vyskúšame rádovo n priamok p – priamku určenú prvou a druhou ovcou, prvou a treťou, ... , prvou a n-tou ovcou. Zakaždým prejdeme všetkých n oviec a overíme či ležia na priamke p. Následne určíme priamku q a pre všetky (zvyšné) ovce overíme, či na nej ležia. Aj toto je v najhoršiom prípade zhruba n oviec. Keďže n krát môžeme robiť až n operácií, výsledný odhad časovej zložitosti v najhoršiom prípade je $O(n^2)$. Teraz si už musíme aj zakaždým pamätať, ktoré ovce nestoja na priamke p, ktorú práve skúšame. To nám však zaberie len $O(n)$ pamäte, pamäťová zložitosť je teda opäť $O(n)$.

Listing programu (C++)

```
#include <iostream>
#include <algorithm> // najvacsi spolocny delitel __gcd
#include <vector> // pole ktoremu mozeme za behu menit velkost
using namespace std;

struct ovca{
    int x,y;
};

ovca zakladny_tvar(ovca vektor){          // upravi smernicu priamky do kanonickeho tvaru
    int gcd = __gcd(vektor.x,vektor.y);   // vydelime dx,dy najvacsim spolocnym delitelom
    vektor.x /= gcd;
    vektor.y /= gcd;

    if(vektor.x < 0){                     // v kanonickom tvare chceme dx kladne
        vektor.x *= -1;
        vektor.y *= -1;
    }
    else if (vektor.x == 0)               // ak je dx 0, chceme dy kladne
        vektor.y = abs(vektor.y);

    return vektor;
}

int n;
vector<ovca> stado;
vector<bool> napriamke;

// oznaci vsetky ovce ktore lezia na priamke urcenej ovcami a,b.
```

```
// ak je 'prva', skusi overit ci su zvysne ovce na priamke urcenej prvymi
// dvoma ovcami ktore na nu nepatria. ak to je uz druha priamka a stale su ovce
// neleziace na priamke, uz vrati zapornu odpoved.

bool over_na_priamke(int a,int b,bool prva){
    napriamke[a] = napriamke[b] = true;

    // toto je priamka urcena ovcami a,b
    ovca vektor = {stado[b].x-stado[a].x,stado[b].y-stado[a].y};
    vektor = zakladny_tvar(vektor);

    // sem dame vsetky ovce ktore nelezia na ziadnej skusanej priamke
    vector<int> zle;

    for(int i=0;i<n;++i){
        if(i==a || i==b) continue;

        ovca tmp = {stado[i].x-stado[a].x,stado[i].y-stado[a].y};
        tmp = zakladny_tvar(tmp);

        // ak je ovca cislo i na rovnakej priamke ako a,b oznacime ju
        // inak, ak nie je oznacena ako na priamke, ju pridame do zoznamu oviec ktore niesu na ziadnej skusanej priamke
        if(vektor.x == tmp.x && vektor.y == tmp.y)
            napriamke[i] = true;
        else if(!napriamke[i]) zle.push_back(i);
    }

    // ak uz skusame druhu priamku, musime mat 0 zlych oviec
    if(!prva) return zle.empty();
    // ak mame najviac 2 ovce, nasli sme riesenie
    if(zle.size()<=2) return true;
    // skusime overit ci vsekty zvysne ovce lezia na priamke urcenej prvymi dvoma zlymi ovcami
    return over_na_priamke(zle[0],zle[1],false);
}

int main(){
    cin >> n;
    if(n<=4){
        cout << "ANO\n";
        return 0;
    }
    stado.resize(n);
    napriamke.resize(n);

    for(int i=0;i<n;++i)
        cin >> stado[i].x >> stado[i].y;

    for(int i=1;i<n;++i){              // budeme skusat priamku urcenu ovcami 0 a i.
        for(int j=0;j<n;++j)           // kazdu ovcu oznacime ako neleziacu na priamke
            napriamke[j] = false;

        if(over_na_priamke(0,i,true)){ // ak najdeme riesenie, hned povieme ano a skoncime
             cout << "ANO\n";
             return 0;
        }
    }

    cout << "NIE\n";                   // ani raz sme nenasli riesenie, povieme nie
    return 0;
}
```

Vzorové riešenie – ešte menej priamok!

Stále ešte overujeme príliš veľa priamok. K vzorovému riešeniu nás privedie pozorovanie podobné tomu, ktoré sme spravili v predošlom riešení. Zoberme si ľubovoľné tri ovce – pre jednoduchosť prvé tri zo vstupu. Predpokladajme, že ovce naozaj stoja na najviac dvoch priamkach a predstavme si, že pre každú ovcu si zaznačíme či leží na prvej alebo na druhej (alebo na oboch, toto však nerobí rozdiel), a pozrime sa na naše tri ovce. Sú len dve možnosti – buď všetky tri naše ovce ležia na spoločnej priamke (ktorá je jedna z dvoch, na ktorých ležia všetky ovce), alebo dve ovce ležia na jednej z našich dvoch priamok a tretia leží na druhej. V oboch prípadoch platí, že na jednej z priamok stoja aspoň dve z našich troch oviec. Povedané inak, nemôže sa stať, že by existovali dve priamky (také že všetky ovce stoja aspoň na jednej z nich), bez toho aby niektoré dve ovce z našich troch stáli na jednej z nich.

Vzorové riešenie je teda skoro rovnaké ako predošlé – overíme však len tri možné priamky p:

- Priamku určenú prvou a druhou ovcou
- Priamku určenú prvou a treťou ovcou
- Priamku určenú druhou a treťou ovcou

Toto overovanie urobíme rovnako ako v minulom riešení. Ak ani jedna z týchto priamok neuspeje, môžeme spokojne oznámiť, že ovce nestoja na najviac dvoch priamkach.

Overenie priamky nám, neprekvapivo, ešte stále zaberie $O(n)$ času, skúšame ich však iba tri – čiže konštantne veľa. Časová zložitosť je teda $O(n)$. Pamäť využívame úplne rovnako ako v predošlom riešení, čiže tá je aj do tretice $O(n)$.

Listing programu (C++)

```
#include <iostream>
#include <algorithm> // najvacsi spolocny delitel __gcd
#include <vector> // pole ktoremu mozeme za behu menit velkost
using namespace std;

struct ovca{
    int x,y;
};

ovca zakladny_tvar(ovca vektor){          // upravi smernicu priamky do kanonickeho tvaru
    int gcd = __gcd(vektor.x,vektor.y);   // vydelime dx,dy najvacsim spolocnym delitelom
    vektor.x /= gcd;
    vektor.y /= gcd;

    if(vektor.x < 0){                     // v kanonickom tvare chceme dx kladne
        vektor.x *= -1;
        vektor.y *= -1;
    }
    else if (vektor.x == 0)               // ak je dx 0, chceme dy kladne
        vektor.y = abs(vektor.y);

    return vektor;
}


int n;
vector<ovca> stado;
vector<bool> napriamke;

// oznaci vsetky ovce ktore lezia na priamke urcenej ovcami a,b.
// ak je 'prva', skusi overit ci su zvysne ovce na priamke urcenej prvymi
// dvoma ovcami ktore na nu nepatria. ak to je uz druha priamka a stale su ovce
// neleziace na priamke, uz vrati zapornu odpoved.

bool over_na_priamke(int a,int b,bool prva){
    napriamke[a] = napriamke[b] = true;

    ovca vektor = {stado[b].x-stado[a].x,stado[b].y-stado[a].y}; // toto je priamka urcena ovcami a,b
    vektor = zakladny_tvar(vektor);

    vector<int> zle;  // sem dame vsetky ovce ktore nelezia na ziadnej skusanej priamke

    for(int i=0;i<n;++i){
        if(i==a || i==b) continue;

        ovca tmp = {stado[i].x-stado[a].x,stado[i].y-stado[a].y};
        tmp = zakladny_tvar(tmp);

        // ak je ovca cislo i na rovnakej priamke ako a,b oznacime ju
        // inak, ak nie je oznacena ako na priamke, ju pridame do zoznamu oviec ktore niesu na ziadnej skusanej priamke
        if(vektor.x == tmp.x && vektor.y == tmp.y)
            napriamke[i] = true;
        else if(!napriamke[i]) zle.push_back(i);
    }

    // ak uz skusame druhu priamku, musime mat 0 zlych oviec
    if(!prva) return zle.empty();
    // ak mame najviac 2 ovce, nasli sme riesenie
    if(zle.size()<=2) return true;
    // skusime overit ci vsekty zvysne ovce lezia na priamke urcenej prvymi dvoma zlymi ovcami
    return over_na_priamke(zle[0],zle[1],false);
}

int main(){
    cin >> n;
    if(n<=4){
        cout << "ANO\n";
        return 0;
    }
    stado.resize(n);
    napriamke.resize(n);

    for(int i=0;i<n;++i)
        cin >> stado[i].x >> stado[i].y;

    for(int i=0;i<2;++i){                     // budeme skusat priamku urcenu ovcami i a j.
        for(int j=i+1;j<3;++j){
            for(int k=0;k<n;++k)              // kazdu ovcu oznacime ako neleziacu na priamke
                napriamke[k] = false;

            if(over_na_priamke(i,j,true)){    // ak najdeme riesenie, hned povieme ano a skoncime
                 cout << "ANO\n";
                 return 0;
            }
        }
    }

    cout << "NIE\n"; // ani raz sme nenasli riesenie, povieme nie
    return 0;
}
```

5. Správne poradie

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

Táto úloha pozostáva z dvoch podproblémov:

- Ako vieme k nejakému zoznamu akcií zistiť poradie, v ktorom ich vieme vykonať
- Ako vieme rýchlo spracovať podmnožiny, v ktorých sa môžu prvky opakovať

Najprv rozoberieme riešenie prvého podproblému a potom to celé poskladáme dohromady. Pre ľahší popis časovej zložitosti budeme celkový počet akcií (súčet q_i zo vstupu) označovať ako s.

Topologické usporiadanie

Prvý problém bol mierne zamaskovaný problém topologického usporiadania. Najprv si uvedomíme, že podmienka „po vykonaní A nemôžeš vykonať B"je rovnaká, ako podmienka „ak chceš vykonať A aj B, musíš najprv vykonať B a až potom A".

Akcie usporiadame nasledovne: Najprv si pre každú akciu spočítame, koľko iných akcií musíme vykonať *pred* ňou, toto číslo budeme dalej volať *počet závislostí*. Niektoré akcie budú mať nula závislostí; ktorúkoľvek z nich vieme vykonať. Keď ju vykonáme, všetkým akciám, ktoré na nej záviseli, znížime počítadlo o jedna. Tým môže niektorým akciám klesnúť počet závislostí tiež na nulu.

Aby sme nemuseli pred vykonávaním každej akcie prejsť cez všetky a hľadať nejakú, ktorú môžeme vykonať, inšpirujeme sa prehľadávaním grafu do šírky[5]. Na začiatku všetky akcie s nula závislosťami hodíme do fronty na spracovanie. S každým znížením počtu závislostí nejakej akcii skontrolujeme, či tento počet neklesol na nulu. Ak klesol, akciu pridáme do fronty.

Vo všeobecnosti by sa mohlo stať, že v niektorom momente nevieme vykonať žiadnu akciu, lebo všetky majú nejakú nevykonanú závislosť. V zadaní je však garantované, že všetky akcie sa dajú vykonať v nejakom poradí a teda takáto situácia nemôže nastať.

Pokiaľ potrebujeme usporiadať n akcií s m závislosťami, celé nám to bude trvať čas $O(n+m)$, lebo každú závislosť práve raz započítame a raz odpočítame, a každú akciu raz pridáme do fronty a raz spracujeme.

Viacero bojovníkov

Keď vieme robiť topologické usporiadanie, môžeme rovno spraviť riešenie polohrubou silou. Pre každého bojovníka vezmeme jeho zoznam akcií, vyberieme relevantné závislosti a topologicky ich usporiadame. Dostaneme tým riešenie s časovou zložitosťou $O((n+m)\cdot q)$.

Lepšie riešenie však dostaneme, keď využijeme nasledujúce pozorovanie: Ak vieme vykonať všetky akcie v nejakom poradí, v tom istom poradí môžeme vykonať akcie každého bojovníka (s tým, že niektoré jednoducho vynecháme).

Na začiatku teda zistíme poradie, v ktorom sa dajú vykonať všetky akcie. Pre každého bojovníka usporiadame jeho zoznam akcií svojim obľúbeným (rýchlym[6]) triediacim algoritmom[7]. Keď nasčítame členy v tvare $q_i \cdot \log(q_i)$, dostaneme časovú zložitosť $O(s\log(s)+n+m)$.

Listing programu (C++)

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

using namespace std;

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

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

    vector<vector<int> > zavislosti(n);
    vector<int> predomnou(n, 0);

    // Nacitame zavislosti a spocitame pocet zavislosti pre kazdu akciu
    for (int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;

        // Akcie budeme vnutorne reprezentovat o jedna mensim cislom
        a --; b--;
        zavislosti [b].push_back(a);
        predomnou [a] ++;
    }
```

[5] https://www.ksp.sk/kucharka/bfs/
[6] https://www.ksp.sk/kucharka/mergesort/
[7] https://www.ksp.sk/kucharka/triedenie/

```
    // Vypocitame topologicke usporiadanie
    vector<int> poradie;
    queue<int> nasprac;

    // Najprv do fronty pridame vsetky akcie, ktore nemali ziadne
    // zavislosti
    for(int i = 0; i < n; i++) {
        if (predomnou [i] == 0) nasprac.push(i);
    }

    // Dalej ich spracujeme
    while (!nasprac.empty()) {
        int aktualny = nasprac.front();
        poradie.push_back(aktualny);
        nasprac.pop();

        // Znizime pocty zavislosti
        for (int zavislost : zavislosti [aktualny]) {
            predomnou [zavislost] --;

            // Ak treba, pridame akciu do fronty
            if (predomnou [zavislost] == 0) nasprac.push (zavislost);
        }
    }

    // Toto pole bude oznacovat, kolka v poradi je ktora akcia,
    // invporadie [7] oznacuje, kolka v poradi sa vykona osma akcia
    vector<int> invporadie(n);
    For(i, n) invporadie [poradie [i]] = i;

    vector<int> bojovnik;

    // Nakoniec spracujeme bojovnikov
    for (int i = 0; i < q; i++) {
        int qi;
        cin >> qi;
        bojovnik.resize(qi);
        for(int j = 0; j < qi; j++) {
            cin >> bojovnik [j];
            bojovnik [j] --;
        }

        // Tu vyuzivame pole invporadie
        sort(bojovnik.begin(), bojovnik.end(), [&invporadie](int a, int b) {return invporadie [a] < invporadie [b];});
        cout << bojovnik [0] + 1;
        for(int j = 1; j < qi; j++) cout << ' ' << bojovnik [j] + 1;
        cout << '\n';
    }

}
```

Optimálne riešenie

Predošlé riešenie stačilo na to, aby sme získali všetky body od testovača. Existuje však komplikovanejšie riešenie, ktoré bude mať lepšiu časovú zložitosť. Využijeme fakt, že bojovníci sa pýtajú stále na „tie isté" akcie.

Najprv nájdeme topologické usporiadanie všetkých akcií v čase $O(n + m)$. Ďalej načítame akcie všetkých bojovníkov, ale uložíme si ich „čudne": Pre každú z n akcií si budeme pamätať zoznam bojovníkov, ktorí ju chceli vykonať (v čase $O(s + n)$).

Nakoniec budeme vytvárať pre každého bojovníka jeho usporiadaný zoznam akcií, na začiatku prázdny zoznam. Spracujeme všetky akcie v ich topologickom usporiadaní. Keď spracúvame akciu X, pre všetkých bojovníkov, ktorí ju chcú vykonať, ju pridáme na koniec ich aktuálnych zoznamov. Táto časť programu sa bude vykonávať v čase $O(n + s)$.

Nakoniec vypíšeme všetky skonštruované zoznamy pre bojovníkov. Celková časová zložitosť bude $O(n+m+s)$.

Listing programu (C++)

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

using namespace std;

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

    vector<vector<int> > zavislosti(n);
    vector<int> predomnou(n, 0);

    // Zaciatok je zhodny s predoslym riesenim
    for (int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;

        a --; b--;
        zavislosti [b].push_back(a);
        predomnou [a] ++;
    }
```

```
    vector<int> poradie;
    queue<int> nasprac;
    for(int i = 0; i < n; i++) {
        if (predomnou [i] == 0) nasprac.push(i);
    }

    while (!nasprac.empty()) {
        int aktualny = nasprac.front();
        poradie.push_back(aktualny);
        nasprac.pop();
        for (int zavislost : zavislosti [aktualny]) {
            predomnou [zavislost] --;
            if (predomnou [zavislost] == 0) nasprac.push (zavislost);
        }
    }

    /** --------------------------
     *   Tu sa zacinaju nove veci
     *  -------------------------- */

    // Najprv pre kazdu akciu vytvorime zoznam bojovnikov, ktori ju chcu vykonat
    vector<vector<int> > bojovnici_s_akciou(n);
    for (int i = 0; i < q; i++) {
        int qi;
        cin >> qi;
        for(int j = 0; j < qi; j++) {
            int x;
            cin >> x;
            x --;
            bojovnici_s_akciou [x].push_back(i);
        }
    }

    // Dalej budeme vytvarat usporiadany zoznam akcii pre kazdeho bojovnika
    vector<vector<int> > vystup(q);
    for(int akcia : poradie) {
        for (int boj : bojovnici_s_akciou [akcia]) {
            vystup [boj].push_back(akcia);
        }
    }

    // Nakoniec vsetko vypiseme
    for (int i = 0; i < q; i++) {
        cout << vystup [i][0] + 1;
        for (int j = 1; j < (int) vystup [i].size(); j++) {
            cout << '␣' << vystup [i][j] + 1;
        }
        cout << '\n';
    }
}
```

Samo

6. Extrémny smrad

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

Našou úlohou bolo pre každú stoličku určiť silu zápachu aký je cítiť, keď si na ňu sadneme. Matematicky preložené: pre každé políčko v štvorci zo vstupu sme mali nájsť najväčšie číslo nachádzajúce sa v štvorci $k \times k$ so stredom v danom políčku

Priamočiare riešenie

Urobíme presne to, čo sa od nás žiada. Pre každé políčko prejdeme všetky políčka v okolí $k \times k$. A vyberieme to najväčšie a to si zapamätáme. Takéto riešenie bude mať časovú zložitosť $O(n^2k^2)$, keďže pre každé z n^2 políčok sme museli skontrolovať celé okolie $k \times k$.

Listing programu (C++)

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

using namespace std;

typedef long long ll;
int n,k;
vector<vector<int> > mapa,vsledok;
int main() {
  ios::sync_with_stdio(false);
  cin>>n>>k;
  mapa.resize(n,vector<int>(n));
  for(int i=0;i<n;i++){
    for(int j=0;j<n;j++)cin>>mapa[i][j];
  }
  vsledok=mapa;
  for(int i=0;i<n;i++){
    for(int j=0;j<n;j++){
      for(int ii=max(0,i-k/2);ii<min(n,i+k/2+1);ii++){
        for(int jj=max(0,j-k/2);jj<min(n,j+k/2+1);jj++)vsledok[i][j]=max(vsledok[i][j], mapa[ii][jj]);
      }
    }
  }
```

```
  for(int i=0;i<n;i++){
    cout<<vsledok[i][0];
    for(int j=1;j<n;j++)cout<<" "<<vsledok[i][j];
    cout <<endl;
  }
  return 0;
}
```

Toto riešenie nie je optimálne, keďže sa na rovnaké políčka zo vstupu pozeráme strašne veľa ráz, poďme sa ale najprv pozrieť na jednoduchšiu verziu úlohy.

Jednorozmerná verzia

Čo ak by sme na vstupe dostali len jeden dlhý riadok a hľadali by sme najväčšie číslo v okolí k políčok?

Jedným z riešení je postupne prechádzať pole zľava doprava, pričom vo vhodnej dátovej štruktúre si budeme stále pamätať posledných k prvkov, ktoré sme videli. V každom kroku do štruktúry pridáme ďalší prvok, vyhodíme najstarší prvok (ktorý je už mimo okolia) a zapíšeme maximum z prvkov, ktoré máme v štruktúre.

Potrebujeme teda nejakú dátovú štruktúru, kde si budeme uchovávať posledných k prvkov a budeme vedieť povedať maximum z nej a robiť operáciu pridania prvku a odobratia najstaršieho prvku. Taká dátová štruktúra, ktorá vie odoberať najstarší prvok a pridávať nový prvok je napríklad fronta. Vo fronte máme prvky vždy zoradené podľa času, kedy prišli. Problém je ale s nájdením maxima.

Všimnime si však, že si nemusíme vždy pamätať všetky z posledných k prvkov. Zaujíma nás totiž iba maximum z ukladaných prvkov, teda ak vieme, že niektorý prvok už určite nebude maximum v žiadnej k-tici, môžeme ho z fronty vymazať. Napríklad keď pridávame prvok do fronty, tak môžeme všetky menšie prvky z fronty vymazať, keďže ich z fronty vyhodíme skôr (sú staršie) a sú menšie, teda nikdy nebudú maximom.

Čo nám to pomôže? Ak budeme dôsledne vyhadzovať z fronty všetky prvky, ktoré sú menšie, než ten práve pridávaný, bude mať naša fronta zaujímavú vlastnosť: jej prvky budú zoradené podľa veľkosti zostupne. Ak je totiž prvok A vo fronte pred prvkom B, znamená to, že A je vo fronte dlhšie. To ale znamená, že A „prežil" pridávanie prvku B, teda určite nie je menší ako B.

Keďže prvky sú zoradené zostupne, vieme ľahko nájsť maximum (je to prvý prvok). Keď pridávame nový prvok, potrebujeme nájsť (a vyhodiť) všetky prvky, ktoré sú menšie. Tieto prvky budú na konci fronty (lebo čísla vo fronte sú zoradené zostupne), teda ich ľahko nájdeme a budú sa aj ľahko vyhadzovať.

Okrem toho ešte potrebujeme v každom kroku vyhodiť prvok, ktorý sme pridali pred k krokmi (ak tam ešte stále je). Ak je tento prvok stále vo fronte, musí to byť najstarší prvok fronty, a teda je určite na začiatku fronty.

S našou frontou teda budeme robiť nasledovné operácie:

- Čítanie z konca fronty a vyhadzovanie prvkov z konca fronty (keď vyhadzujeme z fronty všetky čísla menšie než prvok, ktorý sa chystáme pridať).
- Pridávanie prvkov na koniec fronty (keď pridávame nový prvok z poľa do fronty).
- Čítanie prvkov zo začiatku fronty (keď zisťujeme maximum prvkov vo fronte).
- Kontrolovanie a mazanie prvkov zo začiatku fronty (keď vyhadzujeme príliš staré prvky).

Obyčajná fronta ale nepodporuje mazanie z konca fronty, preto budeme musieť použiť obojsmernú frontu (v C++ `deque`). Tá dokáže robiť všetky spomínané operácie v konštantnom čase.

Zložitosť

Môže sa nám stať, že ak chceme pridať nejaký prvok, musíme zmazať celú frontu (lebo všetky prvky vo fronte sú menšie), čo môže byť až k prvkov. Jeden krok algoritmu teda môže trvať až $\Theta(k)$ času a takýchto krokov robíme $O(n)$, teda celý algoritmus má časovú zložitosť $O(nk)$.

Časová zložitosť nášho algoritmu sa však dá odhadnúť aj tesnejšie. Ak sa pozrieme na algoritmus ako celok, vidíme, že každý prvok pridáme do fronty maximálne raz a maximálne raz ho odtiaľ aj vyhodíme. Všetky vyhadzovania dokopy nám teda zaberú iba $O(n)$ času (aj keď niektoré z nich môžu byť pomalé), teda časová zložitosť celého algoritmu je $O(n)$.

Dvojrozmerná verzia

Teraz sme už iba krok od riešenia. Predstavme si, že pre každý riadok vyriešime 1D verziu. Teraz máme na každom políčko najväčšie číslo v okolí $1 \times k$. Predstavme si, že teraz na tomto upravenom poli budeme zase riešiť 1D úlohu ale tentokrát po stĺpcoch. Takto nájdeme vlastne maximum z k obdĺžnikov veľkosti $1 \times k$ pod sebou, čo je vlastne maximum zo štvorca $k \times k$, presne ako sme chceli. Teda celková časová zložitosť je $O(n^2)$ keďže sme vyriešili 1D úlohu na všetkých n riadkoch a potom n stĺpcoch. Pamäťová zložitosť bude tiež $O(n^2)$ keďže si musíme pamätať celý pôvodný sektor s hodnotami smradu.

Listing programu (C++)

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

using namespace std;

typedef long long ll;
int n,k;
vector<vector<int> > mapa,vsledok,riadky;
deque<pair<int,int> > Q;
void pridaj(pair<int,int> co){
  while(!Q.empty() && Q.back().first>=co.first)Q.pop_back();
  Q.push_back(co);
}

int main() {
  ios::sync_with_stdio(false);
  cin>>n>>k;
  mapa.resize(n,vector<int>(n));
  for(int i=0;i<n;i++){
    for(int j=0;j<n;j++){
      cin>>mapa[i][j];
      //hladame minimum namiesto maxima
      mapa[i][j]*=-1;
    }
  }
  riadky=mapa;
  //po riadkoch
  for(int i=0;i<n;i++){
    Q.clear();
    for(int j=0;j<min(k/2,n);j++)pridaj({mapa[i][j],j});
    for(int j=0;j<n;j++){
      if(j+k/2<n)pridaj({mapa[i][j+k/2],j+k/2});
      if(Q.front().second<j-k/2)Q.pop_front();
      riadky[i][j]=Q.front().first;
    }
  }
  vsledok=riadky;
  // po stlpcoch
  for(int j=0;j<n;j++){
    Q.clear();
    for(int i=0;i<min(k/2,n);i++)pridaj({riadky[i][j],i});
    for(int i=0;i<n;i++){
      if(i+k/2<n)pridaj({riadky[i+k/2][j],i+k/2});
      if(Q.front().second<i-k/2)Q.pop_front();
      vsledok[i][j]=-Q.front().first;
    }
  }
  for(int i=0;i<n;i++){
    cout<<vsledok[i][0];
    for(int j=1;j<n;j++)cout<<"␣"<<vsledok[i][j];
    cout <<endl;
  }
  return 0;
}
```

7. Utrápený Michallius

Buj a Emo
(max. 12 b za popis, 8 b za program)

Koľko bitkoinov vie vyhrať gladiátor a?

Aby sme zistili *koľko* bitkoinov môže zarobiť nejaký gladiátor a, skúsme najskôr zistiť *ktoré* bitkoiny môže a získať.

Aby gladiátor a získal bitkoin, ktorý na začiatku turnaja patril gladiátorovi b, musí k nemu tento bitkoin nejako doputovať. Táto púť bude vyzerať tak, že najskôr b prehrá súboj s nejakým iným gladiátorom, ten (ak to ešte nie je gladiátor a) potom prehrá s ďalším gladiátorom, ten prehrá s ďalším, atď., až nakoniec posledný gladiátor z tejto reťaze prehrá s gladiátorom a.

Povedané formálne, a dokáže získať bitkoin od b, vtedy a len vtedy, keď existuje postupnosť gladiátorov $b = g_1, g_2, \ldots, g_k = a$ taká, že pre všetky prípustné i vie g_{i+1} poraziť g_i v aspoň jednom type súboja.

Máme teda dobre popísaný vzťah "a vie získať bitkoin b". Viac bitkoinov než počet takýchto gladiátorov b gladiátor a nevie získať. Ale to, že a vie získať bitkoiny od $b_1, \ldots, b_m$ ešte neznamená, že ich vie získať všetky **v priebehu jedného turnaja**. Ukazuje sa ale, že sa to dá.

Dobre sa to vysvetľuje z grafového hľadiska. Uvažujme orientovaný graf, v ktorom vrcholy reprezentujú gladiátorov, a je v ňom hrana $x \to y$ vtedy, keď x vie poraziť y aspoň v jednom type súboja. Potom a vie získať bitkoiny práve tých gladiátorov, ktorí sú dosiahnuteľní z a (rozmyslite si).

Postupnosť zápasov, v ktorej a získa bitkoiny od všetkých takýchto gladiátorov, vieme nájsť nasledovne:

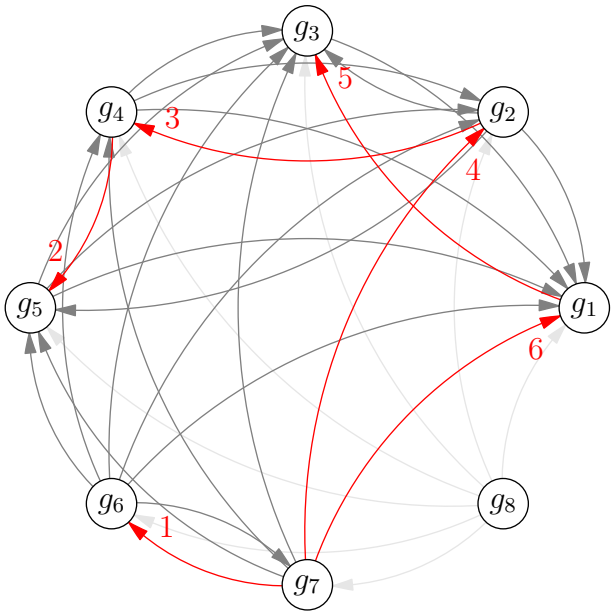
1. Začneme s prázdnou postupnosťou zápasov.

2. Spustime prehľadávanie (napríklad do hĺbky[8]) z a a vždy, keď prejdeme po hrane do nejakého ešte nenavštíveného vrcholu, zafarbime túto hranu na červeno. Po skončení prehľadávania budú červené hrany tvoriť strom obsahujúci práve vrcholy dosiahnuteľné z a.
3. Z tohto stromu odstránime hranu $x \to y$ vedúcu do listu a do postupnosti pridáme zápas medzi x, y, pričom typ zápasu zvolíme tak, aby x vyhral.
4. Opakujeme predchádzajúci krok, kým sú v grafe hrany.

	g_1	g_2	g_3	g_4	g_5	g_6	g_7	g_8
†	1	3	2	5	4	7	6	8
↗	2	5	1	4	3	6	7	8

	1	2	3	4	5	6	7	8
†	g_1	g_3	g_2	g_5	g_4	g_7	g_6	g_8
↗	g_3	g_1	g_5	g_4	g_2	g_6	g_7	g_8

1	2	3	4	5	6
g_7	g_4	g_2	g_7	g_1	g_7
g_6	g_5	g_4	g_2	g_3	g_1
↗	†	↗	†	↗	†



(Prvá tabuľka zobrazuje sily jednotlivých gladiátorov. Druhá tabuľka obsahuje gladiátorov usporiadaných podľa ich síl. Tretia tabuľka zobrazuje zápasy v takom turnaji, v ktorom by gladiátor g_7 získal všetky bitkoiny, ktoré vie získať.)

Prvý správny algoritmus

Predchádzajúca úvaha nám dáva jednoduchý algoritmus na riešenie úlohy: zostrojíme graf a postupne z každého vrcholu a spustíme prehľadávanie, ktorým zistíme, koľko vrcholov z neho vieme dosiahnuť. To je zároveň aj počet bitkoinov, ktorý si vie gladiátor a z turnaja odniesť.

Graf obsahuje n vrcholov a $O(n^2)$ hrán, časová zložitosť je preto $O(n^3)$. Pamäťová zložitosť je $O(n^2)$ (ak v pamäti vytvoríme všetky hrany), prípadne $O(n)$ (ak si pamätáme iba sily gladiátorov).

Listing programu (C++)

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

vector<vector<int>> graf;
vector<int> navstivene;
int pocet;

void dfs(int v) {
  if (navstivene[v]) return;
  navstivene[v] = true;
  pocet++;

  for (int i = 0; i < graf[v].size(); ++i) {
    dfs(graf[v][i]);
  }
}

int main() {
  int n; scanf("%d", &n);
  vector<int> mec(n), kopija(n);
  navstivene.resize(n);

  for (int i = 0; i < n; ++i) {
    scanf("%d%d", &mec[i], &kopija[i]);
  }

  graf.resize(n);
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
```

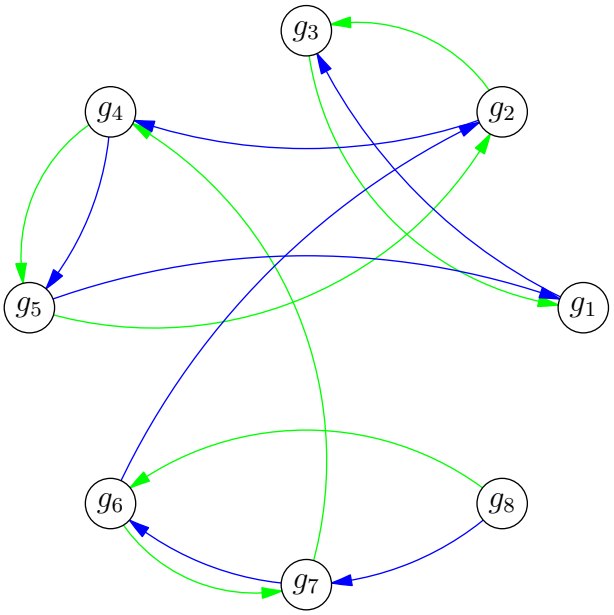
[8] https://www.ksp.sk/kucharka/dfs/

```
      if (mec[i] > mec[j]) graf[i].push_back(j);
      if (kopija[i] > kopija[j]) graf[i].push_back(j);
    }
  }

  for (int i = 0; i < n; ++i) {
    pocet = 0;
    for(int j = 0;j < n;j++) navstivene[j] = 0;
    dfs(i);
    printf("%d\n", pocet);
  }

  return 0;
}
```

Predchádzajúce riešenie možno o rád zrýchliť nasledujúcim trikom: usporiadajme si gladiátorov do postupnosti $g'_1, g'_2, \ldots, g'_n$ podľa rastúcej sily v boji s mečom. Potom nemusíme mať v grafe všetky hrany $g'_i \to g'_j$ pre $i > j$, stačí, ak v ňom budú hrany $g'_i \to g'_{i-1}$ pre všetky prípustné i.

	g_1	g_2	g_3	g_4	g_5	g_6	g_7	g_8
†	1	3	2	5	4	7	6	8
↗	2	5	1	4	3	6	7	8

	1	2	3	4	5	6	7	8
†	g_1	g_3	g_2	g_5	g_4	g_7	g_6	g_8
↗	g_3	g_1	g_5	g_4	g_2	g_6	g_7	g_8



(Graf, v ktorom neuvažujeme zbytočné hrany. Pre každú silu $k > 1$ a každý typ boja máme hranu vedúcu od gladiátora so silou k ku gladiátorovi so silou $k-1$. Zelené hrany zodpovedajú boju s mečom, modré hrany boju s kopijou.)

Prečo? Ak v pôvodnom grafe existovala cesta z a do nejakého b, tak v novom grafe dostaneme cestu z a do b tak, že každú hranu na pôvodnej ceste, ktorá má tvar $g_i \to g_j$ pre $i > j$, nahradíme postupnosťou hrán $g_i \to g_{i-1}, g_{i-1} \to g_{i-2}, \ldots, g_{j+1} \to g_j$. Takže pre ľubovoľný vrchol a je množina vrcholov dosiahnuteľných z a rovnaká, ako v pôvodnom grafe.

Rovnako vieme zredukovať počet hrán, ktoré zodpovedajú súbojom s kopijou. V novom grafe budeme mať $2 \cdot (n-1) = O(n)$ hrán, a vylepšíme tým časovú zložitosť algoritmu na $O(n^2)$.

Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;
typedef pair<int, int> pii;

vector<vector<int>> graf;
vector<int> vysledok, navstivene;
int pocet;

void dfs(int v) {
  if (navstivene[v]) return;
  navstivene[v] = true;
  pocet++;

  for (int i = 0; i < graf[v].size(); ++i) {
    dfs(graf[v][i]);
  }
}

int main() {
  int n; scanf("%d", &n);
  vector<pii> mec(n), kopija(n);
  vysledok.resize(n);
  navstivene.resize(n);
```

```
  for (int i = 0; i < n; ++i) {
    scanf("%d%d", &mec[i].first, &kopija[i].first);
    mec[i].second = kopija[i].second = i;
  }
  sort(mec.begin(), mec.end());
  sort(kopija.begin(), kopija.end());
  graf.resize(n);

  for (int i = 1; i < n; ++i) {
    graf[mec[i].second].push_back(mec[i-1].second);
    graf[kopija[i].second].push_back(kopija[i-1].second);
  }

  for (int i = 0; i < n; ++i) {
    pocet = 0;
    for(int j = 0;j < n;j++) navstivene[j] = 0;
    dfs(mec[i].second);
    vysledok[mec[i].second] = pocet;
  }

  for (int i = 0; i < n; ++i) {
    printf("%d\n", vysledok[i]);
  }

  return 0;
}
```

Vzorové riešenie

Majme gladiátorov $g_1, g_2, \ldots, g_n$ usporiadaných vzostupne podľa sily v boji s mečom. Všimnime si, že všetky bitkoiny, ktoré vie získať g_i, vie získať aj g_{i+1}. To ale znamená, že keď hľadáme vrcholy dosiahnuteľné z g_{i+1}, tak nám stačí hľadať iba také vrcholy, ktoré nie sú dosiahnuteľné z g_i. Vrcholy dosiahnuteľné z g_i totiž môžeme zarátať automaticky.

Budeme teda púšťať prehľadávanie postupne od najslabších gladiátorov k najsilnejším, teda v poradí $g_1, g_2, \ldots, g_n$. Pre každého gladiátora si pamätáme, či sme ho už v niektorom prehľadávaní navštívili, a pamätáme si tiež počet už navštívených gladiátorov. Keď prehľadávame z g_i, tak nenavštevujeme vrcholy, ktoré sme už navštívili v niektorom z predchádzajúcich prehľadávaní.

Rozmyslite si, že takýmto spôsobom sú po i-tom prehľadávaní navštívení práve tí gladiátori, ktorí sú dosiahnuteľní z g_i. Práve od nich vie g_i získať bitkoin. Ich počet je teda počet bitkoinov, ktoré vie g_i získať.

Zložitosť

Na začiatku si potrebujeme utriediť gladiátorov podľa sily. To nám bude trvať $O(n \log n)$. Počas prehľadávaní navštívime každý vrchol grafu najviac raz, čo je $O(n)$ roboty (lebo graf má len n vrcholov a $O(n)$ hrán). Preto je časová zložitosť celého algoritmu $O(n \log n)$. Pamäťová zložitosť ostáva $O(n)$.

Pritom časová zložitosť $O(n \log n)$ je iba kvôli triedeniu, my však máme čísla z rozsahu od 1 po n. Vieme ich teda utriediť count sortom, a dostaneme tak lepší čas $O(n)$.

Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;
typedef pair<int, int> pii;

vector<vector<int>> graf;
vector<int> vysledok, navstivene;
int pocet;

void dfs(int v) {
  if (navstivene[v]) return;
  navstivene[v] = true;
  pocet++;

  for (int i = 0; i < graf[v].size(); ++i) {
    dfs(graf[v][i]);
  }
}

void countSort(vector<pii> &v) {
  vector<pii> utriedene(v.size());
  for (int i = 0; i < v.size(); ++i) {
    utriedene[v[i].first - 1] = v[i];
  }
  v = utriedene;
}

int main() {
  int n; scanf("%d", &n);
  vector<pii> mec(n), kopija(n);
  vysledok.resize(n);
  navstivene.resize(n);

  for (int i = 0; i < n; ++i) {
```

```
    scanf("%d%d", &mec[i].first, &kopija[i].first);
    mec[i].second = kopija[i].second = i;
  }
  countSort(mec);
  countSort(kopija);
  graf.resize(n);

  for (int i = 1; i < n; ++i) {
    graf[mec[i].second].push_back(mec[i-1].second);
    graf[kopija[i].second].push_back(kopija[i-1].second);
  }

  for (int i = 0; i < n; ++i) {
    dfs(mec[i].second);
    vysledok[mec[i].second] = pocet;
  }

  for (int i = 0; i < n; ++i) {
    printf("%d\n", vysledok[i]);
  }

  return 0;
}
```

Baklažán

8. Malé preusporiadanie

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

Hrubá sila

Najjednoduchšie (ale zďaleka nie najefektívnejšie) vyriešime úlohu, ako inak, vyskúšaním všetkých možností. Môžeme vyskúšať všetkých $n!$ možných preusporiadaní stĺpcov, pre každé preusporiadanie skonštruovať výslednú formáciu a skontrolovať, či táto formácia spĺňa podmienky zo zadania. Časová zložitosť takéhoto riešenia je (pri dobrej implementácii) $O(n!n^2)$ a v praktickom testovaní by malo zvládnuť prvú sadu vstupov.

Ako vyzerá výsledok?

Ak sa Spartakova armáda dá preusporiadať do formácie spĺňajúcej podmienku zo zadania, aké vlastnosti bude mať výsledná formácia?

Zo zadania vieme, že v každom rade musia vojaci tvoriť jeden súvislý úsek (okrem prázdnych radov, tie ale nie sú zaujímavé, preto sa nimi veľmi zaoberať nebudeme).

Keďže vymieňame iba stĺpce, každý vojak ostane v tom rade, v ktorom začínal. Pre každý rad teda vieme ľahko zistiť, koľko vojakov v ňom je. Inak povedané, vieme, aký dlhý úsek vojakov musí byť v tomto rade.

Ak teda správne určíme, kde sa nachádzajú (v ktorom stĺpci začínajú) úseky vojakov v jednotlivých riadkoch, jednoznačne tým určíme celú formáciu.

Pre ľubovoľné dva rady A, B vieme zistiť, ako veľmi sa budú ich úseky vojakov prekrývať: stačí nám spočítať, koľko je takých stĺpcov, že aj v rade A aj v rade B majú vojaka.

Ak sa úseky vojakov v dvoch radoch čiastočne prekrývajú (teda majú aspoň jeden spoločný stĺpec, ale každý z nich má aj stĺpec, ktorý ten druhý nemá), určuje to takmer jednoznačne ich vzájomnú polohu. Ak napríklad máme rad A so 7 vojakmi a rad B s 10 vojakmi, ktorých úseky majú 4 spoločné stĺpce, bude ich vzájomná poloha buď takáto:

```
A: *******......
B: ...**********
```

alebo takáto:

```
A: ......*******
B: **********...
```

Ak teda poznáme absolútnu polohu jedného úseku (teda vieme, v ktorom stĺpci formácie začína), sú iba dve možnosti, kde môže začínať druhý úsek.

Ak poznáme absolútnu polohu dvoch čiastočne sa prekrývajúcich úsekov A, B a niekto nám dá tretí úsek C, ktorý sa čiastočne prekrýva s úsekom A, dokážeme už presne určiť polohu úseku C:

- Z toho, že C sa čiastočne kryje s A, máme iba dve možnosti, kde môže začínať C.
- Podľa toho, koľko spoločných stĺpcov majú úseky B a C vieme zistiť, ktorá z týchto dvoch možností je správna.

Ak by napríklad úseky A a B z vyššie uvedeného príkladu boli umiestnené takto:

```
A: ....*******.........
B: .......**********...
```

a vo formácii by bol aj úsek C dĺžky 6, ktorý sa s A prekrýva v piatich stĺpcoch a s B v dvoch stĺpcoch, C musí byť umiestnený takto:

```
A: ....*******.........
B: .......**********...
C: ...******...........
```

Nakoniec si ešte všimnime dve veci:

- Ak je nejaká formácia riešením našej úlohy (dá sa vytvoriť preusporiadaním stĺpcov pôvodnej formácie a každý jej neprázdny riadok obsahuje súvislý úsek vojakov), potom aj jej zrkadlový obraz je riešenie.
- Ak formácia vyhovuje podmienkam zo zadania a obsahuje prázdne stĺpce, potom môžeme všetky tieto stĺpce presunúť úplne doprava a znovu dostaneme formáciu, ktorá je riešením.

Zárodok algoritmu

Na základe týchto pozorovaní už môžeme postaviť efektívny algoritmus, ktorý dokáže riešiť niektoré vstupy:

1. **Pre každý riadok si spočítame, aký dlhý bude jeho úsek vojakov.** *Pri prázdnych riadkoch nie je čo riešiť – musia ostať prázdne. Ďalej sa už budeme zaoberať iba neprázdnymi riadkami.*
2. **Vezmeme si nejaký riadok A a začiatok jeho úseku vojakov umiestnime predbežne do stĺpca 0.** *Toto ešte nie je jeho finálne umiestnenie vo formácii. Ak v nasledujúcich krokoch umiestnime začiatok nejakého úseku do stĺpca x, myslíme tým, že vo finálnej formácii bude tento úsek začínať x stĺpcov napravo od začiatku úseku A. Preto má zmysel hovoriť aj o stĺpcoch so zápornými číslami.*
3. **Nájdeme iný riadok B, ktorý sa čiastočne prekrýva s A a na základe ich prekryvu umiestnime úsek B jedným z dvoch možných spôsobov (je jedno ktorým).** *Od tohto momentu ďalej si budeme pre každý umiestnený riadok pamätať odkaz na jeho* kotvu *– iný umiestnený riadok, s ktorým sa čiastočne prekrýva.* **Kotvou riadku A bude B a kotvou B bude A.**
4. *Kým sa dá, opakujeme nasledujúce kroky:*
   1. **Nájdeme riadok nejaký R, ktorý sme ešte nikam neumiestnili, ale čiastočne sa prekrýva s nejakým už umiestneným riadkom S.**
   2. **Na základe prekryvu R s S a s kotvou riadku S jednoznačne umiestnime úsek v riadku R.** *Môže sa stať, že neexistuje ani jeden vhodný spôsob umiestnenia R – vtedy hneď vieme, že Spartakova armáda sa nedá dobre preusporiadať.*
   3. **Za kotvu riadka R vezmeme riadok S.**
5. *Ak sme mali šťastie a vstup bol dobrého tvaru, v kroku 4 sa nám podarilo umiestniť úseky vojakov vo všetkých riadkoch.* **Všetky úseky posunieme o rovnaký kus doprava tak, aby najľavejší z nich začínal v stĺpci 0.** *Tým dostaneme ich finálne umiestnenie vo formácii.*

Ak pre daný vstup existovala aspoň jedna vyhovujúca formácia, my sme určite zostrojili niektorú z nich: pri umiestňovaní väčšiny riadkov sme si vybrali jedinú možnosť, ktorá mala šancu byť správna.

Vynímkou bol riadok B, kde sme si vyberali z dvoch symetrických možností. Ak však existovala vyhovujúca formácia, kde bol riadok B umiestnený jedným zo spôsobov, potom symetrická formácia bola tiež vyhovujúca a B je v nej umiestnený druhým spôsobom. Preto bolo naozaj jedno, čo sme si vybrali.

Druhým miestom v našom algoritme, kde nevyberáme jedinú možnú správnu možnosť je krok 5. Ak však existuje nejaká vyhovujúca formácia, potom určite existuje aj taká, kde najľavejší úsek začína v nultom stĺpci (napríklad môžeme zobrať formáciu, kde sú všetky prázdne stĺpce úplne napravo).

Mohlo sa však stať, že daný vstup sa nedá vhodne preusporiadať a my sme napriek tomu niečo vytvorili. Preto ešte **musíme overiť, že naša formácia mohla vzniknúť z pôvodnej**:

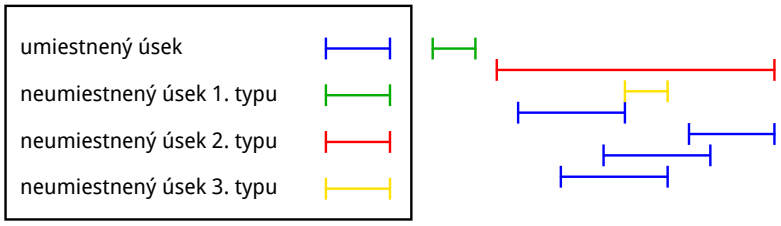
1. **Najprv overíme, že každý úsek vojakov končí najneskôr v stĺpci $n - 1$.** *Ak nie, potom sa naša formácia nezmestí do štvorca $n \times n$ a teda je určite nesprávna.*
2. *Ešte potrebujeme overiť, že naša formácia sa skladá z rovnakých stĺpcov ako pôvodná. To zistíme tak, že* **utriedime jej stĺpce a utriedime aj stĺpce pôvodnej formácie.** *Ak sa formácie skladali z rovnakých stĺpcov, po utriedení už budú úplne identické (a to vieme overiť ľahko).*

Jediným problémom tohto algoritmu je, že funguje, iba ak sa mu v kroku 4 podarí umiestniť všetky úseky vojakov (alebo ak sa mu v tomto kroku podarí zistiť, že sa to nedá).

Vzorové riešenie

V predošlom algoritme sa nám nemusí podariť umiestniť všekty úseky. V takom prípade nám neumiestnené ostanú iba úseky, ktoré sa s umiestnenými úsekmi prekrývajú buď úplne, alebo vôbec. Neumiestnený úsek X môže vyzerať tromi rôznymi spôsobmi:

1. Úsek X sa vôbec neprekrýva so žiadnym umiestneným úsekom (nemajú spoločné stĺpce).
2. Úsek X úplne prekrýva nejaký umiestnený úsek (všetky stĺpce daného úseku sú zároveň stĺpcami X, ale X má aj nejaké ďalšie stĺpce). V takom prípade už musí X prekrývať úplne všetky umiestnené úseky (inak by sa s niektorým z nich čiastočne prekrýval – rozmyslite si!).
3. Úsek X je úplne obsiahnutý v niektorých (aspoň jednom) umiestnených úsekoch, s ostatnými (tých môže byť aj nula) sa vôbec neprekrýva.



Vzniku úsekov druhého typu vieme predísť tým, že si v prvom kroku algoritmu za úsek A zvolíme najdlhší úsek vo formácii.

Zvyšné dva druhy (prvý a tretí) sa dajú šikovne vyriešiť rekurziou.

Neumiestnené úseky prvého typu sa neprekrývajú s umiestnenými úsekmi, ani s úsekmi tretieho typu. Od týchto sú teda úplne nezávislé. Medzi sebou sa však prekrývať môžu. Poumiestňujeme ich tak, že celý algoritmus rekurzívne zavoláme iba na úseky prvého typu a následne ich posunieme tesne napravo od predtým umiestnených úsekov.

Ostáva nám ešte vyriešiť neumiestnené úseky tretieho typu. Všetky tieto úseky sa budú celé nachádzať niekde medzi začiatkom najľavejšieho umiestneného úseku a koncom najpravejšie umiestneného úseku. Ak máme k umiestnených úsekov, pozície ich začiatkov a koncov nám tento interval rozdeľujú na nanajvýš $2k-1$ chlievikov. Každý neumiestnený úsek tretieho typu bude celý vnútri niektorého z týchto chlievikov (inak by sa s nejakým umiestneným úsekom čiastočne prekrýval). Na základe toho, s ktorými umiestnenými úsekmi sa neumiestnené úseky prekrývajú, ich vieme rozdeliť do jednotlivých chlievikov. Ešte potrebujeme umiestniť úseky v rámci ich chlievikov. Na to môžeme pre každý chlievik opäť rekurzívne zavolať náš algoritmus na úseky, ktoré v tomto chlieviku majú byť. Výsledok tohto rekurzívneho volania potom posunieme tak, aby začínal na začiatku chlievika.

Implementácia a zložitosť

Pri implementácii nášho algoritmu si môžeme na začiatku pre každé dva riadky predrátať, v koľkých stĺpcoch majú oba riadky vojaka. To môžeme urobiť priamočiaro v čase $O(n^3)$. Následne už vieme v konštantnom čase pre ľubovoľné dva riadky zistiť, akým spôsobom sa prekrývajú (čiastočne, vôbec, jeden prekrýva druhý, ...).

Na to, ako v kroku 4 hľadáme úseky, ktoré sa čiastočne prekrývajú s už umiesnenými úsekmi, sa dá pozerať ako na prehľadávanie grafu: vrcholmi sú riadky formácie, hrana je medzi dvoma vrcholmi práve vtedy, keď sa ich úseky čiastočne prekrývajú. Tak ho aj môžeme implementovať, takže hľadanie čiastočne sa prekrývajúcich úsekov nám zaberie $O(n^2)$ času (lebo náš graf je hustý – môže mať až rádovo n^2 hrán).

Ostatné kroky algoritmu by už mali byť pomerne jasné.

Poďme sa teraz pozrieť na zložitosť. Predrátanie prekryvov riadkov robíme len raz za život a trvá $O(n^3)$ času. Okrem toho máme nejaký rekurzívny algoritmus, ktorý keď zavoláme na k riadkov, urobí $O(k^2)$ roboty[9], čím umiestni niektoré riadky (aspoň jeden) a na zvyšné sa nejakým spôsobom rekurzívne zavolá. Keďže v každom rekurzívnom zavolaní umiestnime aspoň jeden riadok, volaní bude dokopy nanajvýš n. Keďže v každom z nich urobíme nanajvýš $O(n^2)$ roboty, dokopy to bude $O(n^3)$[10]. Na konci ešte overujeme, či sme naozaj zostrojili riešenie nášho problému. Pri tejto kontrole najdlhšie trvá triedenie stĺpcov formácie, ktoré zaberie $O(n^2 \log n)$

[9] hľadanie riadkov čiastočne sa kryjúcich s už umiestnenými riadkami trvá najdlhšie, ostatné časti algoritmu sú rýchlejšie

[10] poctivejším odhadovaním sa dá ukázať, že táto časť je v skutočnosti dokonca $O(n^2)$, pre nás je to však jedno, lebo aj tak už robíme $\Theta(n^3)$ roboty pri predrátaní

času (na utriedenie n prvkov potrebujeme $O(n \log n)$ porovnaní a jedno porovnanie môže trvať pri najhoršom $O(n)$). Celý náš algoritmus teda beží v čase $O(n^3)$.

Čo sa týka pamäte, pomerne ľahko sa dá vidieť, že nám bude stačiť $O(n^2)$ pamäte.

Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;
#define inf 1023456789

vector<vector<int> > prienik;  // tu si budeme pamatat, kolko spolocnych stlpcov maju dvojice riadkov

struct riadok {
  int id;
  int vojakov;
  int zaciatok, koniec;
  riadok *kotva;

  riadok(string s, int _id) {
    id = _id;
    vojakov = 0;
    for(int i=0; i<s.size(); i++) {
      if(s[i] == '*') vojakov++;
    }
    zaciatok = -1;
    koniec = -1;
    kotva = NULL;
  }

  string vypis(int n) {
    string s;
    for(int i=0; i<n; i++) {
      if(i >= zaciatok && i < koniec) s.push_back('*');
      else s.push_back('.');
    }
    return s;
  }

  bool umiestni(riadok* buduca_kotva);

  void posun(int delta) {
    zaciatok += delta;
    koniec += delta;
  }
};

int prekryv(riadok *a, riadok *b) { // kolko spolocnych stlpcov maju a a b?
  return prienik[a->id][b->id];
}

// Tato funkcia umiestni riadok na zaklade ineho, s ktorym sa ciastocne prekryva.
// Robi teda kroky 4.2 a 4.3 nasho algoritmu, resp. cast kroku 3.
bool riadok::umiestni(riadok* buduca_kotva) {
  if(buduca_kotva == NULL) {
    zaciatok = 0;
    koniec = vojakov;
    return true;
  }
  int p = prekryv(this, buduca_kotva);
  kotva = buduca_kotva;
  zaciatok = kotva->koniec - p;    // skusime umiestnit riadok na pravy koniec kotvy
  koniec = zaciatok + vojakov;

  if(kotva->kotva == NULL) {       // ak sme este len v kroku 3 nasho algoritmu, tak sme spokojni
    kotva->kotva = this;
    return true;
  }

  int pozadovany_prekryv = prekryv(this, kotva->kotva);
  int realny_prekryv = max(min(koniec, kotva->kotva->koniec) - max(zaciatok, kotva->kotva->zaciatok), 0);
  if(pozadovany_prekryv == realny_prekryv) return true;   // skontrolujeme, ci sedi prekryv s kotvou kotvy

  koniec = kotva->zaciatok + p; // skusime umiestnit riadok na lavy koniec kotvy
  zaciatok = koniec - vojakov;
  realny_prekryv = max(min(koniec, kotva->kotva->koniec) - max(zaciatok, kotva->kotva->zaciatok), 0);
  if(pozadovany_prekryv == realny_prekryv) return true;   // skontrolujeme, ci sedi prekryv s kotvou kotvy
  return false;
}

bool ciastocny_prekryv(riadok* a, riadok* b) {   // prekryvaju sa a a b ciastocne?
  int p = prekryv(a, b);
  return p > 0 && p < a->vojakov && p < b->vojakov;
}

int spolocnych(string a, string b) {             // kolko spolocnych stlpcov maju dva riadky?
  int res = 0;
  for(int i=0; i<a.size(); i++) {
    res += (a[i] == '*' && b[i] == '*');
  }
  return res;
}

// V tejto funkcii je v podstate cely nas rekurzivny algoritmus
bool vyries(vector<riadok*> neumiestnene) {
  if(neumiestnene.size() == 0)return true;
  pair<int, int> najdlhsi(-1, -1);  //najdeme najdlhsi riadok
```

```
  for(int i=0; i<neumiestnene.size(); i++) {
    najdlhsi = max(najdlhsi, pair<int, int>(neumiestnene[i]->vojakov, i));
  }

  vector<riadok*> umiestnene;
  neumiestnene[najdlhsi.second]->umiestni(NULL);      //umiestnime najdlhsi riadok
  umiestnene.push_back(neumiestnene[najdlhsi.second]);
  swap(neumiestnene[najdlhsi.second], neumiestnene.back());
  neumiestnene.pop_back();

  // Krok 4 nasho algoritmu. Robime prehladavanie do sirky, pricom pole umiestnene[] pouzivame aj ako frontu
  for(int i=0; i<umiestnene.size(); i++) {
    for(int j=0; j<neumiestnene.size(); j++) {
      if(ciastocny_prekryv(umiestnene[i], neumiestnene[j])) {
        if(!(neumiestnene[j]->umiestni(umiestnene[i]))) return false;
        umiestnene.push_back(neumiestnene[j]);
        swap(neumiestnene[j], neumiestnene.back());
        neumiestnene.pop_back();
        j--;
      }
    }
  }

  int minsur = inf, maxsur = -inf;
  for(int i=0; i<umiestnene.size(); i++) {          // najdeme najlavejsi pouzity stlpec
    minsur = min(minsur, umiestnene[i]->zaciatok);
  }
  for(int i=0; i<umiestnene.size(); i++) {
    umiestnene[i]->posun(-minsur);                  // a posunieme umiestnene riadky, aby zacinali v stlpci 0
    maxsur = max(maxsur, umiestnene[i]->koniec);
  }
  vector<pair<pair<int, int>, riadok*> > podla_intervalu;  // roztriedime neumiestnene riadky do chlievikov
  for(int i=0; i<neumiestnene.size(); i++) {
    int zac = -1, kon = -1;    // kde sa moze nachadzat i-ty neumiestneny riadok? (-1 znamena nedefinovane)

    // ohranicime vyskyt neumiestneneneho useku umiestnenymi usekmi, s ktorymi sa kryje
    for(int j=0; j<umiestnene.size(); j++) {
      if(prekryv(neumiestnene[i], umiestnene[j]) > 0) {
        if(zac == -1 && kon == -1) {
          zac = umiestnene[j]->zaciatok;
          kon = umiestnene[j]->koniec;
        }
        else {
          zac = max(zac, umiestnene[j]->zaciatok);
          kon = min(kon, umiestnene[j]->koniec);
        }
      }
    }

    // ak sa s niecim kryje, tak vezmeme do uvahy aj useky, s ktorymi sa nekryje
    if(zac != -1) {
      for(int j=0; j<umiestnene.size(); j++) {
        if(prekryv(umiestnene[j], neumiestnene[i]) == 0) {
          if(umiestnene[j]->zaciatok <= zac) {
            zac = max(zac, umiestnene[j]->koniec);
          }
          if(umiestnene[j]->koniec >= kon) {
            kon = min(kon, umiestnene[j]->zaciatok);
          }
        }
      }
    }
    podla_intervalu.push_back(make_pair(make_pair(zac, kon), neumiestnene[i]));
  }
  sort(podla_intervalu.begin(), podla_intervalu.end()); // riadky z rovnakeho chlievika dostaneme k sebe
  vector<riadok*> skupina;
  int zac = -2, kon = -2;
  for(int i=0; i<podla_intervalu.size(); i++) {  // rekurzivne volania na jednotlive chlieviky
    if(podla_intervalu[i].first == make_pair(zac, kon)) {
      skupina.push_back(podla_intervalu[i].second);
    }
    else {
      if(skupina.size() > 0) {
        if(!vyries(skupina)) return false;
        int pos = zac == -1 ? maxsur : zac;
        for(int j=0; j<skupina.size(); j++) {
          skupina[j]->posun(pos);
        }
        skupina.clear();
      }
      zac = podla_intervalu[i].first.first;
      kon = podla_intervalu[i].first.second;
      skupina.push_back(podla_intervalu[i].second);
    }
  }
  if(skupina.size() > 0) {
    if(!vyries(skupina)) return false;
    int pos = zac == -1 ? maxsur : zac;
    for(int j=0; j<skupina.size(); j++) {
      skupina[j]->posun(pos);
    }
  }
  return true;
}

vector<string> normalizuj(vector<string> formacia) {
  int n = formacia.size();
  vector<string> vysledok(n);
  for(int x=0; x<n; x++) {
```

```
    for(int y=0; y<n; y++) {
      vysledok[x].push_back(formacia[y][x]);
    }
  }
  sort(vysledok.begin(), vysledok.end());
  return vysledok;
}

bool over(vector<string> vstup, vector<string> vystup) {
  return normalizuj(vstup) == normalizuj(vystup);
}

int main() {
  int n;
  cin >> n;
  string pom;
  getline(cin, pom);

  vector<string> vstup(n);
  vector<riadok*> vsetky;
  vector<riadok*> neprazdne;
  for(int y=0; y<n; y++) {
    getline(cin, vstup[y]);
    vsetky.push_back(new riadok(vstup[y], y));
    if(vsetky.back()->vojakov > 0) {
      neprazdne.push_back(vsetky.back());
    }
  }

  prienik.resize(n, vector<int> (n,0));
  for(int i=0; i<n; i++) {
    for(int j=i+1; j<n; j++) {
      prienik[i][j] = prienik[j][i] = spolocnych(vstup[i], vstup[j]);
    }
  }

  if(!vyries(neprazdne)) {
    cout << "NIE" << endl;
    return 0;
  }

  vector<string> vysledok(n);
  for(int y=0; y<n; y++)vysledok[y] = vsetky[y]->vypis(n);
  if(!over(vstup, vysledok)) {
    cout << "NIE" << endl;
    return 0;
  }
  cout << "ANO" << endl;
  for(int y=0; y<n; y++) {
    cout << vysledok[y] << endl;
  }

  prienik.resize(n, vector<int> (n));
  return 0;
}
```

Lepšie riešenia

Táto úlohá má aj riešenie v $O(n^2)$, je však nad rámec tohoto textu a na plný počet bodov ho nebolo treba. Môžete sa však zamyslieť nad nasledovnou otázkou: ako by sa dalo predrátanie prekryvov riadkov (ktoré vo vzoráku robíme v čase $O(n^3)$) robiť v čase $O(n^3/\log n)$?