Korešpondenčný seminár z programovania
XLII. ročník, 2024/25
FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 2. kola letnej časti

Dušan

1. Poďte si adoptovať psíkov

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

Môžeme sa zamyslieť: Vie nastať situácia, kedy sú v rade aj ľudia, aj psy, ale nikde človek nesusedí so psom? Evidentne nie. Prečo? Lebo každý súvislý úsek ľudí je na krajoch zakončovaný koncom rady alebo psom. Rovnako to funguje aj keď psov a ľudí vymeníme. A keďže máme aspoň dva súvislé úseky (aspoň jeden úsek ľudí a aspoň jeden úsek psov) a iba dva konce radov, tak niekde musí susediť človek so psom.

Čiže pokiaľ existuje v rade aspoň jeden človek a aspoň jeden pes, tak existuje dvojica, ktorú môžeme vymazať. Toto opakujeme, pokiaľ sú v rade aj psy aj ľudia. Každá adopcia zmenší počet ľudí aj psov o 1, čiže na konci dňa ostanú buď iba ľudia alebo iba psy. Takže minimálny počet znakov, ktoré ostanú na konci je $|\text{počet psov} - \text{počet ľudí}|$.

Záleží na poradí ľudí a psov v rade?

Nie, lebo v našich úvahách vyššie sme nikdy nepredpokladali, že ľudia a psy sú zoradené nejako špeciálne. Vždy sme len hovorili, že ak sú v rade aspoň jeden človek a aspoň jeden pes, tak existuje dvojica susediacich ľudí a psov, ktorých môžeme vymazať. Čiže ak by sme zmenili poradie ľudí a psov, tak by sa nič nezmenilo.

Časová zložitosť

Časová zložitosť je $O(n)$, keďže potrebujeme prejsť celý rad a spočítať počet ľudí a psov.

Pamäťová zložitosť

Záleží, či načítavame celý rad do pamäte, a potom ho spracovávame, alebo či načítame iba jeden znak, spracujeme ho a potom načítame ďalší znak. V prvom prípade je pamäťová zložitosť $O(n)$, v druhom prípade je pamäťová zložitosť $O(1)$. Na plný počet bodov stačila implementácia s pamäťovou zložitosťou $O(n)$.

Vzorový python kód má pamäťovú zložitosť $O(n)$, pretože načítava celý rad do pamäte. Vzorový C++ kód má pamäťovú zložitosť $O(1)$, pretože načítava iba jeden znak naraz.

Listing programu (Python)

```python
N = int(input())
s = input()

psi = 0
ludia = 0
for ch in s:
    if ch == 'P':
        psi += 1
    else:
        ludia += 1

print(abs(psi - ludia))
```

Listing programu (C++)

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

```cpp

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

    int psi = 0;
    int ludia = 0;
    for (int i = 0; i < N; i++)
    {
        char x;
        cin >> x;
        if (x == 'P')
            psi++;
        else
            ludia++;
    }

    cout << abs(psi - ludia) << "\n";
}
```

Oliver

2. Sssssss, tak robia hady

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

Riešenie hrubou silou

Úlohu vieme naivne riešiť pomocou hrubej sily. Pre každú súradnicu, kde mohol had začať, môžeme odsimulovať pohyb hada nasledovným spôsobom. Každým krokom si vypočítame zmenu súradníc:

1. `H`: y+=1
2. `D`: y-=1
3. `P`: x+=1
4. `L`: x-=1

Skontrolujeme či sme nevyšli z mriežky, teda skontrolujeme či platia nerovnosti: 1. $0 \leq x \leq m$ 2. $0 \leq y \leq n$
Ešte nám zostáva skontrolovať či had nerazil sám do seba. To vieme napríklad takto.
Zakaždým si uložíme pár súradníc do listu a skontrolujeme či sa tam už nenachádzali predtým. Ak sa v liste už súradnice nachádzajú, tak had narazil sám do seba.

Optimálne riešenie

Základným pozorovaním v úlohe je, že had sa pohybuje iba v obdĺžniku s rozmermi a, b. V úlohe sa teda pýtame koľkými spôsobmi vieme umiestniť tento obdĺžnik do mriežky zo zadania. Zároveň platí, že ak had narazí sám do seba nebude to závisieť od pozície obdĺžnika a narazí zakaždým. Pri kontrole narazení hada do samého seba je však potrebné použiť dátovú štruktúru set (unordered). Takto budeme môcť vkladať súradnice a kontrolovať ich výskyt v časovej zložitosti O(1).[1]
Rozmery tohto obdĺžnika získame tak, že si budeme pamätať minimum a maximum z týchto hodnôt a neskôr ich od seba odčítame.
Počet možností ako vieme umiestniť tento obdĺžnik do mriežky zo zadania vieme vypočítať ako $(m - a) \times (n - b)$ kde:
$a = max_x - min_x$
$b = max_y - min_y$
Umiestnenie obdĺžnika si vieme predstaviť ako umiestnenie ľavého horného bodu obdĺžnika, tak aby pravý dolný bod nevyčnieval z mriežky.

[1] Ak by ste sa chceli dozvedieť viac o dátovej štruktúre set, odporúčam: https://www.geeksforgeeks.org/introduction-to-set-data-structure/

Časová a pamäťová zložitosť

Takéto riešenie bude mať časovú zložitosť $O(t)$ a pamäťovú zložitosť $O(t)$

Všetky set-ové operácie, ktoré vykonávame t-krát budú v O(1) čase. Kvôli kontrole kolízie hada, si budeme potrebovať pamätať všetkých t súradníc.

Plné body môžete získať aj keď využijete ordered set.

Listing programu (Python)

```python
N, M = map(int, input().split())
S = input()

cur = (0, 0)
vis = set()
maxx, minx, maxy, miny = 0, 0, 0, 0

for ch in S:
    if ch == "D":
        cur = (cur[0]-1, cur[1])
    if ch == "H":
        cur = (cur[0]+1, cur[1])
    if ch == "P":
        cur = (cur[0], cur[1]+1)
    if ch == "L":
        cur = (cur[0], cur[1]-1)

    maxx = max(maxx, cur[0])
    minx = min(minx, cur[0])
    maxy = max(maxy, cur[1])
    miny = min(miny, cur[1])

    if cur in vis:
        print(0)
        exit()

    vis.add(cur)

width = maxx-minx
height = maxy-miny

if (M-width) <= 0 or (N-height) <= 0:
    print(0)
    exit()

print((M-width)*(N-height))
```

Listing programu (C++)

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

int main() {
```

```
    cin.tie(0)->sync_with_stdio(0);

    ll m, n;
    cin >>m>>n;
    string s;
    cin >>s;

    set<pair<int,int>> visited;
    ll x = 0; ll y = 0;
    bool narazil = false;

    ll max_x = 0; ll min_x = 0; ll max_y = 0; ll min_y = 0;
    for(int i = 0; i < s.size(); i++){
        if(s[i] == 'H'){
            y++;
        }else if(s[i] == 'D'){
            y--;
        }else if (s[i] == 'P'){
            x++;
        }else{
            x--;
        }

        max_x = max(x, max_x);
        min_x = min(x, min_x);
        max_y = max(y, max_y);
        min_y = min(y, min_y);

        if(visited.count({x,y}) > 0) {
            narazil = true;
        }
        visited.insert({x,y});
    }

    if (narazil or max_x - min_x > m or max_y - min_y > n){
        cout << "0\n";
    }else{
        cout << (m - (max_x-min_x)) * (n - (max_y-min_y))<< '\n';
    }

}
```

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

3. Íš zrozumitelne?

Bruteforce

Na vyriešenie prvej sady postačovalo jednoduché prehľadávanie všetkých možných sekvencií prehľadávaní bruteforceom. To sa dá rekurzívne implementovať v časovej zložitosti $O(n^4)$ (rozmyslite si prečo!).

Niečo lepšie

Stavy dynamiky

Namiesto ukladania všetkých podreťazcov si vytvoríme jeden zoznam $dp[]$ veľkosti $n+1$, kde:

- $dp[i]$ = minimálny počet krokov na vymazanie podreťazca $string[i:]$
- $dp[n] = 0$ (prázdny reťazec nepotrebuje žiadne kroky)

Smer výpočtu

Riešime problémy od najmenších k najväčším.
Ideme od konca reťazca k začiatku. Keď už máme pre nejaký suffix vypočítané hodnoty dp, vieme túto hodnotu vypočítať pre pozíciu hneď pred ním – pozrieme si, aké je na tejto pozícií písmenko a nájdeme všetky zhodné písmenká v suffixe. Následne môžeme pre každú takúto zhodu úsek po ňu zmazať a potom zopakovať postup, ktorý sme našli pre dp nasledujúcej pozície.

Teda $dp[i] = \min_{j \in \{i+1,\dots,n\} \wedge s[j]=s[i]}(dp[j]+1)$.

Listing programu (Python)

```
word = input().strip()
n = len(word)

dp = [float('inf')] * (n + 1)
dp[n] = 0

for start in range(n - 1, -1, -1):
        first_char = word[start]
        for i in range(start, n):
                if word[i] == first_char:
                        dp[start] = min(dp[start], 1 + dp[i + 1])

print(dp[0])
```

Zložitosť riešenia

- **Časová zložitosť:** $O(n^2)$, kde n je dĺžka vstupného reťazca. To preto, lebo pre každý znak prechádzame všetky nasledujúce znaky a $1 + 2 + ... + n = \frac{n(n+1)}{2} \in O(n^2)$.
- **Pamäťová zložitosť:** $O(n)$.

Optimálne riešenie

Všimnime si, že bottleneck predošlého riešenia je to, že pre každý znak prehľadávame všetky znaky, s ktorými by sme ho mohli spojiť. Avšak, to ktorý z tých znakov si skutočne vyberieme je určené jedinou konštantnou hodnotou – minimálnou príslušnou hodnotou dp. Avšak evidentne keď už si raz nevyberieme nejakú pozíciu, znamená to že sme našli lepšiu. Ale tá bude lepšia aj vždy v budúcnosti a tak tú prvú môžeme zahodiť.

Táto idea sa dá interpretovať aj kúsok inak: Pre každý unikátny znak si budeme pamätať minimálny počet krokov, ktoré potrebujeme na odstránenie reťazca končiaceho týmto písmenkom. Tieto hodnoty sa nám budú postupne pri prechádzaní reťazcom upravovať. Všimnime si, že tieto hodnoty určite nikdy neklesajú - keď už sme raz vedeli reťazec končiaci `a` zmazať k krokmi, tak aj akýkoľvek predĺžený reťazec končiaci `a` bude možné zmazať najviac k krokmi – proste len posledné zmazanie neukončíme na tom pôvodnom `a`, ale predĺžime ho až na nový koniec.

Na začiatku vieme len to, že prvé písmenko v reťazci vieme odstrániť jediným krokom. Následne, keď spracovávame ďalší znak, máme vždy dve možnosti:

1. Tento znak sme ešte nevideli. V tomto prípade ale nevieme urobiť nič iné, ako zmazať predošlý reťazec separátne a na tento jeden znak využiť krok navyše. Teda si preň uložíme hodnotu o 1 väčšiu ako predošlý znak v reťazci.
2. Znak sme už videli. Tu máme dve možnosti:

- Buď ho zmažeme separátne rovnako v prvom prípade
- Alebo len predĺžime zmazanie predošlého výskytu tohoto znaku ako sme popísali v predošlom odstavci.

Zložitosť

- **Časová zložitosť:** $O(n)$, kde n je dĺžka vstupného reťazca.
- **Pamäťová zložitosť:** $O(k)$, kde k je počet rôznych znakov v reťazci. To je pretože si potrebujeme pamätať minimálny počet krokov pre každý znak. Všimnime si, že reťazec spracovávame po jednom znaku, teda ho nepotrebujeme ukladať celý do pamäte.

Listing programu (Python)

```python
s = input()

presolve = {s[0] : 1}

for i in range(1, len(s)):
    previous = s[i-1]
    current = s[i]

    if previous in presolve:
        if current not in presolve:
            presolve[current] = presolve[previous] + 1
        else:
            presolve[current] = min(presolve[previous] + 1, presolve[current])

print(presolve[s[-1]])
```

Listing programu (C++)

```cpp
#include <iostream>
#include <string>
#include <unordered_map>
#include <algorithm>

int main() {
    std::string s;
    std::cin >> s;

    std::unordered_map<char, int> presolve;
    presolve[s[0]] = 1;

    for (size_t i = 1; i < s.size(); ++i) {
        char previous = s[i - 1];
        char current = s[i];

        if (presolve.count(previous)) {
            if (!presolve.count(current)) {
                presolve[current] = presolve[previous] + 1;
            } else {
                presolve[current] = std::min(presolve[previous] + 1, presolve[current]);
            }
        }
    }

    std::cout << presolve[s.back()] << std::endl;

```

```
28      return 0;
29  }
```

Strizo

4. Koľko strihaní potrebujeme?

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

Skúšame všetky možnosti

Nič nepokazíme hrubou silou: vygenerujeme všetky možné rôzne poradia miest, tých je $n \times (n-1) \times \ldots 1 = n!$, napríklad pomocou `next_permutation` alebo `itertools.permutations()`, a pre každé takéto poradie overíme, či spĺňa podmienku. Konkrétne si pre každé poradie vieme postupne simulovať vypínanie miest.

Po každom odpojení prvých $1 \leq i \leq n-1$ miest, si treba overiť, či všetky zostávajúce mestá sú stále spojené s elektrárňou. Ako prvé skontrolujeme, či sme náhodou neodpojili aj ju (ak áno, musí byť posledné vypnuté mesto v poradí).

Následne si vieme spustiť prehľadávanie grafu z elektrárne (DFS[2] alebo BFS[3], viď kuchárku). Potom si len stačí overiť, či sme každé miesto už buď odpojili, alebo navštívili.

Pre každé možné poradie spustíme n-krát prehľadávanie celého grafu, ktoré zaberie v najhoršom prípade $O(n+m)$ času. Poradí je $n!$, takže časová zložitosť je $O(n! \cdot n \cdot (n+m))$. Pamäťová zložitosť je $O(n+m)$, keďže si musíme náš graf celý pamätať.

Krok ku riešeniu

Všimnite si, že už len zisťovanie, či nejaké poradie funguje nám zaberie *kvadratický* čas – $O(n(n+m))$, čo je príliš veľa.

Skúsme, namiesto tipovania riešení a skúšania či fungujú, sa zamyslieť nad tým aké riešenia určite fungovať budú. Všimnime si, že mesto ktoré odstránime posledné vieme – je ním mesto, v ktorom je elektráreň. Nazvime ho a_n.

Čo vieme povedať o meste, ktoré odpojíme ako predposledné? Zjavne musí priamo susediť s elektrárňou, inak by stratilo prúd skôr než ho odpoja.

Zoberme si teda **akékoľvek** mesto súvisiace s elektrárňou, nazvime ho a_{n-1}.

Keď už máme mestá ktoré odpojíme ako posledné a predposledné (a_n a a_{n-1}), nájdeme si mesto, ktoré odpojíme ako pred-predposledné, následne nájdeme to pred-pred-posledné, a tak ďalej.

Inak povedané, postupnosť zostavujeme odzadu. A teda namiesto toho, aby sme mestá odstraňovali, budeme mestá postupne pridávať. A chceme ich pridávať tak, aby sieť, ktorú si postupne budujeme, bola vždy súvislá, a nachádza sa v nej elektráreň.

Druhú podmienku, ako sme si už povedali hore, splníme tak, že poslednú vždy odstránime elektráreň. Ako splníme prvú podmienku? Jednoducho, keď pridávame mesto a_i, pridáme mesto ktoré sme ešte nepridali (nie je medzi $a_{i+1}, \cdot a_n$), ale ktoré je priamo spojené s niektorým a_{i+j}.

Na základe tohto nápadu vieme implementovať riešenie bežiace v $O(n \cdot (n+m))$: najskôr nadstavíme $a_n = k$ (mesto s elektrárňou). Potom, budujeme postupnosť odzadu: ak už máme $a_n, a_{n-1}, \ldots, a_{i+1}$, si prejdeme susedov týchto miest, a nájdeme takého ktorý sa ešte v postupnosti nenachádza. Zakaždým prejdeme najviac m hrán, takže dostaneme horeuvedenú časovú zložitosť.

Vzorové riešenie

Odtiaľto sme už len kúsok od vzorového riešenia. Všimnite si, že v riešení hore pozeráme na hrany viackrát. Stačilo by nám, napríklad, si vždy keď do postupnosti pridáme nové a_i, si jeho susedov pridať do fronty (alebo zásobníka). Potom, keď hľadáme nový vrchol do postupnosti, prejdeme si túto frontu (zásobník) vrcholov. Ak sme už vrchol dali do postupnosti, odstránime ho z fronty (zásobníku) a túto hranu už nikdy neuvidíme – nemusíme.

Ak sme ešte vrchol nevideli, nastavíme ho ako nový člen postupnosti a pridáme jeho susedov do fronty (zásobníku).

Takto robíme to, čo predtým, len efektívnejšie: ku každému susedovi sa dostaneme maximálne raz, ale zároveň, keďže je naša mestská sieť súvislá, určite naplníme celú postupnosť. Takto teda dostaneme riešenie bežiace v časovej a aj pamäťovej zložitosti $O(n+m)$.

A nielen to! Záležiac od toho akú dátovú štruktúru ste použili, naprogramovali ste vlastne BFS (v prípade fronty), alebo DFS (v prípade zásobníka)!

[2] https://www.ksp.sk/kucharka/dfs/
[3] https://www.ksp.sk/kucharka/bfs/

Prečo toto riešenie vždy funguje?

Pozrime sa napokon na to, *prečo* je takáto postupnosť odpájania $a_1, \ldots, a_n$ v súlade s požiadavkami zo zadania.

1. **Zachovanie spojitosti**: Keď vypíname mestá v reverznom poradí akom sme ich pridávali (čiže od a_1 po a_k), vždy vypneme najskôr tie mestá, ktoré boli pridané ako posledné pri budovaní siete. Ale pridávali sme ich predsa tak, aby mestá $a_i, a_{i+1}, \ldots, a_n$ boli súvislé, čiže vypnutie prvých i miest nikdy nepreruší spojenie medzi elektrárňou a zvyšnými mestami.
2. **Elektráreň posledná**: Keďže sme začali prehľadávanie v elektrárni, tá sa automaticky stane prvým prvkom v poradí návštev, a teda posledným prvkom v poradí vypínania.
3. **Úplné pokrytie**: Elektrická sieť je súvislá, takže ak nám ostávajú ešte nejaké mestá, je medzi nimi, a už pridanými mestami nejaká hrana.

Listing programu (Python)

```python
n,m,k = map(int,input().split())
susedia = [[] for _ in range(n)]

for _ in range(m):
    a,b = map(int, input().split())
    susedia[a].append(b)
    susedia[b].append(a)

prejdeny = [False] * n
poradie = []
stack = [k]
prejdeny[k] = True

while stack:
    vrchol = stack.pop()
    poradie.append(vrchol)

    for sused in susedia[vrchol]:
        if not prejdeny[sused]:
            prejdeny[sused] = True
            stack.append(sused)

print(*reversed(poradie))
```

Listing programu (C++)

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

using namespace std;

#define FOR(i,n)        for(int i=0;i<(int)n;i++)
#define FOB(i,n)        for(int i=n;i>=1;i--)
#define MP(x,y)        make_pair((x),(y))
#define ii pair<int, int>
#define lli long long int
#define ld long double
#define ulli unsigned long long int
```

```
12 #define lili pair<lli, lli>
13 #ifdef EBUG
14 #define DBG        if(1)
15 #else
16 #define DBG        if(0)
17 #endif
18 #define SIZE(x) int(x.size())
19 const int infinity = 2000000999 / 2;
20 const long long int inff = 4000000000000000999;
21
22 typedef complex<long double> point;
23
24 template<class T>
25 T get() {
26     T a;
27     cin >> a;
28     return a;
29 }
30
31 template <class T, class U>
32 ostream& operator<<(ostream& out, const pair<T, U> &par) {
33     out << "[" << par.first << ";" << par.second << "]";
34     return out;
35 }
36
37 template <class T>
38 ostream& operator<<(ostream& out, const set<T> &cont) {
39     out << "{";
40     for (const auto &x:cont) out << x << ", ";
41     out << "}";
42     return out;
43 }
44
45 template <class T, class U>
46 ostream& operator<<(ostream& out, const map<T,U> &cont) {
47     out << "{";
48     for (const auto &x:cont) out << x << ", ";
49
50     out << "}"; return out;
51 }
52
53 template <class T>
54 ostream& operator<<(ostream& out, const vector<T>& v) {
55     FOR(i, v.size()){
56       if(i) out << " ";
57       out << v[i];
58     }
59     out << endl;
60     return out;
61 }
62
63 bool ccw(point p, point a, point b) {
64     if((conj(a - p) * (b - p)).imag() <= 0) return false;
```

```
65      else return true;
66  }
67
68  void dfs(int v, vector<bool> &seen, vector<int> &poradie, vector<vector<int> >&hrany) {
69      seen[v] = true;
70      for (int w : hrany[v]) {
71          if (!seen[w]) dfs(w, seen, poradie, hrany);
72      }
73      poradie.push_back(v);
74  }
75
76  int main() {
77      cin.sync_with_stdio(false); cin.tie();
78      cout.sync_with_stdio(false); cout.tie();
79      int n = get<int>();
80      int m = get<int>();
81      int k = get<int>();
82      vector<int> poradie;
83      vector<bool> seen(n, false);
84      vector<vector<int> > hrany(n);
85      FOR(i, m) {
86          int a = get<int>();
87          int b = get<int>();
88          hrany[a].push_back(b); hrany[b].push_back(a);
89      }
90
91      dfs(k, seen, poradie, hrany);
92      FOR(i, n) {
93          cout << (i ? " " : "") << poradie[i];
94      }
95      cout << endl;
96
97  }
```

TomasK

5. Obrovitánsky smäd

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

Bruteforce

Najjednoduchšie je odsimulovať zvlášť tok každej kvapky. Vtedy nám stačí udržiavať si pole počtov kvapiek, ktoré už cez jednotlivé údolia pretiekli. Pri každej kvapke potom v každom údolí, do ktorého sa táto kvapka dostane, vieme ľahko spočítať, kam má ďalej tiecť, a to budeme opakovať, až dokým nestečie až k nejakému psíkovi.

Pre všetky údolia na tomto toku si následne zaznačíme, že nimi pretiekla jedna ďalšia kvapka a toto zopakujeme pre každú kvapku.

Takto pre každú z t kvapiek prejdeme potenciálne až tok dĺžky n, teda dostávame riešenie s časovou zložitosťou $O(nt)$.

Spracujme viacero kvapiek naraz

Vo všetkých sadách okrem prvej však musíme spracovať potenciálne až 10^9 kvapiek, preto nám môže napadnúť, že potrebujeme istým spôsobom spracovať viacero kvapiek naraz. Ako to ale spravíme?

Môžeme si všimnúť, že pri veľkom počte kvapiek sa nám často môže stať, že veľa po sebe nasledujúcich kvapiek tečie presne po tej istej ceste. Pre všetky tieto kvapky tak poznáme odpoveď už po odsimulovaní toku tej prvej z nich - všetky ostatné potom stečú k tomu istému psíkovi ako tá prvá.

Stále však potrebujeme simulovať jednotlivé zmeny toku a po každej zmene toku prejsť celý tok odznova a zistiť, pri ktorom psíkovi tento tok končí.

Vieme si ale všimnúť, že zmien toku v údolí i bude za celú dobu stekania kvapiek najviac z_i. Inak povedané, celkový počet zmien toku vo všetkých údoliach dokopy za celú dobu stekania kvapiek bude najviac $\sum z_i$.

Priamočiare riešenie tak mierne vylepšime, aby sme naraz spracovávali všetky po sebe idúce kvapky, ktoré majú úplne identický tok.

Podobne ako v predošlom riešení začneme odsimulovaním toku aktuálnej kvapky. Teraz však pri simulovaní tohto toku súčasne v každom údolí toku určíme, koľko kvapiek doň ešte môže stiecť bez toho, aby sa zmenil smer stekania z daného údolia.

Keď potom vyberieme z týchto hodnôt tú najmenšiu, tak budeme vedieť, koľko ďalších kvapiek ešte stečie po úplne tom istom toku a pri ktorej najbližšej kvapke sa už jej smer stekania v nejakom údolí zmení.

Simulovania celého toku pre tento počet nasledujúcich kvapiek tak vieme preskočiť a všetky tieto kvapky priradiť ku psíkovi, u ktorého skončila aktuálne vyslaná kvapka.

Keď následne budeme chcieť spracovať kvapku, ktorá už má odlišný tok od týchto predošlých, tak musíme prinajhoršom prerátať celý tok odznova (keďže sa napríklad mohol zmeniť smer odtekania už niekde úplne na začiatku toku) a podobne určiť, koľko najbližších kvapiek stečie po tom istom toku. Toto však spravíme nanajvýš $\sum z_i$-krát, keďže nanajvýš toľko zmien v toku kvapiek sa celkovo uskutoční.

Keďže v tomto riešení musíme prerátať celý tok maximálne $\sum z_i$-krát, tak časová zložitosť tohto riešenia je $O(n \cdot \sum z_i)$, čo už stačí aj na druhú sadu.

Optimálne riešenie

Pre získanie plného počtu bodov potrebujeme tok kvapiek simulovať už nie po jednotlivých kvapkách alebo zmenách toku, ale po jednotlivých údoliach. Keď totiž vieme, koľko kvapiek sa celkovo niekedy dostane do nejakého údolia i, tak ľahko vieme tento počet rozdeliť do skupín po k_i (ak počet kvapiek v danom údolí nie je deliteľný k_i, tak nám ostane jedna posledná skupina, v ktorej bude menej než k_i kvapiek) a každú túto skupinu ďalej pošleme do príslušného ďalšieho údolia, do ktorého bude vtedy voda z tohto údolia stekať.

Vieme si všimnúť, že nám pritom vlastne ani nezáleží na tom, v akom presne čase k nám stečie ktorá kvapka, ale stačí nám vedieť tento celkový počet kvapiek, ktoré do daného údolia niekedy stečú.

Treba si však dať pozor na to, v akom poradí z jednotlivých údolí takto ďalej posielame kvapky. Ak by sme totiž poslali ďalej kvapky z nejakého údolia, do ktorého ale ešte nejaké ďalšie kvapky niekedy neskôr stečú, tak sa nám informácia o týchto kvapkách stratí a tieto kvapky sa nedostanú k jednotlivým psíkom.

Tu sa nám však hodí informácia zo zadania, že voda môže stekať len z vyššie položeného údolia do nižšie položeného údolia a že teda nemôže tiecť do cyklu. Ak by sme si teda celú túto úlohu predstavili ako orientovaný graf, kde vrcholy sú jednotlivé údolia a hrana spája vrcholy i a j práve vtedy, keď niekedy steká voda z údolia i do údolia j, tak máme zaručené, že tento graf bude acyklický, teda sa jedná o tzv. DAG (z anglického directed acyclic graph).

Pre každý DAG pritom existuje tzv. topologické usporiadanie jeho vrcholov, teda také usporiadanie jeho vrcholov, pri ktorom sa pre každú hranu jej začiatočný vrchol nachádza v tomto usporiadaní skôr než jej koncový vrchol.

Ak by sme teda v grafe prislúchajúcom tokom kvapiek medzi údoliami našli topologické usporiadanie údolí a posielali by sme kvapky ďalej z jednotlivých údolí v tom poradí, v akom sa tieto údolia nachádzajú v ich topologickom usporiadaní, tak vždy v dobe rozširovania sa z nejakého údolia do ďalších údolí je počet kvapiek v tomto údolí už finálny a vieme ich teda rozposlať ďalej bez toho, aby sa nám nejaké kvapky stratili.

Ostáva nám tak už iba nájsť topologické usporiadanie jednotlivých údolí. To vieme spraviť tak, že pre každé údolie si budeme pamätať, koľko ešte nespracovaných hrán doň smeruje. Pre vrchol, ktorý ako ďalší pridáme do topologického usporiadania, potom musí platiť, že všetky hrany smerujúce doň sme už spracovali a že keď sa k nemu teda v tomto momente dostaneme, tak počet kvapiek v ňom už bude finálny.

Vieme si tak udržiavať pole vrcholov, do ktorých už nesmerujú žiadne nespracované hrany, v každom kroku z tohto poľa nejaký vrchol vybrať a všetky hrany vychádzajúce z neho označiť za spracované tým, že vrcholom, do ktorých smerujú, počet nespracovaných hrán smerujúcich do nich znížime o 1.

Následne nám už pri tomto aktualizovaní počtu vchádzajúcich nespracovaných hrán stačí len kontrolovať, či pri nejakom vrchole tento počet neklesol na nulu a či ho teda nevieme pridať do poľa vrcholov, ktoré vieme priamo pridať do topologického usporiadania ako ďalšie.

Topologické usporiadanie údolí takto nájdeme v čase lineárnom od počtu vrcholov aj hrán v našom grafe, teda v čase $O(n + \sum z_i)$. Následne údoliu s prameňom priradíme t kvapiek a všetkým ostatným údoliam zatiaľ 0 kvapiek.

Tieto údolia budeme potom prechádzať v poradí nájdeného topologického usporiadania a počty kvapiek, ktoré do nich stiekli, budeme ďalej rozdeľovať údoliam, do ktorých tieto kvapky stečú. Takto si pre každé údolie

so psíkom vo výsledku zaznačíme, koľko kvapiek doň celkovo stieklo a tieto hodnoty vypíšeme.

Aj v tejto časti riešenia s rozdeľovaním kvapiek do ďalších údolí spracujeme každý vrchol a každú hranu len raz, teda aj toto prebehne v čase lineárnom od počtu vrcholov a hrán. Celková časová zložitosť tohto riešenia je tak $O(n + \sum z_i)$.

Listing programu (Python)

```python
n, t = map(int, input().split())
K = []
postupnosti = [[] for _ in range(n)]
hrany_do = [0 for _ in range(n)]

for i in range(n):
    A = list(map(int, input().split()))
    K.append(A[0])

    if A[0] != 0:
        for j in range(1, len(A)):
            postupnosti[i].append(A[j])
            hrany_do[A[j]] += 1

toposort = []
for i in range(n):
    if hrany_do[i] == 0:
        toposort.append(i)

kvapky = [0 for _ in range(n)]
kvapky[0] = t

while len(toposort) > 0:
    v = toposort.pop()

    kvapiek = kvapky[v]
    for u in postupnosti[v]:
        hrany_do[u] -= 1
        if hrany_do[u] == 0:
            toposort.append(u)
        if kvapiek > K[v]:
            kvapky[u] += K[v]
            kvapiek -= K[v]
        else:
            kvapky[u] += kvapiek
            kvapiek = 0

for i in range(n):
    if K[i] == 0:
        print(kvapky[i])
```

Listing programu (C++)

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

```cpp
#include <algorithm>

using namespace std;

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

    vector<vector<int>> graf (n);
    vector<int> menia_sa (n);
    vector<int> inc(n,0);
    vector<int> prislo_kvapiek(n,0);
    prislo_kvapiek[0] = t;

    for (int i = 0; i < n; i++)
    {
        cin >> menia_sa[i];
        if (menia_sa[i])
        {
            for (int g = 0; g< (t-1)/menia_sa[i] + 1; g++)
            {
                int to; cin >> to;
                graf[i].push_back(to);
                inc[to] ++;
            }
        }
    }

    vector<int> stack_udoli;
    for (int i = 0; i < n; i++)
    {
        if (inc[i] == 0)
        {
            stack_udoli.push_back(i);
        }

    }

    int ind = 0;

    while (stack_udoli.size() > ind)
    {
        int vtx = stack_udoli[ind];
        int tecie = menia_sa[vtx];
        int kvap = prislo_kvapiek[vtx];
        if (!tecie)
        {
            ind++;
        } else
        {
            for (int i = 0; i < graf[vtx].size(); i++)
            {
                int kam = graf[vtx][i];
                prislo_kvapiek[kam] += min(kvap,tecie);
```

```
                kvap -= min(kvap,tecie);
                inc[kam] -= 1;
                if (!inc[kam])
                {
                    stack_udoli.push_back(kam);
                }
            }
            ind ++;
        }
    }

    for (int i = 0; i < n; i++)
    {
        if (menia_sa[i] == 0)
        {
            cout << prislo_kvapiek[i] << endl;
        }

    }
}
```

Martin M.

6. Vyrovnaná fotka

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

Skúsenejšiemu riešiteľovi by mohlo rýchlo napadnúť riešiť túto úlohu dynamickým programovaním, čo sa ukáže ako dobrý nápad. Veľa úloh, ktoré má otázku "koľko najviac/najmenej" často býva riešených dynamickým programovaním.

Nad úlohou sa budeme ďalej zamýšľať spôsobom, že poznáme minimálnu cenu, za ktorú vieme dostať nejaký prefix postupnosti do požadovaného stavu. Potom budeme počítať prefix dĺžky o 1 viac.

Teda ďalej v riešení uvažujeme, že už sme upravili do požadovaného tvaru postupnosť dĺžky i a zistili najnižšiu cenu za ktorú to dokážeme spraviť. Potom vypočítame odpoveď pre postupnosť, ktorá s vypočítanou postupnosťou má zhodné prvky až po i, ale má jeden ďalší prvok $i+1$.

Stavy dynamického programovania

V tomto riešení budem používať značenie $minc$, aby som zaznačil minimálnu cenu, za ktorú vieme dostať niečo do stavu, ktorého chceme.

Všimnime si, že aby sme zistili $minc$ pre o 1 väčší prefix nás nemusia zaujímať presné úpravy, ktoré sme spravili a ani celý upravený prefix. Zaujímavý je len posledný prvok v upravenom prefixe. Ak chceme rozšíriť prefix upravený do vyhovujúceho tvaru, tak jediné čo potrebujeme zabezpečiť je, aby jeho posledný prvok a novo pridaný prvok mali rozdiel $\leq M$.

Tiež je dobré si uvedomiť, že posledný prvok v upravenom prefixe nemá zmysel nikdy uvažovať väčší, ako $maxA_i$, keďže potom by sme ho mohli nahradiť $maxA_i$ a dostali by sme takú istú, alebo nižšiu cenu.

Z toho už vyplýva, že budeme chcieť vypočítať hodnoty $dp[i][k]$ - najmenšia cena, pomocou ktorej vieme upraviť prefix veľkosti i do požadovaného tvaru tak, že sa bude končiť na prvok k. Finálna odpoveď potom bude najmenšia hodnota poľa $dp[n]$, kde n je dĺžka danej postupnosti.

Ešte si ale môžeme všimnúť, že v optimálnom riešení bude posledný prvok upravenej postupnosti do požadovaného tvaru stále nejaký prvok pôvodnej postupnosti. Ak by to bol vkladaný prvok, tak ho vieme nevložiť a dostať lepšiu cenu za trochu inú, ale tiež vyhovujúcu postupnosť. Toto vieme využiť a trochu upraviť, čo reprezentuje $dp[i][k]$. Pridáme podmienku, že prvok k musí byť upravený prvok pôvodnej postupnosti. S touto podmienkou stále dostaneme správnu odpoveď, ale pomôže nám to neskôr pri vymýšľaní prechodov medzi stavmi.

Riešenie na 6b

Teraz chceme vymyslieť spôsob, ako pomocou hodnôt zapamätaných pre nejaký prefix vypočítať nové hodnoty pre o 1 väčší prefix.

Stále je možnosť zmazať novo pridaný prvok za cenu D. Teda môžeme použiť operácie na najlacnejšiu úpravu prvých i prvkov do požadovaného tvaru a novo pridaný prvok zmažeme. Cenu na začiatku tak môžeme nastaviť $dp[i+1][k] = dp[i][k] + D$ pre každé k a získať minimálne ceny pre každé k, ak použijeme operáciu zmazania.

Teraz ešte ostáva zistiť $minc$, ak novo pridaný prvok nemažeme. Uvažujme teda, že chceme vypočítať $dp[i+1][k]$ pre nejaké konkrétne k. Keďže vieme, že nový prvok nemažeme a tiež sme si dali podmienku, že posledný prvok musí byť z pôvodnej postupnosti, musíme zaplatiť $|A_{i+1} - k|$ za operáciu zmeny prvku. Ešte ostáva zistiť, na akú upravenú postupnosť je najvýhodnejšie nadviazať. Teda aký prvok chceme mať na poslednom mieste v predchádzajúcom prefixe. Uvažujme, že je to l. Môžeme si všimnúť, že v takomto prípade už je presne dané, čo musíme robiť - vložiť medzi posledné 2 čísla čo najmenej prvkov tak, aby nemali žiadne rozdiel väčší ako M. To sa dá už ľahko vypočítať - budeme vkladať prvky tak, aby boli vo vzdialenosti presne M od seba, pokým nebude posledný dostatočne blízko k. Označme túto dopočítanú cenu x. Teda ak nadväzujeme na prefix dĺžky i, ktorý sa po úprave končí na l, najnižšia cena, ktorú vieme dostať bude $dp[i][l] + |A_{i+1} - k| + x$.

Teraz už nie je ťažké vymyslieť, ako to bude celé fungovať. Postupne pre každý prefix veľkosti i pre i od 1 do n pre všetky možné k vyskúšať všetky možné l a podľa popísaného postupu stále dopočítať $dp[i][k]$. Dostaneme tak funkčný algoritmus, pomocou ktorého vieme získať odpoveď ako minimum z hodnôt $dp[n]$. Tento algoritmus bude mať časovú zložitosť $O(n \cdot (maxA_i)^2)$, keďže pre každý prefix prejdeme všetky možné dvojice (l, k). Toto stačilo na program za 6 bodov.

Listing programu (C++)

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

int main() {
    int N, M, I, D;
    cin >> N >> M >> I >> D;
    vector<int> a(N);
    for (int i = 0; i < N; i++) cin >> a[i];
    int K = *max_element(a.begin(), a.end());

    vector<int> dp(K + 1, 0);
    for (int i = 0; i < N; i++) {
        vector<int> new_dp(K + 1, 0);
        for (int j = 0; j <= K; j++) {
            new_dp[j] = dp[j] + D;
        }
        for (int from = 0; from <= K; from++) {
            for (int to = 0; to <= K; to++) {
                int old_to = to;
                if (M == 0) to = from;

                int diff = max(0, abs(from - to) - M);
                int inserts_cost = (M == 0) ? diff : I * ((diff + M - 1) / M);
                int move_cost = abs(a[i] - to);
                int cost = dp[from] + inserts_cost + move_cost;
                new_dp[to] = min(new_dp[to], cost);

                to = old_to;
            }
        }
        dp = new_dp;
    }
    int res = *min_element(dp.begin(), dp.end());
    cout << res << endl;
```

```
}
```

Optimálne riešenie

Vypočítať hodnotu $dp[i+1][k]$ rovno na základe $dp[i]$ v konštantnom čase sa javí ako pomerne ťažká úloha. Skúsme preto nejak efektívne vypočítať medzistav, z ktorého už bude ľahšie získavať odpovede pre $dp[i+1][k]$. Označme tento medzistav $dp_2[i][k]$. Bude nám hovoriť, koľko najmenej musíme zaplatiť, aby sme skončili na prvku k a mali vyhovujúcu postupnosť, ale nemusíme končiť už na prvku pôvodnej postupnosti. Teda je povolené vkladať prvky aj za posledný prvok z pôvodnej postupnosti.

Pozrime sa na vkladanie prvkov pri popísanom algoritme za 6 bodov. Ak sme končili na prvku l, tak sme vkladali buď postupne prvky $l+M, l+2M, ...$ až pokým sme sa nedostali dostatočne blízko ku k, alebo $l-M, l-2M, ...$, až pokým sme sa nedostali dostatočne blízko ku k. Ak by sme teda dostali optimálnu odpoveď pre nejaké $dp_2[i][k]$ vkladaním prvkov menších ako k, tak na jej výpočet nebudeme potrebovať vedieť odpoveď pre $dp_2[i][l]$ také, že sme sa k nej dostali vkladaním prvkov väčších ako l. Ak by sme tieto hodnoty potrebovali, znamenalo by to, že je niekedy optimálne vložiť prvky $..., l+M, l, ..., k-M, k$. Vidíme, že v tomto prípade je zbytočné vložiť prvok s hodnotou l, alebo prvok $l+M$ (podľa toho, či $l < k$, alebo $l > k$) a teda to nemôže byť optimálna odpoveď.

Z toho vyplýva, že môžeme najprv napríklad vypočítať $dp_2[i][k]$ pre prípad, že budeme vkladať prvky menšie ako k $((..., k-2M, k-M)$. Toto by sme chceli vypočítať pre všetky k v čase $O(maxA_i)$. Spravíme to prechodom od najmenších hodnôt po najväčšie s tým, že si budeme pamätať stále najlepší posledný prvok, od ktorého budeme chcieť vkladať hodnoty vo vzdialenosti M od seba tak, že sa dostaneme dostatočne blízko k aktuálne počítanému prvku k. Najlepší posledný prvok si vieme jednoducho udržiavať, keďže pri tomto prechode bude stále len stúpať. Ak sme aktuálne na indexe k vieme povedať, že bude určite lepší pre všetky $q \geq k$, ako aktuálne najlepší index p ak: $dp[i][k] < I \cdot (\lfloor (k-p)/M \rfloor) + dp[i][p]$. Hodnotu $dp_2[i][k]$ teda vieme vypočítať ako $min(dp[i][k], I \cdot (\lfloor (k-p)/M \rfloor) + dp[i][p])$, kde p je najlepší posledný prvok. Následne spravíme to isté ale opačne. Teda vypočítame ceny, ak vkladáme prvky väčšie ako k $(..., k+2M, k+M)$. Teraz už samozrejme budeme zapisovať novú hodnotu, iba vtedy ak je menšia ako tá, ktorú sme vypočítali predchádzajúcim prechodom.

Keď už teraz máme vypočítané hodnoty po všetkých vkladaniach chceme zistiť, aké budú hodnoty $dp[i+1][k]$. Ako v algoritme za 6 bodov, môžeme nastaviť $dp[i+1][k] = dp[i][k] + D$ pre prípad, že novo pridaný prvok zmažeme. Ak prvok nemažeme tak za $dp[i+1][k]$ zaplatíme $min_{j \in \{k-M,...,k+M\}}(dp_2[i][j] + |A_{i+1} - k|)$. Zjavne uvažovať ostatné j nemôžeme, keďže potom by rozdiel posledných prvkov bol väčší ako M. Ostáva už len domyslieť ako postupne efektívne počítať minimá zo súvislých úsekov dĺžky $2M+1$. Toto je pomerne štandardná úloha a dá sa riešiť pomocou minimovej deque dokonca amortizovane v lineárnom čase. V našom prípade teda v $O(maxA_i)$ na vypočítanie $dp[i+1][k]$ pre všetky k. Ako presne sa to dá spraviť sa môžete dočítať napríklad tu[4]

Vymysleli sme algoritmus, ktorý vypočíta hodnoty dp v $O(n \cdot maxA_i)$, keďže hodnoty $dp_2[i]$ aj potom hodnoty $dp[i+1]$ počítame v $O(maxA_i)$ pre i od 1 po n. Pamäťovú zložitosť vieme zlepšiť na $O(n+maxA_i)$, keďže si pri počítaní $dp[i+1]$ si stačí pamätať hodnoty $dp[i]$. Teda nemusíme vytvárať celé dvojrozmerné pole, ale pamätať si stále iba jeho posledné 2 "riadky".

Listing programu (C++)

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

int32_t main() {
    int D, I, M, N;
    cin >> N >> M >> I >> D;
    vector<int> a(N);
    for (int i = 0; i < N; i++) cin >> a[i];
    int K = *max_element(a.begin(), a.end());

    vector<int> dp(K + 1, 0);
    for (int i = 0; i < N; i++) {
```

[4] https://cp-algorithms.com/data_structures/stack_queue_modification.html#queue-modification-method-1

```
        vector<int> inter_dp(dp.begin(), dp.end());
        for (int k = 0; k < M; k++) {
            int best = -1;
            for (int j = k; j <= K; j += M) {
                if (best == -1  I * ((j - best) / M) + dp[best] > dp[j]) {
                    best = j;
                }
                inter_dp[j] = min(inter_dp[j], I * ((j - best) / M) + dp[best]);
            }
        }
        for (int k = 0; k < M; k++) {
            int best = -1;
            for (int j = K - k; j >= 0; j -= M) {
                if (best == -1  I * ((best - j) / M) + dp[best] > dp[j]) {
                    best = j;
                }
                inter_dp[j] = min(inter_dp[j], I * ((best - j) / M) + dp[best]);
            }
        }

        deque<int> q;
        for (int j = 0; j < min(K + 1, M); j++) {
            while (!q.empty() && inter_dp[q.back()] >= inter_dp[j]) {
                q.pop_back();
            }
            q.push_back(j);
        }
        vector<int> new_dp(K + 1, 0);
        for (int j = 0; j <= K; j++) {
            new_dp[j] = dp[j] + D;
        }
        for (int j = 0; j <= K; j++) {
            int last = j + M;
            if (!q.empty() && q.front() + M < j) {
                q.pop_front();
            }
            if (last <= K) {
                while (!q.empty() && inter_dp[q.back()] >= inter_dp[last]) {
                    q.pop_back();
                }
                q.push_back(last);
            }
            new_dp[j] = min(new_dp[j], inter_dp[q.front()] + abs(a[i] - j));
        }
        dp = new_dp;
    }
    int res = *min_element(dp.begin(), dp.end());
    cout << res << endl;
}
```

Johanko

7. Ideálny Žbirkov život

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

Vymýšlanie a vylepšovanie bruteforce riešení prenecháme čitateľovi, ako tréning. My skočíme rovno k zá-

kladným pozorovaniam a postupom.

Počas vzoráku budeme hovoriť o obraze psa, cez nejakú tyč - to bude taký pes, ktorý je presne na opačnej strane tyče (nemusí byť nutne rovnakej farby). Ak takýto pes neexistuje, tak budeme hovoriť, že pes nemá obraz cez danú tyč.

Iba určíme farby

Predstavme si, že by nám niekto povedal, kde sa nachádzajú tyče. Skúsme teraz nájsť rýchly algoritmus, ktorým rozdelíme psíkov na bielych a čiernych, alebo prehlásime, že to nie je možné.

Vyberieme jednu tyč a prehlásime, že všetci psíkovia k nej priviazaní budú čierny. Túto tyč nazveme čiernou, druhú tyč nazveme bielou. Ani jedna farba nie je ničím špeciálna a môžme ich hocikedy vymeniť. Nám sa jednoduchšie bude pracovať s konkrétnou farbou, preto sme si vybrali napríklad čiernu.

Ak máme psa, ktorý nemá obraz cez čiernu tyč, znamená to, že pokiaľ má existovať riešenie, musí v ňom byť tento pes biely. Takýchto psov budeme volať povinne bieli.

Zoberme všetkých povinne bielych psov. Každý z nich musí mať obraz cez bielu tyč. Každý z týchto obrazov musí byť tiež biely pes, a to aj ak nie je povinne biely.

Vyberme si niektorého povinne bieleho psa x a jeho obraz, nech je to pes i. Pes i je vždy jednoznačne určený, lebo nemôžu byť dvaja psi na jednom mieste. Vieme, že pes i je biely, inak by x nemal obraz cez bielu tyč. Pokiaľ má pes i obraz cez čiernu tyč, nech je to pes j, musí aj pes j byť biely. Prečo? Ak by bol pes j čierny, musel by aj jeho obraz cez čiernu tyč – pes i – byť čierny, čo je spor.

V zásade vždy keď prehlásime nejakého psa za bieleho, musíme skontrolovať:

1. či má nejaký obraz cez bielu tyč - ak by nemal, táto voľba tyčí je nesprávna
2. či má nejaký obraz cez čiernu tyč - ak áno, tento obraz tiež prehlásime za biely a skontrolujeme ho.

Na začiatku prehlásime za bielych všetkých povinne bielych, teda takých, ktorí nemajú obraz cez čiernu tyč.

Tento algoritmus má časovú zložitosť $O(n)$ - iba pre každého psa skontrolujeme zopár obrazov.

Listing programu (C++)

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

using namespace std;

typedef long long ll;
typedef pair<ll, ll> pll;

#define For(i, a, n) for (ll i = a; i < (ll)n; i++)
#define all(x) begin(x), end(x)
#define sz(x) (ll) x.size()

ll n;
vector<pll> psy;
map<pll, ll> pozicie;  // pre poziciu aky je na nej pes

pll mid(pll a, pll b) {  // stred medzi bodmi a, b
    // ked vsetko hned na zaciatku prenasobim 2
    // nemusim sa babrat s floatami
    return {(a.first + b.first) / 2, (a.second + b.second) / 2};
}

pll image(pll a, pll s) {
    // obraz bodu a cez s
    return {s.first - (a.first - s.first), s.second - (a.second - s.second)};
}

```

```cpp
bool solve(pll mid_black, pll mid_white) {
    // predpocitam si obrazy cez cierny a biely stred
    vector<pll> sus(n, {-1, -1});

    For(i, 0, n) {
        pll image_black = image(psy[i], mid_black), image_white = image(psy[i], mid_white);
        if (pozicie.count(image_black)) sus[i].first = pozicie[image_black];
        if (pozicie.count(image_white)) sus[i].second = pozicie[image_white];

        // ak nemam ani jeden obraz, mozem skoncit
        if (sus[i].first == -1 && sus[i].second == -1) return false;
    }
    queue<ll> spracuj;

    // ak nemam suseda v ciernom, som povinne biely
    For(i, 0, n) if (sus[i].first == -1) spracuj.push(i);

    vector<ll> solved(n);
    vector<ll> white;
    while (!spracuj.empty()) {  // spracujem postupne bielych,
                                // mozno priadm aj nejakych dalsich
        ll t = spracuj.front();
        spracuj.pop();
        if (solved[t]) continue;
        solved[t] = true;
        white.push_back(t);
        if (sus[t].first != -1) {  // davam sa do white,
                                   // takze moj obraz v black musi byt white
            spracuj.push(sus[t].first);
        }
        if (sus[t].second != -1) {  // som white, takze musim mat obraz vo white
                                    // ten tiez musim niekedy spracovat
            spracuj.push(sus[t].second);
        } else
            return false;  // mal by som mat obraz vo white, ale nemam
    }
    vector<ll> ans(n);
    for (auto i : white) ans[i] = 1;
    cout << setprecision(1) << fixed;
    cout << "zbirka je kral" << endl;
    cout << mid_black.first / (double)2 << ' ' << mid_black.second / (double)2 << ' '
         << mid_white.first / (double)2 << ' ' << mid_white.second / (double)2 << endl;
    for (auto i : ans) cout << i;
    cout << endl;
    return true;
}
```

Pomocou tohoto algoritmu teraz budeme testovať, či je dané rozostavenie tyčí prípustné.. Týchto rozostavení je však veľa, preto si ukážeme, ako odstrániť niektoré, ktoré nemajú šancu byť správne.

Prvé zaujímavé riešenie

Môžeme si všimnúť, že na to, aby nejaká tyč mala šancu byť v správnom riešení, musia existovať nejakí – nie nutne rôzni – psi i a j, ktorí sa na seba cez túto tyč zobrazia (teda tyč je v strede medzi nimi). Ak by

sme v nejakom riešení vybrali tyč nesplňujúcu túto podmienku, určite by sme k nej nemohli priviazať žiadneho psa, lebo nemá obraz cez túto tyč. Preto môžeme túto tyč umiestniť hocikam inam a dostaneme rovnako dobré riešenie. Jednou z možností, kam ju umiestniť, je to isté miesto, na ktorom je druhá tyč tohto riešenia. Týmto spôsobom sme určite dodržali podmienku vyššie. K druhému stĺpu totiž musel byť priviazaný nejaký pes a jeho obraz, teda tento stĺp je uprostred nejakých dvoch psov.

Pre každú tyč teda máme $O(n^2)$ možností, kde môže byť umiestnená. Pre dve tyče spolu dostávame $O(n^4)$ možností, čo spolu s kontrolou platnosti dáva zložitosť $O(n^5)$.

Máme príliš veľa voľnosti

Na to, aby sme mali správne riešenie, musí byť každý pes priviazaný k nejakej tyči. Toto platí aj pre psíka s číslom 0. Aspoň jedna z tyčí teda musí byť taká, že psík 0 má cez ňu obraz.

Tým sa nám zmenšuje počet možností, ktoré musíme kontrolovať. Máme n možností na prvú tyč, tak, aby bola v strede medzi psíkom 0 a nejakým ďalším psíkom (kľudne aj psíkom 0). Druhú tyč budeme umiestňovať rovnako ako doteraz, teda na jej pozície máme $O(n^2)$ možností.

Každú z týchto možností skontrolujeme v čase $O(n)$, dostávame teda $O(n^4)$ riešenie.

Všimnime si ešte, že pes 0 nemusí byť v riešení priviazaný k prvej tyči. Zjavne však ku niektorej tyči priviazaný bude, teda v konečnom dôsledku len neskúšame možnosti, ktoré určite fungovať nebudú.

Zopakujeme starú myšlienku

Ešte stále skúšame niektoré možnosti zbytočne.

Zafixujeme si opäť psa 0 a povedzme, že bude čiernej farby. Vyskúšame všetky možnosti, na ktorého psa sa pes 0 zobrazí. To nám zadá pozíciu čiernej tyče. Podobne ako v prvom odseku vieme teraz zostrojiť množinu psov, ktorí sú povinne bieli.

Teraz použijeme myšlienku z predchádzajúceho odseku. Pes 0 nebol ničím špeciálny. My máme množinu povinne bielych a vieme, že každý z nich musí mať obraz cez bielu tyč. Opäť si môžeme vybrať ľubovoľného povinne bieleho psa – napríklad s najmenším číslom – a skúšať len stĺpy, ktoré sú v strede medzi ním a nejakým ďalším psom. Týmto spôsobom máme aj na pozíciu druhej tyče len $O(n)$ možností.

Máme teda n možností na pozíciu čiernej tyče a pre každú z nich $O(n)$ možností na pozíciu bielej. Spolu máme $O(n^2)$ možností a pre každú vyskúšame v $O(n)$, či vyhovuje. Tým dostávame optimálnu zložitosť $O(n^3)$.

Listing programu (C++)

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

using namespace std;

typedef long long ll;
typedef pair<ll, ll> pll;

#define For(i, a, n) for (ll i = a; i < (ll)n; i++)
#define all(x) begin(x), end(x)
#define sz(x) (ll) x.size()

ll n;
vector<pll> psy;
map<pll, ll> pozicie;  // pre poziciu aky je na nej pes

pll mid(pll a, pll b) {
    return {(a.first + b.first) / 2, (a.second + b.second) / 2};
}

pll image(pll a, pll s) {
    return {s.first - (a.first - s.first), s.second - (a.second - s.second)};
}

```

```cpp
bool solve(pll mid_black, pll mid_white) {
    // predpocitam si obrazy cez cierny a biely stred
    vector<pll> sus(n, {-1, -1});

    For(i, 0, n) {
        pll i1 = image(psy[i], mid_black), i2 = image(psy[i], mid_white);
        if (pozicie.count(i1)) sus[i].first = pozicie[i1];
        if (pozicie.count(i2)) sus[i].second = pozicie[i2];

        // ak nemam ani jeden obraz, mozem skoncit
        if (sus[i].first == -1 && sus[i].second == -1) return false;
    }
    queue<ll> q;

    // ak nemam suseda v ciernom, musim byt biely
    For(i, 0, n) if (sus[i].first == -1) q.push(i);

    vector<ll> solved(n);
    vector<ll> white;
    while (!q.empty()) {  // spracujem postupne bielych,
                          // mozno priadm aj nejakych dalsich
        ll t = q.front();
        q.pop();
        if (solved[t]) continue;
        solved[t] = true;
        white.push_back(t);
        if (sus[t].first != -1) {  // davam sa do white,
                                   // takze moj obraz v black musi byt white
            q.push(sus[t].first);
        }
        if (sus[t].second != -1) {  // som white, takze musim mat obraz vo white
                                    // ten tiez musim niekedy spracovat
            q.push(sus[t].second);
        } else
            return false;  // mal by som mat obraz vo white, ale nemam
    }
    vector<ll> ans(n);
    for (auto i : white) ans[i] = 1;
    cout << setprecision(1) << fixed;
    cout << "zbirka je kral" << endl;
    cout << mid_black.first / (double)2 << ' ' << mid_black.second / (double)2 << ' '
         << mid_white.first / (double)2 << ' ' << mid_white.second / (double)2 << endl;
    for (auto i : ans) cout << i;
    cout << endl;
    return true;
}

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

    cin >> n;
    psy.resize(n);
```

```
    for (auto& i : psy) {
        cin >> i.first >> i.second;
        i.first = i.first * 2;     // ked hodnoty vynasobim 2,
        i.second = i.second * 2;  // nemusim pouzivat double na delenie
        pozicie[i] = sz(pozicie);
    }
    For(j, 0, n) {
        pll mid_black = mid(psy[0], psy[j]);
        ll nejaky_biely = -1;
        // najdem aspon nejakeho bieleho
        For(i, 0, n) if (!pozicie.count(image(psy[i], mid_black))) nejaky_biely = i;

        if (solve(mid_black, mid_black)) return 0;  // mozem mat obe tyce na jednom mieste

        if (nejaky_biely != -1) {
            For(k, 1, n) {  // moj biely sa musi na nieco zobrazit,
                            // tak vyskusam vsetky moznosti
                pll mid_white = mid(psy[nejaky_biely], psy[k]);
                if (solve(mid_black, mid_white)) return 0;
            }
        }
    }
    cout << "zbirka dozbirkoval" << endl;
}
```

Existujú aj rôzne iné prístupy, ktoré sú založené na fixovaní nejakých iných psov, napríklad pravého dolného a ľavého horného. Tiež však v zásade ignorujú vopred zlé alebo už vyskúšané možnosti.

Sebik

8. Absolútne Pesťahovanie

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

Ako prvé dĺžím poďakovanie Martinovi "Medvědovi" Marešovi, ktorý preberá všetky zásluhy za tento príklad a jeho (fakt pekné) vzorové riešenie.

Čo so šírkou?

Pozrime sa ako prvé, ako vyriešime problém, ak sme obmedzení šírkou. Toto riešenie nám ukáže pozorovania, ktoré môžeme využiť neskôr.

Pozorovanie: Ak si zafixujeme výšku prvého poschodia, tak sa nám naň oplatí dať čo najviac psíkov, ktorí sa tam zmestia. Zvyšku problému sa potom môžeme venovať tak, že ignorujeme všetkých psíkov, ktorých sme dali na vrchné poschodie.

Prečo? Máme totiž už fixnú výšku to znamená, že ešte zostal kúsok šírky kam sme mohli psíka dať. Ak teda nejakého psíka môžeme dať na prvé poschodie, a dáme ho tam, riešenie to určite nepohorší.

Jedna zradná pasca, do ktorej sa tu riešiteľ môže chytiť je, že jednoducho si povie, že teda napchám všetky búdky, ktoré sa zmestia do šírky na prvé poschodie, tým ho vyhlásim za hotové a už sa venujem iba zvyšku.

Tento algoritmus by fungoval asi tak, že iteruje cez búdky, a ak sa aktuálny psík zmestí na aktuálne "rozrobené" poschodie, pridám ho naň. Ak sa však nezmestí, vyhlásim poschodie za hotové, pripočítam jeho výšku k priebežnému výsledku, a začnem znova s novým poschodím.

To však nefunguje, viď protipríklad:

```
3
sirka
5
4 2
1 8
4 9
```

Správna odpoveď by totiž bola 11, ale vyššie spomínaný "pažravý program" by vyhlásil, že riešenie je 17.

Problém je, že náš postup funguje pre fixnú výšku poschodia. Popísaný pažravý algoritmus však nastaví výšku tak aby sa využila celá šírka poschodia, čo vôbec nemusí viesť k optimálnemu riešeniu.

Dynamika

Táto pasca sa dá obísť tak, že si všimneme, čo sme vlastne pri pozorovaní zistili. Chceme si oddeliť časť psíkov, a pre zvyšok vyrátať najnižšiu výšku, do ktorej ich môžeme dať.

Myšlienka je taká, že si pre každého psíka spočítame, koľko výšky by potrebovali na ubytovanie všetci psíkovia po neho. Hlavne je potrebné si uvedomiť, že ak už takýto domček postavíme, tento psík bude posledný na poslednom poschodí, a teda značí nejakú hranicu nového poschodia.

Na tejto myšlienke postavme algoritmus. Pre každého psíka sa budeme snažiť vyrátať hodnotu MH (ako minimal height). $MH[i]$ bude značiť, koľko najmenej výšky musíme použiť, aby sme ubytovali prvých i psíkov. Toto nám dáva základ pre dynamické programovanie - budeme hodnoty MH rátať postupne z predošlých hodnôt.

Ako vyrátam stav v dynamike?

Všimnime si, že hodnota $MH[i]$ závisí iba na prvých $i-1$ psíkoch (podľa toho ako sme si to zadefinovali). Predstavme si teda, že už máme vypočítané optimálne hodnoty $MH[0]$ až $MH[i-1]$. Ako vypočítame hodnotu $MH[i]$?

Myšlienka bude takáto: Keďže psíkovia môžu značiť aj "hranicu", kde začína nové poschodie, tak si pre i-tého psíka jednoducho pozrieme, koľko výšky by spotrebovalo, keby predošlé poschodie končilo na nejakom z predošlých psíkov.

Pre každý index $j : j \leq i$ si "odsimulujeme" nasledovné: Chceli by sme, aby všetci psíkovia od psíka j po psíka i bývali spolu na poschodí. Môžu sa tam nezmestiť (poschodie nie je dosť široké), vtedy nerobíme nič.

Ale ak sa tam náhodou zmestia, tak by sme vedeli pre prvých i psíkov skonštruovať domček vysoký $MH[j-1]+maxh([j,i])$, pričom teda $maxh([j,i])$ značí maximálnu **výšku** psíka na intervale $[j,i]$. Toto značenie budem používať aj pre zvyšok príkladu.

Prečo? Z dynamiky vieme, že domček pre prvých $j-1$ psíkov bude vysoký $MH[j-1]$, a pridáme k tomu najnižšie možné poschodie, do ktorého sa určite zmestí aj najväčší psík na intervale $[j,i]$.

A tak máme jednu potenciálnu hodnotu pre číslo $MH[i]$. Ak toto spravíme pre každý index $j : j \leq i$, tak určite nájdeme optimálne riešenie pre $MH[i]$, pretože v optimálnom riešení musí byť nejaký psík prvým na tom istom poschodí, na ktorom je psík i posledný.

Časová zložitosť

Pre výpočet hodnoty $MH[i]$ musíme skontrolovať všetky $j : j \leq i$. Napríklad si vieme priebežne udržiavať najväčšieho psíka, ktorého sme doteraz videli, a taktiež šírku aktuálneho intervalu. Na vypočítanie jednej MH hodnoty teda stačí jeden prechod poľa psíkov – časová zložitosť $O(n)$. To nám dáva celkovú časovú zložitosť $O(n^2)$.

Obmedzenie na výšku pomocou šírky

Už vieme pre nejakú zadanú šírku x zistiť, aký najnižší panelák s takou výškou vieme postaviť. Zároveň však vieme pozorovať, že pre ľubovoľnú šírku väčšiu ako x bude najnižší panelák s takou výškou nižší (alebo rovnako veľký), a opačne, ľubovoľný užší panelák bude aspoň tak vysoký.

Vieme teda binárne vyhľadať najmenšiu šírku, pre ktorú bude najnižší panelák nižší ako naše obmedzenie (teda sa zmestí). Jednoducho si zvolíme nejakú strednú hodnotu (ako šírku), vypočítame pre ňu najnižšiu možnú výšku.

Ak sa zmestí pod naše obmedzenie, tak vieme, že všetky širšie domy nie sú optimálne široké. Ak sa naopak nezmestí pod naše obmedzenie, tak môžeme zahodiť všetky rovnaké a menšie hodnoty.

Toto nás teda dostane na časovú zložitosť $O(n^2 * \log H)$.

Pekná myšlienka do budúcna

Skúsme však spraviť riešenie, ktoré bude závisieť iba od počtu psíkov.

Pozorovanie: optimálna šírka bude určite určená nejakým súvislým úsekom psíkov, ktorí budú spolu na poschodí.

V optimálnom riešení musí byť aspoň na jednom poschodí úplne plno (teda nikde neostane voľné miesto). Ak by totiž ostalo na každom poschodí trošku miesta, môžeme jednoducho zmenšiť optimálne riešenie (čo je teda spor).

Všimnime si taktiež, že súvislých úsekov psíkov je omnoho menej ako H - iba n^2. Vieme si teda všetky tieto súvislé úseky psíkov vyrobiť, utriediť, a binárne vyhľadávať iba na týchto hodnotách.

Výroba týchto úsekov bude trvať $O(n^2)$, utriedenie $O(n^2 \log n^2)$, čo je asymptoticky $O(n^2 \log n)$. Toto nám časovú zložitosť zrazí až na $O(n^2 \log n)$, čo je vzorová časová zložitosť pre sady $1-5$.

Áno, cez testovač prešli aj časové zložitosti závislé od H, ale myšlienka určenia optimálneho riešenia pomocou súvislého úseku psíkov sa nám bude hodiť neskôr, a je fajn sa s ňou pohrať najprv takto.

Zrýchlenie funkcie na výpočtu najnižšej výšky

Časť algoritmu, ktorá nás aktuálne najviac obmedzuje, je funkcia výpočtu najmenšej výšky pre zadanú šírku. Poďme sa teda na ňu pozrieť a skúsiť ju zrýchliť.

Rátame pre každé i, $MH[i]$ ako minimálnu hodnotu z výrazov $MH[j-1] + maxh([j,i])$, pre každé $j \leq i$.

Pozorovanie 1: Pre zafixovaný index i, a zmenšujúci sa index j sa hodnota $maxh([j,i])$ zvyšuje, alebo ostáva rovnaká.

Jednoducho na čím dlhší interval (smerom doľava od i) sa pozeráme, maximum, ktoré sme videli sa určite nezmenší.

Pozorovanie 2: Ak sa práve pozeráme na interval po i, tak pre každé j platí, že $maxh([j,i])$ je rovné prvému prvku na indexe väčšom ako j takému, že doprava nemá žiadny väčší prvok.

To nejak intuitívne dáva zmysel, lebo tento prvok nemá nič väčšie doprava, a ak by existoval prvok na j,i, ktorý by bol väčší ako tento, tak by tiež nemal nič väčšie doprava, a bol by bližšie ku indexu j.

To dáva prvú myšlienku tohto algoritmu: urobiť si z psíkov postupne dátovú štruktúru, v ktorej bude postupnosť prvkov, ktoré pre daný interval (od 0 po nejaké i) nemajú smerom doprava žiadny vyšší prvok, pričom udržiavame ich poradie v pôvodnej postupnosti.

Napríklad pre postupnosť: $5, 4, 5, 8, 6, 7, 5, 2, 3$ by táto dátová štruktúra vyzerala asi takto: $8, 7, 5, 3$.

Všimnime si, že z definície je táto dátová štruktúra utriedená, lebo klesá, a tiež indexy stúpajú. Toto sa volá invariant a takáto dátová štruktúra sa vo všeobecnosti volá monotonický stack - skrátene monostack. Monotónniu sme už overili, vieme že klesá a neskôr ukážeme, že sa táto štruktúra naozaj chová ako stack

Pozorovanie 3: Všimnime si, že pre monostack skonštruovaný po index i a pre nejaký index j platí, že $maxh([j,i])$ je rovné prvku monostacku, ktorý sa vyskytol na najbližšom vyššom indexe od j. Totiž podľa vlastnosti monostacku vieme, že smerom doprava žiadnu väčšiu hodnotu nemá, no a doľava tiež nie (na intervale $[j,i]$), lebo inak by táto väčšia hodnota patrila do monostacku (nemala by doprava väčšiu hodnotu).

To nám však dáva kľúčovú vlastnosť pre zrýchlenie algoritmu: Ak máme dve po sebe idúce prvky monostacku, jeden s výskytom na indexe u a hodnotou t, druhým s výskytom na indexe $q, (q > u)$ s hodnotou m, tak pre interval $[u,q]$ existuje nejaké $j : u < j \leq q$, také že $maxh([j,i]) = m$. Slovne: vieme si prejdený interval rozdeliť na niekoľko podintervalov, pre ktoré je hodnota $maxh([j,i])$ určená najbližším prvkom v monostacku. Predstavte si tú hodnotu $maxh([j,i])$, ktorú ku pole dynamického programovania MH pridávame, ako nejaké schody.

Volajme prvok, ktorý je v monostacku ako "obmedzujúci" pre interval, kde určuje ten $maxh$.

Intervaláč ráta dynamiku

Konečne sa však dostávame ku kľúčovej myšlienke zrýchlenia: budeme si hodnoty $MH[j-1] + maxh([j,i])$ udržiavať v minimovom intervalovom strome. Dáva to zmysel - celý čas hľadáme iba nejaké minimum z nejakých prvkov.

Toto nám zaručí, že hodnotu $MH[i]$ vieme zistiť v čase $O(\log n)$, stačí nám totiž vedieť, (podobne ako pri pomalšom riešení), ktorý najpravejší psík sa nám ešte zmestí na to isté poschodie ako ten aktuálny (psík i). Označme tohoto najpravejšieho r.

Potom sa vieme jednoducho opýtať na minimum z intervalu $[r, i-1]$, čo bude optimálna hodnota $MH[i]$.

Ako však updatnuť celý intervaláč, aby sme rovnako rýchlo vedeli zistiť optimálnu hodnotu pre $MH[i+1]$?

Ako prechádzať medzi výpočtami?

Vyzbrojení našimi pozorovaniami, pridajme psíka i do monostacku. Stane sa jedna z dvoch vecí: 1) Psík je nižší ako predošlý psík v monostacku. Vtedy nemusíme robiť skoro nič, lebo pre všetky j sú $maxh([j,i+1])$ rovnaké ako pre $maxh([j,i])$. Jediná výnimka je interval $[i,i]$, kde predtým nič nebolo. Teraz je tam nový psík, ktorý je nový najvyšší na tomto intervale. Stačí teda updatnuť tento interval so psíkom, ktorý práve pribudol. Vyzeralo by to tak, že pôvodne by na i-tom indexe v intervaláči bola hodnota $MH[i]$ (lebo napravo od tejto pozície nebol žiaden psík), a teraz by sme tam uložili $MH[i] + vyska[i]$. 2) Psík je vyšší ako niektoré psíky v monostacku. To však znamená, že na niektorých intervaloch sa zvýši $maxh([j,i+1])$ oproti $maxh([j,i])$. My však presne podľa pozorovania vieme, ktoré intervaly to sú! Ak je psík vyšší ako nejaký prvok v monostacku,

my presne vieme o koľko, a pre ktorý interval bol tento prvok v monostacku obmedzujúci. Stačí teda každú hodnotu v tomto intervale navýšiť o rozdiel medzi novým psíkom a starým (čo pomocou lazy intervaláču hravo zvládneme v $O(\log n)$).

Rekapitulácia algoritmu

Udržujeme si monostack, kde sú práve tí psíkovia, ktorí nemajú žiadneho väčšieho psíka napravo pre aktuálny interval: 0 až nejaké i.

Každý z týchto psíkov je obmedzujúcim pre nejaký interval hodnôt - to znamená, že pre ľubovoľné číslo j patriace do tohoto intervalu bude $maxh([j, i]) =$ veľkosť obmedzujúceho psíka.

V intervalovom strome si na pozícii $p \leq i$ pamätáme nasledovné: $MH[p] + maxh([p, i])$. $MH[i]$ vypočítame ako minimum z nejakého intervalu (toho intervalu, ktorý sa ešte zmestí do danej šírky).

Ak potom chceme posunúť i, tak musíme updatnuť monostack pridaním nového psíka. Tento psík sa stane novým obmedzujúcim psíkom pre každý interval, ktorého aktuálny obmedzujúci psík je menší ako tento.

Druhý sčítanec ($maxh([p, i])$) teda pre tieto intervaly lazy updatneme o rozdiel medzi aktuálnym a minulým obmedzujúcim psíkom.

Pfuh. Akú má toto celé časovú zložitosť?

Pre každý prvok urobíme operáciu zistenie minima z nejakého intervalu na intervaláči, čo nás stojí $O(\log n)$.

Tiež však robíme nejaké range update operácie na našom intervaláči. Koľko ich bude?

Jednu musíme spraviť práve vtedy, keď pridávame nového psíka do monostacku a keď odstraňujeme nového psíka z monostacku. A keďže každého psíka môžeme pridať a odstrániť len raz, dokopy to spravíme najviac $2n$ krát, teda nás to celé bude stáť $O(n \log n)$.

Myšlienka z minula

No dobre. Tak teraz vieme pomocou tejto novej vylepšenej funkcie v čase $O(n \log n)$ nájsť najnižší domček s obmedzenou šírkou.

Čo ak sme však obmedzení výškou?

Vždy vieme spraviť rovnakú vec ako minule a binárne vyhľadať šírku, pre ktorú je najnižší domček s touto šírkou nižší ako obmedzenie.

Teda, vieme jednoducho vyriešiť príklad v $O(n \log n \log H)$.

To však nie je optimálna zložitosť, závislá iba od n. Keď sa však pokúsime aplikovať myšlienku s generovaním a utriedením všetkých súvislých intervalov (aby sme dostali plný počet za teoretické), zistíme, že generovanie týchto intervalov nám pôvodne trvalo $O(n^2 \log n)$, takže by sme si novou zrýchlenou funkciou vlastne vôbec nepomohli.

My však vieme tento postup ešte stále zrýchliť.

Priemerne dobré binárne vyhľadávanie

Hlavná myšlienka tejto optimalizácie je nasledovná: pri binárnom vyhľadávaní je síce optimálne vždy vyberať ako pivot prvok "v strede", ale aj pri výbere rovnomerne náhodného prvku ako pivota je časová zložitosť priemerne logaritmická.

Keď na utriedenom poli binárne vyhľadávame, stačí, že vyberieme náhodný prvok a podľa neho zahodíme alebo si ponecháme zvyšné prvky. Nerozdelíme vždy pole na polovicu, ale priemer počtu "zahodených" prvkov ostane rovnaký.

Prečo nás toto však zaujíma? My si totiž súvislé úseky psíkov nemusíme generovať - my si iba môžeme pamätať, že existujú.

Pre každého psíka i si pamätajme nasledovný interval: ak by poschodie s psíkom i na začiatku určovalo šírku paneláku, psíky v tomto intervale by mohli byť tie posledné na tomto poschodí.

Pre každého psíka i si jednoducho pamätáme množinu psíkov, ktorí môžu byť pravou stranou súvislého úseku psíkov, ktorí určuje šírku paneláku. Táto množina bude súvislý úsek.

A keďže je táto množina súvislý úsek, môžeme si ju pamätať pomocou dvoch čísel - pravého a ľavého konca. Volajme tento interval množina pravých (Kubov) koncov.

Keď si takto kompaktne pamätáme všetky súvislé úseky, ktoré ešte môžu určovať šírku paneláku, rovnomerne náhodne z nich jeden vyberieme.

Zistíme, či je panelák takejto šírky možný alebo nie. Pre každého psíka potom upravíme množinu pravých koncov podľa výsledku. Možno na prvý pohľad nie je jasné, ako by sme to mali robiť, ale v skutočnosti stačí pole psíkov prejsť pomocou dvoch bežcov. A môžeme opakovať.

Akú má celé toto časovú zložitosť? Tak funkcia "zisti pre zadanú šírku najnižšiu výšku paneláku" nás stojí $O(n \log n)$, vybrať rovnomerne náhodný úsek zvládneme tiež v $O(n)$, a podľa výsledku updatnuť množinu

pravých koncov pre každého psíka nás tiež stojí $O(n)$. Dokopy túto celú procedúru budeme volať v priemere $O(\log n)$ krát, čo nám dá finálnu časovú zložitosť $O(n \log^2 n)$.

Listing programu (C++)

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <random>

using namespace std;
vector<long long> pref_sums;

long long psum(int l, int r) {
    return pref_sums[r+1] - pref_sums[l];
}

mt19937_64 mt;
long long mt_range(long long lo, long long hi) { return lo + mt() % (hi - lo + 1); }

struct Intervalac
{
    public:
    vector<long long> tree;
    vector<long long> lazy_update;
    int offset;

    Intervalac(vector<long long> pole){
        int next2 = 1;
        while (next2 < pole.size())
        {
            next2 *=2;
        }

        offset = next2;
        next2 *=2;
        tree.resize(next2);

        for (int i = 0; i < pole.size(); i++)
        {
            tree[i+offset] = pole[i];
        }

        for (int i = offset-1; i > 0; i--)
        {
            tree[i] = min(tree[i*2], tree[i*2+1]);
        }
        lazy_update.resize(next2);
    }

    long long mr(int l, int r, int dl, int dr, int vtx) {
        if (lazy_update[vtx] != 0)
        {
```

```
            tree[vtx] += lazy_update[vtx];
            if (vtx*2 < tree.size())
            {
                lazy_update[vtx*2] += lazy_update[vtx];
                lazy_update[vtx*2+1] += lazy_update[vtx];
            }
            lazy_update[vtx] = 0;
        }

        if (dl <= l && r <= dr)
        {
            return tree[vtx];
        }

        if (dr < l  r < dl)
        {
            return 1e17;
        }

        int mid = (l+r)/2;
        long long out = min(
            mr(l,mid,dl,dr,vtx*2),
            mr(mid+1,r,dl,dr,vtx*2+1)
        );

        if (vtx*2 < tree.size())
        {
            tree[vtx] = min(tree[vtx*2], tree[vtx*2+1]);
        }
        return out;
    }

    long long min_range(int l, int r) {
        return mr(0, offset-1, l, r, 1);
    }

    void ur(int l, int r, int dl, int dr, int vtx, long long val) {
        if (dl <= l && r <= dr)
        {
            lazy_update[vtx] += val;
        }

        if (lazy_update[vtx] != 0)
        {
            tree[vtx] += lazy_update[vtx];
            if (vtx*2 < tree.size())
            {
                lazy_update[vtx*2] += lazy_update[vtx];
                lazy_update[vtx*2+1] += lazy_update[vtx];
            }
            lazy_update[vtx] = 0;
        }

```

```
102         if (dl <= l && r <= dr)
103         {
104             return;
105         }
106 
107         if (dr < l  r < dl)
108         {
109             return;
110         }
111 
112         int mid = (l+r)/2;
113 
114         ur(l,mid,dl,dr,vtx*2, val);
115         ur(mid+1,r,dl,dr,vtx*2+1, val);
116 
117         if (vtx*2 < tree.size())
118         {
119             tree[vtx] = min(tree[vtx*2], tree[vtx*2+1]);
120         }
121     }
122 
123     void update_range(int l, int r, long long val) {
124         ur(0, offset-1, l, r, 1, val);
125     }
126 
127 };
128 
129 
130 long long min_height(vector <pair<long long, long long>> &psiky, long long wid) {
131     int n = psiky.size();
132     vector<long long> init(n+1,0);
133 
134     int l = 0; int r = 0;
135     vector<pair<long long, pair<int,int>>> monostack;
136     monostack.push_back({0,{0,0}});
137     Intervalac intervalac = Intervalac(init);
138     long long curr_wid = 0;
139 
140     for (int i = 0; i < n; i++)
141     {
142         r++;
143         long long curr_top = psiky[r-1].second;
144         curr_wid += psiky[r-1].first;
145 
146         int interval = r;
147         int x = monostack.size();
148         for (int g = 0; g < x; g++)
149         {
150             long long porovnavam = monostack.back().first;
151             if (porovnavam < curr_top)
152             {
153                 long long diff = curr_top - porovnavam;
```

```
154                 intervalac.update_range(monostack.back().second.first,
    ↪ monostack.back().second.second, diff);
155                 interval = monostack.back().second.first;
156                 monostack.pop_back();
157             } else
158             {
159                 break;
160             }
161         }
162         monostack.push_back({curr_top,{interval,r-1}});
163         monostack.push_back({0,{r,r}});
164 
165         while (curr_wid > wid)
166         {
167             curr_wid-= psiky[l].first;
168             l++;
169         }
170 
171         intervalac.update_range(r,r, intervalac.min_range(l, r-1));
172     }
173 
174     return intervalac.tree[intervalac.offset+n];
175 }
176 
177 void update_intervals(vector<pair<long long, long long>> &psiky, vector<pair<int,int>> &konce, bool
    ↪ vacsie, long long sol) {
178     int n = psiky.size();
179 
180     int l = 0;
181     bool found = false;
182     for (int i = 0; i < n; i++)
183     {
184         long long wid = psum(l,i);
185         if (vacsie)
186         {
187             if (wid <= sol)
188             {
189                 konce[l] = {max(konce[l].first, i+1), konce[l].second};
190             } else
191             {
192                 while (psum(l,i) > sol)
193                 {
194                     konce[l] = {max(konce[l].first, i), konce[l].second};
195                     l++;
196                 }
197                 konce[l] = {max(konce[l].first, i+1), konce[l].second};
198             }
199 
200             if (i == n-1)
201             {
202                 while (l < i)
203                 {
204                     l++;
```

```
                    konce[l] = {konce[l].second, konce[l].second};
                }
            }

        } else
        {
            if (found)
            {
                while (psum(l,i) >= sol)
                {
                    konce[l] = {konce[l].first, min(i, konce[l].second)};
                    l++;
                }
            } else
            {
                while (true)
                {
                    long long widd = psum(l,i);
                    if (widd == sol)
                    {
                        found = true;
                        break;
                    } else if (widd > sol)
                    {
                        konce[l] = {konce[l].first, min(i, konce[l].second)};
                        l++;
                    } else
                    {
                        break;
                    }
                }
            }
        }
    }
}

int main() {
    int n; cin >> n;
    string typ; cin >> typ;
    long long obmedzenie; cin >> obmedzenie;
    vector <pair<long long, long long>> psiky(n);
    for (int i = 0; i < n; i++)
    {
        cin >> psiky[i].first;
        cin >> psiky[i].second;
    }


    if (typ == "sirka")
    {
        cout << min_height(psiky, obmedzenie) << endl;
        return 0;
    }
```

```
258
259     pref_sums.resize(n+1,0);
260     long long mini = 0;
261     for (int i = 0; i < n; i++)
262     {
263         pref_sums[i+1] = pref_sums[i]+psiky[i].first;
264         mini = max(mini, psiky[i].first);
265     }
266
267     vector<pair<int,int>> jakub_konce(n);
268
269     for (int i = 0; i < n; i++)
270     {
271         jakub_konce[i] = {i,n};
272     }
273
274     update_intervals(psiky,jakub_konce,true, mini-1);
275     vector<int> pmoz(n,0);
276     while (true)
277     {
278         long long moznosti = 0;
279         pmoz.resize(n,0);
280
281         for (int i = 0; i < n; i++)
282         {
283             pmoz[i] = jakub_konce[i].second - jakub_konce[i].first;
284             if (pmoz[i] < 0)
285             {
286                 pmoz[i] = 0;
287             }
288             moznosti += pmoz[i];
289
290         }
291
292         if (moznosti == 1)
293         {
294             break;
295         }
296
297         long long nahodna = mt_range(0,moznosti-1);
298
299         long long wid = 0;
300         for (int i = 0; i < n; i++)
301         {
302             if (nahodna - pmoz[i] < 0)
303             {
304                 wid = psum(i,jakub_konce[i].first+nahodna);
305                 break;
306             }
307             nahodna -= pmoz[i];
308         }
309
310         long long sol = min_height(psiky, wid);
```

```
311
312         if (sol > obmedzenie)
313         {
314             update_intervals(psiky, jakub_konce, true, wid);
315         } else
316         {
317             update_intervals(psiky, jakub_konce, false, wid);
318         }
319     }
320
321     for (int i = 0; i < n; i++)
322     {
323         if (pmoz[i])
324         {
325             cout << psum(i,jakub_konce[i].first) << endl;
326             return 0;
327         }
328     }
329 }
```