Korešpondenčný seminár z programovania
XXXIV. ročník, 2016/17
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 2. kola letnej časti

Mário a MikX

1. Zábava musí byť

(max. 6 b za popis, 4 b za program)

Riešenie tejto úlohy sa síce dalo ľahko vymyslieť a aj ľahko a rýchlo naprogramovať, no dôležité bolo správne odargumentovať správnosť vášho riešenia.

Nezáleží na poradí operácií

Zostreľovanie terčov si môžeme predstaviť ako zoznam operácií, v ktorom sa dá každá operácia zapísať ako `px` – puška zostrelí jeden terč na pozícii x, alebo `bx` – brokovnica zostrelí po jednom terči na pozíciách x a $x+1$.

Pre každú pozíciu $1, 2, \ldots, n$ si môžeme spočítať, koľko výstrelov na ňu dopadne pri konkrétnom zozname operácií. Napríklad pri zozname `p1, b3, p4` dopadne po jednom výstrele na pozície 1, 3, dva výstrely na pozíciu 4 a žiadny výstrel na pozíciu 2.

Vďaka takémuto pohľadu na našu úlohu si môžeme uvedomiť, že ak v nejakom zozname preusporiadame poradie použitých operácií, na každú pozíciu dopadne stále rovnaký počet výstrelov ako pred preusporiadaním. Zoznamy operácií `p1, b3, p4` a napríklad `b3, p4, p1` budú mať teda rovnaký výsledný efekt.

Pri hľadaní riešenia nás teda **nebude zaujímať poradie operácií**, ale len to, koľkokrát musíme pozíciu x zasiahnuť puškou a koľkokrát brokovnicou, teda len počet operácií `bx` a `px` pre každé x.

Ako vymyslieť riešenie?

Puška zhodí v jednej operácii jeden terč, no brokovnica dva. Keďže chceme minimalizovať celkový počet použitých operácií, chceli by sme **čo najviackrát použiť brokovnicu** (a zasiahnuť tak dva terče naraz v čo najviac operáciách). Pri rozmiestňovaní výstrelov si ale musíme dať pozor, aby sme sa nepripravili o možnosť použiť brokovnicu.

Predstavme si, že na každej z pozícií $1, 2, 3, 4$ stojí jeden terč. Všetky terče vieme zostreliť dvoma operáciami, napríklad `b1, b3`. Ak by sme ale brokovnicou strelili do stredu (použijeme `b2`), tak pri zostreľovaní zvyšných terčov nám už brokovnica nepomôže a potrebujeme aspoň 3 operácie (terče 1 a 4 zostrelíme puškou). Problém vznikol tým, že sme skupinku terčov vedľa seba (rozostavenie, ktoré sa dobre zostreľuje brokovnicou), rozdelili jednou *dierou* na dve skupinky, ktoré boli ďaleko od seba a už sa pre ne brokovnica nedala použiť.

Ako využiť potenciál brokovnice naplno? Ako dobrý nápad sa môže zdať, že budeme zostreľovať terče zľava doprava, pričom brokovnicou strieľame, kým sa dá (kým na pozícii x aj $x+1$ sú terče) a ak sa náhodou minú terče na pozícii $x+1$ skôr ako terče na x, zvyšok terčov na x zostrelíme puškou. Takto nikdy nevytvoríme žiadnu *dieru* navyše a použijeme brokovnicu čo najviackrát.

Ako dokázať správnosť riešenia?

Naša predošlá úvaha nás síce doviedla k riešeniu, no vysvetlenie, prečo toto riešenie nájde minimálny počet operácií je trochu nejasné. Ukážeme si však, ako jasne dokázať, že riešenie je správne.

Ak na prvej pozícii stojí k terčov, v ľubovoľnom (teda aj v tom najlepšom) zozname operácií, ktorý má zhodiť všetky terče, musí byť **práve** k operácií, pri ktorých $x = 1$. Čo najviac z týchto terčov zostrelíme brokovnicou, a keď už brokovnicu nemôžeme použiť (lebo už nie sú terče na pozícii 2), zvyšok terčov zostrelíme puškou. Takto sa dostávame do stavu, kde na prvej pozícii nezostal žiadny terč, použili sme najmenší počet operácií, ktoré boli nutné na zostrelenie všetkých k terčov na pozícii 1 a zároveň sme použili brokovnicu najviackrát ako sa dalo.

Dôležitý bod zamyslenia je, že síce sme použili brokovnicu najviackrát pri zostreľovaní terčov na prvej pozícii, no nemohli by sme dosiahnuť celkovo lepšie riešenie (celkovo väčší počet výstrelov brokovnicou a menší počet operácií), ak by sme na prvú pozíciu použili brokovnicu menejkrát? Nie. Ak by sme použili brokovnicu menejkrát, na zvyšných $n-1$ pozíciách by tak celkovo zostalo viac terčov ako v našom riešení. Na zostrelenie zvyšných terčov by tak bolo potrebných aspoň toľko operácií ako v našom riešení a tak vieme, že sa naše riešenie určite nedá zlepšiť.

Keďže na pozícii 1 už nezostáva žiadny terč, môžeme sa na našu úlohu pozrieť tak, ako keby sme ju riešili už len pre $n-1$ terčov a použiť znova tú istú úvahu: Nech je na pozícii 2 m terčov, potrebujeme aspoň m operácií, čo najviac z nich chceme spraviť brokovnicou, atď.

Implementácia po lopate

Postupne prechádzame rady terčov od $i = 1$ po $i = n$: Pre rad i použijeme brokovnicu kým sa dá (kým je v rade $i + 1$ aspoň jeden terč), a potom dorazíme ostatné terče v rade i puškou. Pokračujeme na ďalší rad. Toto sa dá implementovať jedným cyklom.

Časová zložitosť je teda lineárne závislá od počtu radov terčov a ničoho iného: $O(n)$.

Pamäťová zložitosť bude kvôli pamätaniu počtu terčov v jednotlivých radoch tiež: $O(n)$.

Listing programu (Python)

```
n = int(input())
A = [int(x) for x in input().split()]

vystrelov, vysledok = 0, 0
for x in A:
    # kolko novych striel spravim v tomto kole?
    # ratam s tym, ze uprednostnujem brokovnicu, teda co najviac
    # z tych, co som vystrelil v minulom kole zasiahlo aj tento rad
    if x < vystrelov:
        vystrelov = 0
    else:
        vystrelov = x - vystrelov
    # pripocitame pocet vystrelov v ramci spracovavania radu
    vysledok += vystrelov

print(vysledok)
```

Všímaj, že pamäť ušetriť môžeš!

Ešte raz si prečítajte predchádzajúci odsek. V každom momente sa pri výpočte pozeráme len na práve spracovávaný a nasledujúci rad terčov, teda nám stačí pamätať si len počty terčov v týchto dvoch radoch, čím dostávame konštantnú pamäťovú zložitosť: $O(1)$.

Listing programu (C++)

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

using namespace std;

int main () {
  // a = predchadzajuci, b = terajsi
  long long n, a, b, pocet_vystrelov = 0LL, vystreli = 0LL;
  cin >> n >> a;

  for (int i = 0; i < n - 1; ++i) {
    cin >> b;
    // brokovnica
    vystreli = min(a, b);
    a -= vystreli;
    b -= vystreli;
    pocet_vystrelov += vystreli;
    // puska ak treba
    if (a > b) {
      pocet_vystrelov += a;
    }
    // terajsi bude dalej ako predchadzajuci
    a = b;
  }
  pocet_vystrelov += a; // dorazime este posledneho puskou, tento by sa inak nespracoval

  cout << pocet_vystrelov << endl;
}
```

Iné riešenie(a)

Optimálnych riešení tu je viac (keďže nezáleží na poradí operácií). Napr. by bolo rovnako správne najprv všetko čo sa dá postrieľať brokovnicou a potom všetko ostatné doraziť puškou. Toto by však nešlo s konštantnou pamäťovou zložitosťou.

2. Zákerní súrodenci

Šandyna
(max. 6 b za popis, 4 b za program)

Prvé a najľahšie riešenie, ktoré nám napadne, je vykonávať operácie a po každej zmene skontrolovať, či je

reťazec palindróm[1] alebo nie. Pri kontrole možného palindrómu nás vždy[2] zaujímajú dvojice znakov - prvý a posledný, druhý a predposledný, atď. Aby bol reťazec palindrómom, musia byť všetky tieto dvojice zhodné.

Keď sa pokúsime pri každej zmene prechádzať celý reťazec, zistíme, že je to príliš pomalé. Každé skontrolovanie, či je reťazec momentálne palindróm, si totiž v najhoršiom prípade (ak palindróm je) vyžaduje $O(n)$ porovnaní znakov. Celé to teda nášmu programu môže trvať až $O(n \times q)$, čo nestíhame.

Čo robíme navyše? Keďže sa nám vždy mení len jeden znak, nie je vôbec potrebné kontrolovať zvyšok reťazca; ten bude presne taký ako pred danou zmenou.

Pozrime sa radšej na ten znak, ktorý meníme a na jeho náprotivok. Zaujíma nás, ako sa zmenil stav dvojice: z rovnakej na rozdielnu, z rovnakej na rovnakú, z rozdielnej na rovnakú alebo z rozdielnej na rozdielnu.

Chceme teraz zistiť, kedy nastane situácia, že každá dvojica je zhodná. Na začiatku si prejdeme celý reťazec a budeme si pamätať, koľko párov sa navzájom nezhoduje. Pri každej zmene znaku toto číslo príslušne upravíme: ak sme si nejakú dvojicu pokazili, tak počet zlých párov stúpne a ak sme si dvojicu opravili, počet zlých dvojíc klesne. Treba si dať pozor, aby sme hodnotu nemenili, ak sa zmení rozdielny na rozdielny alebo rovnaký na rovnaký! Keď máme počet nezhôd 0, vieme, že náš reťazec je palindróm.

Načítanie a prvotné spočítanie nezhodných dvojíc nám zaberie $O(n)$ krokov, a každú z q otázok vyriešime a odpovieme v $O(1)$ – časová zložitosť je teda $O(n + q)$.

V oboch riešeniach sme si museli pamätať celý reťazec, ale otázky stačilo riešiť postupne, teda sme si mohli pamätať len jednu otázku naraz. To nám udáva pamäťovú zložitosť $O(n)$.

Listing programu (C++)

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

int main () {
  long long q, nesedi = 0;
  string vstup;
  cin >> vstup >> q;
  //pociatocne pocitanie nezhodnych znakov
  for (int i = 0; i < int(vstup.size() + 1) /2; i++) {
    if (vstup[i] != vstup[vstup.size() - i - 1]) {
      nesedi ++;
    }
  }
  long long index;
  char znak;
  for (int i = 0; i < q; i++) {
    cin >> index >> znak;
    //nesediaci -> sediaci
    if (vstup[index] != vstup[vstup.size() -1 - index] &&
        vstup[vstup.size() -1 - index] == znak) {
      nesedi --;
    } else {
    //sediaci -> nesediaci
      if ((vstup[index] == vstup[vstup.size() -1 - index]) &&
          (vstup[vstup.size() -1 - index] != znak) &&
          (vstup.size() - 1 - index != index)) {
        nesedi ++;
      }
    }
    vstup[index] = znak;
    if (nesedi == 0) {
      cout << "ano\n";
    } else {
      cout << "nie\n";
    }
  }
  return 0;
}
```

Emo

3. Zblúdilá krava

(max. 6 b za popis, 4 b za program)

Táto úloha bola iná ako ostatné. Pomáhali ste Denisovi v jeho probléme, pričom váš program načítaval postupne pozície Rysule, a vypisoval, ktorý plot sa má postaviť. Na vyriešenie tejto úlohy sme potrebovali vymyslieť "nepriestrelnú" stratégiu, ktorá nedovolí Rysuli ujsť.

Kedy vyhrá Rysuľa

Skúsme sa na úlohu pozrieť z pohľadu Rysule. Ona sa snaží ujsť, avšak my jej staváme prekážky (ploty), cez ktoré nevie prejsť.

[1] Palindróm je reťazec, čo sa číta odzadu rovnako ako spredu.
[2] Okrem prípadu, keď je reťazec nepárnej dĺžky.

Základné pozorovanie je, že okrajových políčok pastviny je $2(r+s)-4$ a plotov, čo musíme postaviť je $2(r+s)$. Vyplýva to z toho, že rohové políčko musí byť oplotené z dvoch strán.

Jedna z možných stratégií pre Rysuľu je rozbehnúť sa k najbližšiemu okraju pozemku. Ak sa jej nepodarí utiecť cez prvé okrajové políčko, na ktoré príde (lebo ho Denis stihne oplotiť), začne bežať po obvode pozemku. Ak sa jej počas toho naskytne možnosť ujsť (Denis nestihne postaviť nejakú časť plota včas), využije ju.

Predpokladajme na chvíľu, že sa Rysuľa bude správať podľa tejto stratégie. Na to, aby Denis vyhral, musí stihnúť dokončiť plot najneskôr v momente, keď Rysuľa stúpi na posledné obvodové políčko, kde dovtedy nebola. Ak by totiž aj počas nasledujúceho Rysulinho ťahu bola v plote diera, znamenalo by to, že Rysuľa cez túto dieru mohla ujsť, keď okolo nej šla.

Keďže Denis na oplotenie celého pozemku potrebuje $2(r+s)$ ťahov, môže vyhrať jedine v prípade, že Rysuľa bude na dobehnutie k okraju pozemku a následnú prechádzku po jeho obvode potrebovať aspoň $2(r+s)-1$ ťahov (keďže Denis začína, po $2(r+s)-1$-vom Rysulinom ťahu bude nasledovať $2(r+s)$-tý Denisov ťah). Ak sa Rysuľa svojím tretím ťahom dostane na okrajové políčko, prechádzku po obvode dokončí svojím $2(r+s)-2$-hým ťahom (obvod má $2(r+s)-4$ políčok a prvé dva ťahy ešte nejde po obvode), takže Denis nemá šancu stihnúť postaviť plot.

To znamená, že ak sa Rysuľa vie na 3 alebo menej ťahov dostať na okrajové políčko, má zaručený spôsob ako ujsť.

Kedy vyhrá Denis

Už sme zistili, že Denis nevie vyhrať, ak Rysuľa začína 3 alebo menej políčok od okraja a je inteligentná. Stačí mu, aby začínala 4 tahy od okraja? Stačí, lebo môže hrať napríklad podľa nasledovnej stratégie:

V prvých štyroch ťahoch Rysuľa Denisovi určite neujde, keďže na štyri ťahy sa stihne dostať maximálne na okrajové políčko. Denis za tieto štyri ťahy postaví jednu časť plotu v každom rohu. Potom každé políčko bude potrebovať už len jedno oplotenie. A teda vždy keď sa Rysuľa dostaví na nejaké okrajové políčko, Denis jej tam postaví plot.

(Keďže zadanie hovorí, že vždy vieme zabrániť Rysuli ujsť, tak najmenšie pastviny majú rozmery 9×9 a Rysuľa začína v ich strede)

Implementácia

Úloha je vcelku prísna na to čo vypíšeme. V každom ťahu musíme niečo postaviť, nemôžme postaviť plot tam kde už je postavený a tiež si musíme dať pozor aby súradnice ktoré vypisujeme boli naozaj na okraji pastviny.

Najprv musíme vyriešiť rohy pastviny. Tieto pozície budeme poznať hneď ako načítame rozmery pastviny. Vieme potom vybrať jednu časť z každého rohu (rozmyslite si prečo treba z každého), a tú oplotiť. Potom začneme v nejakom krajnom políčku, a bude oplocovať pažravo dookola. Keď sa Rysuľa dostaví na kraj, prerušíme pažravé stavanie, a postavíme plot pri políčku na ktorom sa nachádza. Ak by bola taká hlúpa a opustila kraj, tak pokračujeme v pôvodnom pažravom stavaní.

Ostáva už len vyriešiť ako si pamätať už postavené ploty? Najjednoduchšie je pouziť v C++ `set` alebo `unordered_set` (tu si musíte vytvoriť vlastnú hashovaciu funkciu), ale dajú sa tieto údaje počítať aj v obyčajnom **poli**.

Listing programu (C++)

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

const int N = 20010;
// rozmery mapy, pozicia Rysuli
int r,s,x,y;
// doprava, hore, dolava, dole
int xp[] = {1,0,-1,0}, yp[] = {0,-1,0,1};
// tu si pametame, ktore ploty sme uz postavili
bool uz_postavene[4*N];

// funkcia ktora dostane usek plotu a vrati index do pola uz_postavene
int zahashuj(pair<int, int> plot) {
    int a = plot.first, b = plot.second;
    if (a == 0)return b;
    if (a == s+1) return N + b;
    if (b == 0) return 2*N + a;
    return 3*N + a;
}

// vrati true ak sa policko (x,y) nachadza na Denisovej pastvine
int v_pastvine(int x, int y) {
    return x >= 1 && y >= 1 && x <= s && y <= r;
}

// prezre ci sa v poli nachadza dana hodnota
int nachadza_sa(pair<int, int> pole[4], pair<int, int> hodnota) {
```

```
    for (int i = 0; i < 4; ++i) {
        if(pole[i] == hodnota)return true;
    }
    return false;
}

// vrati smer, ktorym vie Rysula ujst na 1 tah, alebo -1 ak nevie
int vie_rysula_ujst() {
    for (int i = 0; i < 4; ++i){
        int nx = x + xp[i];
        int ny = y + yp[i];
        if(!v_pastvine(nx, ny) && uz_postavene[zahashuj({nx, ny})] == 0)return i;
    }
    return -1;
}

int main() {
    // tieto prikazy je uzitocne vediet, lebo podstatne zrychlia nacitavanie a vypisovanie
    ios_base::sync_with_stdio(false);
    cin.tie(0);cout.tie(0);
    cin >> r >> s;
    // predpocitame si pozicie rohovych plotov
    pair<int, int> rohy[] = {{0, 1}, {0, r}, {s+1, 1}, {s+1, r}};
    for (int i = 0; i < 4; i++) {
        cin >> x >> y;
        uz_postavene[zahashuj(rohy[i])] = true;
        cout << rohy[i].first << " " << rohy[i].second << endl;
    }

    // nepovolene policka, teda tie na ktorych sa chceme otocit
    pair<int, int> nepovolene[] = {{0, 0}, {0, r+1}, {s+1, 0}, {s+1, r+1}};
    // najprv oplocujme smerom doprava, zatial sme oplotili 4 casti plotu
    int smer = 0, pocet = 4;
    // zacnime v lavom hornom rohu
    int plotx = 1, ploty = r + 1;
    while(pocet < 2 * (r+s)) {
        cin >> x >> y;
        // ak je krava na kraji vedla neoploteneho pozemku, musime oplotit ten
        int smer_uniku = vie_rysula_ujst();
        if (smer_uniku != -1){
            int nx = x + xp[smer_uniku];
            int ny = y + yp[smer_uniku];
            uz_postavene[zahashuj({nx,ny})] = true;
            cout << nx << " " << ny << endl;
        } else {
            // chceme prejst cez vsetky uz postavene policka, pripadne sa otocit ak sme v rohu
            while (uz_postavene[zahashuj({plotx, ploty})] || nachadza_sa(nepovolene, {plotx, ploty})) {
                if (nachadza_sa(nepovolene, {plotx, ploty})) smer++;
                // posunieme sa zvolenym smerom
                plotx += xp[smer];
                ploty += yp[smer];

            }
            // postavime plot na pozicii [plotx, ploty]
            uz_postavene[zahashuj({plotx,ploty})] = true;
            cout << plotx << " " << ploty << endl;
        }
        pocet++;
    }
    return 0;

}
```

Listing programu (Python)

```
# funkcia, ktora vracia prvy prvok z hashsetu
def prvyPrvok(hashset):
  return next(iter(hashset))

# nacitame vstup a prekonvertujeme na inty
r,s = map(int, input().split())
# vyberieme si nejaky plot na kazdom rohu
rohy = [(0, 1), (0, r), (s+1, 1), (s+1, r)]
# vytvorime (hash)set a nahadzeme donho vsetky ploty
ploty = set()
ploty.update([(0, i) for i in range(1, r+1)])
ploty.update([(s+1, i) for i in range(1, r+1)])
ploty.update([(i, 0) for i in range(1, s+1)])
ploty.update([(i, r+1) for i in range(1, s+1)])

# pre kazdy roh nacitame vstup, vypiseme a odstranime dany roh
for roh in rohy:
  x,y = map(int, input().split())
  # vstup musime flushovat
  print(roh[0], roh[1], flush=True)
  ploty.remove(roh)

# dokym sa v sete nachadzaju este ploty, pozreme sa ci sa aktualna
# pozicia Rysule nachadza na neoplotenom kraji. Ak ano, vypiseme
# dany plot, inak vypiseme lubovolny z nasej tabulky.
while ploty:
  x,y = map(int, input().split())
  # zistime kam sa vie Rysula pohnut
  susedne = set([(x,y+1), (x, y-1), (x+1,y), (x-1,y)])
  # zistime prienik a susednych policok a plotov
  inter = susedne.intersection(ploty)
```

```
# ak je tento prienik neprazdny, zobereme prvok z prieniku, inak
# lubovolny zo susednych policok
plot = prvyPrvok(inter) if len(inter) else prvyPrvok(ploty)
print(plot[0], plot[1], flush=True)
ploty.remove(plot)
```

Zložitosť

Časová zložitosť algoritmu je $O(r+s)$ ak použijeme unordered_set alebo pole, a $O((r+s) \cdot log(r+s))$ ak použijeme set, kedže musíme oplotiť celú pastvinu, a odpoveď ktorú časť plotu postaviť vieme zodpovedať v čase $O(1)$. Pamäťová zložitosť tohto algoritmu je tiež $O(r+s)$, keďže je potrebné pamätať si už postavené ploty.

4. Zimný problém sysľa Mariána

Hodobox
(max. 9 b za popis, 6 b za program)

Pred čítaním vzorového riešenia tejto úlohy silno odporúčame mať prečítané články z kuchárky o grafoch[3] a prehľadávaní do hĺbky[4].

Formálne si popíšeme, čo je vlastne našou úlohou. Sysľova nora je zakorenený strom (koreňom je izba číslo l). Pre každý vrchol v tomto strome nás zaujíma dĺžka najdlhšej cesty v jeho podstrome.

Riešenie prvej sady

Spomenieme stručne riešenie prvej sady – v nej nám bolo sľúbené, že z každej izby vedie (najviac) jeden tunel do hlbšej izby. Keď sa v takejto nore Marián v niektorej izbe rozhoduje, kam si má ľahnúť a následne kam utiecť, má jednoduchú optimálnu stratégiu – ľahne si do najhlbšej izby v nore, po príchode žaby utečie do tej v ktorej začal; vyššie vyjsť nemôže, smerom nadol je najviac vzdialená predsa najhlbšia izba. Tá bude mať v prvej sade vždy hĺbku n (keďže sa nora nerozvetvuje, ale stále iba klesá nadol). Jediné, čo teda potrebujeme zistiť, je pre každú izbu, ako hlboko vlastne je – označme si hĺbku i-tej izby h_i; odpoveď pre izbu i bude $n - h_i$.

Hĺbku každej izby vieme zistiť jednoduchým prehľadávaním do hĺbky – nastavíme hĺbku izby l na 1, začneme z nej prehľadávať a izbe ktorá je s ňou spojená nastavíme hĺbku 2. Následne pokračujeme v prehľadávaní z nej, teda izbe ktorá je spojená s ňou (a nie je to izba l) nastavíme hĺbku 3. Teraz pokračujeme v prehľadávaní z nej, a tak ďalej.

Z každej izby budeme prehľadávať práve raz, teda časová zložitosť je $O(n)$. Noru si môžeme pamätať ako graf so zoznamami susedov pre každý vrchol, pamäťová zložitosť bude teda tiež $O(n)$ (v prvej sade bolo však vrcholov málo, takže ľubovoľná implementácia bola v poriadku).

Listing programu (C++)

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

struct vrchol
{
    bool videl = false;
    int hlbka;
    vector<int> hrany;
};
vector<vrchol> graf;

void dfs(int v)
{
    graf[v].videl = true;

    for(int i=0;i<graf[v].hrany.size();++i)
    {
        int k = graf[v].hrany[i];
        if(graf[k].videl) continue; // ak sme vrchol k videli, je to izba nad nami
        graf[k].hlbka = graf[v].hlbka + 1; // inak je pod nami a ma hlbku o 1 vacsiu
        dfs(k);
    }
}

int main()
{
    int n,l;
    cin >> n >> l;
    --l;
    graf.resize(n);
    for(int i=0;i<n-1;++i)
    {
        int a,b;
        cin >> a >> b;
        --a,--b;
```

[3] https://www.ksp.sk/kucharka/grafy_uvod/
[4] https://www.ksp.sk/kucharka/dfs/

```
        graf[a].hrany.push_back(b);
        graf[b].hrany.push_back(a);
    }

    graf[l].hlbka = 1;
    dfs(l);

    for(int i=0;i<n;++i)
        cout << n-graf[i].hlbka << "\n";

    return 0;
}
```

Čo nám chýbalo pre rozvetvujúcu noru?

Predstavme si, že Marián si pre zvolenú začiatočnú miestnosť už vybral, kam sa najprv uloží spať. Jeho úniková cesta môže byt len troch rôznych typov:

1. vráti sa do začiatočnej izby a tu ostane
2. prejde cez začiatočnú izbu a bude pokračovať niekam inam
3. do začiatočnej izby sa nevráti a ani cez ňu neprejde

Žiadne iné scenáre sa neodohrajú, takže ak každý z nich rozoberieme, určite sme na nič nezabudli a máme celé riešenie. Pre každú izbu samozrejme povieme najväčší výsledok z hore uvedených možností.

V prvom type cesty utečie Marián rozdiel hĺbok izby v ktorej začal a v ktorej si najprv ľahol. Spočítame si teda hĺbky izieb rovnako ako v predošlom riešení (prehľadávame vždy z každej doteraz nevidenej izby) a pre každú izbu si zapamätáme najväčšiu hĺbku izby, ktorá je v jej podstrome – označme si túto hodnotu i-tej izby $maxh_i$. Jedna možná odpoveď pre izbu i je teda $maxh_i - h_i$ (v druhom príkladovom vstupe to bola odpoveď napríklad pre izby 1,4,6,7 – obrázok sa nachádza na konci vzorového riešenia).

Toto riešenie sa však dá triviálne vylepšiť na druhý spomínaný typ cesty, ak je izba **priamo** spojená s aspoň dvoma hlbšími izbami. V takom prípade, po ľahnutí do najhlbšej možnej izby a vrátení sa do začiatočnej, sa Marián naviac poberie cez inú, priamo spojenú hlbšiu izbu a ľahne si do niektorej izby dosiahnuteľnej z nej. Do ktorej izby sa mu oplatí si ľahnúť? Čím hlbšie pôjde Marián cez inú izbu ako tú, pod ktorou si najprv ľahol, tým bude ďalej od žaby. Určite si teda bude chcieť nakoniec ľahnúť v čo najhlbšej izbe, ako sa len dá. Zapamätáme si dve najväčšie max_h izieb s ktorými sme priamo spojení v izbe i; nech je to $maxh_j$ a $maxh_k$. Najprv teda pôjdeme z izby i v hĺbke h_i do izby s hĺbkou $maxh_j$ cez izbu j. Následne od žaby utečieme tak, že sa vrátime naspäť do izby i (zatiaľ teda vzdialenosť $maxh_j - h_i$) a potom pôjdeme do izby v hĺbke $maxh_k$ cez izbu k – tým si prilepšíme o vzdialenosť $maxh_k - h_i$. Dokopy sme teda prešli od žaby $maxh_j - h_i + maxh_k - h_i = maxh_j + maxh_k - 2 \times h_i$ (v druhom príkladovom vstupe to bola odpoveď napríklad pre izby 2,3).

Posledný prípad je taký, že si Marián chce najprv ľahnúť niekam hlbšie a potom utiecť do druhej izby bez toho aby sa vrátil do začiatočnej – vlastne chce využiť nejakú existujúcu najdlhšiu cestu ktorá neobsahuje tú izbu, v ktorej práve je. To je jednoducho len maximum z už vypočítaných odpovedí všetkých priamo spojených hlbších izieb. Keby v druhom príkladovom vstupe bol od izby 1 tunel do hlbšej izby 8 a od izby 6 tunel do hlbšej izby 9, odpoveď pre izbu 2 je 4 (druhým spôsobom) – Marián si ľahne napríklad do izby 8 a potom utečie do izby 9. Pre izbu 5 je odpoveď rovnaká – Marián má na výber buď spôsob 1 (utiecť do najhlbšej izby s hĺbkou o 3 väčšou) alebo využiť už nájdenú odpoveď pre izbu 2 – a tá je teda lepšia.

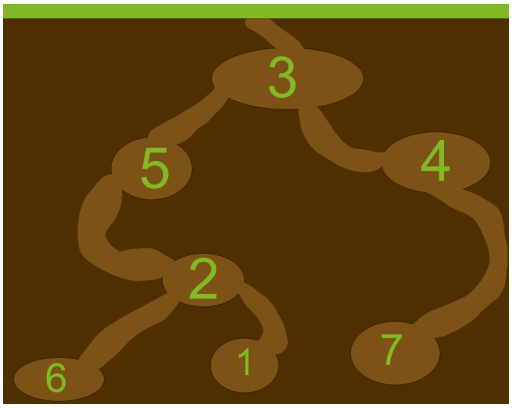
Graf si budeme pamätať rovnako ako v riešení prvej sady - ako zoznamy susedov, s konštantne veľa premennými pre každý vrchol navyše $(h_i, maxh_i, odpoved_i)$. Pamäťová zložitosť je teda $O(n)$.

Časová zložitosť bude tiež rovnaká – $O(n)$ – keďže opäť prehľadávame z každej izby práve raz, a pre každú raz spočítame $h_i, maxh_i$ a $odpoved_i$.

Listing programu (C++)

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

using namespace std;

struct vrchol
{
    vector<int> tunely;
    bool videna = false;
    int hlbka,maxcesta;
};
vector<vrchol> nora;

// vypocita odpoved pre vrchol v, a vrati najvacsiu dosiahnutelnu hlbku cez vrchol v
```

```
int dfs(int v)
{
    nora[v].videna = true;

    int maxcesta_synov = 0;
    int hlbky[2] = {0,0};
    int synovia = 0;

    for(int i=0;i<nora[v].tunely.size();++i)
    {
        int k = nora[v].tunely[i];
        if(nora[k].videna) continue;
        synovia++;
        nora[k].hlbka = nora[v].hlbka + 1;
        int hlbkasyna = dfs(k);
        maxcesta_synov = max(maxcesta_synov,nora[k].maxcesta);
        if(hlbkasyna>hlbky[0]) swap(hlbky[0],hlbkasyna);
        if(hlbkasyna>hlbky[1]) swap(hlbky[1],hlbkasyna);
    }

    if(synovia==0)
    {
        //som list, nemam v podstrome ziadnu cestu
        nora[v].maxcesta = 0;
        return nora[v].hlbka;
    }

    //bud pouzijem maximalnu cestu v podstrome mojich synov
    nora[v].maxcesta = maxcesta_synov;


    //alebo spravim cestu najhlbsi_syn -> ja ( -> druhy najhlbsi,iny syn ak mam viac synov)
    if(synovia==1)
        nora[v].maxcesta = max(nora[v].maxcesta, hlbky[0] - nora[v].hlbka);
    else
        nora[v].maxcesta = max(nora[v].maxcesta, hlbky[0]+hlbky[1] - 2*nora[v].hlbka);

    return hlbky[0];
}

int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);

    int n,l,i;
    cin >> n >> l;

    nora.resize(n);
    l--;

    for(i=0;i<n-1;++i)
    {
        int a,b;
        cin >> a >> b;
        --a,--b;
        nora[a].tunely.push_back(b);
        nora[b].tunely.push_back(a);
    }

    nora[l].hlbka = 1;
    dfs(l);

    for(i=0;i<n;++i)
        cout << nora[i].maxcesta << "\n";

    return 0;
}
```



Dávid

5. Obrovská rekonštrukcia

(max. 8 b za popis, 7 b za program)

Pozrieme sa, ako sa dá táto úloha riešiť hrubou silou, a pomocou niekoľkých dôležitých pozorovaní sa dostaneme ku vzorovému riešeniu.

Bruteforce

Jednoduchý bruteforce môžeme naprogramovať tak, že budeme robiť presne to, čo hovorí zadanie. Pre každú linku sa pozrieme aké najmenšie číslo na nej chýba, (napríklad tak že sa pre každé číslo pozrieme na celý interval a hľadáme či tam je) a vypíšeme najmenšie z nich. Takéto riešenie má časovú zložitosť $O(m \cdot n^2)$ a dostane dva až tri body.

Listing programu (C++)

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

using namespace std;

int main(){
    int n,m;
    cin >> n >> m;
    vector<int> dni(n+1);
    for(int i = 1; i <= n; i++) cin >> dni[i];
    int mi=n+2;
    for(int i = 0; i < m; i++){
        int a, b;
        cin >> a >> b;
        for(int j = 0; j <= b-a+1; j++){
            bool mam=0;
            for(int k = a; k <= b; k++){
                if (dni[k] == j){
                    mam = 1;
                    break;
                }
            }
            if (!mam){
                mi = min(mi, j);
                break;
            }
        }
    }
    cout << mi << endl;
}
```

Vzorové riešenie

Môžeme si všimnúť, že náš bruteforce robí veľa krát to isté. Najme, ak sa intervaly prekrývajú, alebo dokonca opakujú. Niektoré prípady sa dajú vyriešiť jednoducho. Napríklad, ak sa nejaký interval presne zopakuje, môžeme si pamätať aký bol výsledok a druhý krát sa naň len pozrieť. Tiež nie je zložité vyriešiť prípady, keď je jeden interval celý obsiahnutý v inom. Vtedy stačí zistiť riešenie pre vnútorný, lebo vonkajší obsahuje všetko to čo vnútorný (možno niečo viac).

Na plný počet to však stále nestačí. Musíme vyriešiť aj prípady, keď sa intervaly čiastočne prekrývajú. Spravíme to tak, že si budeme jeden interval posúvať a postupne pri tom pokryjeme každý interval, ktorý v sebe nemá obsiahnutý žiaden menší interval.

Koniec nášho intervalu budeme posúvať od začiatku po koniec, a vždy sa pozrieme iba na najmenší z intervalov, ktoré končia na tomto políčku. Všetky ostatné čo na ňom končia ho celý obsahujú. Môžeme si všimnúť, že ak sa v jednom kroku pozeráme na interval, ktorý začína na políčku z, ďalej nás zaujímajú len intervaly, ktoré začínajú ďalej. Ak narazíme na interval ktorý nezačína ďalej, znamená to, že obsahuje celý interval na ktorý sme sa už pozerali.

Ak ďalší začína ďalej, potrebujeme začiatok z predošlého kola posunúť. Spravíme to tak, aby sme rovno zistili výsledok pre nový interval. Budeme si pamätať pre každé číslo, koľko krát sa nachádza v našom intervale, a pri posúvaní ľavého konca ich budeme z tade vyhadzovať (pri posúvaní pravého pridávať). Takto budeme vždy vedieť, aké čísla sú v intervale, na ktorý sa práve pozeráme. Stále však nevieme, či tam nejaké číslo chýba. Preto si budeme okrem toho pamätať aj koľko rôznych čísel tam je. Ak zistíme, že pole je dlhšie ako počet rôznych čísel, znamená to, že tam nejaké chýba. Tiež to znamená, že v ďalšom výpočte nás nemusia zaujímať čísla od neho väčšie, a preto môžeme pole v ktorom si ich pamätáme skracovať, kým naša podmienka nezačne platiť (počítame len čísla ktoré boli do teraz v každom intervale). Ak pri tom vyhodíme nejaké číslo ktoré sa v intervale nachádzalo, musíme aj zmenšiť počet rôznych čísel ktorý si pamätáme.

Keď takýmto spôsobom prejdeme celé pole a všetky intervaly, stačí nám vypísať počet rôznych čísel, lebo v každom intervale na ktorý sme sa pozerali, ich aspoň toľko je.

Časová zložitosť takéhoto riešenia je $O(m+n)$, pretože začiatok aj koniec intervalu na ktorý sa práve pozeráme posúvame iba doprava, a to sa dá najviac n - krát. Keď nesedí počet rôznych čísel, pole vždy len zmenšujeme, čo sa tiež dá najviac n - krát.

Pamäťová zložitosť je $O(n)$, pretože pre každé políčko v poli nám stačí pamätať si začiatok najkratšieho intervalu, ktorý na ňom končí.

Listing programu (C++)

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

using namespace std;

int main(){
    int n,m;
    cin >> n >> m;
    vector<int> dni(n + 1);
    for(int i = 1; i <= n; i++) cin >> dni[i];

    vector<int> zacina(n + 1, -1);
    for(int i = 0; i < m; i++){
        int a, b;
        cin >> a >> b;
        zacina[b] = max(zacina[b], a);
    }
    vector<int> kolkokrat(n + 1, 0);
    int kolko = 0;
    int lavy = 1;
    for (int pravy = 1; pravy <= n; pravy++){
        if (dni[pravy] < kolkokrat.size()){
            kolkokrat[dni[pravy]]++;
            if (kolkokrat[dni[pravy]] == 1) kolko++;
        }
        if(zacina[pravy] == -1) continue;
        while(lavy < zacina[pravy]){
            if(dni[lavy] < kolkokrat.size()){
                kolkokrat[dni[lavy]]--;
                if(kolkokrat[dni[lavy]] == 0)kolko--;
            }
            lavy++;
        }
        while (kolko < kolkokrat.size()){
            if(kolkokrat[kolkokrat.size() - 1] != 0)kolko--;
            kolkokrat.pop_back();
        }
    }
    cout<<kolkokrat.size()<<endl;
}
```

6. Oválna pracovňa

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

Zadanie od nás vyžaduje postaviť taký múr, že rozdiely susedných stĺpov budú všade rovnaké. To ale znamená, že výšky našich stĺpov majú tvoriť aritmetickú postupnosť. A síce výšky stĺpov vyberáme z množiny danej množiny, ak si stĺpy usporiadame podľa výšky, zmení sa náš problém na hľadanie najdlhšej aritmetickej podpostupnosti. Dokonca v zadaní je napísané, že hľadáme neklesajúcu postupnosť a preto usporiadaním stĺpov nestratíme žiadne riešenie.

Aké aritmetické podpostupnosti sa tu nachádzajú?

Aritmetická postupnosť je definovaná 2 číslami – prvým prvkom a diferenciou. Na to, ktorý prvok vybrať ako prvý máme n možností. A pri určovaní diferencie si uvedomme, že diferencia musí byť naozaj rozdiel niektorých dvoch prvkov z našej postupnosti. V opačnom prípade totiž nevytvoríme ani podpostupnosť dĺžky 2. Máme teda najviac n^2 možných diferencií. Pre každý začiatočný prvok a diferenciu vieme zistiť, aká dlhá aritmetická postupnosť má takúto charakteristiku a vybrať to najdlhšie riešenie. Zložitosť takéhoto riešenia bude $O(n^4)$.

Rýchlejšie riešenie

Treba si ale uvedomiť, že prvý prvok nás až tak nezaujíma. Stačí si určiť hodnotu diferencie d. Následne si vieme klásť otázku, aká by bola dĺžka najdlhšej aritmetickej podpostupnosti s diferenciou d, ktorej posledný prvok leží v usporiadanom poli výšok na pozícii x. Túto najväčšiu dĺžku si označme $P[d][x]$.

Pre dané d budeme túto hodnotu počítať postupne pre čoraz väčšie hodnoty x – teda naše pole budeme predchádzať zľava doprava. Nech x-tá výška má hodnotu $A[x]$. Potom vieme, že predposledný prvok tejto aritmetickej postupnosti musel mať hodnotu $A[x]-d$. Musíme, preto zistiť, na ktorom mieste v našom poli sa nachádza táto hodnota.

Jedna možnosť je, že hodnota $A[x]-d$ sa medzi našimi výškami nenachádza. V takom prípade je $P[d][x]=1$, lebo pre danú hodnotu diferencie nemohol existovať skorší prvok. V druhej možnosti, nech je hodnota $A[x]-d$ na

indexe y. Potom zase vieme, že dĺžka najdlhšej podpostupnosti končiacej na indexe x bude $P[d][x] = 1 + P[d][y]$. A keďže $y < x$, tak hodnotu $P[d][y]$ už máme vypočítanú.

Pre každú kladnú diferenciu d (0 ošetríme zvlášť) teda prejdeme poľom `A[]` a vyplníme pole `P[d][]`. Jediné, čo potrebujeme vedieť robiť rýchlo je povedať, na ktorom indexe, ak vôbec, leží nejaké číslo. Toto môžeme robiť pomocou `setu` v čase $O(\log n)$, alebo pomocou hash tabuľky v čase $O(1)$. Dokopy dostaneme časovú zložitosť $O(n^3)$.

Listing programu (C++)

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

int main() {
  int n;
  scanf("%d", &n);
  vector<int> stlpy(n);
  for (int i = 0; i < n; i++)
    scanf("%d", &stlpy[i]);
  sort(stlpy.begin(), stlpy.end());
  int odpoved = 1;
  for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
      int d = stlpy[j] - stlpy[i];
      // v tejto hash mape si na indexe x pamatame dlzku
      // najdlhsej aritmetickej postupnosti konciacej prvkom s hodnotou x
      unordered_map<int, int> najdlhsia;
      for (int k = 0; k < n; k++) {
        int moznaodpoved = 1, hodnotapredosleho = stlpy[k] - d;
        if (najdlhsia.find(hodnotapredosleho) != najdlhsia.end()) {
          moznaodpoved += najdlhsia[hodnotapredosleho];
        }
        odpoved = max(odpoved, moznaodpoved);
        najdlhsia[stlpy[k]] = moznaodpoved;
      }
    }
  }
  printf("%d\n", odpoved);

  return 0;
}
```

Vzorové riešenie

To už ale nie sme ďaleko od vzorového riešenia. Stačí si uvedomiť, že diferenciu si nepotrebujeme určovať dopredu. Aritmetickú postupnosť predsa jednoznačne určujú aj jej posledné dva prvky.

Takže našu otázku upravíme: aká by bola dĺžka najdlhšej aritmetickej postupnosti, ak by posledné dva prvky boli na indexoch x (posledný prvok) a y (predposledný prvok)? Označme si túto hodnotu $D[x][y]$.

Je jasné, že diferencia takejto postupnosti je $d = A[x] - A[y]$. Opäť nás teda bude zaujímať, na ktorom indexe, ak vôbec, leží číslo $A[y] - d$. V prípade, že táto hodnota sa v poli nenachádza, tak $D[x][y] = 2$, pretože tieto dva prvky tvoria našu postupnosť a nič iné pred nimi nemôže byť. Ak sa však hľadaná hodnota nachádza na indexe z, tak platí, že $D[x][y] = 1 + D[y][z]$.

Tak ako v predchádzajúcom riešení budeme pole `D[][]` vypĺňať postupne od najmenších hodnôt x a y, aby sme na riešenie s veľkými hodnotami mohli použiť tie s malými. Na určenie indexu, na ktorom sa nachádza nejaká hodnota použijeme hash tabuľku. Každú hodnotu v našej tabuľke preto vieme vypočítať v konštantnom čase, čo vedie k zložitosti $O(n^2)$.

Listing programu (C++)

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

using namespace std;
int n, vysledok = 1;
vector<int> vysky;
int main() {
  cin >> n;
  vector<vector<int>> D(n, vector<int>(n, 1));
  vysky.resize(n);
  for (int i = 0; i < n; i++)
    cin >> vysky[i];
  sort(vysky.begin(), vysky.end());
  for (int i = 0; i < n; i++) {
    unordered_map<int, int>
        videne; // pre kazdu hodnotu si pamatame kde v poli sa nachadza
    for (int j = 0; j < i; j++) {
      int diferencia = vysky[i] - vysky[j];
      auto it = videne.find(vysky[j] - diferencia);
      if (it == videne.end())
        D[i][j] = 2;
      else
        D[i][j] = 1 + D[j][it->second];
      videne[vysky[j]] = j;
```

```
    }
  }
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++)
      vysledok = max(vysledok, D[i][j]);
  }
  cout << vysledok << endl;
  return 0;
}
```

Rasťo

7. Organizácia projektov

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

Táto úloha sa dala riešiť viacerými spôsobmi. V tomto vzorovom riešení si postupne ukážeme tri rôzne prístupy, ktoré sa dali na riešenie použiť. Napriek tomu, že len jeden je dostatočne rýchly, každý z nich prináša iný pohľad na ten istý problém.

Dynamické programovanie

V našej úlohe sa vlastne snažíme rozdeliť ľudí do troch skupín – tím A, tím B a skupinu, ktorú musíme prepustiť. Pri dynamickom programovaní sa snažíme rozdeliť problém na menšie podproblémy. Môžeme sa inšpirovať problémom batohu[5], kde sme sa snažili rozdeliť veci na dve kopy – veci, ktoré dáme do batoha a veci, ktoré do batoha nedáme. Podproblémom bolo, že chceme nájsť optimálne riešenie pre prvých i vecí a kapacitu batoha j.

V našom prípade by sme si mohli povedať, že chceme nájsť optimálne riešenie pre prvých i programátorov, pričom v tíme A sa z nich bude nachádzať j programátorov a v tíme B sa bude nachádzať k programátorov. Označme si maximálny súčet skúseností oboch tímov pre takýto podproblém ako $P[i][j][k]$. Náš podproblém je jednoznačne určený trojicou čísel (i, j, k).

Už sme si definovali, čo je náš podproblém a teraz nám už ostáva iba sa zamyslieť, ako vieme vypočítať nové riešenie z prechádzajúcich hodnôt. Zoberme si i-teho programátora. Kde sa môže tento programátor nachádzať v optimálnom riešení? Samozrejme, máme tri možnosti – buď je v tíme A, v tíme B, alebo sa nenachádza v žiadnom tíme. Rozoberme si všetky tieto možnosti.

Nech sa nachádza v tíme A. Potom odobratím tohto programátora z tímu A získame optimálne riešenie pre podproblém $(i-1, j-1, k)$. Prečo? Môžeme to dokázať sporom. Nech hodnota tohto riešenia je r. Predpokladajme, že toto nie je optimálne riešenie pre $(i-1, j-1, k)$, čiže $r < P[i-1][j-1][k]$. Potom vieme zobrať optimálne riešenie pre $(i-1, j-1, k)$ a pridať i-teho programátora do tímu A, čím dostaneme lepšie riešenie ako $P[i][j][k]$, lebo $P[i][j][k] = r + a_i < P[i-1][j-1][k] + a_i$. Tým sme sa však dostali do sporu. Čiže odobratím i-teho programátora sme museli nutne dostať riešenie s hodnotou $P[i-1][j-1][k]$. Tým pádom vieme povedať, že ak programátor i skončí v tíme A, tak $P[i][j][k] = P[i-1][j-1][k] + a_i$.

Čo ak sa programátor i nachádza v optimálnom riešení pre (i, j, k) v tíme B? Potom ak ho odoberieme z tímu B, tak dostaneme optimálne riešenie pre podproblém $(i-1, j, k-1)$. Dôkaz je znova podobný tomu predchádzajúcemu. V takomto prípade bude platiť, že $P[i][j][k] = P[i-1][j][k-1] + b_i$.

Posledná možnosť je, že i-ty programátor sa nenachádza v žiadnom tíme. Potom platí,že toto riešenie je optimálnym riešením aj pre $(i-1, j, k)$, čiže $P[i][j][k] = P[i-1][j][k]$.

Ak si to zhrnieme, tak potom $P[i][j][k]$ vieme vypočítať ako maximum z troch hodnôt: $P[i-1][j-1][k] + a_i$, $P[i-1][j][k-1] + b_i$ a $P[i-1][j][k]$.

Ešte si musíme vyjasniť, ako inicializujeme pole `P[][][]`. Triviálnym prípadom je podproblém, keď máme 0 programátorov. V takom prípade vieme inicializovať $P[0][0][0] = 0$. Ostatné hodnoty $P[0][j][k]$ inicializujeme na mínus nekonečno, lebo tieto prípady nemajú riešenie. Ak totiž máme nula programátorov, tak neviem mať v žiadnom tíme nenulový počet programátorov.

Časová zložitosť tohto riešenia je $O(n \cdot x \cdot y)$. Pamäťová zložitosť je tiež $O(n \cdot x \cdot y)$, ale dá sa zlepšiť, ak si všimneme, že na výpočet hodnoty $P[i][j][k]$ potrebujeme iba niekoľko políčok okolo a zvyšné môžeme zabudnúť.

Párovanie a toky

Ďalší pohľad je grafový, využívajúci niekoľko pomerne štandardných algoritmov. Keďže toto riešenie stále nie je vzorové, tak tieto algoritmy nevysvetľujeme do podrobnosti. Ak ich teda nepoznáte, nič si z toho nerobte. Vo vzorovom riešení ich vôbec používať nebudeme.

Tento problém sa dá preformulovať aj ako problém maximálneho váhovaného párovania. Zostrojme si bipartitný graf. Vrcholy v prvej partícii zodpovedajú programátorom a vrcholy v druhej partícii zodpovedajú pozíciám v tíme. Teda prvá partícia má n vrcholov a druhá $x + y$ vrcholov. Medzi každými dvoma vrcholmi z

[5] knapsack – https://en.wikipedia.org/wiki/Knapsack_problem

rôznych partícií vedie hrana, pričom ak máme hranu z i-teho programátora do j-tej pozície v tíme A, tak cena tejto hrany je a_i a ak máme hranu do tímu B, tak cena tejto hrany je b_i. Na takýto graf potom vieme použiť niektorý z algoritmov na hľadanie maximálneho váhovaného párovania na bipartitnom grafe. Ani jeden však nebude dostatočne rýchly, keďže náš graf je pomerne veľký a hlavne obsahuje až $n(x+y)$ hrán.

Problém maximálneho párovania sa však dá preformulovať na problém maximálneho toku. Stačí nám pridať dva špeciálne vrcholy. Prvý vrchol bude pospájaný so všetkými vrcholmi v prvej partícii a druhý vrchol bude pospájaný so všetkými vrcholmi v druhej partícii. Prvý vrchol je zdroj (source) a druhý je odtok (sink). Kapacita každej hrany je jedna.

Môžeme si všimnúť, že v druhej partícii máme zbytočne veľa vrcholov. Všetky pozície v tíme A predsa vieme skomprimovať do jedného vrchola a z tohto vrchola pridať hranu do odtoku s kapacitou x. A rovnako pre vrcholy patriace tímu B.

Na tomto grafe potom môžeme spustiť nejaký všeobecný algoritmus na hľadanie maximálneho toku s maximálnou cenou[6]. Zmenšením druhej partície z $x+y$ na 2 sme zmenšili počet hrán medzi týmito partíciami z $(x+y)n$ na $2n$, čím dostaneme lepšiu časovú zložitosť. Bohužiaľ, ani toto nestačí na vzorové riešenie.

Vzorové riešenie

Vzorové riešenie je v podstate greedy algoritmus. Doteraz sme riešili o dosť všeobecnejšie problémy. Teraz budeme postupovať trocha ináč. Pozrieme sa na to, ako fungujú niektoré špeciálne prípady nášho problému.

Zamyslime sa nad prípadom, keď bude $y = 0$. V tomto prípade sa nám oplatí utriediť programátorov podľa hodnoty a_i zostupne a zobrať prvých x programátorov.

Čo ak sa $y = 1$? Nech máme znova utriedených programátorov podľa a_i. Potom sa nám môže oplatiť zobrať jedného z prvých x programátorov a dať ich do tímu B a doplniť programátora číslo $x+1$ do A. V opačnom prípade zoberieme prvých x programátorov do tímu A a zo zvyšku zoberieme programátora s najvyšším b_i a dáme ho do tímu B.

Čo ak sa $y = 2$? Určite vieme povedať, že prvých x programátorov bude v tíme A alebo B. Nechceme ich teda prepustiť. Vyberme prvého programátora do tímu B. Môže sa nám oplatiť zobrať niektorého z prvých x programátorov (opäť usporiadaných podľa a_i). V takom prípade potrebujeme doplniť tím A, čo samozrejme spravíme $(x+1)$-vým programátorom. Ostáva ešte druhý človek do tímu B. A opäť to môže byť niekto z tímu A, alebo niekto úplne mimo. Ak je to niekto z A, tak tento tím doplníme $(x+2)$-hým programátorom.

Všimnime si nasledovný fakt: Ak si usporiadame všetkým programátorov podľa čísla a_i, tak v každom optimálnom riešení existuje taká hodnota k, že všetci programátori z tímu A sú medzi prvými k programátormi. Naviac vieme, že ak máme najmenšie také k, tak presne $k-x$ z týchto k programátorov musíme umiestniť do tímu B a zo zvyšných $n-k$ programátorov musíme umiestniť $y-(k-x)$ programátorov do tímu B. Otázkou ostáva, či pre zadané k vieme efektívne vypočítať, ktorých programátorov kam umiestniť.

Z posledných $n-k$ programátorov chceme do tímu B vybrať $y-(k-x)$ tých, ktorý majú najväčšiu hodnotu b_i. Musíme sa teda ešte zamyslieť, ako rozdeliť do tímov prvých k programátorov. Predstavme si, že sme všetkých k programátorov dali do tímu A. Čo sa stane, ak i-teho z nich presunieme do tímu B? Z tímu A stratíme a_i skúseností a do tímu B pribudne b_i skúseností. Takže celkový zisk/strata bude $b_i - a_i$. No a zjavne chceme presunúť tých $k-x$ programátorov, pri ktorých získame čo najviac, teda pre ktorých bude číslo $b_i - a_i$ čo najväčšie.

Nájsť optimálne riešenie teda vieme nasledovne: Programátorov si zoradíme podľa hodnoty a_i. Potom vyskúšame každú prípustnú hodnotu k, teda $x \leq k \leq x+y$. Pre dané k vyberieme z prvých k programátorov $k-x$ tých, ktorý majú najväčšiu hodnotu $b_i - a_i$ a týchto programátorov dáme do tímu B. Zvyšných z prvej k-tice dáme do tímu A. Následne z ostatných programátorov vyberieme $y-(k-x)$ programátorov s najvyšším b_i, ktorých zaradíme do tímu B. Pre každú hodnotu k dostaneme nejaké riešenie a optimálne bude to najlepšie z nich.

Ostáva už len navrhnúť rýchly algoritmus na zostrojenie tohto riešenia. V prvej časti spočítame pre každé k, koľko najviac vieme získať ak zoberieme prvých k do tímu A a následne $k-x$ z nich hodíme do tímu B. V druhej časti spočítame pre každé k ako vieme najlepšie nahrabať zvyšných $y-(k-x)$ programátorov do tímu B. Aké dátové štruktúry budeme potrebovať? Aké operácie budeme často vykonávať? Pre oba problémy sa nám bude hodiť dátová štruktúra, ktorá dovolí efektívne vkladať hodnoty a vyberať najväčší prvok – teda halda.

V prvej časti utriedime programátorov podľa a_i. Potom prechádzame cez takto utriedených programátorov a rátame si súčet a_i. V halde si udržiavame pre každého programátora v tíme A držať rozdiel $b_i - a_i$. Chceme v nej teda mať najviac x prvkov.

[6] https://en.wikipedia.org/wiki/Minimum-cost_flow_problem

Postupne prechádzame programátorov, začínajúc od tých s najvyšším a_i. V každej iterácii priradíme ďalšieho programátor do tímu A a jeho rozdiel vložíme do haldy. Ak máme v halde viac ako x prvkov, tak vyberieme von programátora s najväčším rozdielom a preradíme ho do tímu B. Počítame si súčet rozdielov $b_i - a_i$ pre programátorov, ktorých sme vybrali z haldy a tiež si počítame súčet $\sum_{i=1}^{k} a_i$. Súčet rozdielov pre nejaké k je náš zisk, ktorý dostaneme ak daných programátorov presunieme do tímu B. K tomuto číslu ešte môžeme pričítať $\sum_{i=1}^{k} a_i$. Toto dokopy tvorí súčet skúseností tímu A a tímu B, ktorý sme doteraz dosiahli pre dané k.

V tomto čísle sa nenachádza súčet zvyšných $y - (k - x)$ programátorov, ktorých musím pridať do tímu B. Toto spočítame v druhej fáze. Aby sme tieto súčty spočítali, tak budeme iterovať cez programátorov odzadu v poradí v ako sa nachádzajú v utriedenom poli hodnôt a_i. Iterujeme od $i = n$ až po $i = x$. Tentokrát si v halde udržiavame hodnoty b_i. V každej iterácii pridáme novú hodnotu b_i. Ak $i < x + y$, tak vyberieme jedného programátora s najväčším b_j. Počítame si súčet b_j pre vybratých programátorov. Všimnime si, že pre $k = x + y$ nevyberáme ešte žiadneho programátora. Pre $k = x + y - 1$ však už potrebujeme do tímu B doplniť jedného programátora a preto vyberieme toho s najväčším b_i. Súčet hodnôt b_i vybranných programátorov korešponduje súčtu skúseností zvyšných $y - (k - x)$ programátorov, ktorých pridáme do tímu B.

Zhrňme si to na záver. V prvej fáze algoritmu sme počítali koľko skúseností získame ak zoberieme prvých k programátorov a z nich $k - x$ s najväčším rozdielom $b_i - a_i$ dáme do B a zvyšných x dáme do A. V druhej fázi sme pre každé k počítali, koľko skúseností vieme ešte navyše získať ak doplníme $y - (k - x)$ programátorov do tímu B. Ak sčítame súčet, ktorý sme dostali pre nejaké k v prvej fáze sčítame s číslom, ktoré sme dostali v druhej fáze pre to isté k, tak dostaneme celkový najväčší súčet skúseností, ktorý vieme dostať pri rozdelení definovanom hodnotou k. Z týchto všetkých k nám už ostáva iba vybrať to najlepšie.

Celková časová zložitosť nášho algoritmu je $O(n \log n)$, lebo sme potrebovali triediť a používali sme haldu. Pamäťová zložitosť je $O(n)$.

Listing programu (Python)

```
from itertools import accumulate
from heapq import heappop, heappush


def top(ppl_indices, vals, start):
    """ Pre kazde i>=start vypocitaj sucet top (i-start) hodnot z pola vals.
    Pole vals iteruj v poradi urcenim polom ppl_indices."""
    queue = []
    res = [0 for i in range(len(ppl_indices))]
    for k, idx in enumerate(ppl_indices):
        # Pythonova halda je MIN-halda a my chceme MAX-haldu.
        # Preto vkladame zaporne hodnoty do haldy.
        heappush(queue, -vals[idx])
        if k >= start:
            res[k] = res[k-1] - heappop(queue)

    return res


n, a_size, b_size = map(int, input().split())
a = list(map(int, input().split()))
b = list(map(int, input().split()))

conversion_gain = [y - x for x, y in zip(a, b)]
ordered_by_a = sorted(zip(a, range(n)), reverse=True)
prefix_sums_a = list(accumulate([x for x, _ in ordered_by_a]))
# Kolko ziskame konverziou z timu A to timu B?
conversions = top([idx for _, idx in ordered_by_a], conversion_gain, a_size)
# Dopln zvysnych do timu B.
rest_of_bs = list(reversed(top([idx for _, idx
                                in reversed(ordered_by_a[a_size:])],
                               b, n - a_size - b_size))) + [0]

# Scitaj vsetko dokopy.
sol = max(prefix_a + convert + add_bs
          for prefix_a, convert, add_bs
          in zip(prefix_sums_a[a_size-1:a_size+b_size],
                 conversions[a_size-1:a_size+b_size],
                 rest_of_bs))

print(sol)
```

Baklažán

8. Oporné múry

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

Naivné riešenie

Jedno možné priamočiare riešenie je postupne simulovať stavanie jednotlivých múrov. Počas simulácie si budeme pamätať zoznam všetkých už postavených múrov. Vždy, keď postavíme nejaký múr A, overíme, či

je riskantný. To urobíme jednoducho tak, že pre všetky múry postavené skôr než A skontrolujeme, či ich A ohrozuje.

Ostáva ešte doriešiť ako skontrolovať, či múr A ohrozuje nejaký iný múr B.

Pozícia múru M je daná trojicou čísel $x_{M,1}, x_{M,2}, y_M$. Aby sme overili, či nejaký múr A ohrozuje iný múr B, potrebujeme podľa zadania skontrolovať dve veci:

1. Či je múr A na vyššej y-ovej súradnici než B.
2. Či sa projekcie múrov na x-ovú os prekrývajú.

Prvá z týchto podmienok sa dá formálne napísať ako $y_A > y_B$ a druhá sa dá po kratšom zamyslení napísať ako $x_{A,2} > x_{B,1} \wedge x_{A,1} < x_{B,2}$ (kde $\wedge$ znamená logický AND). Obe tieto podmienky vieme teda overiť v konštantnom čase.

Zložitosť

Keď staviame prvý múr, nemusíme kontrolovať nič. Keď staviame druhý, musíme overiť, či neohrozuje ten prvý. Keď staviame tretí, musíme urobiť dve overenia, atď. Dokopy teda budeme robiť $0+1+2+\cdots+n-1 = \frac{(n-1)(n)}{2}$ overení, či nejaký múr ohrozuje nejaký iný múr. To nám zaberie $O(n^2)$ času.

Okrem toho musíme ešte načítať vstup ($O(n)$ času) a pri stavaní každého múru ho pridať do zoznamu postavených (dokopy tiež $O(n)$ času). Celková časová zložitosť teda bude $O(n^2)$.

Pamäťová zložitosť je lineárna: pamätáme si vstup ($O(n)$ pamäte), zoznam už postavených múrov (tiež $O(n)$ pamäte) a ešte konštantný počet pomocných premenných.

Listing programu (C++)

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

struct mur {
        int x1, x2, y;
};

bool ohrozuje(mur A, mur B) {// je pravda, ze mur A ohrozuje mur B?
        return A.y > B.y && A.x1 < B.x2 && A.x2 > B.x1;
}

int main() {
        int n;
        scanf("%d", &n);
        vector<mur> mury(n);
        for(int i=0; i<n; i++) {
                scanf("%d %d %d", &mury[i].x1, &mury[i].x2, &mury[i].y);
        }

        vector<int> postaveny;
        vector<bool> riskantny(n); // riskantny[i] = je mur cislo i riskantny?
        for(int i=0; i<n; i++) {
                int d;
                scanf("%d", &d);
                d--;
                riskantny[d] = false;
                for(int j=0; j<postaveny.size(); j++) {
                        if(ohrozuje(mury[d], mury[postaveny[j]])) {
                                riskantny[d] = true;
                                break;
                        }
                }
                postaveny.push_back(d);
        }

        for(int i=0; i<n; i++) {
                if(riskantny[i])printf("ANO\n");
                else printf("NIE\n");
        }
        return 0;
}
```

Vzorové riešenie

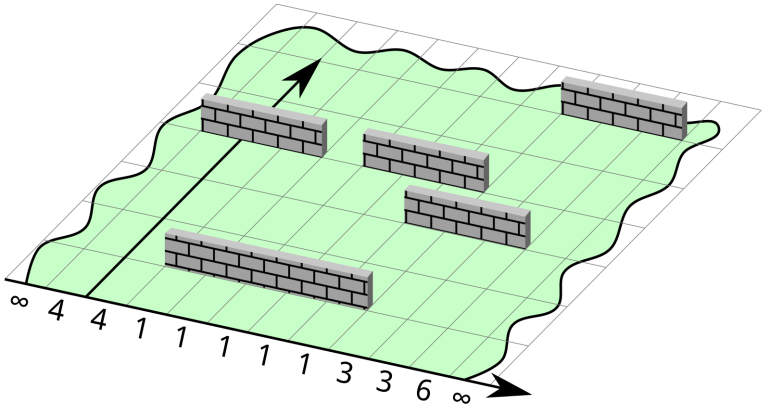
Podobne ako v predošlom riešení, budeme simulovať stavanie múrov a popri tom kontrolovať, či je práve stavaný múr riskantný. Budeme si však pamätať inú informáciu a kontrolu riskantnosti budeme robiť šikovnejšie.

Os x (vrstevnicu na úpätí kopca) si môžeme nasekať na kúsky jednotkovej dĺžky. Pre každý takýto dielik sa potom môžeme pýtať otázku:

"Ak sa postavíme na tento dielik a pozrieme sa po spádnici smerom nahor, uvidíme priamo nad nami nejaký múr? Ak áno, ako vysoko bude najnižší takýto múr?"

Keďže všetky súradnice na vstupe sú celočíselné, odpoveď na túto otázku bude v rámci jedného dielika všade rovnaká. Mohli by sme si teda vytvoriť pole, v ktorom by sme si pre jednotlivé dieliky pamätali odpovede na našu

otázku (výšku najnižšieho múru nad daným dielikom alebo nekonečno, ak taký múr neexistuje). Samozrejme, nemôžeme si pamätať *všetky* dieliky na x-ovej osi (keďže tá je nekonečná). Našťastie, nám budú stačiť iba tie, nad ktorými má šancu byť nejaký múr (čo sú podľa obmedzení zo zadania dieliky ležiace v intervale $[-10^9, 10^9]$). To, že je to stále trochu veľa (dve miliardy dielikov), vyriešime neskôr.



Na začiatku simulácie sú teda v našom poli samé nekonečná (keďže ešte nič nie je postavené).

Keď staviame múr vo výške y s x-ovými súradnicami koncov x_1, x_2, stačí sa nám do nášho poľa pozrieť na dieliky, ktoré ležia v intervale $[x_1, x_2]$. Ak na niektorom z týchto dielikov nájdeme číslo menšie než y, znamená to, že pod múrom, ktorý práve staviame, je už niečo postavené, a teda je riskantný. Ak bude na všetkých dielikoch číslo väčšie než y, potom náš múr nie je riskantný.

Následne ešte musíme naše pole aktualizovať: všetkým dielikom z intervalu $[x_1, x_2]$, ktoré si pamätali číslo väčšie než y (prípadne nekonečno) zmeníme ich hodnotu na y.

Takto sme sa dopracovali k riešeniu, kde pri stavbe každého múru potrebujeme skontrolovať (a následne aktualizovať) nanajvýš dve miliardy prvkov poľa. To je ešte pomalšie než naivné riešenie, spotrebuje však o to viac pamäte. Dá sa to však zachrániť.

Kompresia súradníc

Najprv vyriešme problém s príliš veľkým poľom. Uvedomme si, že nás v skutočnosti nezaujímajú presné x-ové súradnice múrov – potrebujeme iba vedieť, ktoré dvojice múrov sa v x-ovom smere prekrývajú. Na to nám stačí vedieť, v akom poradí budú zaujímavé body (začiatky a konce múrov), ak ich budeme čítať zľava doprava.

Predstavme si, že by sme všetkým múrom zmenili x-ové súradnice začiatkov aj koncov na nejaké iné, tak, aby sa zachovalo pôvodné poradie, teda aby platilo:

- Ak bol nejaký bod (začiatok, alebo koniec múru) R pôvodne naľavo od nejakého iného bodu S, aj v nových súradniciach bude R naľavo od S.
- Ak bol bod R na rovnakej x-ovej súradnici ako S, aj po novom budú na rovnakej x-ovej súradnici.

Dostali by sme síce trochu inú situáciu, ale principiálne by vyzerala rovnako – tie isté dvojice múrov by sa prekrývali. To znamená, že aj riskantnosť múrov by bola rovnaká, teda zmenené zadanie by malo rovnaké riešenie, ako to pôvodné.

Presne to aj spravíme. Konkrétne to urobíme tak, aby všetky x-ové súradnice boli po novom z rozsahu 0 až $2n - 1$. To sa určite dá, keďže zaujímavých bodov je presne $2n$ a niektoré z nich ešte môžu byť na rovnakej x-ovej súradnici.

Technicky to bude vyzerať tak, že všetky začiatky a konce múrov (vždy s pribaleným číslom daného múru) si spolu utriedime podľa x-ovej súradnice. Následne ich všetky prejdeme zľava doprava a prečíslujeme (zmeníme im súradnice): doteraz najľavejšia x-ová súradnica bude odteraz 0, druhá najľavejšia bude 1, a tak ďalej. Pri tom si ešte musíme dávať pozor, aby sme bodom s rovnakou x-ovou súradnicou dali aj po novom rovnakú. Celá táto maškaráda nám zaberie $O(n \log n)$ času, keďže musíme triediť (okrem toho robíme už len konštantný počet lineárnych prechodov).

Dostali sme sa do situácie, keď sú všetky x-ové súradnice z rozumne malého rozsahu, takže namiesto nášho dvojmiliardového poľa nám už stačí iba pole veľkosti $2n$. S takýmto poľom bude mať náš algoritmus časovú zložitosť $O(n^2)$, keďže pri stavaní jedného múru potrebujeme skontrolovať nanajvýš $2n$ prvkov poľa. Pamäťová zložitosť bude $O(n)$, teda v čase aj pamäti sme sa dostali na úroveň naivného riešenia.

Intervalový strom

Teraz skúsme náš algoritmus zrýchliť. Všimnime si, že v našom poli robíme počas behu algoritmu dva druhy operácií:

- Nájdi v poli najmenšie číslo s indexom z intervalu $[x_1, x_2)$.
- Všetky prvky s indexami z intervalu $[x_1, x_2)$, ktoré sú väčšie než y, zmeň na y.

Skúsenejší riešitelia si pravdepodobne spomenú, že na takéto operácie existuje špeciálna dátová štruktúra – intervalový strom. Konkrétne budeme potrebovať minimový intervalový strom s lazy propagation (keďže chceme meniť celý interval naraz). Táto dátová štruktúra dokáže simulovať pole, v ktorom obe uvedené operácie dokáže robiť v logaritmickom čase od počtu prvkov poľa. Ak túto dátovú štruktúru ešte nepoznáte, prečítajte si v našej Kuchárke najskôr o základnom intervalovom strome[7] a potom aj o lazy intervalovom strome[8].

Zložitosť

Keď namiesto obyčajného poľa použijeme intervalový strom, budeme schopní postaviť jeden múr v čase $O(\log n)$, keďže pri stavbe múru potrebujeme spracovať iba dve požiadavky do nášho intervalového stromu. Celá simulácia stavby nám teda bude trvať $O(n \log n)$ času. Kompresia súradníc nám tiež trvala $O(n \log n)$ času a načítanie vstupu je lineárne, celý algoritmus má teda časovú zložitosť $O(n \log n)$.

Pamäťová zložitosť bude $O(n)$, keďže si pamätáme vstup a intervalový strom so zhruba $2n$ listami.

Listing programu (C++)

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

const int inf = 1023456789;

struct intervalac
{
        vector<int> strom, lazy;
        vector<int> zac, kon;

        intervalac(int sz) {
                int offset;
                for(offset = 1; offset < sz; offset *= 2);
                strom = vector<int>(2*offset, inf);
                kon.resize(2*offset);
                zac.resize(2*offset);
                for(int i=0; i<offset; i++) {
                        zac[i+offset] = i;
                        kon[i+offset] = i+1;
                }
                for(int i=offset-1; i>0; i--) {
                        kon[i] = kon[i*2+1];
                        zac[i] = zac[i*2];
                }
                lazy = vector<int>(2*offset, inf);
        }

        void nebud_lenivy(int vrchol) {
                if(lazy[vrchol] != inf) {
                        for(int i=0; i<2; i++) {
                                lazy[vrchol*2+i] = min(lazy[vrchol*2+i], lazy[vrchol]);
                                strom[vrchol*2+i] = min(strom[vrchol*2+i], lazy[vrchol]);
                        }
                        lazy[vrchol] = inf;
                }
        }

        int minimum(int a, int b, int vrchol = 1) {
                if(a >= kon[vrchol] || b <= zac[vrchol])return inf;
                if(a <= zac[vrchol] && b >= kon[vrchol])return strom[vrchol];
                nebud_lenivy(vrchol);
                return min(minimum(a,b,vrchol*2), minimum(a, b, vrchol*2+1));
        }

        void zmen(int a, int b, int hod, int vrchol = 1) {
                if(a >= kon[vrchol] || b <= zac[vrchol])return;
                if(a <= zac[vrchol] && b >= kon[vrchol]) {
                        lazy[vrchol] = min(lazy[vrchol], hod);
                        strom[vrchol] = min(strom[vrchol], hod);
                        return;
                }
                nebud_lenivy(vrchol);
                zmen(a,b,hod, vrchol*2);
                zmen(a,b,hod, vrchol*2+1);
                strom[vrchol] = min(strom[vrchol*2], strom[vrchol*2+1]);
        }

};
```

[7] https://ksp.sk/kucharka/intervalovy_strom
[8] https://ksp.sk/kucharka/lazy_intervalovy_strom

```
struct mur
{
        int x1, x2, y;
        void zmen(int stare_x, int nove_x) {
                if(x1 == stare_x) x1 = nove_x;
                if(x2 == stare_x) x2 = nove_x;
        }
};

int main()
{
        int n;
        scanf("%d", &n);
        vector<mur> mury(n);
        vector<pair<int,int> > podla_x;
        for(int i=0; i<n; i++) {
                scanf("%d %d %d", &mury[i].x1, &mury[i].x2, &mury[i].y);
                podla_x.push_back(make_pair(mury[i].x1, i));
                podla_x.push_back(make_pair(mury[i].x2, i));
        }
        sort(podla_x.begin(), podla_x.end());
        int x = 0;
        for(int i=0; i<podla_x.size(); i++) {
                if(i > 0 && podla_x[i].first > podla_x[i-1].first) x++;
                mury[podla_x[i].second].zmen(podla_x[i].first, x);
        }

        intervalac strom(x);
        vector<bool> riskantny(n);
        for(int i=0; i<n; i++) {
                int d;
                scanf("%d", &d);
                d--;
                riskantny[d] = strom.minimum(mury[d].x1, mury[d].x2) < mury[d].y;
                strom.zmen(mury[d].x1, mury[d].x2, mury[d].y);
        }

        for(int i=0; i<n; i++) {
                if(riskantny[i])printf("ANO\n");
                else printf("NIE\n");
        }

        return 0;
}
```

Iné riešenie

Iné riešenie, tiež v čase $O(n \log n)$, dostaneme, ak v predošlom riešení vymeníme úlohy času a y-ovej súradnice. Múry by sme teda nespracovávali v poradí stavby, ale podľa y-ovej súradnice od najnižšiehio po najvyšší. V intervalovom strome si potom budeme pamätať odpovede na otázku "Kedy najskôr bude nad týmto dielikom postavený múr?", pričom vždy berieme do úvahy iba tie múry, ktoré sme už spracovali.

Ešte iné riešenie (doplnil Mišof)

Predstavme si, že kým staviame prvú polovicu múrov, farbíme ich na modro, a keď staviame druhú polovicu múrov, tie už farbíme na červeno. Ak existuje nejaká dvojica múrov, z ktorých druhý ohrozuje prvý, tak sú len tri možnosti: tvoria ju dva modré múry, tvoria ju dva červené múry, alebo nejaký červený múr ohrozuje nejaký modrý múr

Naše nové riešenie bude založené na princípe *rozdeľuj a panuj*. Spravíme v ňom nasledovné:

1. Ak máme viac ako jeden modrý múr, rekurzívnym volaním nájdeme všetky riskantné múry medzi modrými múrmi.
2. Ak máme viac ako jeden červený múr, rekurzívnym volaním nájdeme všetky riskantné múry medzi červenými múrmi (čiže všetky červené múry, ktoré ohrozujú iný, skôr postavený červený múr).
3. Nejak šikovne pre každý červený múr zistíme, či neohrozuje nejaký modrý.

Ako vieme šikovne spraviť krok 3?

Všetky múry si usporiadame podľa výšky zdola hore. V tomto poradí ich následne spracujeme. Počas spracúvania si udržiavame zjednotenie už spracovaných modrých múrov. Na to nám stačí buď intervalový strom, alebo si napríklad môžeme toto zjednotenie pamätať ako usporiadanú množinu (set) disjunktných intervalov. No a vždy, keď spracúvame červený múr, pozrieme sa, či má neprázdny prienik so zjednotením dovtedy spracovaných modrých.

Akú to má dokopy časovú zložitosť? Podobá sa to tak trochu na MergeSort: tiež robíme dve rekurzívne volania na problém polovičnej veľkosti. MergeSort však navyše spraví len $O(n)$ práce, zatiaľ čo naše riešenie jej počas zametania v kroku 3 spraví až $O(n \log n)$. A tento "logaritmus navyše" ostane aj vo výsledku: zatiaľ čo časová zložitosť MergeSortu výjde $O(n \log n)$, tento náš algoritmus má časovú zložitosť $O(n \log^2 n)$. Toto riešenie tiež stačilo na plný počet bodov.