Korešpondenčný seminár z programovania
42. ročník, 2024/25
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 1. kola zimnej časti

Vladaso

1. Hlúpa intergalatická pošta!

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

Ako najjednoduchšie riešenie je si sitúaciu odsimulovať. Spraviť si pole `[1, 2, 3, 4, 5, 6, 7, 8, 9, 0]` a pamätať si pozíciu, na ktorej sa nachádzame. Túto pozíciu meníme a pre každé číslo v pin-e si pomocou cyklu cez pole zistíme, ktorá strana je bližšie a na tú stranu sa vydáme. Pri takomto počítaní nemôžeme zabudnúť, že pole je ako keby tvaru kruhu a preto musíme myslieť na pozície na krajoch (1 -> 0 a 0 -> 1). Toto však nie je úplne najlepšie riešenie.

Problém si chceme simulovať, ale nechceme manuálne pohybovať ukazovátkom. Vlastne, z každého bodu máme dve cesty kadiaľ sa vieme vybrať. Buď doľava alebo doprava. Minimum pohybov potrebných na napísanie pinu dostaneme vtedy, keď si vždy vyberieme tú kratšiu cestu. Ešte však treba vymyslieť logiku za tým, koľko krokov potrebujeme na to, aby sme sa dostali na číslo z nejakej pozície.

Stále si potrebujeme pamätať našu pozíciu, no nejdeme krok po kroku, ale vypočítame si počet krokov, ktoré potrebujeme. V našom cykle pozíciu potom vždy updatujeme iba na miesto, kam sa chceme presunúť.

Treba si uvedomiť že k počtu krokov môžeme ihneď pripočítať dĺžku pin kódu, keďže určite budeme musieť každé číslo stlačiť.

Ak sa chceme dostať z čísla 5 na 6 alebo 6 na 5 je to vlastne $\text{abs}(6-5)$ alebo $\text{abs}(5-6)$, čo je rovnaké číslo. Musíme však rátať aj s tým, že cesta druhou stranou vie niekedy byť rýchlejšia. Cestu druhou stranou dostaneme keď od celkového počtu políčok odčítame vzdialenosť, ktorú by sme prešli ak by sme nepoužili cestu z 1 na 0. Teda pre nás je to $10-\text{abs}(5-6)$, čo nám dá výsledok 9, čo je naozaj počet krokov, keby putujeme druhou stranou. Vzorec na výpočet najkratšej vzdialenosti z pozície i na pozíciu x bude teda vyzerať takto: $\min(\text{abs}(x-i), 10-\text{abs}(x-i))$

Menšiu z týchto dvoch možností (ísť doľava alebo doprava) pripočítame k počtu krokov a opakujeme pre všetky čísla v pin-e. Keď prejdeme všetkými číslami v pin-e, vypíšeme celkový počet krokov.

Listing programu (Python)

```python
def main():
    code = input()
    res = 0
    prev = 1
    res += len(code)  # pridame pocet znakov, kedze aj zadat znak stoji 1
    for c in code:
        num = int(c)
        if num == 0:
            num = 10  # toto aby 0 bola ako keby na desiatej pozicii
        # pridame minimalnu vzdialenost medzi cislami
        res += min(abs(num - prev), 10 - abs(num - prev))  # 10-abs(num-prev) je vzdialenost ak
        ↪ pouzijeme cyklus
        prev = num  # aktualizujeme nasu poziciu
    print(res)

if __name__ == "__main__":
    main() # je dobre pouzit main, python potom bezi rychlejsie
```

Listing programu (C++)

```cpp
#include <bits/stdc++.h>
using namespace std;
int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    string code;
    cin >> code;
    long long res = 0;
    int prev = 1;
    res+=code.size(); // pridame pocet znakov, kedze aj zadat znak stoji 1
    for(char c : code){
        int num = c - '0'; // prevedieme znak na cislo
        if(num == 0){
            num = 10; // toto aby 0 bola ako keby na desiatej pozicii
        }
        res += min(abs(num-prev), 10-abs(num-prev)); // pridame minimalnu vzdialenost medzi cislami
        //10-abs(num-prev) je vzdialenost ak pouzijeme cyklus
        prev = num; // aktualizujeme nasu poziciu
    }
    std::cout << res << std::endl; //vypiseme vysledok a newline!!
}
```

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

2. Vladkova hra

Simulácia

Priamočiarym riešením by mohlo byť si Vladkove hranie odsimulovať. Prechádzať po políčkach zľava doprava, vždy zmeniť stav plošinky a po postavení sa na políčko bez plošinky, vrátiť sa na začiatok. Počas tohto si budeme počítať, koľko krokov sme spravili. Toto budeme opakovať dovtedy, kým sa Vladko nedostane na koniec.

Toto riešenie funguje, avšak je pomerne pomalé.

Potrebujeme to simulovať?

Skúsme sa pozrieť na niektoré hry. Ak Vladkov level začína v stave `0 0 0`, po prvom prejdení skončí v stave `1 0 0`, potom `0 1 0`, `1 1 0`, `0 0 1`, `1 0 1`, `0 1 1` a skončí v `1 1 1`. Vladko tento level dokončí po 7-ich pokusoch.

Môžeme si všimnúť, že ak by sme level mali nakreslený opačne, tak postupnosť `0 0 0`, `0 0 1`, `0 1 0`... vyzerá presne ako postupnosť binárnych čísiel od 0 do 7. Teraz si potrebujeme uvedomiť, že každý level, ktorý sa skladá iba z políčok bez plošiniek sa správa rovnako, a teda ho vieme reprezentovať ako postupnosť binárnych čísiel od 0 po x. Následne potrebujeme zistiť, aké je to x pre daný počet políčok. Ak máme v leveli n políčok, aké najväčšie binárne číslo v ňom vieme zobraziť? To bude n jednotiek. Ak n binárnych jednotiek chceme premeniť na číslo v desiatkovej sústave, dostaneme $2^0 + 2^1 + \cdots + 2^{n-1}$. Alternatívne, vieme, že ak máme n binárnych jednotiek, tak číslo o jedna väčšie bude jednotka a n núl, čiže 2^n v desiatkovej sústave. Potom naše pôvodné číslo bude $2^n - 1$.

Čo ale, ak nezačíname so samými nulami? Vtedy môžeme počiatočný stav vyjadrený ako binárne číslo odpočítať od celkového počtu pokusov, keby všetky políčka boli nulové.

Alternatívne môžeme počiatočný stav ako binárne číslo znegovať (`0 1 0` -> `1 0 1`) a premeniť do desiatkovej sústavy, čím dostaneme počet pokusov.

Časová aj pamäťová zložitosť takéhoto riešenia je $O(n)$. Technicky si nemusíme pamätať celý vstup, ale spracovávať ho postupne, vtedy by sme vedeli dostať pamäťovú zložitosť aj $O(1)$.

Listing programu (Python)

```python
n = int(input())
states = list(map(bool, map(int, input().split())))

flips = 0

for i, s in enumerate(states):
        if not s:
                flips += 2**i

print(flips)
```

Listing programu (C++)

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main(){
    int n;
    cin >> n;
    ll helper = 0;
    ll power = 1;

    char c;
    for (int i = 0; i < n; i++){
        cin >> c;
        helper += power*(c - '0');
        power *= 2;
    }

    cout << (power - helper - 1) << "\n";
}
```

Mato

3. Intraplanetárne čížmy

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

Ak sa nachádzame na pozícii x a vieme prejsť ešte d, vieme sa dostať na všetky pozície až do $x+d$. Spomedzi čižiem na týchto pozíciách si tak chceme vybrať nejaké čižmy, ktoré nás dovedú k optimálnej odpovedi.

Intuitívne dáva zmysel, že to budú také čižmy i, že ich $A_i + B_i$ je najväčšie spomedzi všetkých dostupných čižiem. Uvažujme ale, že by sme tieto čižmy nezobrali a zobrali nejaké iné čižmy j. V následujúcom kroku by sme mali na výber z čižiem z nejakej podmnožiny tých, ku ktorým sme sa vedeli dostať pomocou čižiem i, pretože by sme sa mohli dostať na všetky čižmy, okrem tých, ktoré sú na ceste na pozíciách $x + A_j + B_j + 1$ až $x + A_i + B_i$. (pre každé j musí platiť, že $A_j + B_j \leq A_i + B_i$, keďže $A_i + B_i$ je maximálne).

Keďže teraz vieme, že máme na výber stále iba z podmnožiny čižiem dosiahnuteľných z i, ak zoberieme čižmy i, vieme, že výber čižiem v ďalšom kroku bude najväčší možný. Z toho vyplíva, že po k krokoch budeme vždy najďalej ako sa dá, pretože stále robíme najväčší krok, ktorý nám aj ako sme si ukázali, najviac zväčší výber. To nám zaručí, že dôjdeme na koniec cesty v najmenšom počte krokov, ak je to možné. Ak to možné nie je, zistíme to tak, že z aktuálnych čižiem sa už nie je možné dostať do iných čižiem, ktorými vieme dôjsť ďalej a nie je možné sa s nimi dostať ani do cieľa.

Prvé riešenie

Popísaný postup môžeme simulovať a dostaneme sa k nejakému riešeniu. Z každej pozície, kde sme si obuli nové čižmy sa pozrieme na čižmy v dosiahnuteľnej vzdialenosti vpravo. Spomedzi týchto čižiem následne vyberieme také, ktorými sa dostaneme najďalej a tento proces opakujeme, až dokým nedôjdeme na koniec cesty.

Časová zložitosť

Aj keď tento algoritmus možno na prvý pohľad vyzerá, ako $O(n^2)$, vieme dokázať, že na žiadne čižmy sa nepozrieme viac ako dvakrát. Ak sa pozeráme na čižmy i, prezrieme všetky čižmy na úseku A_i až po $A_i + B_i$. Z týchto vyberieme najlepšie čižmy j. Následne sa pozrieme na všetky čižmy na úseku od A_j až po $A_j + B_j$. Druhýkrát sa počas toho pozrieme na čižmy na úseku od A_j po $A_i + B_i$. Nikdy sa ale nepozrieme na tieto čižmy tretíkrát, pretože na tomto úsek už nemôžu byť žiadne lepšie čižmy (s ktorými by sme sa dostali ďalej ako $A_j + B_j$), pretože inak by sme ich zobrali namiesto j. Teda nutne vyberieme nejaké, čo sa nachádzajú až za $A_i + B_i$. Toto platí pre každú dvojicu po sebe idúcich vybratých čižiem a teda na všetky úseky na ktoré sa pozrieme druhýkrát sa nepozrieme už tretíkrát.

Z toho už vyplýva, že časová zložitosť je $O(2n) = O(n)$, keďže čižmy sú už na vstupe zoradené. Pamäťová zložitosť je tiež $O(n)$.

Druhé riešenie

Existuje aj iné, rovnako efektívne riešenie s ľahšou analýzou časovej zložitosti. Môžeme si všimnúť, že zaujímajú nás iba také čižmy i, že $A_i + B_i > A_j + B_j$ pre všetky $i > j$. Čižmy tak stačí postupne prejsť a zapamätať si iba tie pre ktoré platí spomenutá podmienka. Overovať všetky j nie je treba, pretože posledné zapamätané čižmy budú stále tie s najväčším dosahom a tak stačí podmienku skontrolovať pre nich.

Následne postupne prejdeme novo vytvorený zoznam a stále keď máme čižmy i, zoberieme posledné zapamätané čižmy j, ktore sú za i, a ktorých $A_j < A_i + B_i$. Tými sa už dostaneme najďalej ako je to možné zo všetkých dosiahnuteľných čižiem od čižiem i. Toto opakujeme podobne ako v prvom riešení. Druhé riešenie je teda veľmi podobné tomu prvému, akurát odstránime nejaké čižmy, aby sme sa v druhom cykle pozreli na všetky ostávajúce čižmy iba raz. Časová aj pamäťová zložitosť je rovnaká ako pri prvom riešení.

Listing programu (Python)

```python
[N, D, B] = map(int, input().split())
a, b, = [0] * (N + 1), [0] * (N + 1)
a[0] = 0
b[0] = B
for i in range(N):
    [a[i + 1], b[i + 1]] = map(int, input().split())

last, ans = 0, 0
while a[last] + b[last] < D:
    best = last
    for i in range(last + 1, N + 1):
        if a[i] > a[last] + b[last]:
            break
        if a[i] + b[i] > a[best] + b[best]:
            best = i
    if best == last:
        ans = -1
        break
    ans += 1
    last = best
print(ans)
```

Listing programu (C++)

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int prsum(pair<int, int> pr) {
```

```
    return pr.first + pr.second;
}

int main() {
    cin.tie(0)->sync_with_stdio(0);

    int N, D, B;
    cin >> N >> D >> B;
    vector<pair<int, int>> a(N + 1);
    a[0] = { 0, B };
    for (int i = 1; i <= N; i++) {
        cin >> a[i].first >> a[i].second;
    }

    int last = 0, ans = 0;
    while (prsum(a[last]) < D) {
        int best = last;
        for (int i = last + 1; i <= N && a[i].first <= prsum(a[last]); i++) {
            if (prsum(a[i]) > prsum(a[best])) {
                best = i;
            }
        }
        if (best == last) {
            ans = -1;
            break;
        }
        ans++;
        last = best;
    }
    cout << ans << endl;
}
```

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

4. Existenčná kríza

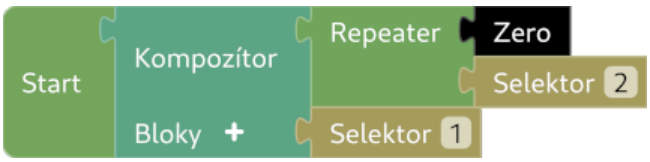
Pokiaľ pri podúlohe nie je uvedené inak, tak si vstupy označíme postupne a, b, c, ...

a. nula

Na to, aby sme zavolali Zero bez vstupov, využijeme Kompozítor a Repeater.

Kompozítor vyžaduje aspoň dva bloky, z čoho vyplýva, že `final` blok dostane vždy aspoň jeden vstup. Kompozítor tak vieme využiť na eliminovanie počtu vstupov na jeden.

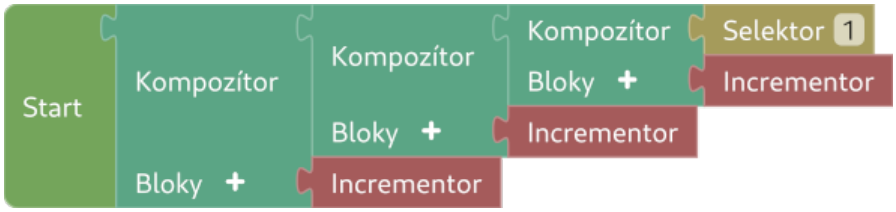
Teraz sa pozrieme na Repeater - ako prvý vstup berie počet opakovaní a ostatné vstupy sú voliteľné. Keď Repeater zavoláme s jedným vstupom, tak `init` nedostane žiadny vstup, čiže ako `init` vieme dať Zero. Už len si potrebujeme túto hodnotu udržať, a tak `step` musí vždy vrátiť priebežnú hodnotu. Na to už len použijeme Selektor.



b. +3

Použijeme vnorené Kompozítory, aby sme postupne 3× zavolali Increment na vstupe.

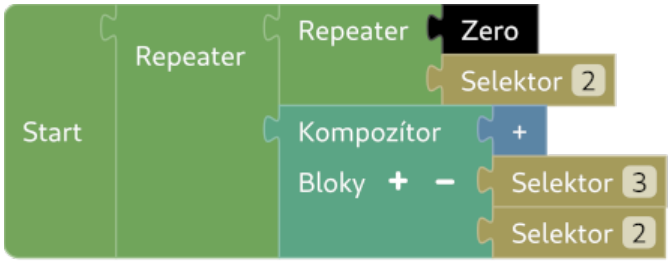
Skúste si uvedomiť, že pomocou podúlohy a. takto vieme vyrobiť ľubovoľnú nezápornú konštantu s ľubovoľným počtom vstupov.



c. násobenie

Násobenie spravíme pomocou postupného sčítania (to je vysvetlené v tutoriáli). Pomocou Repeatera budeme volať sčítavanie, ktoré zoberie predchádzajúcu hodnotu a druhý vstup Repeatera.

A čo bude počiatočná hodnota Repeatera? Takýmto spôsobom a krát pripočítame b, čiže začiatočná hodnota musí byť 0. Na to môžeme využiť nulu z podúlohy a.

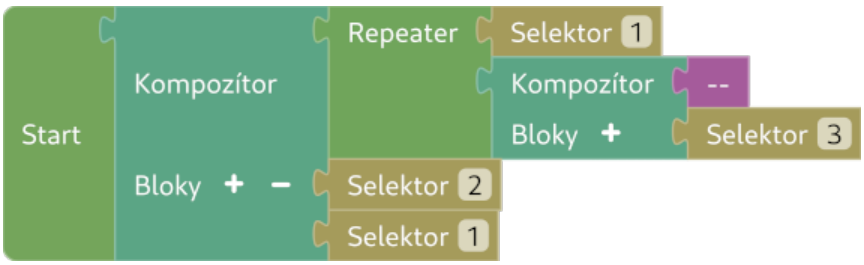


d. decrement

Trik spočíva v tom, že Repeater počíta iterácie od 0 po $n-1$. Použijeme Repeater s počiatočnou hodnotou 0 (špeciálny prípad) a `step` vráti číslo iterácie.

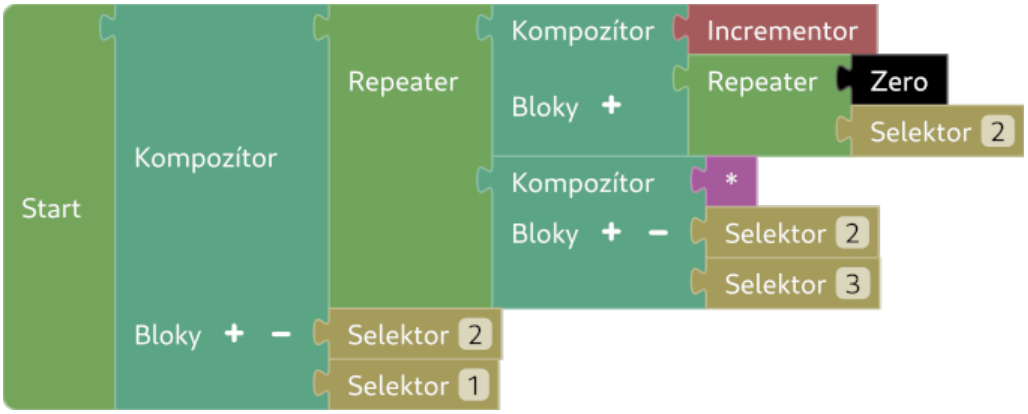
e. odčítanie

Odčítanie spravíme podobne ako sčítanie, ale použijeme decrement namiesto Incrementora a vymeníme poradie vstupov pomocou Kompozítora, aby sme sme odčítali b číslo od a.



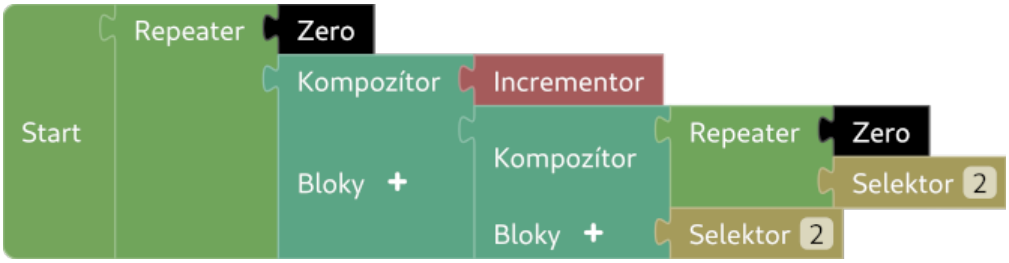
f. umocňovanie

Obdobne ako sme pri násobení použili sčítanie, teraz použijeme násobenie. Avšak tento krát počiatočná hodnota bude 1 (keby sme nechali 0, tak výsledok bude vždy 0), čiže použijeme pozorovanie z podúlohy b. Nakoniec ešte musíme vymeniť poradie vstupov, aby sme b krát vynásobili a a nie naopak.



g. sign

Potrebujeme vrátiť 0, ak je vstup 0, inak vrátiť 1. Na to použijeme Repeater - ak $a = 0$, tak Repeater vráti počiatočnú hodnotu cyklu, ktorú nastavíme na 0. Ak $a > 0$, tak Repeater by mal Repeater vrátiť 1 - na to vieme opäť použiť pozorovanie z podúlohy b.

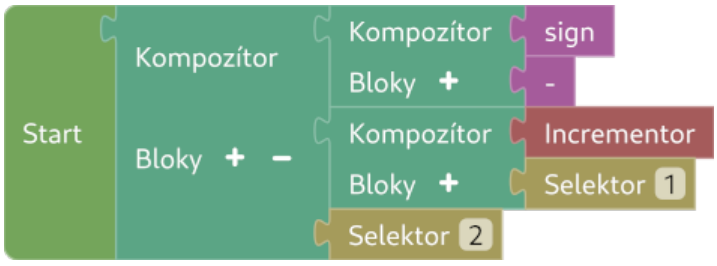


h. >=

$a - b$ nám vráti 0 ak $a \leq b$, v opačnom prípade vráti kladné číslo. My ale chceme vrátiť iba 0 alebo 1 a na to vieme použiť `sign` z podúlohy g. Takto dostaneme 0 ak $a \leq b$ a v opačnom prípade 1.

Ak chceme dostať 0 ak $a < b$ a v opačnom prípade 1, musíme rovnosť posunúť. Zväčšením a o jedna, dostaneme $a + 1 - b$, ktoré vráti 0 ak $a < b$ a inak vráti 1 (po použití `sign`).

Skúste si uvedomiť, že podobným spôsobom vieme vytvoriť aj funkcie `<`, `<=` a `>`.



i. if

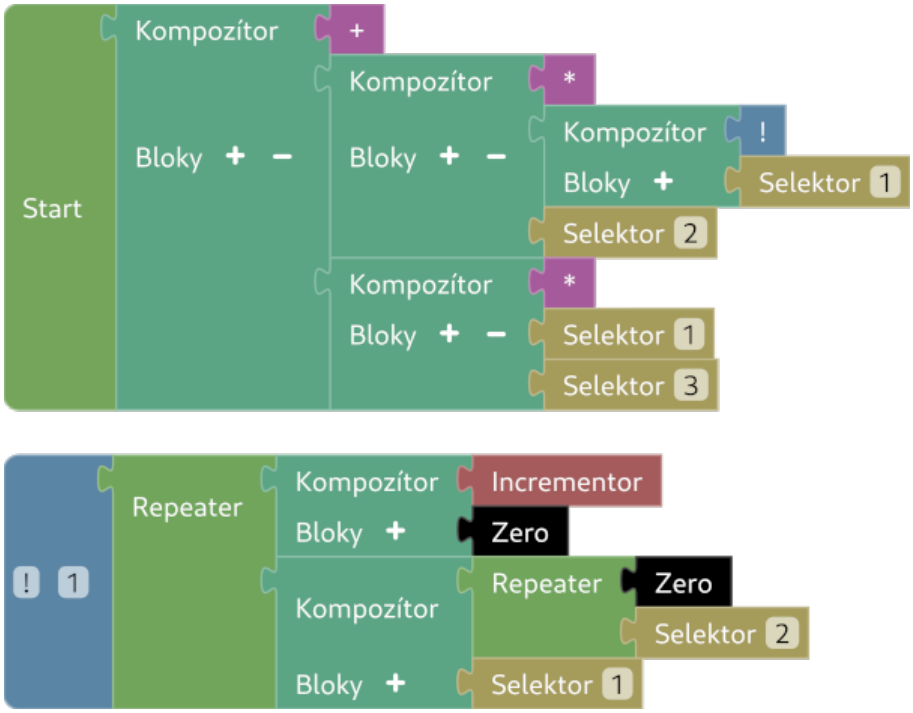
Označme si vstupy postupne c (condition), i (if) a e (else).

Začnime tým, že si definujeme funkciu `!` , ktorá zneguje svoj jediný vstup - ak je 0, tak vráti 1, inak vráti 0. Môžeme si všimnúť, že je to vlastne `sign`, len s vymenenými konštantami - `init` bude 1 a `step` bude 0.

Teraz, keď už vieme znegovať c, môžeme si skúsiť úlohu vyjadriť matematicky:

$$v = !a * c + b * c$$

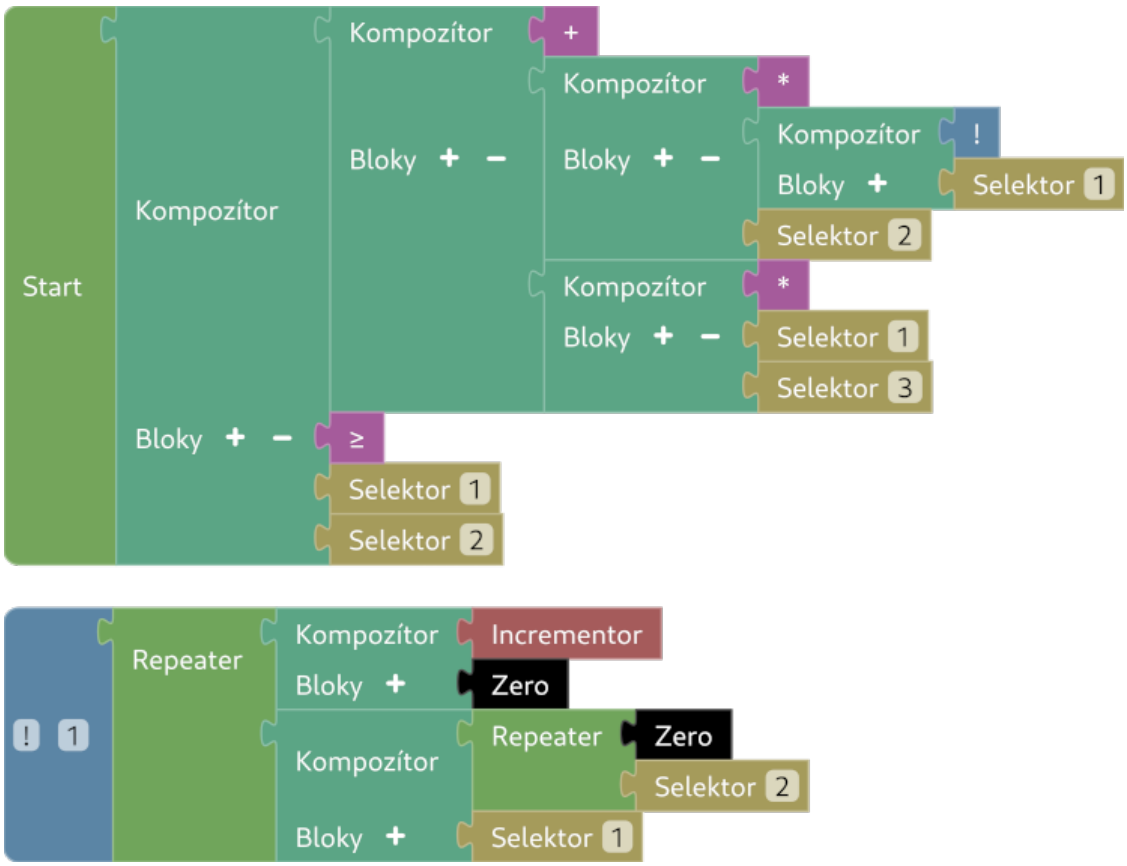
Ak $c = 0$, tak $v = b$, inak $v = a$, čo je to, čo máme urobiť. A to už vieme urobiť pomocou predchádzajúcich podúloh a vnorených Kompozítorov.



j. min

Táto podúloha sa veľmi podobá na podúlohu i, až na to, že nemáme c na vstupe. c si vieme získať pomocou podúlohy h. Čiže nám stačí pomocou Kompozítora a pretransformovať vstup pre program z podúlohy i.

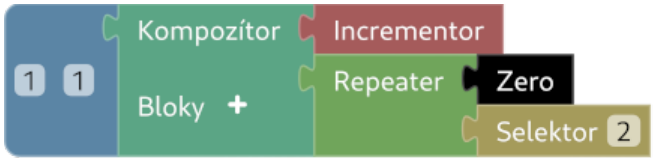
Skúste si uvedomiť, že `max` vieme vytvoriť použitím `<=` z podúlohy h.

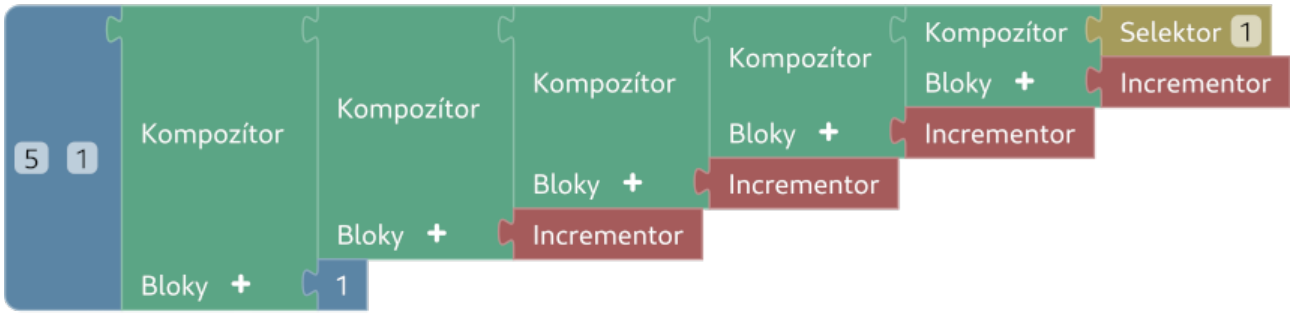


k. bonus

Pre detaily ohľadom riešenie pôvodnej úlohy si pozrite vzorák prvej úlohy.

Najskôr si definujme konštanty `10` a `1` s jedným vstupom:



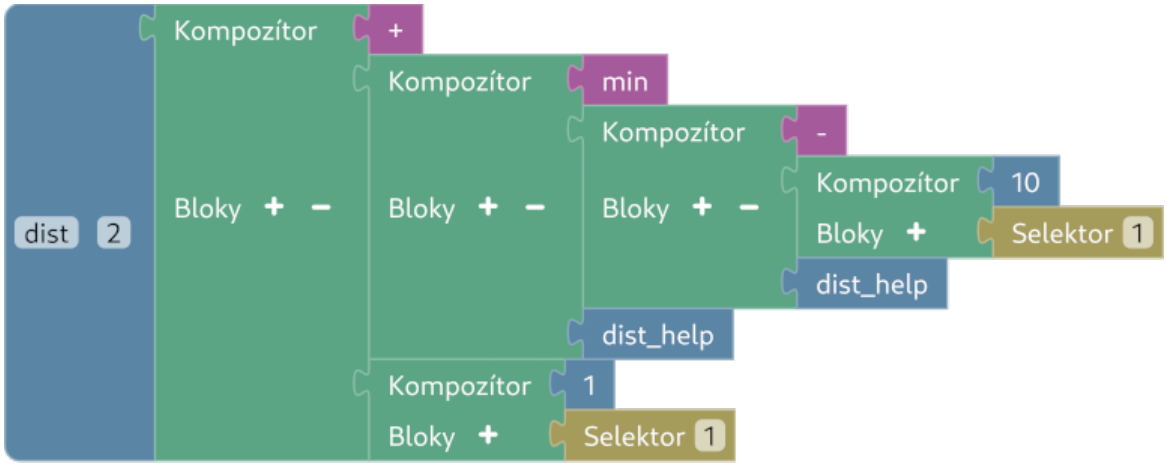


Ďalej si potrebujeme vyrobiť funkciu `dist` s dvoma vstupmi a a b, ktorá vypočíta vzdialenosť dvoch pozícií (vstupov) + 1 (kliknutie).

Vzdialenosť dvoch pozícií je $\min(10 - \text{abs}(a - b), \text{abs}(a - b))$

Vieme spraviť všetko, okrem `abs`. My ale `abs` nepotrebujeme, stačí nám vždy odčítavať menšie číslo od väčšieho, čo vieme spraviť ako: $\max(a, b) - \min(a, b)$. Túto funkciu si pomenujme `dist_help`.

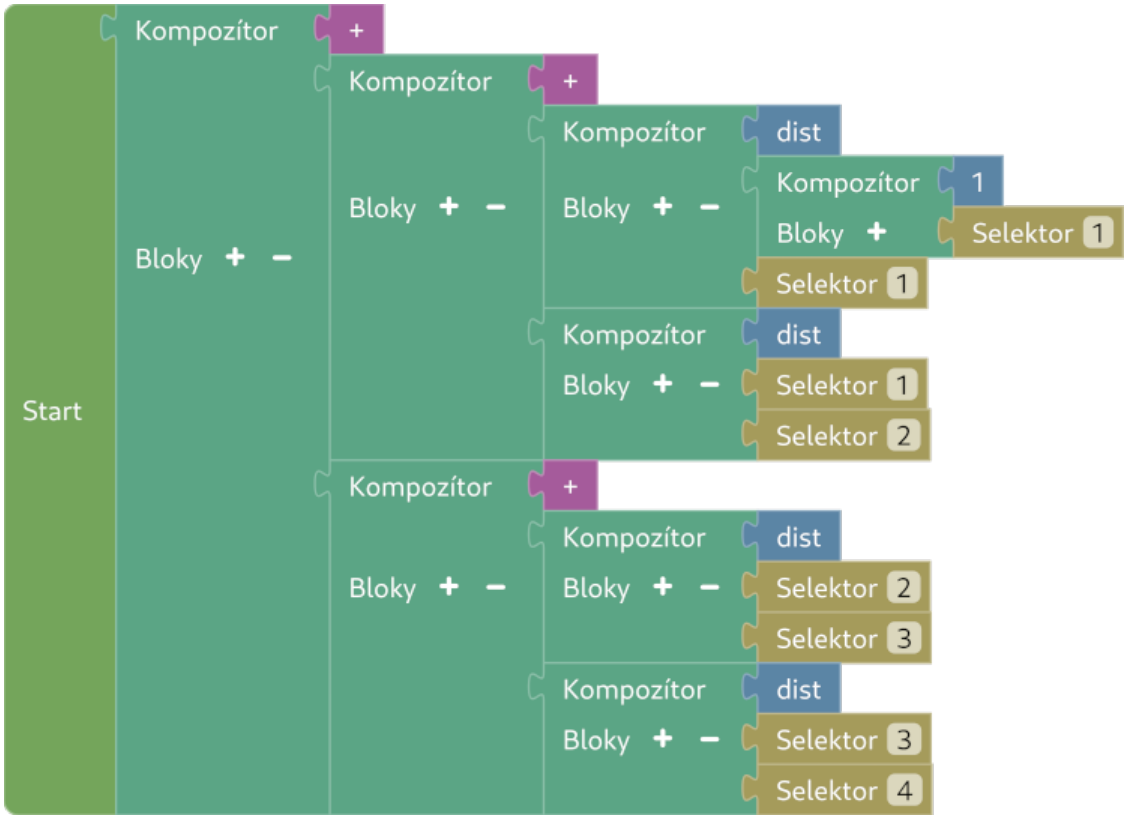
Počítanie vzdialenosti tak vyzerá nasledovne:



Keď už vieme vypočítať vzdialenosť dvoch pozícií, už len potrebujeme dostať finálny výsledok. To vieme tak, že postupne sčítame medzivýsledky:

$$\text{dist}(1, a) + \text{dist}(a, b) + \text{dist}(b, c) + \text{dist}(c, d)$$

Takto dostaneme konečný výsledok (1 je počiatočná pozícia, ako je definované v zadaní):



Strizo

5. Zašifrovávať budeme

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

Bruteforce

Prvé čo nám napadne je, že však poďme si to odsimulovať a tým pádom na konci len vypíšeme k-te pismenko z nášho výsledneho stringu. Toto riešenia má časovú zložitosť $O(D^N)$ kde D je, že koľko najviac písmenok sa vie stať z jedného písmenka, a N je počet "zámien" ktoré máme spraviť. Pamäťová zložitosť je $O(D^N)$, keďže až po takto veľký náš výsledok vie narásť.

Niečo lepšie

Ďalej si môžeme všimnúť, že my si nemusíme odsimulovávať vždy celý string, ale môžme ísť postupne po písmenkách. Čiže namiesto toho, aby sme z "abc" si vyvorili "aakfhjioufdljfskmdklmdscdk", sa môžme pozrieť iba na prvé písmenko, a na prvé písmenko čo nám vznikne z tohto písmenka. Týmto pádom nám vznikne niečo ako strom kde každý vrchol je písmenko a z neho vedú hrany k písmenkam ktoré z neho môžu vzniknúť. Tento spôsob nám ale dovoľuje aby sme zastali v generovaní vtedy keď sa dostaneme na k-tu pozíciu a tým pádom nemusíme generovať celý výsledný string. A akú to má časovú zložitosť? Takže je nám jasné, že ak by sme sa pýtali na posledné písmenko zo stringu, tak by sme ho museli vygenerovať celý, a teda worst-case by bola $O(D^N)$, ale keďže vieme prerušiť generovanie keď dôjdeme k nášmu k-temu prvku, tak časová zložitosť bude $O(min(K, D^N))$, čo keď si všimneme obmedzenia by nám malo vedieť prejsť prvými troma sadami. A pamäťová zložitosť sa dá optimalizovať na $O(KD)$, keďže si musíme pamätať ku každému písmenku za čo ho zamieňame a potom už len nejaký zásobník veľkosti &n& aby sme sa vedeli hýbať po našom strome.

Zaujímavá myšlienka

My ale v skutočnosti nepotrebujeme vedieť, aké boli tie písmenká pred našim k-čkom. To znamená že ak nájdeme rozumný spôsob ako povedať, že koľko písmenok vznikne z tohto nášho za J vnorení, potom vieme povedať, či sa nám ho oplatí úplne preskočiť, alebo je naše riešenie v ňom a teda ho pôjdeme generovať

Optimálne riešenie

Čiže čo sa snažíme zistiť je, že pre každé písmenko, koľko sa z neho stane po J vnoreniach. Znie povedome? Áno, vytvorili sme si podproblémy, ktoré na seba rozumne nadväzujú. Je nám totiž jasné, že keď viem pre každé

písmenko koľko z neho vznikne iných po J opakovaniach, tak potom keď chcem to isté vedieť pre niektoré moje písmenko po $J+1$ opakovaniach, tak sa mi len stačí pozrieť na aké všetky sa zmení po jednej premene a spočítať hodnoty tých, na ktoré by sa premenila, ktoré majú po J opakovaniach. A keď už všetky tieto informácie mám, tak potom viem postupne prechádzať cez úvodný string, a ak moje K-te pismenko sa vytvori z toho písmenka, na ktoré sa pozerám, tak si ho rozložím a znova sa pozriem, či sa moje k-te písmenko nachádza pod prvým alebo druhým atď. To, pod ktorým písmenkom sa nachádza moje, viem povvedať tak, že ak mám spočítané, koľko mi dajú všetky písmenká pred ním, tak ak to písmenko, na ktorom som + tie, ktoré som už prešiel, je väčšia alebo rovná ako K tak vtedy sa k vytvorí z tohto môjho písmenka, keďže je nám jasné, že suma tých pred týmto mojim je menej ako K, inak by sme sa zastavili na nejakom predchádzajúcom písmenku. Toto riešenie má časovú zložitosť $O(D_N)$, keďže generujeme tak veľkú súčtovú tabuľku a nájdenie výsledku je taktiež najhoršie $O(D_N)$, keďže by sme N krát mohli prejsť cez celý string čo vytvárame z daného písmenka ktorý je maximálne D. A pamäťová zložitosť je taktiež $O(DN)$, keďže túto súčtovú tabuľku si musíme pamätať.

Listing programu (Python)

```python
povodny_string = input()
pocet_vnoreni, kolkate_chcem = [int(i) for i in input().split()]
poctova_matica = [[0] * 26 for i in range(pocet_vnoreni)]
pismenk_matica = []
preskocene = 0
kolkate_pismenko = -1
possible = False

# vytvorenie si tabulky na co sa mi meni kazde pismeno
for i in range(26):
    pis = input()
    if pis == "#":
        pismenk_matica.append("")
    else:
        pismenk_matica.append(pis)
    # nahodenie to rovno aj do prveho riadku suctovej tabulky
    poctova_matica[0][i] = len(pismenk_matica[i])

# vytvorenie suctovej tabulky
for i in range(1, pocet_vnoreni):
    for j in range(26):
        poctova_matica[i][j] = sum(
            [poctova_matica[i - 1][ord(ch) - ord("a")] for ch in pismenk_matica[j]]
        )

# prejdenie cez prve slovo
i = -1
for j in range(len(povodny_string)):
    if (
        preskocene + poctova_matica[i][ord(povodny_string[j]) - ord("a")]
        < kolkate_chcem
        or pismenk_matica[ord(povodny_string[j]) - ord("a")] == ""
    ):
        preskocene += poctova_matica[i][ord(povodny_string[j]) - ord("a")]
    else:
        # ak sme nasli pismenko, ktore chceme tak nastavime, ze je to mozne a zapiseme si ktore
        ↪ pismenko to je
        possible = True
        kolkate_pismenko = ord(povodny_string[j]) - ord("a")
```

```
        break

if possible:
    # prejdenie az na poslednu vrstvu
    for i in range(pocet_vnoreni - 2, -1, -1):
        j = 0
        # dokym nenajdeme pismenko z ktoreho vznikne nase k-te pismenko
        while (
            preskocene
            + poctova_matica[i][ord(pismenk_matica[kolkate_pismenko][j]) - ord("a")]
            < kolkate_chcem
            or pismenk_matica[kolkate_pismenko] == ""
        ):
            preskocene += poctova_matica[i][
                ord(pismenk_matica[kolkate_pismenko][j]) - ord("a")
            ]
            j += 1
        # zapiseme si ktore pismenko to je
        kolkate_pismenko = ord(pismenk_matica[kolkate_pismenko][j]) - ord("a")

    # z poslednej vrstvy urcenie finalneho pismenka
    kolkate_pismenko = pismenk_matica[kolkate_pismenko][kolkate_chcem - preskocene - 1]

    print(kolkate_pismenko)

else:
    print("Neexistuje")
```

Listing programu (C++)

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

struct nefinalne_pismenko{
        long long kolko_este;
        char pismenko;
};

int main(){

        string slovo;
        cin >> slovo;

        long long n,k;
        cin >> n >> k;

        vector<string> rules(26);
        for(long long i = 0; i < 26; i++){
                cin >> rules[i];
                if(rules[i] == "#"){
```

```
                rules[i] = "";
        }
    }

    vector<vector<long long>> lengths(26);
    for(long long i = 0; i < 26; i++){
        lengths[i].push_back(1);
    }

    for(long long i = 1; i <= n; i++){
        for(long long j = 0; j < 26; j++){
            string change = rules[j];
            long long result = 0;
            for(long long k = 0; k < change.length(); k++){
                result += lengths[change[k]-'a'][i-1];
            }
            lengths[j].push_back(result);
        }
    }

    vector<nefinalne_pismenko> stack;
    for(long long i = slovo.length()-1; i >= 0; i--){
        nefinalne_pismenko nove;
        nove.pismenko = slovo[i];
        nove.kolko_este = n;
        stack.push_back(nove);
    }

    long long este_zjest = k-1;

    while(true){

        if(stack.empty()){
            cout << "Neexistuje" << endl;
            break;
        }

        nefinalne_pismenko dalsie = stack[stack.size()-1];
        long long velkost_dalsieho = lengths[dalsie.pismenko-'a'][dalsie.kolko_este];

        if(velkost_dalsieho <= este_zjest){
            stack.pop_back();
            este_zjest -= velkost_dalsieho;
        }else{
            if(dalsie.kolko_este == 0){
                cout << dalsie.pismenko << endl;
                break;
            }
            stack.pop_back();
            long long new_depth = dalsie.kolko_este-1;
            string replacing_string = rules[dalsie.pismenko-'a'];
            for(long long i = replacing_string.length()-1; i >= 0; i--){
                nefinalne_pismenko nahrada;
```

```
                                nahrada.pismenko = replacing_string[i];
                                nahrada.kolko_este = new_depth;
                                stack.push_back(nahrada);
                            }
                        }

                }

        }
```

Viktor

6. Diskrétny Banket

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

Zadrátované riešenie

Najprv sa pozrime na veličiny, ktoré v úlohe vystupujú. Všimnime si, že otázok je síce veľa, ale čísla v nich budú pomerne malé. Vyzerá to teda, že optimálne bude výsledky pre čísla počítať dopredu, respektíve ich vypočítať, keď ich potrebujeme, a odvtedy si ich už pamätať. Možných otázok je totiž pomerne málo… V prvej sade dokonca platí, že žiadne čísla nie sú zakázané - teda v tejto sade bude odpoveď na dané číslo zakaždým rovnaká. Prvú sadu teda vieme jednoducho zriešiť tak, že najprv si nejakým ľubovoľne pomalým riešením vygenerujeme odpovede pre prvých 42 čísel, a potom len napíšeme program, ktorý podľa čísla na vstupe zvolí správnu odpoveď. Problémom bude, keď sa v neskorších sadách objavia aj zakázané čísla. Vtedy totiž taká istá otázka môže mať rôzne odpovede, v závislosti od toho, ktoré čísla sú zakázané… Časová zložitosť je $O(1)$, a pamäťová zložitosť je $O(a)$.

Bruteforce

Mohli by sme pre každé číslo na vstupe vygenerovať všetky možné podpostupnosti $2, 3, 4...a$ a pre každú otestovať, či spĺňa všetky zadané podmienky. Týchto podpostupností je rádovo 2^a. Každú z nich musíme otestovať - ako to spraviť pomerne rýchlo? Najprv overíme, či naozaj skákaním z posledného čísla skončíme na 1. To vieme overiť jediným prechodom - prejdeme postupnosť od konca, a vždy, keď narazíme na číslo, na ktorom teraz sme, pozrieme sa na jeho index a ten si odteraz pamätáme. Pokiaľ pred dorazením na 1 nenájdeme v zozname nejaké číslo, ktoré sme dostali, vieme, že postupnosť nevyhovuje. Popri tom ešte musíme kontrolovať, či neprechádzame cez zakázané číslo. Prvotný nápad je zakaždým prezrieť celý zoznam zakázaných čísel, či sa v ňom to naše nenachádza. Stačí nám ale použiť techniku dvoch bežcov: keďže zakázaný zoznam je zoradený, a čísla, ktorých prítomnosť v ňom kontrolujeme, sa vždy znižujú, stačí nám pamätať si pozíciu medzi zakázanými číslami, teda "medzeru", do ktorej by momentálne číslo patrilo. Na začiatku bude na konci, a posunieme ju bližšie k začiatku vždy, keď kontrolujeme číslo, ktoré je pod spodným okrajom tejto medzery. Takto toto miesto posúvame iba na začiatok, a teda počas celého kontrolovania postupnosti prejdeme jeho r prvkov najviac raz. Teda okontrolovať postupnosť má zložitosť $O(r + a)$. V každej z k otázok ale testujeme 2^a postupností, teda celková zložitosť algoritmu je $O(k * 2^a * (r + a))$. Pamäťová zložitosť je $O(r + a)$, lebo si pamätáme postupnosť dĺžky až $a - 1$, a r zakázaných čísel.

Optimálne riešenie

Pomôže nám dynamické programovanie. Budeme si pamätať pre všetky dvojice a, n počet postupností končiacich na a, ktoré majú práve n prvkov ($n >= 2$). Zakaždým, keď príde otázka, sčítame počty pre dané a pre všetky možné dĺžky n, ak ich už poznáme. Ak ešte nie, postupne budeme počítať tieto hodnoty pre najnižšie nevypočítané a, až kým sa k otázke nedostaneme.

Dobre teda, ale ako vypočítame odpoveď pre dvojicu a, n? Využijeme pri tom, samozrejme, odpovede pre nižšie a. Majme teda postupnosť končiacu na a dĺžky \$n%. Hneď vieme, aký bude prvý krok pri skákaní po tejto postupnosti. Keďže a je na pozícií n, skočíme na číslo n. Na akej pozícií môže byť toto n? Na akejkoľvek od 1 po $n - 1$. Prejdime všetkými a pre každú vypočítajme, koľko takých postupností existuje. Označme pozíciu, na ktorej je n, ako s. Uvedomme si, že prvých s čísel, až po číslo n, bude tiež vyhovujúcou postupnosťou (začneme na jej konci a doskáčeme na jednotku). Teda takýchto postupností je už vypočítané množstvo - dvojica n, s. Ale čo pozície po tej s-tej? Po žiadnom z jej čísel nikdy nebudeme skákať, teda môžu to byť úplne ľubovoľné čísla medzi n a a. Teda predpočítanú dvojicu n, s vynásobíme kombinačným číslom $a - n - 1$ nad $n - s - 1$. Kombinačné čísla si tiež predpočítame - zakaždým, keď potrebujeme nové, sčítavame tie pred ním, využívajúc vzťah $(a - 1nadb) + (a - 1nadb - 1) = (anadb)$.

Ešte jeden detail nám chýba - zakázané čísla. Jednoducho, vždy keď počítame počet postupností pre dvojicu a, n, tak ak náhodou a je zakázané, namiesto počítania hneď vrátime nulu. Na rozdiel od minulého riešenia, tentokrát sa čísla, na ktorých zakázanie sa pýtame, zvyšujú, tak použijeme rovnakú techniku, ako minule, len sa poľom zakázaných čísel hýbeme od začiatku po koniec.

Aká je teda zložitosť tohto riešenia? Potrebujeme najviac raz prejsť pole zakázaných čísel, teda $O(r)$. Musíme vypočítať odpovede pre všetky dvojice posledného čísla a dĺžky postupnosti, čo je $O(a^2)$ - pretože pre každú dvojicu stačí vynásobiť premennú kombinačným číslom v konštantnom čase. No a samozrejme, predpočítavame kombinačné čísla. Teda musíme vypočítať všetky kombinačné čísla až po (*anad?*), a tých je $O(a^2)$. Teda celková časová zložitosť je $O(a^2 + r)$. Pamäťová zložitosť je tiež $O(a^2 + r)$, pretože toľko výsledkov si predpočítavame, a k tomu máme pole zakázaných čísel.

Listing programu (Python)

```python
#!/usr/bin/python

p = 10**9 + 7

c = [[1]]
def comb(n,k):
    #vypocitame kombinacne cislo cez pascalov trojuholnik
    while len(c) <= n:
        past = 0
        newrow = []
        for i in c[-1]:
            newrow.append((i+past) % p)
            past = i
        newrow.append(past)
        c.append(newrow)

    return c[n][k]

r,k = input().split()
#pocet bezpecnych cisel
r = int(r)
#pocet testov
k = int(k)

#zostrojime zoznam bezpecnych cisel
s = []
for i in range(r):
    s.append(int(input()))

#ukazujeme na poziciu v tomto zozname, kedze ho prechadzame zaradom
spos = 0

#dyn pre kazde cislo a obsahuje zoznam dlzky a-1: vsetky mozne pocty moznosti, ak okrem daneho
↪  cisla mame este pred nim v mnozine 0 az a-2 cisel
dyn = [0,0]
#answers obsahuje celkovy sucet moznosti pre kazde cislo - rovny suctu jeho zoznamu v dyn
answers = [0,0]

while k:
    k-=1
    #pre kazdy test sa vykona tento cyklus
```

```
41
42      a = int(input())
43
44      #pokial este nemame odpoved pre vstup, dorobime zoznamy pre nizsie cisla
45      while a >= len(answers):
46          #n je cislo, ktoreho pocet moznosti teraz pocitame
47          n = len(answers)
48          #pokial n je bezpecne, hodnota bude vzdy 0
49          while spos < len(s) and s[spos] < n:
50              spos+=1
51          if spos < len(s) and n == s[spos]:
52              #same nuly, pretoze je to zakazane
53              entry = [0]*(n-1)
54              dyn.append(entry)
55              answers.append(0)
56              continue
57
58          #entry bude obsahovat zoznam poctu moznosti pre vsetky velkosti mnoziny
59          entry = []
60          #pokial okrem n mame v mnozine 0 cisel, tak je jedina moznost, a vzdy spravna
61          entry.append(1)
62
63          #pokial v mnozine chce byt cisel viac, treba vypocitat pocet moznosti komplikovanejsie:
64          for before in range(1,n-1):
65              #postupne spocitame vsetky moznosti, total si pamata ich pocet
66              total = 0
67              #vieme, ze n je (before+1)-te najvacsie, preto niekde v mnozine urcite bude cislo
                ↪  before+1
68              pivot = before+1
69
70              #pozrime sa na kazdu moznost, kolko cisel z mnoziny moze ist pred pivotom
71              #teoreticky sa zan (vratane) zmesti len obmedzene mnozstvo cisel:
72              space_after_pivot = n-pivot
73              #teda taketo je najnizsie mnozstvo cisel, ktore mozu byt pred pivotom:
74              least_before_pivot = before-space_after_pivot
75              #prejdeme teraz vsetky pripustne moznosti (hornu hranicu netreba osetrovat, before-1 sa
                ↪  vzdy zmesti pred pivot, pretoze pivot je before+1)
76              for before_pivot in range(max(0,least_before_pivot),before):
77                  #pocet takychto moznosti - sucin moznosti, ako pred pivot dostat before_pivot cisel
                    ↪  (vdaka dynamike uz vieme), no a zvysok je umiestnitelny lubovolne, kombinacnym
                    ↪  cislom:
78                  total += (dyn[before+1][before_pivot]) *
                    ↪  (comb(space_after_pivot-1,before-1-before_pivot))
79                  total %= p
80              entry.append(total)
81
82          #entry prihodime do pamate
83          dyn.append(entry)
84          #zapiseme aj odpoved
85          answers.append(sum(entry)%p)
86
87
```

```
    print(answers[a])
```

Listing programu (C++)

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

vector<vector<ll>> c;
ll p = 1000000007;

ll comb(ll n,ll k){
    //vypocitame kombinacne cislo cez pascalov trojuholnik
    while(c.size() <= n){
        vector<ll> newrow;
                if(c.size() == 0){
                        newrow.push_back(1);
                }else{
                        ll past = 0;
                        ll prev_entry = c.size()-1;
                        for(ll i = 0; i < c[prev_entry].size(); i++){
                                ll curr = c[prev_entry][i];
                                newrow.push_back((curr+past)%p);
                                past = curr;
                        }
                        newrow.push_back(past);
                }
        c.push_back(newrow);
        }

    return c[n][k];
}

int main(){

        cin.tie(0);
        ios::sync_with_stdio(0);

        ll r,k;
        cin >> r >> k;
        vector<ll> safe;
        for(ll i = 0; i < r; i++){
                ll s;
                cin >> s;
                safe.push_back(s);
        }
        ll safe_pointer = 0;

        vector<ll> answers(2);
        vector<vector<ll>> dynamic(2);

        while(k){
```

```
49                  k--;
50
51                  ll a;
52                  cin >> a;
53
54                  while(answers.size() <= a){
55
56                          ll n = answers.size();
57
58                          while(safe_pointer < safe.size() && safe[safe_pointer] < n){
59                                  safe_pointer++;
60                          }
61                          if(safe_pointer < safe.size() && safe[safe_pointer] == n){
62                                  vector<ll> empty_entry(n-1);
63                                  dynamic.push_back(empty_entry);
64                                  answers.push_back(0);
65                                  continue;
66                          }
67
68                          vector<ll> new_entry;
69                          ll entry_sum = 0;
70
71                          new_entry.push_back(1);
72                          entry_sum++;
73                          entry_sum%=p;
74
75                          for(ll i = 1; i < n-1; i++){
76                                  ll total = 0;
77                                  ll pivot = i+1;
78                                  ll fit_before_pivot = pivot-2;
79                                  ll fit_after_pivot = n-pivot-1;
80                                  ll needed_before_pivot = i-1-fit_after_pivot;
81                                  if(needed_before_pivot < 0){
82                                          needed_before_pivot = 0;
83                                  }
84                                  for(ll before_pivot = needed_before_pivot; before_pivot <=
    ↪  fit_before_pivot; before_pivot++){
85                                          ll after_pivot = i-1-before_pivot;
86                                          total += (dynamic[pivot][before_pivot]) *
    ↪  (comb(fit_after_pivot,after_pivot));
87                                          total %= p;
88                                  }
89                                  new_entry.push_back(total);
90                                  entry_sum += total;
91                                  entry_sum %= p;
92                          }
93
94                          dynamic.push_back(new_entry);
95                          answers.push_back(entry_sum);
96                  }
97
98                  cout << answers[a] << "\n";
99
```

```
100         }
101
102 }
```

fezjo

7. Nevítaný hosť s chlpatým kožuchom

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

Zadanie úlohy je podobné Game of Life, ale s inými pravidlami. Pravidlá sú nasledovné:

1. Bunka (x, y) je v novom kroku živá, ak aspoň dve z troch buniek $\{(x, y), (x-1, y), (x, y-1)\}$ sú v aktuálnom kroku živé.
2. Inak je bunka v novom kroku mŕtva.

Plocha bude mŕtva v konečnom čase

Najprv si skúsme uvedomiť, že v každej situácii všetky bunky zomrú v konečnom čase. Všimnime si, že ak sa pozrieme na minimálne a maximálne súradnice všetkých buniek, tak žiadna bunka mimo obdĺžnika $(x_{min}, y_{min}, x_{max}, y_{max})$ nebude nikdy živá. Pre ľubovoľný vstup je teda konečne veľa buniek, ktoré môžu byť potenciálne živé. Pozrime sa na najhorší možný prípad, kedy na začiatku sú všetky bunky v tomto obdĺžniku živé (toto je naozaj najhorší prípad, keďže na rozdiel od Game of Life bunky nezomierajú, ak ich je niekde veľa). V každom čase, kedy existuje aspoň jedna živá bunka existuje aj niekoľko buniek, ktorá majú najmenšie y a v rámci nich existuje práve jedna bunka, ktorá má zároveň najmenšie x spomedzi nich. Táto bunka určite zomrie a už nikdy neožije. Keďže v každom kroku zomrie aspoň jedna bunka, žiadne bunky nepribúdajú a buniek je konečne veľa, všetky bunky zomrú v konečnom čase.

Simulácia – sada 1/5

Vyriešme prvú sadu priamočiarou simuláciou. Pamätajme si všetky bunky v 2D mriežke. V každom kroku sa pozrieme na každú bunku a podľa pravidiel zo zadania určíme, či bude živá alebo nie.

Toto by malo časovú zložitosť $O([\text{veľkosť mriežky}] \cdot [\text{odpoveď}])$. Nech rozmer plochy je S. Najhorší vstup čo môžeme dostať je mriežka $S \times S$ plná živých buniek. Dá sa ukázať, že na tejto ploche všetky bunky zomrú za $2 \cdot S - 1$ krokov. Teda časová zložitosť je $O(S^3)$ a pamäťová $O(S^2)$.

Tento kód môžeme trochu zoptimalizovať tak, že si nám stačí pamätať iba živé bunky (tých by malo byť zvyčajne oveľa menej ako tých mŕtvych) – na to môžeme použiť množinu (*Python* `set`, *C++* `std::unordered_set`). V každom kroku sa potom nemusíme pozerať na každé políčko, ale iba na tie, ktoré sú blízko živých. Konkrétne nám stačí pre každú živú bunku skontrolovať, či zostane žiť a či vďaka nej neožije jej pravý sused. Avšak pre hore spomenutý najhorší vstup $(S \times S)$ bude mať toto riešenie rovnakú časovú zložitosť.

Záleží na komponentoch

Dve bunky alebo obdĺžniky nazveme susedné, ak zdieľajú hranu alebo pravohorný-lavodolný ↗↙ roh. Teda nasledujúce konfigurácie buniek/obdĺžnikov nazývame susedné:

```
Fig. 1
     A. AB .A
     B. .. B.
```

Ďalej budeme hovoriť, že ak sú dve bunky/obdĺžniky susedné, tak patria do toho istého komponentu. Táto vlastnosť je *tranzitívna*, teda ak $A - B$ patrí do toho istého komponentu a $B - C$ patria do toho istého komponentu, tak aj $A - C$ patria do toho istého komponentu.

Tvrdíme, že komponenty sú vždy súvislé a navzájom nezávislé. Ak si skúsime vytvoriť vstupy pre túto úlohu, môžeme si všimnúť, že ak na začiatku existovali nejaké komponenty, tieto komponenty sa počas celej simulácie nikdy neovplyvnili, nespojili ani nerozdelili na viac komponentov.

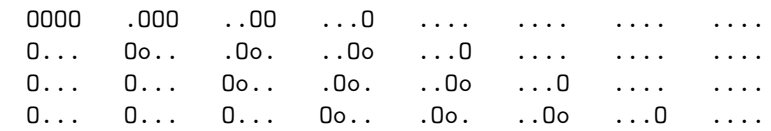
Ak je toto pravda, úlohu vieme riešiť samostatne pre každý komponent a výsledok bude maximum z výsledkov pre jednotlivé komponenty.

Dôkaz uvedieme neskôr.

Záleží na diagonálach

Podobne ak sa pozeráme na simuláciu jedného komponentu tak si všimneme, že bunky zomierajú a vznikajú v skupinách na diagonálach. Pozrite sa ako sa správa tento vstup:

Fig. 2



V každom kroku zomrie jedna diagonála a jedna nová ožije. Komplikovanejšie vstupy sa budú správať podobne.

Spomeňme si na argument, ktorý sme spravili na začiatku, že v každom kroku zomrie aspoň jedna bunka. Tento argument môžeme rozšíriť na celú diagonálu – v každom kroku zomrú všetky bunky na najľavohornejšej diagonále. Navyše, keďže išlo o najľavšiu diagonálu, tieto bunky už nikdy neožijú.

Odbočka – Simulácia s kladivom – sada 2/5

Pozrime sa na jeden komponent. Simulovať bunku po bunke je pomalé, ale už sme si všimli, že diagonály sú zaujímavé. Predstavme si, že na ploche sa nachádza jedna diagonála. Čo sa s ňou stane v ďalšom kroku? Táto jedna diagonála celá umrie, lebo žiadna bunka nemá naľavo a hore živé bunky. Ale vedľa tejto diagonály ožije nová diagonála, ktorá bude o jedno políčko kratšia.

Na diagonále naraz ožívajú, zostávajú žiť alebo umierajú veľké intervaly políčok. Vieme sa na tento proces pozrieť zametaním. Začneme v najľavejšej diagonále a zametáme smerom doprava dole. Implementovať krok vieme nasledovne:

- pre každý interval živých buniek sa tento interval v ďalšom kroku zmenší o jedna
- tým, že na ploche sú pôvodne nejaké obdĺžniky, medzi jednotlivými krokmi musíme manuálne pridať nejaké intervaly
- táto kolekcia intervalov a zmeny na nich sa dajú implementovať pomocou usporiadanej množiny alebo intervalového stromu.

Akú to bude mať časovú zložitosť? Časovú zložitosť budú dominovať operácie s intervalovým stromom. Jednak doňho pridávame nové diagonály a dvak v každom kroku zmenšujeme všetky intervaly. Nech počet buniek na začiatku je Z. Ukážeme, že počet týchto operácií je $O(Z)$, a teda časová zložitosť $O(Z \log Z)$. Uvedomme si, že jediný spôsob akým pribúdajú políčka do intervalového stromu je tým, že pridávame diagonály. Pravidlá množenia sa plesne sme transformovali na pravidlo, že živý interval sa v ďalšom kroku presunie do susednej diagonály a skráti o jedna. Navyše jediný spôsob ako vie políčko zomrieť je tak, že zmizne skrátením intervalu o jedna. Počet týchto skrátení bude teda presne toľko, koľko políčok pridáme do intervalového stromu a to je nanajvýš Z. Máme teda horný odhad. Zároveň je to presný odhad, keďže to vieme dosiahnuť napríklad vstupom:

Fig. 3

```
1 1 1 1000000
1 1 1000000 1
```

Pre obmedzenia druhej sady (obmedzenia na veľkosť súradníc a počet živých políčok) je najhorší vstup nasledovného tvaru (veľa vyššie spomenutých vstupov (Fig. 3) vložených do seba):

Fig. 4

```
AAAAAAAA...A
B
B CCCCCC...C
B D
B D EEEE...E
B D F
B D F GG...G
B D F H.
. . . . .
. . . .  .
. . . .   .
B D F H    .
```

Dá sa ukázať, že časová zložitosť je $O(D \log D)$, kde D je počet živých diagonálnych intervalov na vstupe, avšak v tomto najhoršom prípade sú D a Z skoro rovnaké. Znamená to ale, že na vstupe s približne štvorcovými počiatočnými plochami (hore uvedený vstup má mimoriadne neštvorcové plochy) by mal tento algoritmus zložitosť $O(\sqrt{Z} \log Z)$.

Zatiaľ sme však nespomenuli, ako hľadať komponenty.

Zjednodušenie úlohy

Simuláciou sa veľmi ďalej nedostaneme, treba si úlohu zjednodušiť. Znova sa pozrime na jeden komponent. Ukázali sme, že ako prvá zomrie bunka v najľavejšej diagonále. Otázkou teda zostáva, kedy zomrie posledná bunka. Ak si trochu nasimulujeme ako sa hra správa, vieme si všimnúť, že ako posledná zomrie bunka, ktorej súradnice (x, y) sú maximum x-ových a y-ových súradníc všetkých buniek. Úloha sa nám teda zjednoduší na hľadanie najspodnejšej a najpravejšej bunky komponentu (dôkaz neskôr). A to je triviálne, ak vieme, že máme na ploche iba jeden komponent. Zostáva nám teda pre každú bunku určiť, do ktorého komponentu patrí. A tu sa reálne začína úloha.

Zadanie: *Na vstupe máme N potenciálne prekrývajúcich sa obdĺžnikov. Ak sa dva obdĺžniky prekrývajú, dotýkajú hranou alebo pravohornými-ľavodolnými ↗↙ rohmi, tak hovoríme, že obdĺžniky sú susedné. Susedné obdĺžniky patria do rovnakého komponentu. Nájdi komponenty.*

Určenie komponentov na úrovni buniek – sada 2/5

Toto riešenie dostane rovnako veľa bodov, ako vyššie spomínaná simulácia kladivom, avšak je oveľa jednoduchšie na nakódenie.

V druhej sade máme zaručené, že súčet plôch všetkých obdĺžnikov je malý. Môžeme teda každú bunku považovať za vrchol, vytvoriť hrany medzi susednými bunkami a nájsť komponenty v tomto grafe. Pre každú bunku vieme, že je k nej iba konštantne veľa susedných buniek (6 smerov, navyše pre každú bunku stačí zohľadniť iba polovicu z nich a zvyšok zohľadní susedná bunka). Tento graf bude mať teda $O(Z)$ vrcholov a hrán.

Určenie komponentov na úrovni obdĺžnikov – sada 3/5

Teraz už ideme konečne opísať riešenie, ktoré nám dá viac bodov a je dokonca stále veľmi jednoduché na naprogramovanie.

Podobne ako v predchádzajúcom riešení, snažíme sa zostrojiť graf a nájsť v ňom komponenty. Snažíme sa postaviť graf, kde každý obdĺžnik je vrchol a hrana vedie medzi obdĺžnikmi, ktoré sú susedné.

V tejto sade však vieme, že počet obdĺžnikov je taký malý, že si vieme dovoliť $O(N^2)$ riešenie. Môžeme sa teda pozrieť na každú dvojicu obdĺžnikov, určiť v $O(1)$ čase, či sú susedné, postaviť graf a nájsť komponenty. Dostaneme graf s $O(N)$ vrcholmi a v najhoršom prípade $O(N^2)$ hranami.

Určiť či sú obdĺžniky susedné je priamočiare a pozostáva iba z porovnania súradníc oboch obdĺžnikov a kontroly, či sa pretínajú, dotýkajú hranami alebo správnymi rohmi.

Hľadanie komponentov v postavenom grafe

Na hľadanie komponentov v grafe vieme použiť rôzne metódy – ľubovoľné prehľadávanie (DFS, BFS) alebo Union-Find.

Časová zložitosť DFS a BFS je $O(V + E)$ a časová zložitosť Union-Find je $O(E\alpha^{-1}(E))$. Čo je ale zaujímavé je, že Union-Find je nielen rýchlejšie naprogramovať, ale reálne aj beží rýchlejšie. Pre účely popisu však povieme, že použitím prehľadávania dostávame lineárnu zložitosť vzhľadom na počet hrán.

Bez ohľadu na to, aký prístup zvolíme, nie je nutné si graf explicitne konštruovať. Ak máme funkciu, ktorá nám pre dva vrcholy povie, či je medzi nimi hrana (teda napríklad funkciu susednosti dvoch obdĺžnikov), graf vieme používať iba implicitne.

Neexistuje $O(N^2\alpha^{-1}(N))$

A dokonca ani $O(N \cdot \log N \cdot \alpha^{-1}(N))$.

Čo tým myslím? Keď si pozrieme Wikipédiu, tá hovorí, že časová zložitosť M operácií Union-Findu je $O(M\alpha^{-1}(N))$. Avšak ja tvrdím, že ak zavoláme funkciu `UF.find` alebo `UF.join` $N \log N$ krát, časová zložitosť bude iba $O(N \log N)$, nie $O(N \cdot \log N \cdot \alpha^{-1}(N))$. Väčšina týchto volaní sa dokázateľne vykoná v $O(1)$ čase.

Spomeňme si na analýzu Union-Find algoritmu, kde funkcia `UF.find` robí *path compression* a `UF.join` joinuje buď podľa *ranku* alebo *veľkosti* komponentu.

Potom vieme, že najhlbší strom, ktorý vie vzniknúť, bude mať hĺbku $O(\log N)$ (kvôli *rank join*). A z toho vidno, že pre každý vrchol sa funkcia `UF.find` rekurzívne zavolá tiež maximálne iba $O(\log N)$ krát (kvôli *path compression*). Ak sme pre nejaký vrchol $\log N$ krát *rekurzívne* zavolali `UF.find` (teda že tento vrchol nebol priamo pod koreňom), tak už určite bude navždy priamo pod koreňom a všetky ďalšie volania nebudú rekurzívne.

Počet rekurzívnych `UF.find` volaní vieme teda ohraničiť ako $O(N \log N)$. Pre M operácií bude časová zložitosť $O(\max(M, \min(M, N) \log N))$, čo pre $M \in O(N \log N)$ znamená $O(N \log N)$ a nie $O(N \cdot \log N \cdot \alpha^{-1}(N))$.

Zložitosť pre tretiu sadu je teda bez ohľadu na to, či použijeme BFS, DFS alebo Union-Find $O(N^2)$.

Ďalšie zjednodušenia

Aby sa nám v ďalších sadách pracovalo ľahšie, zjednodušíme si vstup.

V prvom rade si súradnice skomprimujeme. Teda si pozbierame všetky x-ové súradnice, utríedime ich a prečíslujeme na hodnoty 0 až počet rôznych x-ových súradníc. Počet rôznych x-ových súradníc bude nanajvýš $2N$.

Avšak, ak by sme to spravili takto, pokazili by sme si riešenie. Totiž, ak sme mali napríklad obdĺžniky `(10, 20, 13, 21)` a `(17, 20, 19, 21)`, po prečíslovaní by sa z nich stali `(0, 0, 1, 1)` a `(2, 0, 3, 1)`. Zatiaľ čo originálne dva obdĺžniky nie sú susedné, prečíslované obdĺžniky už sú. Preto, ak chceme takto prečíslovávať, pre každú x_1 súradnicu by sme mali do prečíslovacieho zoznamu pridať aj $x_1 - 1$. Potom budú dané obdĺžniky vyzerať nasledovne – `(1, 1, 2, 2)` a `(4, 1, 5, 2)`. Počet súradníc bude teda nanajvýš $3N$.

V druhom rade prestaneme používať uzavreté intervaly na obdĺžniky. Nepopierateľne sa s tým ľahšie programuje. Ak by sme túto zmenu spravili už v riešení tretej sady, funkcia na susednosť by bola oveľa krajšia. Avšak v tomto prípade nám to pomôže aj pri kompresii. Ak prestaneme používať uzavreté intervaly, zmizne hore popísaný problém a znova budeme mať iba nanajvýš $2N$ rôznych súradníc (čo nám prakticky zrýchli program o viac ako 33 %).

Rovnako upravíme aj y-ové súradnice.

Obdĺžnikov je veľa, ale neprekrývajú sa – sada 4/5

Čo nám pomáha, že sa obdĺžniky neprekrývajú? Čo je problém s predchádzajúcim riešením? Ak sa všetky obdĺžniky pretínajú, vznikne graf s $O(N^2)$ hranami (a taký vstup vieme triviálne zostrojiť). Čo však v prípade, že sa obdĺžniky nepretínajú? Dá sa ukázať, že počet hrán je v tomto prípade $O(N)$.

Dôkaz málo susedností

Pozrime sa na dva susedné neprekrývajúce sa obdĺžniky. Ako môžu vyzerať? Buď je hrana jedného podúsekom hrany toho druhého, alebo sa obe prekrývajú čiastočne, alebo sa dotýkajú rohom.

```
Fig. 5
   AAA       AAAAAAAAAA    AAAAAA          AAAAAA       AAA
BBBBBBBBB      BBB           BBBBBB    BBBBBB       BBB
```

Vidíme, že pre `B` sa môže takmer každá z týchto situácií stať iba nanajvýš raz (napríklad nemôžu sa dva obdĺžniky dotýkať jeho pravého horného rohu). Jediné, čo sa môže stať viackrát je, že viacero obdĺžnikov má hranu ako podúsek jeho hrany. V tejto situácii sa na to však vieme pozrieť z opačnej strany a znova povedať, že pre tieto menšie obdĺžniky môže existovať iba jeden nadobdĺžnik. Každá susednosť nám "vyrieši" aspoň jednu hranu alebo roh a keďže tých je $O(N)$, tak celkovo môže existovať iba $O(N)$ susedností.

Zametanie

Ako efektívne hľadať tieto susednosti? Celkom určite zametaním. Vďaka vlastnosti, ktorú sme hore ukázali, by malo fungovať veľa rôznych prístupov – či už riešenie používajúce `std::map` alebo intervalový strom. Dôležité je, že si obdĺžniky pozametáme a budeme efektívne hľadať susednosti na okrajoch.

Najjednoduchšie je spraviť dve zametania, jedno vertikálne a jedno horizontálne. Zamerajme sa teraz na vertikálne zametanie. Potom sa v rámci tohto zametania stačí sústrediť iba na susednosti, ktoré sú spôsobené situáciami zobrazenými vyššie (Fig. 5).

Ak robíme zametanie, budeme mať nejakú štruktúru, v ktorej si budeme pre aktuálny riadok pamätať aktuálne aktívne obdĺžniky. Potom, keď ideme na ďalší riadok, budeme musieť pridať niektoré nové obdĺžniky a niektoré nové obdĺžniky musíme odobrať. Najprv vyhodnotíme všetky susednosti, potom odstránime všetky obdĺžniky, ktoré treba odstrániť, a nakoniec pridáme všetky, ktoré treba pridať. Čo je zaujímavé je, že keďže robíme dve zametania, netreba vyhodnocovať žiadne susednosti medzi obdĺžnikmi, ktoré práve pridávame. Tie sa vyhodnotia v kolmých zametaniach.

Ako nájsť všetky susednosti? Očakávame, že naša zvolená štruktúra podporuje vrátenie všetkých prvkov, ktoré sa pretínajú s požadovaným intervalom. Napríklad pre `std::set`, kde kľúč je `pair<x_1, x_2>` vieme tieto prvky nájsť pomocou funkcie `lower_bound` – ak zavoláme `set.lower_bound({x_2, -1})`, dostaneme prvok, najľavejší interval, ktorý je napravo od a určite nesusedí s intervalom (x_1, x_2). Podobne `set.lower_bound({x_1, -1}) - 2` nám dá najpravější interval, ktorý je naľavo a určite nesusedí s intervalom (x_1, x_2) (zamyslite sa, prečo `-2`). Stačí nám zavolať jedno z týchto volaní a potom iterovať cez všetky vedľajšie intervaly, až kým neprestaneme susediť. Vyššie sme ukázali, že susedností je $O(N)$, takže celkovo týchto iterácií spravíme $O(N)$.

Musíme implementovať nasledujúce operácie:

1. pridať interval
2. odobrať interval
3. iterovať cez všetky intervaly, ktoré susedia s daným intervalom

Celkovo bude časová zložitosť $O(N \log N)$, keďže triedime a používame nejakú stromovú štruktúru.

Obdĺžniky sa ale môžu prekrývať – sada 5/5

Pozrime sa, aké všelijaké problémy nám môžu vzniknúť, keď sa obdĺžniky prekrývajú:

```
Fig. 6
     1      AAA      CC D   F
     2   BBBabaBBB   CCoDoooFoooo
     3   BBBbabBBB    ooDoEoFoGoHH
     4      AAA  Y    ooooooFoGoo
     5      X               F G
```

Pridajme obdĺžnik do prázdneho lazy intervalového stromu[1]. Keďže je to lazy strom, na pridanie stačí navštíviť iba $O(\log N)$ vrcholov (až $2 \log N$ bude obsahovať lazy informáciu a po ceste sme navštívili až $2 \log N$ ďalších predkov). V predchádzajúcej sade, kde sa žiadne obdĺžniky neprekrývali, sme mali vlastne zaručené, že každý vrchol je pokrytý nanajvýš jedným obdĺžnikom. V tejto sade, žiaľ, už túto peknú vlastnosť nemáme. Pamätajme si teda v každom vrchole pole obdĺžnikov. Keď lenivo pridávame nový obdĺžnik do $O(\log N)$ vrcholov, je možné, že v ňom už nejaký obdĺžnik je. V tomto prípade by sme radi tiež dali tieto dva obdĺžniky do jedného komponentu.

Spravíme v tomto lazy intervalovom strome jednu zvláštnu vec, a to že si lazy informácie nebudeme propagovať ďalej. Ak štandardne máme vo vrchole zapamätanú premennú `hodnota` a `lazy`, tak v našom intervalovom strome nám stačí `lazy` (ktorá je pole identifikátorov obdĺžnikov). Prečo to robíme? Keďže chceme podporovať aj odoberanie obdĺžnikov, ľahšie sa nám to bude robiť, ak sa obdĺžniky nerozlezú až hlboko do listov. Ak by sme informácie propagovali, explicitné upratanie obdĺžnikov by potenciálne trvalo až $O(N)$. Trochu sa nám týmto komplikuje vyhodnocovanie pretnutí pri pridávaní obdĺžnikov, ale to zvládneme vyriešiť.

Nazvime množinu vrcholov, ktoré pokrývajú interval prislúchajúci pridávanému obdĺžniku J (teda práve tie vrcholy, do ktorých by sme uložili lazy informáciu). Čo by sa zmenilo, keby sme každý obdĺžnik rozdelili na množinu nových obdĺžnikov tak, že každý pokrýva práve jeden vrchol z J. Celý algoritmus by mal naďalej fungovať (keďže takýto vstup existuje), iba sa nám zložitosti zhoršia o logaritmus, pretože máme o logaritmus krát viac obdĺžnikov. Robiť to teda nebudeme, ale bez ujmy na všeobecnosti nad tým tak môžeme uvažovať.

Pozrime sa teda iba na jeden vrchol z J. Pridaný obdĺžnik prejde v intervalovom strome nejakú cestu. Naším cieľom je ho spojiť so všetkými ostatnými obdĺžnikmi, ktoré sa s ním pretínajú. Teda nájsť všetky obdĺžniky, ktorých cesta je buď podmnožina (starý obdĺžnik pokrýva celý nový obdĺžnik) alebo nadmnožina (nový obdĺžnik pokrýva celý starý obdĺžnik). Každú dvojicu ciest stačí spracovať iba raz. Rozumné miesto kde to robiť je v koncovom vrchole kratšej cesty.

```
Fig. 7                1       2       3       4       5
     1   AAAAAAAA      A       A       A       A       .
     2   AAAABBAA     / \     / \     / \     / \     / \
     3   AAAABBCC    .   .   .   .   .   C   .   C   .   C
     4   AAAACCCC      / \     / \     / \     / \     / \
     5       CCCC      .   .   B   .   B   .   .   .   .   .
```

Na obrázku (Fig. 7) vidíme, ako by vyzeral intervalový strom v jednotlivých časových bodoch zametania. Vidíme, že cesta do vrcholu s obdĺžnikom B je nadmnožina cesty do vrcholu pre A (rovnako aj pre dvojice $C - A$ a $B - C$). O spojenie $A - B$ a $A - C$ sa teda chceme postarať v A a o spojenie $B - C$ v C.

Pozrime sa čo sa deje v tomto koncovom vrchole V pre nový obdĺžnik. Chceme pridať obdĺžnik do V a celý ho pokryť. Pod V môže byť viacero doposiaľ disjunktných obdĺžnikov a týmto ich všetky dostaneme do jedného komponentu:

```
Fig. 8
           A  B
```

[1] https://www.ksp.sk/kucharka/lazy_intervalovy_strom/

```
       A  B
XX...XXaXXbXXXX...XX
       A  B
       A  B
WW...WWaWWbWWgW...WW
       A  B  G
       A  B
YY...YYaY B  H
       A  B
       A  B
       A ZbZZZZ...ZZ
       A  B
```

Povedzme, že v tomto prípade (Fig. 8) zametáme zhora nadol, X pokrýva nejaký veľký interval a tým, že ho pridáme, spojíme obdĺžniky A, B a X do jedného komponentu. Keďže A a B sú veľmi úzke, v intervalovom strome je J_A a J_B iba jeden list. Ak by sme pri pridávaní X do intervalového stromu išli až po úroveň listov, vykonali by sme až $O(N)$ operácií pre jeden obdĺžnik. Namiesto toho, aby sme teda chodili do všetkých potomkov J, pamätajme si informáciu o prítomnosti obdĺžnika aj vo všetkých predkoch J. Tých je iba nanajvýš $O(\log N)$. Informáciu o prítomnosti obdĺžniku si teda pamätáme na celej spomínanej ceste a zjednotenie všetkých ciest pre J má veľkosť iba $O(\log N)$.

Ak by sme sa však pokúsili spojiť nový obdĺžnik so všetkými obdĺžnikmi vo V, mali by sme stále zlú časovú zložitosť. Obdĺžniky sa nám vo vrcholoch hromadia a každý môže obsahovať až $O(N)$ obdĺžnikov (avšak všetky vrcholy dokopy majú stále iba $O(N \log N)$). Preto je dôležité, že nový obdĺžnik pokrýva celé intervaly prislúchajúce vrcholom z J. Vieme si teda predstaviť, že aj keď máme vo vrchole V zapamätaných povedzme tisíc obdĺžnikov, pridaním tohoto nového obdĺžnika sa všetkých tisíc spojí do jedného komponentu. Pri pridávaní ďalšieho obdĺžnika by ho stačilo spojiť s týmto veľkým komponentom, namiesto spájania s tisíc obdĺžnikmi. Nový obdĺžnik *vyrieši/vyčistí* všetky vrcholy z J, tak že v nich zostane iba jeden komponent. Tento prístup je teda ako keby sme si v každom vrchole pamätali nezávislú Union-Find štruktúru.

Problémy nastanú, keď sa pokúsime obdĺžnik odstrániť – ako na obrázku (Fig. 8). Ak odstránime obdĺžnik X, nastanú dva problémy. Jednak sme potenciálne odstránili koreň nejakého Union-Find komponentu. Dvak, keďže X pokrýval celý interval od A po B, ako zaručíme, že nebudeme znova vykonávať veľa práce pri pridávaní obdĺžnika W?

Prvý problém nie je problém, za koreň zvolíme ľubovoľného potomka. Pre druhý problém je dôležité to, že obdĺžnik W pokrýva celý interval daného vrcholu. Keďže odstránením vrcholu sa komponenty Union-Findu nerozpadli a W pokrýva celý interval, stačí W spojiť iba so všetkými koreňmi Union-Find komponentov, čím znova *vyčistíme* daný vrchol a amortizovane spotrebujeme $O(1)$ času.

Týmto by sme mali celkom vyriešené spájanie s nadmnožinovými cestami. Čo tie podmnožinové? Teda ako sa zachovať keď pridávame obdĺžniky G a H? Stále ich chceme vyriešiť v koncovom vrchole tej kratšej cesty. To znamená, že po ceste dole intervaláčom sa pozeráme, či aktuálny vrchol nie je koncovým pre nejaký obdĺžnik. Môže byť dokonca koncovým pre viacero, ale znova si uvedomme, že v tom prípade budú určite všetky patriť do toho istého komponentu. A ak taký existuje, tak to musí znamenať, že všetky obdĺžniky v tomto vrchole patria do jedného komponentu. Pozor, ide o implikáciu, nie ekvivalenciu – to, že existuje iba jeden komponent neznamená, že existuje obdĺžnik čo pokrýva celý interval. Zamyslite sa, ako zistiť, či existuje pokrývajúci obdĺžnik. V tom prípade nový obdĺžnik tiež pridáme do tohoto veľkého komponentu, inak vytvoríme nový komponent obsahujúci iba nový obdĺžnik.

Všimnime si, že nám nevadí vytvárať veľa malých komponentov. Je možné, že vo vrchole pre interval dĺžky 16 budeme mať tisíc jednoprvkových komponentov. Očividne by sa počet komponentov dal zredukovať na nanajvýš 16. My však chceme byť efektívny, a teda lenivý. Túto redukciu si odložíme na moment, keď pridáme obdĺžnik, ktorý bude mať tento vrchol vo svojom J. Tým, že by sme to spravili skôr si nič nepomôžeme, keďže všetky dôležité spojenia komponentov sa vyriešili na hlbších vrstvách intervalového stromu.

Teraz si môžeme všimnúť, že vo vrchole si vždy pamätáme jeden veľký komponent a kopu jednoprvkových komponentov, ktoré sa doňho dostali z jeho potomkov. Môžeme tomu teda prispôsobiť implementáciu a nemusíme si vo vrchole pamätať Union-Find, ale iba nejaké dve množiny (napríklad `std::unordered_set`) – jednu pre veľký komponent a druhú pre jednoprvkové komponenty. Množinu používame, aby sme vedeli rýchlo odstraňovať prvky.

`std::unordered_set` má síce $O(1)$ operácie, nie je však taký rýchly ako pole. Implementáciu vieme teda vylepšiť tým, že si budeme pamätať dve polia a hodnoty z nich nemažme. Hodnoty v poliach totiž používame iba ak ich chceme všetky spojiť do jedného komponentu, teda zobrať všetky hodnoty z druhého pola a nasypať

ich do prvého. Keď už toto robíme, vieme si pre každý prvok skontrolovať, či je ešte živý (`rects[ri].y2 >= sweep_time`) a ak nie, jednoducho ho odignorovať. Dajme si pozor, aby sme nehýbali s prvkami vo veľkom komponente (to by nám pokazilo časovú zložitosť). Na to, či vo veľkom komponente existuje ešte aspoň jeden aktívny obdĺžnik si budeme pamätať maximálnu koncovú súradnicu zo všetkých obdĺžnikov v ňom (podobne riešime zamyslenie nad existencie pokrývajúceho obdĺžnika). Celé toto nás bude stáť zopár riadkov, ale kód by mal byť oveľa rýchlejší.

Týmto sme vyriešili pridávanie a odoberanie obdĺžnikov a spájanie tam, kde sa prekrývajú. Spájanie tam kde sa dotýkajú doriešime všeliako na kolene, keďže stačí skontrolovať iba $O(\log N)$ vrcholov, ktorých intervaly sa dotýkajú pridávaného obdĺžnika.

Premeníme intervaly vstupe na polootvorené a skomprimujeme si súradnice. Zametáme, pričom si pamätáme globálnu Union-Find štruktúru. Máme lazy intervalový strom v ktorom nepropagujeme a vo vrcholoch si pamätáme dve polia a pomocné agregačné informácie o tom prvom. Obe polia obsahujú indexy nejakých obdĺžnikov, pričom v prvom poli sú všetky súčasťou jedného komponentu a v druhom sú všekty zatiaľ ako samostatný komponent. Agregačné informácie pre prvé pole sú: o aký komponent ide, dokedy je ešte niečo aktívne a dokedy je ešte nejaký plne pokrývajúci prvok aktívny. Keď pridávame obdĺžnik, po ceste kontrolujeme, či je vrchol pokrytý jedným obdĺžnikom a podľa toho pridáme nový obdĺžnik do prvého alebo druhého pola. Keď prídeme do vrcholu, ktorý nový obdĺžnik celý pokrýva, skončíme rekurziu. Presypeme druhé pole do prvého, pričom ignorujeme neaktívne prvky. Aktualizujeme agregačné informácie. Vždy keď dávame prvok do prvého pola, spojíme ho v globálnom Union-Finde s daným komponentom. Počas rekurzie vyriešime aj dotýkajúce sa obdĺžniky. Obdĺžniky pri zametaní teda explicitne nemažeme. Skončime s Union-Find štruktúrou, ktorá určuje rozdelenie obdĺžnikov do nezávislých komponentov.

No a toto je prakticky celé riešenie. Časová a pamäťová zložitosť sú $O(N \log N)$.

Dá sa to aj lepšie

Predsa len, komu sa chce pamätať si pole alebo nebodaj `std::unordered_set` vo vrcholoch intervalového stromu. Keď namiesto toho sa môžeme ďalej zamyslieť nad tým, čo sa ako deje a v každom vrchole si pamätať iba $O(1)$ informácie.

Znova majme lazy intervalový strom, ale tentokrát budeme informácie aj propagovať. To preto, lebo z neho nebudeme explicitne mazať veci. Pre každý vrchol si budeme snažiť zapamätať, aký najdlhšie žijúci komponent v ňom je, do akého času bude žiť a ktorý obdĺžnik ho reprezentuje — `(life, kompid, rectid)`. `life` je v tomto prípade najväčšia koncová súradnica spomedzi všetkých obdĺžnikov na danom intervale (ľubovoľný čo doň zasahuje). Ak máme takúto informáciu pre dva intervaly, tak informácia pre ich zjednotenie je jednoducho ich maximum. Keď pridávame nový obdĺžnik, túto informáciu si vypýtame pre celý jeho interval. Vďaka nej potom vieme povedať, či sa obdĺžnik s niečím pretne alebo nie (ak `life >= y_1`, tak áno, inak nie).

Zistili sme, že sa pretneme s nejakým obĺžnikom R. Predpokladáme, že on už je pospájaný do jedného komponentu so všetkými ostanými obdĺžnikmi, ktoré sú celé v ňom. Interval nového obdĺžnika teda môžeme rozdeliť na dva nezávislé kratšie úseky naľavo a napravo od R a rekurzívne sa pýtať intervalového stromu. Tu je problém, že odpovede naľavo a napravo vedia patriť do toho istého komponentu ako R a pokazíme si časovú zložitosť.

Pre každý komponent si pamätajme intervaly, ktoré pokrýva. Potom nebudeme deliť interval nového obdĺžnika na dva menšie podúseky, ale na veľa podúsekov, ktoré sú pomedzi intervaly komponentu $komp[R]$. Na tieto podintervaly sa stále vieme zavolať rekurzívne. Samozrejme, týchto podintervalov môže byť veľa a môžu byť aj prázdne. To sa môže zdať ako strata času. Dôležité je, že pridaním tohto nového obdĺžnika tieto intervaly zaplníme a odstránime zo zoznamu intervalov pre $komp[R]$. Ak bol interval prázdny alebo sa nič nepretlo, tak nový obdĺžnik bude určite žiť na tomto intervale v intervalovom strome dlhšie ako čokoľvek čo tam bolo doteraz. Ak sa obdĺžnik pretne, počet komponentov klesne o jedna. Amortizovane bude táto operácia trvať rovnako dlho, koľko existuje intervalov a komponentov. Teraz si už iba stačí uvedomiť, že pridaním nového obdĺžnika vieme vytvoriť iba konštantne veľa nových intervalov. Na to, aby sme vytvárali, pridávali a odoberali nové intervaly, si budeme pre každý komponent pamätať `std::set` a robiť tieto operácie efektívne na ňom. Intervalov a komponentov bude nanajvýš $O(N)$, a teda časová aj pamäťová zložitosť budú $O(N \log N)$.

Zamyslite sa, ako bude tento algoritmus prebiehať na takomto vstupe pri zametaní zhora nadol:

```
Fig. 8
     AAAAAAAAAAA
     B         C
     B  DDDDD  C
     B  DDDDD  C
     B         C
```

Aj ak sa nepretneme, musíme sa rekurzívne zavolať na podintervaly, aby sme správne modifikovali zoznamy intervalov pre komponenty.

Naozaj prosím žiadne sety

Ale komu sa chce spájať a rozpájať nejaké intervaly v sete. Vystačíme si iba s intervalovým stromom. Problém bol, že prvým zavolaním sme efektívne našli pretínajúci komponent, avšak ďalšie volania nám mohli dávať ten istý komponent. Napríklad ak by hodnoty `life` v intervalovom strome boli `1 2 3 4 5 4 3 2 1` a všetky patrili do toho istého komponentu, tak by sme vykonali $O(N)$ volaní.

My sme však ukázali, že intervalov, ktoré patria do rôznych komponentov, je celkovo iba $O(N)$. Rovnakú myšlienku sme použili na riešenie štvrtej sady! Tam sme ukázali, že existuje nanajvýš $O(N)$ disjunktných intervalov rôznych *obdĺžnikov*. Tu sme ukázali, že existuje nanajvýš $O(N)$ disjunktných intervalov rôznych *komponentov*. Na dosiahnutie dobrej časovej zložitosti nám stačí dané riešenie prispôsobiť na komponenty, teda efektívne implementovať 3. operáciu — iterovanie.

Pre ľubovoľný list intervalového stromu chceme efektívne ($O(\log N)$) nájsť najbližší list napravo od neho, ktorý patrí do iného komponentu (alebo žiadneho). So správnymi údajmi môžeme výsledok "binárne vyhľadávať". Vo vrcholoch si budeme pamätať dodatočné informácie:

1. či je celý interval pokrytý jedným komponentom alebo nie
2. do akého komponentu patrí najľavejší list tohto intervalu
3. do akého komponentu patrí najpravý list tohto intervalu

Ak máme túto informáciu pre dva susedné intervaly, vieme ju jednoducho zistiť aj pre ich zjednotenie (zmena je buď v ľavom, pravom alebo na hranici intervalu).

Treba sa zamyslieť, ako tieto informácie ovplyvnia lazy updaty. Lazy update robíme iba ak pokrývame celý interval. V tom prípade v danom intervale určite nebude žiadna zmena komponentu a ľavý aj pravý okraj budú mať hodnotu nového komponentu.

Ako teda binárne vyhľadávať? Sme v nejakom liste a chceme nájsť najbližšiu zmenu komponentu napravo od nás. Budeme bublať smerom dohora. Ak sme pravé dieťa, tak s tým nič nenarobíme. Ak sme ľavé dieťa, tak najbližšia zmena je potenciálne v našom súrodencovi alebo na hranici medzi nami. To vieme okamžite zistiť tak, že sa pozrieme na predpočítanú informáciu. Inak pokračujeme v bublaní dohora. Ak je nejaká zmena v našom súrodencovi, tak bubleme dodola. Ak je zmena v jeho ľavom synovi, tak ideme tam, inak je možno na rozmedzí a inak musí byť v pravom synovi. Potrvá nám $O(\log N)$ času vybublať hore a dole a nájsť najbližší odlišný komponent. Počas tohoto si treba dať pozor na správne propagovanie lazy informácii.

Zmenili sme intervali na polootvorené, skomprimovali sme si súradnice, vytvorili Union-Find štruktúru a lazy intervalový strom, zametáme jedným smerom. Pridanie jedného obdĺžnika potrvá $O(\log N)$ (Union-Find operácie sa amortizujú na $O(1)$). Obdĺžniky neodoberáme, iba ich začneme ignorovať, ak sú príliš staré (ich koncová súradnica je malá). Nájdenie najbližšieho susedného obdĺžnika trvá $O(\log N)$ a zavoláme to $O(N)$ krát. Celková časová zložitosť je $O(N \log N)$, pamäťová $O(N)$ a `set` sme použili iba pri kompresií súradníc.

Dôkazy

Diagonála je určená ako všetky bunky, ktorých $x + y = C$.

Komponent s jednou živou bunkou je triviálny prípad, takže predpokladáme, že každý komponent má aspoň dve živé bunky.

Lema 1: Ak máme jeden komponent a diagonála $x + y = C$ je najmenšia diagonála, ktorá obsahuje živé bunky, potom v ďalšom kroku žiadna bunka nebude na diagonále C a nejaká bunka bude na diagonále $C + 1$.

Dôkaz: Pozrime sa na bunku s najväčším y na diagonále C. Keďže ide o jeden komponent s aspoň dvoma bunkami, musí existovať aspoň jedna bunka na pozíciách $\leftarrow\rightarrow\uparrow\downarrow\nearrow\swarrow$. Pozície $\uparrow\leftarrow\swarrow$ sú v rozpore s tým, že ide o bunku s najväčším y na diagonále C. Pozície $\downarrow\rightarrow$ sú bunky na diagonále $C + 1$, ktoré vďaka tejto bunke tento krok prežijú, čo spĺňa Lemu. Nakoniec pozícia $\nearrow$ znamená, že ak na pozícii $\rightarrow$ nie je živá bunka, v ďalšom kroku tam ožije, čo spĺňa Lemu.

Lema 2: Súvislý komponent v ďalšom kroku zostane súvislý.

Dôkaz: Sporom. Predpokladajme, že sa komponent rozdelil.

Jedna možnosť je, že sa vytvoril nový komponent alebo komponenty, ktoré nie sú napojené navzájom alebo k pôvodnému. Na začiatok si uvedomme, že nemôžu vzniknúť dva rôzne komponenty a zároveň všetky bunky z predchádzajúceho komponentu zaniknúť. Stačí sa teraz sústrediť na to, či môže vzniknúť nový komponent, ktorý nie je napojený na na nejakú preživšiu bunku.

Prepokldajme, že sa také stalo. Tento komponent bude mať tvar iba jednej diagonály. Bez ujmy na všebecnosť zoberme najmenšiu možnú diagonálu, teda jednu bunku (obdobný dôkaz sa aplikuje na ľubovoľne dlhú diagonálu).

Keďže vznikla nová bunka, ktorá nie je na nič napojená, obe bunky ←↑ čo ju stvorili musia zomrieť a v jej okolí ←→↑↓↗↙ nesmú zostať žiť žiadne:

```
  -   >     A   =     A
 -†   >    †-   =    †a
-†    >   †x-   =   †x-
      >  B--    =  Bb-
```

Kde † reprezentujú ←↑ bunky, na `-` nesmie byť živá bunka a na `A` alebo `B` musí byť aspoň jedna živá bunka (inak je toto celý komponent a Lema platí). To ale spôsobí, že na `a` alebo `b` vznikne bunka, čím dostávame spor s tým, že vznikla diagonála dĺžky iba jedna.

Alternatívne sa komponent rozdelí tým, že niektorá bunka zomrie.

```
 -A
-†B
DC
```

V tomto prípade † je bunka čo v ďalšom kroku zomrie, na `-` nesmie byť bunka a na `ABCD` musí byť aspoň jedna živá bunka. Ak by na `ABCD` bola práve jedna alebo všetky štyri živé bunky, † nespôsobí rozdelenie komponentu. Ak na `ABCD` budú tri živé bunky, tak je to iba jednoduchšia situácia ako dve živé bunky. Ak na `ABCD` budú práve dve živé bunky, máme 6 možností. Vieme si overiť, že pre žiadnu z nich sa komponent lokálne nerozpadne, pretože buď zostanú niektoré `ABCD` bunky žiť alebo vďaka † vzniknú nové. Prípady kedy `AB`, `BC` alebo `CD` žijú sú triviálne. Názorne si ukážeme prípad kedy `AD` sú živé a ostatné prípady by sa dali ukázať obdobne. V ďalšom kroku určite budú bunky `BC` žiť a niektoré z `AD` možno zomrú. Ak niektorá z buniek `AD` prežije, na danej strane komponentu ide o nezaujímavý prípad, keďže komponent sa tam nerozpadne. Ak však povedzme `A` zomrie, argument vieme posunúť o jedna ďalej, prepísať `†->D, A->†` a pozrieť sa rovnako na túto situáciu. Takto to vieme posúvať ďalej a ďalej. Avšak buniek je konečne veľa, takže musíme prísť do prípadu, ktorý je triviálny a spĺňa Lemu. Následne spätne ukážeme, že aj všetky poposúvané situácie teda spĺňali Lemu.

Nie je to úplne najelegantnejší, ale hádam to je valídny dôkaz…

Lema 3: Dva rôzne komponenty sa nikdy nespoja.

Dôkaz: Keďže komponenty sú rôzne, musí vzniknúť nejaká nová bunka, ktorá ich spojí. Na to, aby vznikla bunka, musia existovať bunky ←↑. Keďže tieto dve bunky sú susedné v ↗↙ smere, musia byť z toho istého komponentu (inak spor). Nová bunka musí susediť s druhým komponentom. Ak by susedila v jednom z ↗↙ smerov, tak máme spor, že komponenty sú rôzne (susedia s ←↑ bunkami). Ak je druhý komponent v smeroch →↓, tak nám to nepomôže, keďže v čase, keď bunka vznikne, tieto bunky zomrú. Dva rôzne komponenty sa teda nikdy nemôžu spojiť. Preto sa rôzne komponenty ani nikdy neovplyvnia.

Lema 4: V súvislom komponente sa maximálna súradnica X a Y živých buniek v ďalšom kroku nezmení.

Dôkaz: Ukážeme dôkaz pre maximálne Y a dôkaz pre maximálne X je obdobný. Pozrime sa na bunku s maximálnym Y. Keďže ide o súvislý komponent s aspoň dvoma bunkami, existuje aspoň jedna živá bunka v smeroch ←→↑↓↗↙. Smery ↓↙ sú v rozpore s tým, že táto bunka má maximálne Y. Smery ←↑ spôsobia, že táto bunka v ďalšom kroku prežije, čo spĺňa Lemu. Smery →↗ spôsobia, že bunka → v ďalšom kroku bude žiť, čo spĺňa Lemu.

Spojením Lemy 1 a 4 sme ukázali, že dĺžka prežitia komponentu je $X + Y - C$, kde X je maximálna x súradnica komponentu, Y maximálna y súradnica komponentu a C je najmenšia diagonála, teda najmenšie $x + y$ spomedzi všetkých buniek komponentu.

Listing programu (C++)

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

struct UF {
    vector<int> e;
    UF(int n): e(n, -1) {}
    int find(int x) { return e[x] < 0 ? x : e[x] = find(e[x]); }
    bool join(int a, int b) {
        a = find(a), b = find(b);
        if (a == b) return false;
```

```
        if (e[a] > e[b]) swap(a, b);
        e[a] += e[b];
        e[b] = a;
        return true;
    }
};

struct STval {
    // x coordinate, color, contains color change
    int maxi, c, firstc, lastc;
    bool ch;

    constexpr STval(): maxi(-1), c(-1), firstc(-1), lastc(-1), ch(false) {}
    STval(int maxi, int c, int firstc, int lastc, bool ch):
        maxi(maxi), c(c), firstc(firstc), lastc(lastc), ch(ch) {}
    bool isempty() const { return c == -1; }

    static STval merge(const STval &a, const STval &b) {
        bool ch = a.ch  b.ch  (a.lastc != b.firstc);
        return a.maxi > b.maxi ?
            STval(a.maxi, a.c, a.firstc, b.lastc, ch) :
            STval(b.maxi, b.c, a.firstc, b.lastc, ch);
    }
    void update(const STval &b) {
        if (b.maxi > maxi) {
            maxi = b.maxi;
            c = b.c;
        }
        ch = false;
        firstc = b.firstc;
        lastc = b.lastc;
    }
};

// lazy ST with range set
struct ST {
    using T = STval;
    static constexpr T unit = STval();

    int n;
    vector<T> val, lazy;
    vector<pair<int, int>> bound;

    ST(int _n) {
        for (n = 1; n < _n; n *= 2);
        val.resize(2 * n, unit);
        lazy.resize(2 * n);
        bound.resize(2 * n);
        for (int i = 0; i < n; ++i)
            bound[n + i] = {i, i+1};
        for (int i = n - 1; i > 0; --i)
            bound[i] = {bound[2 * i].first, bound[2 * i + 1].second};
    }
```

```
64
65      void push(int i) {
66          if (lazy[i].isempty()) return;
67          val[i].update(lazy[i]);
68          if (i < n) {
69              lazy[2 * i].update(lazy[i]);
70              lazy[2 * i + 1].update(lazy[i]);
71          }
72          lazy[i] = unit;
73      }
74
75      void update(int cur, int qlo, int qhi, T v) {
76          if (qhi <= bound[cur].first  bound[cur].second <= qlo) return;
77          if (qlo <= bound[cur].first && bound[cur].second <= qhi) {
78              lazy[cur].update(v);
79              return;
80          }
81          push(cur);
82          for (int i: {0, 1}) update(2 * cur + i, qlo, qhi, v), push(2 * cur + i);
83          val[cur] = STval::merge(val[2 * cur], val[2 * cur + 1]);
84      }
85
86      T query(int cur, int qlo, int qhi) {
87          push(cur);
88          if (qhi <= bound[cur].first  bound[cur].second <= qlo) return unit;
89          if (qlo <= bound[cur].first && bound[cur].second <= qhi) return val[cur];
90          return STval::merge(query(2 * cur, qlo, qhi), query(2 * cur + 1, qlo, qhi));
91      }
92
93      // find leftmost index ind in from qlo for which val[ind-1] != val[ind]
94      int find_next_change(int qlo) {
95          if (qlo >= n) return n + 1;
96          {
97              int path[30], len = 0;
98              for (int cur = n + qlo; cur > 0; cur /= 2)
99                  path[len++] = cur;
100             for (int i = len - 1; i >= 0; --i)
101                 push(path[i]);
102         }
103         int res = n + 1;
104         for (int cur = n + qlo; cur > 0; cur /= 2) {
105             if (!val[cur].ch) continue;
106             assert(cur < n);
107             push(cur * 2); push(cur * 2 + 1);
108             if (val[cur * 2].lastc != val[cur * 2 + 1].firstc) {
109                 int r = bound[cur * 2 + 1].first;
110                 if (r > qlo) res = min(res, r);
111             }
112         }
113         int cur = n + qlo;
114         for (; cur > 0; cur /= 2) {
115             if (cur % 2  !val[cur + 1].ch) // we are right child or there is no change
116                 continue;
```

```
            cur += 1;
            break;
        }
        if (cur == 0) return n + 1;
        assert(cur < n);
        assert(val[cur].ch);
        while (cur < n) {
            push(cur * 2); push(cur * 2 + 1);
            if (val[cur * 2].ch) cur = cur * 2;
            else if (val[cur * 2].lastc != val[cur * 2 + 1].firstc) {
                cur = bound[cur * 2 + 1].first;
                break;
            } else if (val[cur * 2 + 1].ch) cur = cur * 2 + 1;
            else assert(false); // st is mangled up
        }
        assert(cur < n); // we should not end up in leaf
        res = min(res, cur);
        return res;
    }
};

struct Rect {
    int x1, y1, x2, y2, i;
};

UF find_components(vector<Rect> Rs) {
    set<int> xs, ys;
    for (auto &r: Rs)
        xs.insert(r.x1), xs.insert(r.x2), ys.insert(r.y1), ys.insert(r.y2);

    unordered_map<int, int> xmap, ymap;
    {
        int i = 0;
        for (auto x: xs) xmap[x] = i++;
        i = 0;
        for (auto y: ys) ymap[y] = i++;
    }

    for (auto &r: Rs) {
        r.x1 = xmap[r.x1];
        r.x2 = xmap[r.x2];
        r.y1 = ymap[r.y1];
        r.y2 = ymap[r.y2];
    }

    sort(Rs.begin(), Rs.end(), [](const Rect &a, const Rect &b) {
        return a.x1 < b.x1;
    });

    ST st(ymap.size());
    UF uf(Rs.size());
    for (auto &r: Rs) {
        { // topleft corner is special
```

```
            auto qres = st.query(1, r.y1-1, r.y1);
            if (qres.maxi > r.x1) uf.join(r.i, qres.c);
        }
        int qlo = r.y1, qhi = qlo + 1, maxq = r.y2;
        while (qlo <= maxq) {
            auto qres = st.query(1, qlo, min(qhi, maxq + 1));
            if (qres.maxi >= r.x1) uf.join(r.i, qres.c);
            qlo = qhi;
            qhi = st.find_next_change(qlo);
        }
        int col = uf.find(r.i);
        st.update(1, r.y1, r.y2, {r.x2, col, col, col, false});
    }

    return uf;
}

void solve() {
    int Rn;
    cin >> Rn;

    vector<Rect> Rs(Rn);
    for (int i = 0; i < Rn; i++)
        Rs[i].i = i;
    for (auto &r: Rs) {
        cin >> r.x1 >> r.y1 >> r.x2 >> r.y2;
        --r.x1, --r.y1;
    }

    auto uf = find_components(Rs);

    vector<int> minxy(Rn, 2.1e9), maxx(Rn, -1), maxy(Rn, -1);
    for (int i = 0; i < Rn; i++) {
        int c = uf.find(i);
        minxy[c] = min(minxy[c], Rs[i].x1 + Rs[i].y1);
        maxx[c] = max(maxx[c], Rs[i].x2 - 1);
        maxy[c] = max(maxy[c], Rs[i].y2 - 1);
    }

    int res = 0;
    for (int i = 0; i < Rn; i++)
        res = max(res, 1 + maxx[i] + maxy[i] - minxy[i]);

    cout << res << '\n';
}

int main() {
    cin.tie(0)->sync_with_stdio(0);

    int TC;
    cin >> TC;
    while (TC--) solve();
```

```
};
```

Eliška

8. Aquamarínové guličky

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

Prerekvizity: Kruskalov algoritmus (vedieť, ako a prečo funguje), union-find, binary-lifting (budeme používať takú tú tabuľku čo staviame keď chceme hladať LCA) a DFS.

Implementácia tejto úlohy je pomerne náročná a je ľahké sa v nej zamotať, ak si po prečítaní vzoráku stále nie si istý, ako na to, odporúčam začať preriešením si nasledujúcich úloh z testovača KSP: Sneh, Kostry, Počet komponentov, Najnižší spoločný predok, Vzdialenosť v strome.

A už dosť rečí, ide sa na vec!

Dolný odhad na odpoveď

Najprv sa zamyslíme, že koľko hrán musíme pridať po odobratí vrchola i a všetkých hrán, ktoré s ním susedia.

Nech jeho stupeň (teda počet hrán, ktoré s ním susedia) je d_i. To znamená, že máme d_i komponentov, ktoré musíme pospájať. Ľahko si rozmyslíme, že vždy vieme pridať hranu ktorá spojí dva komponenty, a zároveň nikdy nevieme hranou naraz spojiť viac ako dva komponenty, preto v optimálnom riešení pridáme $d_i - 1$ hrán.

Teraz musíme určiť, aký najmenší súčet hrán novo pridaných hrán vieme dosiahnuť. V prvom rade, keďže každá hrana spájajúca dva rôzne vrcholy má cenu aspoň 1, celkový súčet musí byť aspoň $d_i - 1$. V prípade, že i nie je 1 ani n, vieme urobiť ďalšie jednoduché pozorovanie: ak neexistuje hrana, ktorá spája vrchol s číslom menším ako i s vrcholom s číslom väčším ako i, tak jedna novo pridaná hrana bude mať určite cenu aspoň dva. Prečo? Môžeme si uvedomiť, že každá takáto hrana má cenu aspoň dva, no a zároveň určite aspoň jednu takúto hranu potrebujeme, ináč by nový graf nebol súvislý (vieme ho rozdeliť na tie malé a na tie veľké vrcholy). To znamená, že v tomto prípade celkový súčet musí byť aspoň d_i.

Vieme nájsť aj nejaké ďalšie dolné ohraničenia na to, že aká veľká musí byť tá výsledná cena? Nie? Ako uvidíme neskôr, tento odhad je skutočne optimálny. Zároveň existuje niekoľko spôsobov ako implementovať vyššie opísané počítanie hrán a cien, podľa presnej časovej zložitosti môžeme za toto riešenie získať 2 až 4 body.

Listing programu (C++)

```cpp
// O(n log n) riesenie, len pocitame odpoved
#include <algorithm>
#include <iostream>
#include <map>
#include <queue>
#include <random>
#include <set>
#include <string>
#include <vector>
typedef long long ll;
using namespace std;

const int inf = 1e9 + 5; // nekonecno
struct interval
    // reprezentuje interval [l, r], pruky pridavame cez upd () a [l, r] je vzdy najmensi interval
    ↪ ktory pokryva vsetky pridane pruky
{
    int l, r;
    interval() { l = inf, r = -inf; }
    void upd(int x) { l = min(l, x), r = max(r, x); }
};
vector<vector<int> > g; // vstupny graf
vector<vector<interval>> m;
// pre kazdy vrchol si zistime intervaly pokryvajuce komponenty na ktore sa strom rozpadne ked ho
↪ odstranime
```

```cpp

vector<interval> s; // pre kazdy podstrom si zistime najmensie a najvacsie cislo vrcholu v nom
void dfs1(int u, int p)
{
    if (p != u)
    {
        g[u].erase(find(g[u].begin(), g[u].end(), p));
        // odstranime otca z mojich susedov aby tam ostali len moje deti
        // inak pozor ked pouzivas funkcie find a erase, casova zlozitost
        // je linearna od velkosti vrcholu z ktoreho mazes
    }
    s[u].l = s[u].r = u;
    for (int v : g[u])
    {
        dfs1(v, u);
        m[u].push_back(s[v]); // pridame interval podstromu dietata medzi moje intervaly
        s[u].upd(s[v].l);
        s[u].upd(s[v].r);
    }
}
// ideme pre kazdy vrchol spocitat interval prisluchajuci podstromu ktory ma jeho otec ked zoberieme prec tento vrchol
void dfs2(int u, interval otcov)
{
    vector<interval> pf(g[u].size() + 1), sf(g[u].size() + 1);
    pf[0] = otcov, sf[0] = otcov;
    pf[0].upd(u), sf[0].upd(u);
    // ok teraz detom musim pomoct spocitat tie intervaly tiez
    for (int i = 0; i < g[u].size(); i++)
    {
        pf[i + 1] = pf[i], pf[i + 1].upd(s[g[u][i]].l), pf[i + 1].upd(s[g[u][i]].r);
        sf[i + 1] = sf[i], sf[i + 1].upd(s[g[u][g[u].size() - 1 - i]].l), sf[i + 1].upd(s[g[u][g[u].size() - 1 - i]].r);
    }
    for (int i = 0; i < g[u].size(); i++)
    {
        interval now = pf[i];
        now.upd(sf[g[u].size() - i - 1].l), now.upd(sf[g[u].size() - i - 1].r);
        if (otcov.l != inf) m[u].push_back(otcov);
        dfs2(g[u][i], now);
    }
}
int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0); // rychle nacitavanie
    int t;
    cin >> t;
    while (t--)
    {
        int n;
        cin >> n;
        // ideme vycistit vsetky polia - velmi dolezite pri ulohach s viacerymi vstupmi!
```

```
75          g.assign(n, {}), s.assign(n, interval()), m.assign(n, {});
76          // nacitame strom a spustime predpocitavanie
77          for (int i = 0, u, v; i < n - 1; i++) cin >> u >> v, g[--u].push_back(--v),
    ↪ g[v].push_back(u);
78          dfs1(0, 0);
79          dfs2(0, interval());
80          for (int i = 0; i < n; i++)
81          {
82              if (g[i].empty())
83              {
84                  // som list - strom zostane cely aj bez mna, takze netreba nic robit
85                  cout << "0 0\n";
86                  continue;
87              }
88              int edges = g[i].size();
89              if (!i) edges--; // koren nema otca
90              int coins = edges;
91              bool overlap = false; // mame interval ktory prekryva i
92              for (interval in : m[i])
93              {
94                  if (in.l <= i && i <= in.r)
95                  {
96                      overlap = true;
97                  }
98              }
99              if (!overlap && 0 < i && i + 1 < n)
100             {
101                 coins++;
102             }
103             cout << edges << " " << coins << "\n";
104             for (int j = 0; j < edges; j++) cout << "0 0\n";
105             cout << "\n";
106         }
107     }
108     return 0;
109 }
```

Ako skonštruovať hľadanú množinu hrán?

Najprv taká úvaha na úvod, aký algoritmus by sme asi mohli použiť na postavenie toho nového grafu? Pridávame hrany tak aby sme mali súvislý graf a chceme čo najmenšiu cenu... To znie presne ako Kruskalov algoritmus. Čo by sme teda mohli urobiť je, že nejakým spôsobom nájdeme množinu hrán a potom na nich pustíme Kruskalov algoritmus aby našiel najlacnejší spôsob, ako vybrať podmnožinu hrán tak, aby bol výsledný graf súvislý.

Jeden jednoduchý príklad na takúto množinu sú hrany $(1,2),(2,3),...,(i-1,i+1),...,(n-1,n)$. Ľahko si môžeme rozmyslieť, že keďže sami o sebe tvoria súvislý graf (až na odstránený vrchol i), tak ak na rozpadnutom grafe + tejto množine spustíme Kruskalov algoritmus, dostaneme tiež súvislý graf. Dôležité pozorovanie: Iné hrany ako tie v tejto množine sa nám ani neoplatí používať. Skutočne, ak by sme v optimálnom riešení mali hranu medzi $(a,b), a+1<b$ (pre jednoduchosť uvažujme prípad $b<i$, ostatné sa dokazujú podobne), mohli by sme ju nahradiť hranami $(a,a+1),(a+1,a+2),...,(b-1,b)$, cena sa nezmení a ak boli predtým dva vrcholy spojené, určite budú spojené aj teraz. No lenže graf bol súvislý už aj predtým, a my sme teraz nejaké hrany pridali, to znamená, že sa vytvoril aspoň jeden nový cyklus. To znamená, že nejaké z nových hrán môžeme odstrániť, graf bude naďalej súvislý a dokonca ešte lacnejší a to je spor s tým, že sme predpokladali, že riešenie bolo optimálne.

Teraz už vieme urobiť $O(n^2 \log n)$ riešenie za 4 body - postupne skúsime odobrať každý vrchol i a hrany s

ním susediace, ostatným pôvodným hranám priradíme cenu 0, následne pridáme hrany $(1,2),(2,3),...,(i-1,i+1),...,(n-1,n)$ (každej priradíme cenu rovnú jej kontrastu) a nájdeme minimálnu kostru, ako sme ukázali, to je optimálna množina hrán ktorú chceme v grafe mať po tomto odobraní. Ak ešte tieto sady vyriešené nemáš, odporúčam si ísť toto riešenie ihneď naprogramovať skôr než sa pustíš do čítania zvyšku vzoráku. Môže sa ti neskôr hodiť aj ako bruteforce, proti ktorému možno testovať nefungujúce rýchlejšie riešenia.

Vzorové riešenie

Ako uvidíme zachvíľu, riešenie na plný počet bodov má v princípe rovnakú myšlienku, ale nemôžeme si dovoliť zakaždým staviať také veľké kostry. Označme si množiny vrcholov patriacich do toho istého komponentu po odobratí vrcholu i ako $C_i(1), C_i(2), ..., C_i(d_i)$. Prichádza asi najtrikovejšie pozorovanie celej úlohy: Existuje optimálne riešenie, ktoré pridáva len hrany spomedzi $(min(C_i(1))-1, min(C_i(1)), ..., (min(C_i(d_i))-1, min(C_i(d_i)), (max(C_i(1))-1, max(C_i(1)), ..., (max(C_i(d_i))-1, max(C_i(d_i)), (i-1, i+1)$.

Odporúčam sa v tomto momente zastaviť, nakresliť si niekoľko príkladov, že ako táto množina vyzerá pre konkrétne grafy a pokúsiť sa najprv samostatne dokázať, prečo je optimálna.

Dôkaz trikového pozorovania: Najprv ukážeme, že po pridaní všetkých navrhovaných hrán bude graf opäť súvislý. Ak by to nebola pravda, ozančme si a najmenší vrchol taký, že vrcholy a a $a+1$ sú v rôznych komponentoch. Ak $a \neq i, i-1$ tak potom $a+1$ je zjavne najmenší vrchol vo svojom komponente (lebo zjavne všetky vrcholy $1, ..., a$ sú v rovnakom komponente) a preto existuje hrana z $a+1$ do a ktorú sme mohli pridať a zvýšiť tým počet súvislých komponentov. Ak $a = i$ alebo $a = i+1$, spor sa ukáže veľmi podobne, stačí sa pozrieť buď na dvojicu $(a, a+2)$, alebo na dvojicu $b, b+1$, kde b je druhý najmenší vrchol taký, že b a $b+1$ nie sú v rovnakom komponente. Teraz už len stačí ukázať, že kostra ktorú nájdeme bude skutočne najlacnejšia. To si vieme rozmyslieť tak, že sa pozrieme na to, čo spoja hrany dĺžky 1 - uvidíme, že dva komponenty, a teda potreba pridať hranu dĺžky 2, vznikne len a len vtedy, ak v pôvodonom grafe neexistovala hrana spájajúca vrchol s menším číslom ako i a vrchol s väčším číslom ako i, teda cena nájdená týmto riešením sa zhoduje s dolným odhadom na cenu nájdeným na začiatku vzoráku.

Na to, aby sme efektívnejšie vedeli simulovať hladanie kostry ju nebudeme staviať na celom pôvodnom grafe, ale len na susedoch vrcholu i. Keď rozmýšlame, či pridať nejakú hranu, nahradíme čísla vrcholov jej koncov číslami susedov vrcholu i, do ktorých komponentov tieto vrcholy patria. Toto vieme robiť pomocou techniky známej ako binary lifting.

Pre samotný Kruskalov algoritmus si postavíme union find, ale aby sme nemuseli to pole predkov a veľkostí vždy vytvoriť nanovo, použijeme zakaždým to isté pole, ale medzi jednotlivými iteráciami mazania vrcholov vždy napravíme tie hodnoty, ktoré sme práve menili (čiže políčka zodpovedajúce susedom vrcholu i).

Tiež si na samotnom začiatku pre každú hranu spočítame minimum a maximum v dvoch komponentoch ktoré dostaneme po jej odobratí, toto sa dá napríklad pomocou dvoch DFS: zakoreníme si strom v ľubovoľnom vrchole a najprv spočítame minimum a maximum pre podstrom každého vrcholu a potom pomocou nich spočítame tieto hodnoty aj pre tie "opačné" podstromy.

Vďaka tomuto predpočítavaniu vieme nájsť riešenie po zmazaní vrcholu i v časovej zložitosti $O(d_i \log n)$, čo sa sčíta na celkovú časovú zložitosť $O(n \log n)$.

Pamäťová zložitosť je tiež $O(n \log n)$, bottle-neck je pole predpočítaných hodnôt pre binary lifting.

Listing programu (C++)

```cpp
// O(n log n) vzorove riesenie
#include <algorithm>
#include <iostream>
#include <map>
#include <queue>
#include <random>
#include <set>
#include <string>
#include <vector>
typedef long long ll;
using namespace std;

const int inf = 1e9 + 5, logn = 17; // nekonecno a log n (aby sme vedeli najst k-teho predka)
struct interval
```

```
    // reprezentuje interval [l, r], pruky pridavame cez upd () a [l, r] je uzdy najmensi interval ktory pokryva usetky pridane pruky
{
    int l, r;
    interval() { l = inf, r = -inf; }
    void upd(int x) { l = min(l, x), r = max(r, x); }
};
vector<vector<int> > g; // vstupny graf
vector<int> d; // hlbky vrcholov
vector<vector<int> > l; // binary lifting
vector<vector<interval>> m;
// pre kazdy vrchol si zistime intervaly pokryvajuce komponenty na ktore sa strom rozpadne ked ho odstranime
vector<int> p; // rodicia v union finde (pozor, nie rodicia v strome)
// nasleduju klasicke union find funkcie
int root(int i) { return i == p[i] ? i : p[i] = root(p[i]); }
bool merge(int i, int j)
{
    i = root(i), j = root(j);
    if (i == j) return false;
    p[i] = j;
    return true;
}
// koniec union findu
vector<interval> s; // pre kazdy podstrom si zistime najmensie a najvacsie cislo vrcholu v nom
void dfs1(int u, int p)
{
    if (p != u)
    {
        g[u].erase(find(g[u].begin(), g[u].end(), p));
        // odstranime otca z mojich susedov aby tam ostali len moje deti
        // inak pozor ked pouzivas funkcie find a erase, casova zlozitost
        // je linearna od velkosti vrcholu z ktoreho mazes
    }
    // ideme vypocitat skoky 1, 2, 4, 8, ... vrcholov nahor aby sme vedeli hladat k-teho predka
    l[0][u] = p;
    for (int i = 1; i < logn; i++) l[i][u] = l[i - 1][l[i - 1][u]];
    s[u].l = s[u].r = u;
    for (int v : g[u])
    {
        d[v] = d[u] + 1;
        dfs1(v, u);
        m[u].push_back(s[v]); // pridame interval podstromu dietata medzi moje intervaly
        s[u].upd(s[v].l);
        s[u].upd(s[v].r);
    }
}
// funkcia ktora zisti, v podstrome ktoreho suseda u sa nachadza vrchol v
int cislo_suseda(int u, int v)
{
    if (d[v] <= d[u]) return (u == v ? u : l[0][u]);
    int k = d[v] - d[u] - 1;
    for (int i = logn - 1; i >= 0; i--) if ((1 << i) <= k) k -= (1 << i), v = l[i][v];
```

```
66      if (l[0][v] == u) return v;
67      return l[0][u];
68  }
69  // ideme pre kazdy vrchol spocitat interval prisluchajuci podstromu ktory ma jeho otec ked
    ↪  zoberieme prec tento vrchol
70  void dfs2(int u, interval otcov)
71  {
72      vector<interval> pf(g[u].size() + 1), sf(g[u].size() + 1);
73      pf[0] = otcov, sf[0] = otcov;
74      pf[0].upd(u), sf[0].upd(u);
75      // ok teraz detom musim pomoct spocitat tie intervaly tiez
76      for (int i = 0; i < g[u].size(); i++)
77      {
78          pf[i + 1] = pf[i], pf[i + 1].upd(s[g[u][i]].l), pf[i + 1].upd(s[g[u][i]].r);
79          sf[i + 1] = sf[i], sf[i + 1].upd(s[g[u][g[u].size() - 1 - i]].l), sf[i +
    ↪  1].upd(s[g[u][g[u].size() - 1 - i]].r);
80      }
81      for (int i = 0; i < g[u].size(); i++)
82      {
83          interval now = pf[i];
84          now.upd(sf[g[u].size() - i - 1].l), now.upd(sf[g[u].size() - i - 1].r);
85          if (otcov.l != inf) m[u].push_back(otcov);
86          dfs2(g[u][i], now);
87      }
88  }
89  int main()
90  {
91      ios::sync_with_stdio(false);
92      cin.tie(0); // rychle nacitavanie
93      int t;
94      cin >> t;
95      while (t--)
96      {
97          int n;
98          cin >> n;
99          // ideme vycistit vsetky polia - velmi dolezite pri ulohach s viacerymi vstupmi!
100         g.assign(n, {}), s.assign(n, interval()), m.assign(n, {}), l.assign(logn, vector<int>(n,
    ↪  0)), d.assign(n, 0), p.assign(n, -1);
101         // nacitame strom a spustime predpocitavanie
102         for (int i = 0, u, v; i < n - 1; i++) cin >> u >> v, g[--u].push_back(--v),
    ↪  g[v].push_back(u);
103         dfs1(0, 0);
104         dfs2(0, interval());
105         for (int i = 0; i < n; i++)
106         {
107             if (g[i].empty())
108             {
109                 // som list - strom zostane cely aj bez mna, takze netreba nic robit
110                 cout << "0 0\n";
111                 continue;
112             }
113             if (i) g[i].push_back(l[0][i]);
114             for (int j : g[i]) p[j] = j;
```

```
115             int edges = g[i].size() - 1, coins = 0;
116             vector<pair<int, int> > ans; vector<pair<int, int> > v;
117             // najdeme relevantne hrany a skusime ich pridat
118             for (interval j : m[i]) v.push_back({ j.l, j.l - 1 }), v.push_back({ j.r, j.r + 1 });
119             for (pair<int, int> j : v)
120             {
121                 if (j.second >= 0 && j.second < n && j.second != i)
122                 {
123                     // ideme si vypocitat ktorych dvoch mojich susedov sa tyka tato hrana
124                     int a = cislo_suseda(i, j.first), b = cislo_suseda(i, j.second);
125                     // pozrieme sa ci ju chceme zobrat
126                     if (merge(a, b)) ans.push_back(j);
127                 }
128             }
129             // skontrolujeme ci este treba pridat tu specialnu hranu s cenou dva
130             if (i && i + 1 < n && merge(cislo_suseda(i, i - 1), cislo_suseda(i, i + 1)))
    ↪ ans.push_back({ i - 1, i + 1 });
131             // spocitame kolko minci nas to vlastne stalo a vypiseme odpoved
132             for (pair<int, int> j : ans) coins += abs(j.first - j.second);
133             cout << edges << " " << coins << "\n";
134             for (pair<int, int> j : ans) cout << j.first + 1 << " " << j.second + 1 << "\n";
135             cout << "\n";
136         }
137     }
138     return 0;
139 }
```