Korešpondenčný seminár z programovania
XLI. ročník, 2023/24
FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 2. kola letnej časti

Tomi

1. Divný sen

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

Hľadáme nejaký neprázdny súvislý podúsek vstupu i až j, kde o hodnotách $v_{i\ldots j}$ platí, že majú súčet $v_i + \ldots + v_j = s$, a o hodnotách $k_{i\ldots j}$, že prvá je 1 (les) a všetky ostatné sú 0 (pole).

Nemá zmysel skúšať všetkých $n \cdot (n+1)/2$ možností – každý možný začiatok a koniec. Keď sa pozrieme na ľubovoľné miesto kde by sme mohli skončiť, zjavne jediný začiatok čo prichádza do úvahy je najbližší predošlý les. A ak ešte žiaden les nebol, toto miesto nemôže byť koniec. Takže potenciálnych začiatkov+koncov, ktoré by sa zišlo vyskúšať, je maximálne n.

Keby sme od každého konca išli naspäť, hľadali najbližší začiatok a spravili zakaždým nový súčet všetkých prvkov medzi nimi, tiež by sme sa zbytočne namakali.

Radšej proste prejdeme vstupom a budeme si pamätať tri veci: priebežný súčet výšok v_i od posledného lesa, počet možností čo sme zatiaľ objavili, a či už sme vôbec videli aspoň jeden les. Každý prvok vybavíme v konštantnom čase. Vždy, keď priebežný súčet výšok dosiahne hodnotu s, a už sme videli les, zvýšime počet možností o jedna.

Toto riešenie má časovú zložitosť $O(n)$, a pamäťovú zložitosť $O(1)$, lebo si ani len nemusíme vopred načítať všetky riadky vstupu do poľa. Môžeme každý riadok spracovať okamžite po tom, čo ho načítame.

Listing programu (Python)

```python
s = int(input())
n = int(input())
vyska = 0
videl_les = False
moznosti = 0
for i in range(n):
  v, k = map(int, input().split())
  if k == 1:
    vyska = 0
    videl_les = True
  vyska += v
  if vyska == s and videl_les:
    moznosti += 1
print(moznosti)
```

Listing programu (C++)

```cpp
#include <iostream>
using namespace std;
using ll = long long;

int main() {
  ll s, n;
  cin >> s >> n;
  ll vyska = 0;
  bool videl_les = false;
```

```
  ll moznosti = 0;
  for (ll i = 0; i < n; i++) {
    ll v, k;
    cin >> v >> k;
    if (k == 1) {
      vyska = 0;
      videl_les = true;
    }
    vyska += v;
    if (vyska == s && videl_les) {
      moznosti++;
    }
  }
  cout << moznosti << endl;
}
```

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

2. Incident na diaľnici

Jedno možné riešenie

Pozrime sa na to, čo sa stane, ak kladné čísla dáme do kladného kamiónu, záporné do záporného a nuly do nulového. Keďže vieme, že riešenie existuje, tak v zápornom kamióne bude aspoň jedno číslo, keďže na to, aby súčin bol záporný potrebujeme aspoň jedno záporné číslo. Taktiež v nulovom kamióne bude aspoň jedna nula. V kladnom kamióne teoreticky nemusí byť ani jedno číslo. V prípade, že tam nie je ani jedno, tak môžeme presunúť dve ľubovoľné záporné čísla zo záporného kamióna do kladného. Tým sa nám súčin čísel v zápornom kamióne nezmení, ale kladný kamión už bude obsahovať nejaké čísla (ktorých súčet je kladný), takže náš problém je vyriešený. Ešte môže nastať situácia, že v zápornom kamióne je párny počet záporných čísel. V tom prípade môžeme presunúť jedno záporné číslo do nulového kamiónu, aby súčin v zápornom bol záporný a v nulovom stále ostane nulový. Týmto spôsobom sme vyriešili úlohu.

Ďalšie riešenie

Ďalšie riešenie bolo založené na tom, že do záporného kamiónu dáme prvé záporné číslo. Do kladného kamiónu dáme 1 kladné číslo (alebo 2 záporné ak kladné neexistuje) a do nulového kamiónu zvyšné čísla. Takýto prístup tiež bez problémov vyriešil úlohu.

Zložitosti

V oboch prípadoch je pamäťová zložitosť riešenia $O(n)$, keďže si náš program musí pamätať všetky čísla. Je to preto, lebo čísla musíme ich na konci vypísať a nevieme to urobiť tak, že ich vypisujeme postupne, keďže musíme napísať, koľko ich dokopy v ktorom kamióne bude. Časová zložitosť je tiež $O(n)$, keďže prechádzame čísla po jednom a pre každé z nich urobíme iba konštantné množstvo operácií (jednoducho ho iba zaradíme do niektorej množiny).

Listing programu (Python)

```
N = int(input())
A = list(map(int, input().split()))

neg, zero, pos = [], [], []

for i in range(N):
    if (A[i] < 0):
        neg.append(A[i])
    elif (A[i] == 0):
        zero.append(A[i])
```

```
    else:
        pos.append(A[i])

if (len(pos) == 0):
    pos.append(neg[-1])
    neg.pop(-1)
    pos.append(neg[-1])
    neg.pop(-1)

if (len(neg)%2 == 0):
    zero.append(neg[-1])
    neg.pop()

print(len(neg), " ".join(map(str, neg)))
print(len(pos), " ".join(map(str, pos)))
print(len(zero), " ".join(map(str, zero)))
```

Listing programu (C++)

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

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

    int N;
    cin >> N;
    vector<int> neg;
    vector<int> zero;
    vector<int> pos;

    for (int i = 0; i < N; i++)
    {
        int a; cin >> a;
        if (a < 0) neg.push_back(a);
        else if (a == 0) zero.push_back(a);
        else pos.push_back(a);
    }

    if (pos.size() == 0)
    {
        pos.push_back(neg[neg.size()-1]);
        neg.pop_back();
        pos.push_back(neg[neg.size()-1]);
        neg.pop_back();
    }

    if (neg.size()%2 == 0)
    {
        zero.push_back(neg[neg.size()-1]);
        neg.pop_back();
```

```
    }

    cout << neg.size();
    for (int i = 0; i < neg.size(); i++) cout << " " << neg[i];
    cout << "\n";

    cout << pos.size();
    for (int i = 0; i < pos.size(); i++) cout << " " << pos[i];
    cout << "\n";

    cout << zero.size();
    for (int i = 0; i < zero.size(); i++) cout << " " << zero[i];
    cout << "\n";
}
```

Marcel

3. Aspoň jedna cesta von

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

Vašou úlohou bolo zistiť, či vieme orientovať cesty tak, aby z každého mesta vychádzala aspoň jedna cesta. Tí z vás, ktorí už počuli o grafoch[1], si určite všimli, že túto úlohu vieme reprezetovať pomocou grafu, kde mestá sú vrcholy a cesty sú hrany. V ďalšom texte teda budeme pre jednoduchosť používať grafovú terminológiu. V grafovej terminológii teda zadanie úlohy znie, že máme zistiť, či vieme orientovať hrany v neorientovam grafe tak, aby z každého vrcholu vychádzala aspoň jedna hrana.

Pozorovania

Dôležitým pozorovaním pri riešení tejto úlohy je, že si môžeme graf rozdeliť na komponenty súvislosti a každý komponent súvislosti skúmať samostatne. Je to preto, lebo to, či vieme správne orientovať hrany v jednom komponente nijako neovplyvňuje ostatné komponenty. Ak aspoň v jednom komponente nevieme orientovať hrany podľa zadania, tak potom je odpoveď `nie`. Vďaka tomuto sa môžeme v ďalšom texte zoberať iba grafmi, ktoré sú tvorené presne jedným komponentom súvislosti.

Začnime riešenie úlohy tak, že si napíšeme niektoré prípady, kedy učite nevieme hrany orientovať podľa zadania. Ak máme jeden komponent súvislosti, ktorý sa skladá z jediného samostatného vrcholu, do ktorého nejdú žiadne hrany, tak v ňom určite nevieme orientovať hrany podľa zadania (tento vrchol nemá žiadnu hranu, takže z neho nemá aká vychádzať). Rovnako, ak napríklad máme komponent, ktorý sa skladá presne z dvoch vrcholov spojených hranou, z jedného z nich hrana vychádzať nemôže.

Ak si takto rozoberieme ešte niekoľko prípadov zistíme, že ak je komponent grafu strom, tak v ňom nevieme orientovať hrany podľa zadania. Predstavme si totiž ľubovoľný zakorenený strom[2]. Ak v ňom orientujeme hrany zvrchu dole (od koreňa), tak žiadna hrana nebude vychádzať z listov. Ak orientujeme hrany opačne (od listov), tak žiadna hrana nebude vychádzať z koreňa. A samozrejme, ak orientujeme hrany hocijako inak, tak niekde v strede grafu nám vznikne vrchol/vrcholy, z ktorých žiadna hrana nevychádza.

Hlavná myšlienka

Už vieme, že ak je komponent grafu strom, tak v ňom nevieme zorientovať hrany podľa zadania. Otázka je, že či vo všetkých ostatných druhoch grafov vieme správne zorientovať hrany. Odpoveď je áno. Stačí ak doplníme do stromu na akékoľvek miesto jednu hranu. Potom môžeme strom zakoreniť v jednom z týchto vrcholov, kam sme pridali hranu. Teraz sa dajú v strome všetky hrany orientovať z listov smerom hore. To znamená, že ak by náš graf bol strom (bez pridanej hrany), tak jediný vrchol z ktorého by nevychádzala hrana by bol koreň. Do koreňa vedie ale ešte jedna hrana naviac (tá, ktorá je v grafe naviac oproti stromu). Túto hranu vieme orientovať smerom z koreňa. Z vrcholu kam hrana vedie už nejaká hrana vychádza (ide smerom hore do jeho rodiča), takže tým nič nepokazíme.

Postup sa teda dá zhrnúť takto: - rozdelíme si graf na komponenty súvislosti - prejdeme všetky komponenty súvislosti a ak každý obsahuje oproti stromu aspoň jednu hranu naviac, tak odpovedáme `ano` - inak odpovedáme `nie`

[1] https://www.ksp.sk/kucharka/grafy_uvod/
[2] https://www.ksp.sk/kucharka/grafy_uvod/#wiki-toc-zakorenene-stromy

Ako nájsť "hrany naviac"?

Teraz sa pozrime na to, ako zistiť, či graf obsahuje nejakú "hranu naviac" oproti stromu. Predstavme si napríklad, ako prehľadáva tento strom prehľadávanie do hĺbky[3]. Toto prehľadávanie postupne prechádza všetky vrcholy a v každom vrchole sa pozrie na všetkých jeho susedov. Týchto susedov môžeme rozdeliť do troch kategórií. Susedný vrchol môže byť: - ešte nenavštívený - ten vrchol z ktorého sme do aktuálneho vrcholu prišli (a teda je navštívený) - hociktorý iný už navštívený vrchol

Ak si predstavíme ako postupne prechádza prehľadávanie do hĺbky *stromom*, tak každú hranu v strome vidíme dva krát: raz, keď po nej ideme do nového vrcholu a druhý krát, keď by sme sa po nej vrátili do vrcholu z ktorého sme do aktuálneho vrcholu prišli (vedie do rodiča aktuálneho vrcholu). Tieto dva prípady zodpovedajú prvým dvom kategóriám susedných vrcholov.

Ak máme v grafe nejakú hranu naviac oproti stromu, tak táto hrana nevedie do nového vrcholu a ani nevedie do rodiča aktuálneho vrcholu (keďže podľa zadania môžeme predpokladať, že v grafe neexistujú násobné hrany).

Hrany naviac teda vieme nájsť pomerne jednoducho: stačí spustiť prehľadávanie do hĺbky z ľubovoľného vrcholu. V každom vrchole si potrebujeme pamätať z ktorého vrcholu sme sa do aktuálneho vrcholu dostali (nášho rodiča). Ak nájdeme susedný vrchol, ktorý je už navštívený a nie je to náš rodič, tak sme našli "hranu naviac" a vieme, že v tomto komponente vieme zorientovať hrany podľa zadania.

Zhrnutie a zložitosti

Ako sme už písali vyššie, tak postupne prechádzame všetky komponenty súvislosti grafu a o každom sa rozhodujeme samostatne. Ak každý komponent súvislosti obsahuje oproti stromu aspoň jednu hranu naviac, tak odpovedáme `ano`, inak odpovedáme `nie`. To, že či komponent obsahuje nejakú hranu naviac oproti stromu zisťujeme prehľadávaním do hĺbky.

Tento program si okrem pár premenných potrebuje pamätať vstupný graf, ktorý obsahuje n vrcholov a m hrán, takže pamäťová zložitosť je $O(n+m)$. Čo sa týka časovej zložitosti, tak na grafe niekoľko krát (pre každý komponent jeden krát) spustíme prehľadávanie do hĺbky. Je dobré si uvedomiť, že nezáleží na tom, koľko komponentov súvislosti vlastne graf obsahuje. Na určenie časovej zložitosti stačí, že vieme, že dokopy vo všetkých prehľadávaniach prejdeme celý graf (budeme v každom vrchole presne raz a na každú hranu sa pozrieme konštantný počet krát). Časová zložitosť je teda $O(n+m)$.

Listing programu (C++)

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

vector<vector<ll> > G;
vector<bool> visited;

bool dfs(ll v, ll parent){
        bool answer = false;
        visited[v] = true;
        for(ll i=0;i<G[v].size();i++){
                ll u = G[v][i];
                if(u!=parent && visited[u])
                        answer = true;
                if(!visited[u]){
                        answer = dfs(u, v)  answer;
                }
        }
        return answer;
}

int main() {
```

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

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

    ll n, m;
    cin >> n >> m;
        G.resize(n);
        visited.resize(n, false);

    for (ll i=0;i<m;i++){
                ll a, b;
        cin >> a >> b;
        G[a].push_back(b);
        G[b].push_back(a);
    }

    for(ll i = 0; i<n; i++){
            if(!visited[i]){
                    if(!dfs(i, -1)){
                            cout<<"nie"<<endl;
                            return 0;
                        }
            }
    }
        cout<<"ano"<<endl;
    return 0;
}
```

Listing programu (Python)

```
from sys import setrecursionlimit

setrecursionlimit(1000000)

def dfs(v, parent):
    answer = False
    visited[v] = True
    for u in G[v]:
        if u!=parent and visited[u]:
           answer = True
        if not visited[u]:
            answer = dfs(u, v) or answer
    return answer

def solve():
    for i in range(n):
        if not visited[i]:
            result = dfs(i, -1)
            if not result:
                print("nie")
                return
    print("ano")

n, m = map(int, input().split())
```

```
G = [[] for _ in range(n)]
visited = [False for _ in range(n)]
for i in range(m):
    a, b = map(int, input().split())
    G[a].append(b)
    G[b].append(a)

solve()
```

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

4. Len tak ďalej Adam

Offline riešenie

Nejaké body ste dostali, ak ste úlohu vyriešili offline, avšak tento postup je aj programátorsky skoro náročnejší ako postup na plný počet bodov, preto veľmi odporúčam rovno preskočiť na ďalší odstavec, kde sa dozviete ako túto úlohu riešiť online.

Úlohu budeme riešiť odsimulovaním si Adamovho cvičenia. Najskôr načítame všetky čísla zo vstupu (váhy činiek, na ktorých boli zavesené činky s ktorými Adam cvičil). Na to, aby sme odsimulovali Adamove cvičenie, si musíme čísla na vstupe uložiť v opačnom poradí v akom sme ich dostávali napríklad do poľa C. Takto budeme mať na mieste $C[0]$ uloženú váhu činky, na ktorej bola zavesená činka, s ktorou cvičil Adam ako prvou.

Keď už máme vhodne uložený vstup, tak sa môžeme pustiť do simulovania Adamovho cvičenia. Keď chceme zistiť, s ktorou činkou cvičil Adam ako i-tou, tak potrebujeme vedieť, na ktorých činkách nie je nič zavesené a ktoré sú už zobraté. Činky, na ktorých nie je nič zavesené sú tie, ktoré sa už nevyskytujú na vstupe. Činky, s ktorými už Adam cvičil si budeme ukladať do poľa. Takže, keď chceme zistiť, s ktorou činkom cvičil Adam ako i-tou, tak si vytvorím pole A veľkosti $n+1$. Na j-tom indexe bude 1 vtedy, keď je možné danú činku použiť. Potom si do poľa pridáme 0 na miesta, ktorých váhy už boli zobrané. Potom prejdeme C od $i-1$-tého indexu a zaznačíme si do príslušných miest v A nuly. Nakoniec nám už iba stačí nájsť najmenšie miesto, kde je 1 a pridať túto váhu do už použitých váh.

Toto riešenie je dosť dobré na to, aby prešlo prvými dvomi sadami na vstupe. Riešenie, pri ktorých potrebujeme poznať celý vstup vzhľadom na charakter úlohy môžu dostať iba polovicu bodov, takže sa nimi už nebudeme viacej zaoberať.

Poďme na to online

Postupne spracúvame vstup, teda určujeme v akom poradí idú činky, ale odzadu. Keď sa teda pozeráme na i-te číslo, vieme už posledných $i-1$ činiek. Nech je i-te číslo na vstupe x. Môžu nastať 2 prípady:

1) Adam ešte necvičil s činoku x. To znamená, že Adam s ňou musel cvičiť ako i-tou od konca. Neskôr s ňou nemohol cvičiť, pretože tie už poznáme. Skôr s ňou nemohol taktiež cvičiť, pretože na nej je neskôr niečo zavesené.
2) Adam už cvičil s činkou x. V tomto prípade Adam cvičil s najťažšou činkou, s ktorou necvičil neskôr. Označme si túto činku p.Musíme sa však zamyslieť nad tým, či táto činka už nebola na vstupe. To by znamenalo, že s ňou Adam nemohol cvičiť ako i-tou od konca, pretože na nej neskôr bolo niečo zavesené. Ideme to dokázať sporom. Vieme, že Adam ešte necvičil s činkou p. Pozrime sa na prvý výskyt činky p na vstupe. Vieme, že pred tým s ňou necvičil. To znamená, že sa jedná o prípad číslo 1 a tým pádom s ňou Adam vtedy cvičil. To je ale v spore s predpokladom a to znamená, že činka p ešte nebola na vstupe a Adam s ňou môže cvičiť.

To znamená, že každé číslo zo vstupu nám jednoznačne povie, s ktorou činkou Adam cvičil ako i-tou od konca. Na konci zostane jedna činka nepoužitá (keďže na vstupe je $n-1$ činiek - s ňou cvičil Adam ako prvou).

Teraz to nakódime

Spravíme si pole U veľkosti $n+1$. V tomto poli bude na políčku j jednotka, ak už Adam cvičil s činkou s váhou j. Pozrime sa na i-te číslo na vstupe. Nech je to číslo x. Ak Adam ešte necvičil s činkou x, tak s ňou musel cvičiť ako i-tou a vypíšeme ju. Do poľa U si zapíšeme na pozíciu x jednotku. Ak už Adam cvičil s činkou x, tak na vstup vypíšeme najťažšiu činku, s ktorou ešte necvičil. Túto nájdeme tak, že budeme prechádzať pole U od najväčších indexov k najmenším a budeme hľadať 0. Povedzme, že ju nájdeme na indexe u. Vypíšeme u, zapíšeme do poľa na u-ty index 1. Keď neskôr hľadamé najťažšiu činku, stačí nám pokračovať od indexu u.

Listing programu (Python)

```python
#!/usr/bin/python3

n = int(input())

uz = [0] * (n+1)
t = n
for i in range(n-1):
    x = int(input())
    if uz[x] == 1:
        while uz[t] == 1:
            t -= 1
        uz[t] = 1
        print(t, flush=True)
    else:
        uz[x] = 1
        print(x, flush=True)
while uz[t] == 1:
    t -= 1
uz[t] = 1
print(t)
```

Listing programu (C++)

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

using namespace std;

int main()
{
    int n;
    cin >> n;
    int t = n;
    vector<int> uz(n+1, 0);
    for(int i = 1; i < n; i++)
    {
        int x;
        cin >> x;
        if(uz[x])
        {
            while(uz[t]) t--;
            uz[t] = 1;
            cout << t << endl;
            continue;
        }
        uz[x] = 1;
        cout << x << endl;
    }
    while(uz[t]) t--;
    cout << t << endl;
}
```

Johanko

5. Neskutočná zápcha

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

Začneme okľukou

Väčšinou riešime úlohy od prvej sady postupne vyššie. Teraz si však vyriešime najskôr tretiu sadu, lebo zvyšné na seba nadväzujú.

V tejto sade sú všetky autobusy rovnako veľké, teda počet ľudí, ktorí prejdú na jednu zelenú križovatkou je vždy rovnaký – je jedno, ktorý autobus je momentálne prvý, lebo sú všetky rovnaké. Môžme teda napríklad odsimulovať prvú zelenú (viď bruteforce nižšie) a to následne vynásobiť k, ale v skutočnosti si to môžme aj jednoducho vypočítať.

Označme si počet ľudí v autobuse a (toto bude rovnaké pre všetky autobusy), potom na každú zelenú prejde križovatkou $\lfloor r/a \rfloor$ autobusov, avšak križovatkou môže na každú zelenú autobus prejsť len raz, čize v skutočnosti $\min(\lfloor r/a \rfloor, n)$ autobusov. V každom autobuse je a ľudí, teda za jednu zelenú prejde $\min(\lfloor r/a \rfloor, n) \times a$ ľudí. Zeleých je k, čiže spolu $\min(\lfloor r/a \rfloor, n) \times a \times k$ ľudí.

Časová aj pamäťová zložitosť tohoto programu bude $O(1)$.

Bruteforce

Teraz ideme riešiť prvú sadu. Keďže je zelených aj autobusov málo, môžme si dovoliť všetky zelené simulovať. Každú zelenú odsimulujeme presne tak, ako by to robil semafór na križovatke. Sčítavame si, koľko ľudí sme zatiaľ križovatkou pustili, a pokiaľ môžme pustiť aj ďalší autobus, pustíme ho. Toto riešenie bude mať zložitosť $O(kn)$ – pre každú zelenú môžeme pustiť až n autobusov.

Jedna možnosť, ako to implementovať, je použiť dátovú štruktúru rad / fronta (C++ `std::queue`), prípadne obojstranná fronta (C++ `std::deque`, Python `collections.deque`), a naozaj autobusy vyberať spredu a vkladať dozadu.

Druhá možnosť je mať autobusy celý čas uložené v pôvodnom poradí, a pamätať si stav len ako jedno číslo: index autobusu, ktorý momentálne stojí ako prvý pred semafórom. Obe možnosti sú rovnako dobré, ale tento pohľad sa nám bude hodiť v ďalších sadách.

Podľa implementácie mohlo toto riešenie zbehnúť na 2 alebo 4 body.

Načo toľko počítame?

Pri predchádzajúcom riešení sme si mohli všimnúť dôležité pozorovanie. A to že stav po zelenej závisí iba od stavu pred zelenou. Ak sme sa počas deja viackrát nachádzali v nejakom stave i (teda v situácii že autobus číslo i čakal hneď pred semafórom), zakaždým semafór pustí rovnako veľa ľudí, a zakaždým bude po stave i nasledovať ten istý ďalší stav.

Preto by sa nám hodilo predpočítať polia, čo o každom stave/autobuse povedia: "Ak pred zelenou čakal prvý v rade autobus x, tak počas zelenej prejde c_x ľudí, a po nej bude čakať prvý autobus s_x." S nimi budeme vedieť každú zelenú vybaviť v konštantnom čase.

Predpočítať takéto polia c, s by sme dokázali napríklad pomocou prefixových súčtov a binárneho vyhľadávania, ale my to spravíme rýchlejšie, a dokonca bez nejakých pokročilých techník, ako prefixové súčty a binárne vyhľadávanie!!

Všimnime si, že pole s je neklesajúce – o dvoch susedných autobusoch x, $x+1$ platí že $s_x \leq s_{x+1}$. Inak povedané, ak by som pred zelenou začínal o jeden autobus neskôr, tak po zelenej skončím na rovnakom alebo neskoršom autobuse. Platí to, pretože ak autobusy od x po $s_x - 1$ mohli prejsť (ich súčet ľudí je maximálne r), tak zaručene aj od $x+1$ po $s_x - 1$ môžu prejsť (ich súčet tiež bude maximálne r), a možno sa zmestia aj ďalšie, takže s_{x+1} musí byť aspoň s_x.

(Výnimka: autobusy tvoria cyklus. Chceme sa tváriť že za posledným autobusom $n-1$ ide zase 0 a to je stále "neklesajúci" krok. V niektorých úlohách sa môže pri kódení zísť trik, prilepiť k sebe dve kópie pôvodného zoznamu. Indexy n až $2n-1$ znamenajú, že sme na 0 až $n-1$, ale už sme prešli otočku. Tu to nebolo treba.)

Vďaka neklesajúcosti môžeme použiť techniku dvoch bežcov (two pointers). Pre prvý autobus 0 vypočítame c_0, s_0 normálne – skúšame autobusy až kým neprekročíme r. Pre zvyšné autobusy pri výpočte c_{x+1}, s_{x+1} zoberieme c_x, s_x predošlého suseda, odčítame ľudí z autobusu x, a začneme skúšať možné s_{x+1} až od s_x ďalej. Predstavme si to ako dvoch bežcov, kde zakaždým keď zadný bežec spraví jeden krok, predný bežec spraví nula alebo viac krokov tak, aby udržal náskok r cestujúcich.

Aké je to celé rýchle? Zadný bežec spraví presne n krokov. Predný bežec spraví maximálne $2n$ krokov, lebo jeho náskok nikdy nemôže byť väčší ako n autobusov, lebo každý autobus môže na zelenú prejsť len raz. Spolu bežci spravia $O(n)$ krokov, každý spracujeme v konštantnom čase.

Máme teda riešenie v $O(n+k)$, čo pohodlne zbehne na 4 body.

Optimálne riešenie

Na optimálne riešenie si stačí uvedomiť, že stavy sa nám určite budú počas deja opakovať. Máme totiž len n možných stavov (možných autobusov ktoré môžu čakať ako prvý), a každý z nich sa vždy správa rovnako. Najviac po $n+1$ zelených sa teda dostaneme do stavu v ktorom už sme boli. Odteraz sa bude všetko správať ako predtým, teda sa do neho znovu dostaneme, a tak ďalej…

Ako to využiť? Budeme si simulovať zelené podobne ako v predošlom riešení a keď sa dostaneme do stavu, v ktorom už sme boli – teda že ako prvý je autobus, ktorý už sme ako prvý niekedy mali – vieme že sa bude stále dookola opakovať sekvencia stavov, ktorú sme videli, odkedy sme sem prvýkrát prišli.

Keď do stavu i prídeme prvýkrát, zapamätajme si hodnotu d_i – počet zelených, po koľkých sme do toho stavu prvýkrát prišli, a p_i – počet cestujúcich, čo sme do prvého príchodu videli. Keď sa vrátime do už navštíveného stavu, zistíme vďaka tomu dĺžku cyklu (počet zelených od prvej návštevy), aj koľko cestujúcich prejde za jeden cyklus.

A s týmito informáciami už vieme spočítať odpoveď. Nech dĺžka cyklu je D, počet ľudí, ktorí za cyklus prejdú je P a cyklus začína autobusom a. Potom sa celý cyklus vykoná $\lfloor (k-d_a)/D \rfloor$ krát – najskôr prejdeme na začiatok cyklu, teda sa nám zmenší k o d_a. V každom cykle prejde P ľudí, spolu teda $\lfloor (k-d_a)/D \rfloor \times P$ ľudí. Nakoniec nám ešte zostalo niekoľko zelených konkrétne $(k-d_a) \mod D$, ktoré treba dokončiť. Keďže však $D \leq n$, bude aj počet zelených ktoré treba vykonať nanajvýš n.

Keď si to celé zhrnieme, najskôr pomocou dvoch bežcov spočítame úseky autobusov, ktoré prejdú cez križovatku, ak bude rad začínať jednotlivými autobusmi – $O(n)$. Potom odsimulujeme najviac $n+1$ zelených, kým najdeme cyklus – $O(n)$, urobíme nejaké výpočty s cyklom – $O(1)$, po čom musíme ešte odsimulovať nanajvýš n ďalších zelených – $O(n)$.

Celková časová zložitosť teda bude $O(n)$, pamäťová bude rovnaká.

Listing programu (C++)

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

using namespace std;

int main(){
    int r, k, n; cin>>r>>k>>n;

    vector<int> a(n);
    for(int i =0;i< n;i++)cin>>a[i];
    vector<long long> next(n), pocet(n); //skupina co bude po procese prva, a pocet aut ktore za proces prejdu, ak zacinala skupina i

    long long sum = 0;
    int end = 0;
    for(int i =0;i< n;i++){
        while(sum + a[end] <=r){
            sum += a[end];
            end = (end + 1)%n;
            if(end == i) break;
        }
        next[i] = end;
        pocet[i] = sum;
        sum-=a[i];
    }

    int first_bus = 0;
    long long cycle_distance = 0, cycle_pocet = 0; //D a P zo vzoraku
    vector<long long> d(n, -1), p(n, -1); //polia d a p zo vzoraku

```

```
    while(true){
        d[first_bus] = cycle_distance;
        p[first_bus] = cycle_pocet;
        cycle_distance++;
        cycle_pocet += pocet[first_bus];
        first_bus= next[first_bus];
        if(p[first_bus]!= -1)break;
    }

    long long ans = 0;

    if(d[first_bus] <= k){
        k -= d[first_bus];
        ans += p[first_bus];
    }

    cycle_distance -= d[first_bus]; cycle_pocet -= p[first_bus];

    long long reps = k/cycle_distance;

    k-=reps*cycle_distance;

    ans += reps*cycle_pocet;

    while(k--){
        ans += pocet[first_bus];
        first_bus = next[first_bus];
    }
    cout<< ans<<endl;
}
```

Tomi

6. Ilúzia luxusu

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

Hľadanie šachovníc

Najprv zabudnime na to, že situácia sa bude časom meniť tým že budeme vyrezávať. Pozrime sa na úvodný stav a poďme v ňom hľadať najväčšie šachovnice. O každom políčku by sme chceli vedieť, aká veľká šachovnica v ňom môže končiť. Presnejšie, chceme vyrobiť pole $S[y][x]$, kde bude pre každé políčko zapísaná veľkosť najväčšej možnej šachovnice, ktorá má v tom políčku pravý dolný roh.

Toto sa dá spraviť cez dynamické programovanie. Netreba žiadne exotické triky, bude to klasická 2D dynamika v ktorej každá hodnota závisí len od hodnôt nášho ľavého, horného, a šikmého ľavého horného suseda. Len ten vzorec môže byť trochu neintuitívny.

Ak sme v prvom riadku alebo stĺpci, $S[y][x]$ je zjavne 1. Ak má nejaký náš sused nesprávnu farbu, tiež to bude 1. Ak majú všetci traja vyhovujúcu farbu, použijeme tento vzorec:

$$S[y][x] = 1 + \min(S[y-1][x],\ S[y][x-1],\ S[y-1][x-1])$$

Zamyslime sa, prečo to platí. $S[y][x]$ musí byť aspoň toľko čo hovorí tento vzorec, lebo bez ohľadu na to ktorý sused je ten najmenší, spolu s ostatnými susedmi sa poskladajú na o 1 väčší štvorec. Napríklad povedzme že vľavo odo mňa je hodnota presne 42 a hore aj vľavo hore odo mňa je aspoň 42 alebo viac. Keď sa tie tri šachovnice zjednotia, a pridáme naše políčko, vznikne šachovnica veľká (aspoň) 43.

$S[y][x]$ určite nemôže byť viac ako hovorí vzorec, pretože by z toho vyplynulo že ten sused má mať väčšiu hodnotu ako má. Napríklad povedzme že moja hodnota je 47 ale môj ľavý sused má iba 45. No ale ak ja mám pravý dolný roh platnej šachovnice veľkej 47, táto šachovnica pokrýva aj celú tú šachovnicu veľkú 45 môjho ľavého suseda, a aj prečnieva na ďalší riadok a stĺpec, takže ľavý sused by mal byť aspoň 46. Čo je spor.

$O(r^2c^2)$ riešenie

Opakujeme kým je čo vyrezávať: Na základe momentálneho stavu vyrobíme naším vzorcom pole S. Nájdeme v ňom najväčšiu hodnotu $S[y][x]$ (a najmenšie y, a najmenšie x). Tým sa dozvieme, že teraz máme vyrezať šachovnicu s touto veľkosťou a týmto pravým dolným rohom. O každom políčku tejto šachovnice si zapamätáme, že už je vyrezané (v ďalších kolách odteraz budú mať $S[y][x] = 0$ a tým ovplyvnia svojich susedov).

Šachovníc môže byť najviac rc a v každej iterácii odznova vypočítame celé pole S v čase $O(rc)$, takže dokopy nám to potrvá $O(r^2c^2)$.

Ktoré hodnoty sa vlastne zmenia?

Nemusíme zakaždým odznova počítať celé pole S. Vždy keď vyrežeme ďalší štvorec, zmenia sa iba tie políčka, ktorých doterajšia maximálna šachovnica podľa S sa prekrývala s tou našou vyrezanou, a tým pádom odteraz už nie je dostupná. Vtedy sa hodnota v S zmenší (zväčšiť sa nemôže).

Pre políčka vľavo od ľavého okraja alebo hore od horného okraja vyrezanej šachovnice sa hodnota v S nezmení. Berieme vždy pravý dolný roh, takže sa nemajú ako prekrývať.

Pre ostatné políčka sa hodnota v S zmeniť môže, ale iba ak sú k vyrezanej šachovnici relatívne blízko. Je to tak preto, že šachovnice vyrezávame od najväčšej veľkosti. V momente keď režeme šachovnicu veľkú $d \times d$, už nikde nenájdeme väčšiu – už sú všetky $S[y][x] \leq d$. Takže nás trápia len šachovnice, ktoré sa prekrývajú s vyrezanou, a sú veľké d alebo menej. Také šachovnice nemôžu byť vzdialené viac ako d.

Takže vždy keď vyrežeme šachovnicu veľkú d s pravým dolným rohom (y, x), stačí aktualizovať hodnoty S vo štvorci veľkom $2d \times 2d$ ktorý siaha od $y - d$ po $y + d$ a od $x - d$ po $x + d$ (uhhhh plusmínus jedna…).

To si môžeme dovoliť! Obsah štvorca $2d \times 2d$ je štvornásobok obsahu štvorca $d \times d$. Dá sa povedať, že vyrezanie jedného políčka nás stojí štyri updaty v S. Koľko dokopy vyrežeme políčok? Samozrejme rc. Takže koľkokrát dokopy budeme niečo prepisovať v S? $4rc$, čiže $O(rc)$.

$O(rc \cdot \min(r, c))$ riešenie

Vyskúšame fakt každú možnú šachovnicu v predpísanom poradí. Čiže:

Pre každú veľkosť d od $\min(r, c)$ po 1: Pre každé y: Pre každé x: Pozrime sa, či je $S[y][x] = d$. Ak áno, super, našli sme šachovnicu ktorú chceme vyrezať. Zapíšeme si o každom jej políčku že je vyrezané, prepočítame hodnoty S v okolitom štvorci veľkom $2d \times 2d$, a ideme ďalej.

Tento algoritmus je $O(rc \cdot \min(r, c))$. Za prepočítanie malého okolia S neplatíme nič extra, lebo dokopy všetkých zmien v S robíme len $O(rc)$, a novú hodnotu v S náš vzorec spočíta v konštantnom čase.

(Un)fun fact: Tento bruteforce mi ako organizátorovi spôsobuje samé problémy. Asymptotická časová zložitosť nie je optimálna, ale má úžasnú konštantu. V praxi je hrozne rýchly a nie veľmi chápem prečo. :(Nepodarilo sa mi nastaviť veľkosť vstupov a časový limit tak, aby prešlo všetko čo má prejsť, ale tento neprešiel. Smutný život.

$O(rc \cdot \log(rc))$ riešenie

Spravíme vyvažovaný vyhľadávací strom, napríklad `std::set`, ktorý nám poslúži ako "prioritná fronta s editovaním". Bude obsahovať trojicu (d, y, x) vždy práve vtedy keď platí $S[y][x] = d$. Vhodne skonštruujeme trojice (dáme tam záporné d) alebo definujeme custom porovnávaciu funkciu, aby boli vo vhodnom poradí a na začiatku setu bola vždy najlepšia šachovnica. Vždy keď v S niečo zmeníme, upravíme aj set (zmažeme starú trojicu a pridáme novú).

Opakujeme kým je čo vyrezávať: Zistíme zo setu najlepšiu existujúcu šachovnicu. Zapíšeme si o každom jej políčku že je vyrezané, prepočítame hodnoty S v okolitom štvorci veľkom $2d \times 2d$, upravíme set, a ideme ďalej. Toto trvá $O(rc \cdot \log(rc))$, lebo veľkosť setu je najviac rc, takže každá zmena v ňom trvá $\log(rc)$, a robíme $O(rc)$ zmien v S a teda aj v sete.

Pre záujem zopár optimalizácii čo sa tu dá spraviť. V KSP sa väčšinou snažíme aby ich nebolo treba, a toto aj tak nie je optimálne riešenie, ale možno sa vám niekedy v živote zídu.

- `std::set` má veľkú konštantu. Môžeme namiesto neho použiť haldu, napríklad `std::priority_queue`, alebo `std::make_heap` a spol. Jediná komplikácia je, že z haldy nejde zmazať ľubovoľný prvok, iba vrchný. Nevadí, tak ich proste pri zmene S nebudeme mazať. Naša halda bude mať nielen trojice (d, y, x) také že $S[y][x] = d$, ale aj všelijaké smeti, o ktorých to *kedysi* platilo. Vždy keď vyberieme vrchnú trojicu z haldy, najprv skontrolujeme či ešte stále $S[y][x] = d$. Veľkosť haldy aj vrátane smetí je najviac $O(rc)$, takže časová zložitosť bude rovnaká, ale halda je v praxi o dosť rýchlejšia (aspoň v tejto úlohe).

- Namiesto `std::tuple<int, int, int>` alebo `structu` s 3 číslami môžete trojice namačkať po bitoch do jedného intu. Vďaka tomu sa viac dát zmestí do CPU cache, a nejakú rýchlosť tým vyžmýkate. Pri `std::priority_queue` to môže spraviť skoro dvojnásobok.
- Operácie na `std::set` aj `std::priority_queue` sú drahé. Nemeňte ich pre úplne každé políčko v $2d \times 2d$ štvorci, iba tie, kde sa hodnota v S fakt zmenila.

Riešenie, ktoré implementuje tieto 3 zlepšováky, bolo u nás asi 4x rýchlejšie ako bez nich.

$O(rc)$ riešenie

Podobne ako v $O(rc \cdot \min(r,c))$ riešení, vo vonkajšom cykle spracujeme samostatne každú veľkosť d od $\min(r,c)$ po 1. Podobne ako v $O(rc \cdot \log(rc))$ riešení s haldou, budeme si v nejakej štruktúre pamätať políčka podľa toho, akú majú hodnotu v S (alebo sú to smeti čo ju kedysi mali), aby sme nemuseli skúšať každé y a x.

Tá štruktúra bude normálne pole. Každá veľkosť d bude mať svoje vlastné pole Q_d, ktoré obsahuje tie dvojice (y,x) o ktorých platí $S[y][x] = d$, alebo kedysi platilo. Tieto polia budú zo začiatku neutriedené. Keď ideme spracovávať veľkosť d, utriedime pole Q_d a prejdeme ho. O každom prvku najprv skontrolujeme, či tá šachovnica stále existuje, alebo sú to smeti. Ak existuje, vyrežeme ju, prepočítame okolie $2d \times 2d$, a zmeny v S si poznačíme v poliach $Q_?$. Hodnoty v S môžu iba klesať, a pri spracovávaní d už nie je žiadna väčšia ako d, takže v poli Q_d už nebude nič pribúdať po tom, čo sme ho utriedili.

Trik je v tom, že na triedenie použijeme radix sort. Na dvojicu (y,x) sa dá pozrieť ako dvojciferné číslo v $\max(r,c)$-kovej sústave. Zoradiť n-prvkové pole l-ciferných čísel v s-kovej sústave trvá $O(l \cdot (n+s))$. Takže utriediť jedno pole Q_d potrvá $O(2 \cdot (|Q_d| + \max(r,c))) = O(|Q_d| + \max(r,c))$. Sčítajme prvý člen: súčet všetkých $O(|Q_d|)$, čiže súčet dĺžok polí Q_d, sa rovná počtu zmien čo sme robili v S, čo je $O(rc)$. Sčítajme druhý člen: robíme $\min(r,c)$ triedení a každé trvá $O(\max(r,c))$, čiže $O(\min(r,c)\max(r,c))$, čo je samozrejme $O(rc)$. A ako v predošlých riešeniach, všetko vyrezávanie objavených šachovníc a upravovanie S tiež dokopy trvá $O(rc)$. Hurá, všetko je to $O(rc)$!

Pamäťová zložitosť je samozrejme tiež $O(rc)$.

Radix sort

Radix sort je špecializovaný triediaci algoritmus. Dokáže triediť rýchlejšie ako klasické v $O(n \log n)$. Daň za to je, že nevie triediť podľa ľubovoľného kritéria, ale len podľa celých čísel.

Najprv utriedi čísla podľa poslednej (najmenej dôležitej) cifry, potom predposlednej, a tak ďalej až po prvú. Každý krok zachová vzájomné poradie tých čo sa tam rovnajú, takže na konci budú úplne utriedené.

Triedenie podľa konkrétnej cifry vyzerá takto: Spočítame počet, ktorú číslicu sme videli koľkokrát. Z počtov spravíme prefixové súčty. Tie nám povedia o každej číslici, odkiaľ pokiaľ majú byť vo výstupe umiestnené čísla, čo ju majú. Prejdeme znova vstupné čísla a skopírujeme ich na správne miesto vo výstupnom poli, pričom si pamätáme, koľko sme zatiaľ videli s každou číslicou (alebo proste inkrementujeme pozíciu v poli prefixových súčtov).

Listing programu (C++)

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

using dvojica = array<int, 2>;

int H, W;
int minHW, maxHW;
vector<vector<char>> mapa;
vector<vector<int>> S;
vector<vector<dvojica>> Q;
vector<pair<int, int>> results;
```

```
16
17  void vypocitaj(int y, int x) {
18    if (mapa[y][x] == 2) {
19      S[y][x] = 0;
20    } else if (y == 0  x == 0
21        mapa[y-1][x] != 1-mapa[y][x]
22        mapa[y][x-1] != 1-mapa[y][x]
23        mapa[y-1][x-1] != mapa[y][x]) {
24      S[y][x] = 1;
25    } else {
26      S[y][x] = 1 + min(min(S[y-1][x], S[y][x-1]), S[y-1][x-1]);
27    }
28  }
29
30  vector<dvojica> sortuj_podla(int cifra, const vector<dvojica>& vstup) {
31    vector<int> kolko(maxHW, 0);
32    for (dvojica d : vstup) kolko[d[cifra]]++;
33
34    vector<int> kdepatri(maxHW, 0);
35    for (int i = 1; i < maxHW; i++) kdepatri[i] = kdepatri[i-1] + kolko[i-1];
36
37    vector<dvojica> vystup(vstup.size());
38    for (dvojica d : vstup) {
39      vystup[kdepatri[d[cifra]]] = d;
40      kdepatri[d[cifra]]++;
41    }
42    return vystup;
43  }
44
45  int main() {
46    cin >> H >> W;
47    minHW = min(H, W);
48    maxHW = max(H, W);
49    mapa.resize(H, vector<char>(W));
50    S.resize(H, vector<int>(W));
51    Q.resize(minHW + 1);
52
53    for (int y = 0; y < H; y++) {
54      cin.ignore(); // koniec riadku
55      for (int x = 0; x < W; x += 4) {
56        char ch = cin.get();
57        int val = ch >= 'A' ? ch - 'A' + 10 : ch - '0';
58        for (int sub = 0; sub < 4; sub++) {
59          mapa[y][x+sub] = (val & (8>>sub)) ? 1 : 0;
60        }
61      }
62    }
63
64    for (int y = 0; y < H; y++) {
65      for (int x = 0; x < W; x++) {
66        vypocitaj(y, x);
67        Q[S[y][x]].push_back({y, x});
68      }
```

```
  }

  for (int velkost = minHW; velkost > 0; velkost--) {
    if (Q[velkost].empty()) continue;

    vector<dvojica> zoradeny = sortuj_podla(0, sortuj_podla(1, Q[velkost]));

    int pocet = 0;
    for (auto [rohy, rohx] : zoradeny) {
      if (S[rohy][rohx] == velkost) {
        pocet++;

        for (int y = rohy - velkost + 1; y <= rohy; y++) {
          for (int x = rohx - velkost + 1; x <= rohx; x++) {
            mapa[y][x] = 2;
          }
        }

        for (int y = rohy - velkost + 1; y <= rohy + velkost - 1 && y < H; y++) {
          for (int x = rohx - velkost + 1; x <= rohx + velkost - 1 && x < W; x++) {
            int old = S[y][x];
            vypocitaj(y, x);
            if (S[y][x] != old && S[y][x] != 0) {
              Q[S[y][x]].push_back({y, x});
            }
          }
        }
      }
    }

    if (pocet) results.push_back({velkost, pocet});
  }

  cout << results.size() << endl;
  for (auto [velkost, pocet] : results) {
    cout << velkost << ' ' << pocet << endl;
  }
}
```

paulinka

7. Celistvé tabule

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

Ako ste si mohli všimnúť, táto úloha mala vela rôznych limitov pre jednoduché sady. Ako zvyčajne, znamená to existenicu veľa medzistupňových riešení ktoré váš mohli navnadiť ku vzoráku ;)

Poďme sa na ne postupne pozrieť:

Aspoň nejaké body

Prvý bod: Totálny Bruteforce

Na prvý bod si stačí poriadne prečítať ako počítame možnosti a pomocou štyroch forcyklov skúsime všetky možností pre ľavú, strednú, a pravú tabulu, a pozíciu strednej tabule. Následne zistíme či pasuje (napríklad obyčajným porovnávaním podreťazcov). Ak vám to stále nejde, zrejme ste sa oblbli niekde pri indexovaní ;)

Akú má toto riešenie zložitosť? Pamätová je lineárna – pamätáme si vstup. Pre výpočet časovej zložitosti, pozrime sa, čo robíme: pre každú možnú trojicu (w_l, w_p, w_s) nám stačí skontrolovať $|w_s| - 1$ pozícií, pre každé overenie rovnosti podreťazcov treba porovnať $|w_s|$ písmenok.

Takže celková zložitosť je $O(n^2 \sum_i |w_i|^2)$, čo stačilo v prvej sade.

Tri body: trochu stringológie

Všimnime si, čo vlastne v horeuvedenom bruteforce robíme: hľadáme stringy v stringoch.

Ak ste už videli trochu stringológie, možno ste už počuli o KMP[4].

Je to spôsob akým nájsť všetky výskyty stringu t v stringu s v lineárnom (teda $O(|s|+|t|)$) čase. KMPčkom by sme tak mohli zrýchliť prechádzajúci bruteforce, tak že pre každú možnosť trojice tabúľ (w_l, w_p, w_s) nájdeme všetky možné pozície kde možno dať strednú tabuľu v čase $|w_l| + |w_r| + |w_l|$, teda celková časová zložitosť je $O(\sum_i \sum_j \sum_l |w_l| + |w_r| + |w_l|) = O(n^2 \sum_i |w_i|)$. Pamäťová zložitosť je stále lineárna.

Toto riešenie vie prejsť na prvých troch sadách.

Štyri body: lepšia stringológia

Ak ste počuli o KMP, možno vás už strašili aj Aho-Corasickom[6]. To je algoritmus založený na podobnom princípe ako KMP, ale vie vyhľadávať veľa stringov naraz v lineárnom čase.

Ak ste sa tu zľakli, nebojte sa, existuje vzorové riešenie bez Aho-Corasicka a môžete túto sekciu preskočiť. Existuje však aj vzorové riešenie na základe Aho-Corasicka tak si tento algoritmus zbežne popíšeme aj v tejto sekcii:

Zo slov ktoré ideme v texte vyhľadávať (v našom prípapade sú to nápisy na všetkých tabuliach) si postavíme písmenkový strom[8]: strom, kde každý vrchol reprezentuje *prefix* nejakého (alebo viacerých) z nápisov. Pre každý vrchol si vypočítame pointer na vrchol reprezentujúci *najdlhší prefix ktorý je zároveň sufixom prefixu zodpovedajúcemu vrcholu* (ale nie sám na seba). Tieto pointery si vieme vypočítať efektívne v lineárnom čase napríklad pomocou prehľadávania do šírky. Pomocou tejto štruktúry si potom vieme efektívne vypočítať (v čase lineárnom od celkovej dĺžky reťazcov a počtu výskytov) koľkokrát sa niektorý z hľadaných reťazcov vyskytoval v danom reťazci.

Takže pre každú dvojicu (ľavá tabuľa, pravá tabuľa) vieme zbehnúť Aho-Corasicka pre všetky ostatné tabule (dobrá správa je, že nám stačí konečný automat predpočítavať iba raz), a tak nájsť všetky pozície v čase $O(\sum_i |w_i| + \sum_i \sum_j |w_i|) = O(n \sum_i |w_i|)$ plus počet odpovedí (čo sa pre prvé štyri sady ukáže ako dostatočne nízke číslo). Pamäťová zložitosť je stále linárna.

Pár detailov na záver: pri implementácii tohto riešenia si treba dávať pozor aby ste nezabudli že nie všetky výskyty sa rátajú (slová sa musia prekrývať), a že keď nájdeme výskyt, možno existuje iné slovo ktoré je sufixom dlhšieho výskytu, a navyše môže existovať rovnakých slov naviac. No, nie je to najvďačnejšie štvorbodové riešenie ;)

Stretnime sa v strede

Všimnime si, že doterajšie riešenia vždy počítali riešenia cez počítanie možností pre každú kombináciu ľavej a pravej tabule. Tých môže byť dokopy až pol milióna, takže vzorák musí robiť niečo trochu iné.

Čo keby sme išli a iterovali cez *strednú tabuľu*?

Spýtajme sa nasledovnú otázku: koľko je možností takých že w_i je stredná tabuľa, a jej prvých k znakov ide doľava a zvyšných $|w_i| - k$ doprava?

Všimnime si, že možnosti pre ľavú a pravú tabuľu sú *nezávislé*: ak existuje L tabúľ takých že ich posledných k znakov sa zhoduje s prvými k znakmi w_i, a R tabúľ takých že sa ich prvých $|w_i| - k$ znakov zhoduje s prefixov w_i rovnakej dĺžky, potom odpoveď na horeuvedenú otázku je $R \cdot L$.

Všimnime si, že keby sme vedeli odpovedať na každú z týchto otázok, stačí nám odpovede sčítať, a dostaneme takto výsledok. Navyše týchto otázok je dokopy $\sum_i |w_i| - 1$, čo je lineárne v dĺžke vstupu.

Takže ak by sme na ne vedeli odpovedať napríklad v priemerne konštantom čase, boli by sme vcelku spokojní (a vyriešili by sme úlohu).

Tri body: späť ku hrubej sile

Čísla R a L si vieme vypočítať hrubou silou: prejdeme si každého kandidáta na ľavú (alebo pravú) tabuľu, a porovnáme či sedia.

Pre každé porovnanie dokopy použijeme $O(|w_i|)$ operácií, takže časová zložitosť bude $O(\sum_i \sum_j |w_i|) = O(n \sum_i |w_i|)$, a pamäťová ostáva lineárna.

[4] Ak nie, nájdete ho v kuchárke[5]

[6] Tutoriál napríklad tu[7]

[8] https://en.wikipedia.org/wiki/Trie

L, R môže byť veľa, nájdime všetky naraz

Problém je, že nemáme čas skúšať všetky možné pravé (alebo ľavé) reťazce. Musíme na to ísť trochu múdrejšie. Poďme sa na otázku ešte raz pozrieť pod lupou: chceme vlastne vedieť, pre každý prefix, koľko sufixov sa s ním zhoduje (a naopak). Namiesto porovnania s každým možným reťazcom, pamätajme si pre každý prefix počet slov pre ktoré je to prefix. Podobne pre sufixy (v separátnej mape / dicte).

Takto vieme každú z otázok zodpovedať v $O(|w_i|)$ čase: pozrieme sa do mapy obsahujúcej všetky sufixy (resp. prefixy), zistíme koľkokrát sa náš prefix (resp. sufix) nachádza ako sufix (resp. prefix), a toto číslo si zapamätáme ako L (resp. R).

Takže nám dokopy bude trvať na otázky zodpovedať $O(\sum_i |w_i|^2)$ v čase aj pamäti, čo stačí na štyri body.

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
#define lili pair<lli, lli>
#ifdef EBUG
#define DBG        if(1)
#else
#define DBG        if(0)
#endif
#define SIZE(x) int(x.size())
const int infinity = 2000000999 / 2;
const long long int inff = 4000000000000000999;

typedef complex<long double> point;

template<class T>
T get() {
    T a;
    cin >> a;
    return a;
}

template <class T, class U>
ostream& operator<<(ostream& out, const pair<T, U> &par) {
    out << "[" << par.first << ";" << par.second << "]";
    return out;
}

template <class T>
ostream& operator<<(ostream& out, const set<T> &cont) {
    out << "{";
    for (const auto &x:cont) out << x << ", ";
    out << "}";
```

```
42      return out;
43  }
44
45  template <class T, class U>
46  ostream& operator<<(ostream& out, const unordered_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 increase_val(string key, unordered_map<string, int> &mapa) {
69      if (mapa.count(key)) mapa[key] ++;
70      else mapa[key] = 1;
71  }
72
73  lli return_val(string key, unordered_map<string, int> &mapa) {
74      if (mapa.count(key)) return mapa[key];
75      else return 0;
76  }
77
78  int main() {
79      cin.sync_with_stdio(false);
80      cout.sync_with_stdio(false);
81      int n = get<int>();
82      vector<string> slova;
83      FOR(i, n) slova.push_back(get<string>());
84      int L = 0;
85      FOR(i, n) L = max(L, SIZE(slova[i]));
86
87      vector<unordered_map<string, int> > suff_by_len(L), pref_by_len(L);
88      FOR(i, n) {
89          int l = SIZE(slova[i]);
90          FOR(j, l) {
91              increase_val(slova[i].substr(l - j - 1, j + 1), suff_by_len[j]);
92              increase_val(slova[i].substr(0, j + 1), pref_by_len[j]);
93          }
94      }
```

```
95
96      DBG FOR(i, L) cerr << suff_by_len[i].size() << " " << pref_by_len[i].size() << endl;
97
98      DBG cout << "Suffixes\n " << suff_by_len << "Prefixes:\n " << pref_by_len;
99
100     lli cnt = 0;
101     FOR(i, n) {
102         int l = SIZE(slova[i]);
103         for(int j = 0; j < l - 1; j ++) { // not entirely in one
104             lli pf_matches = return_val(slova[i].substr(0, j + 1), suff_by_len[j]);
105             if (pf_matches) {
106                 cnt += pf_matches * return_val(slova[i].substr(j + 1, l - j - 1), pref_by_len[l - j
    ↪  - 2]);
107             }
108         }
109     }
110
111     cout << cnt << endl;
112
113
114
115
116 }
```

Špeciálne prípady: AAAAAA

Ak sú všetky písmenká na ceduliach rovnaké, potom si nám v mapách stačí pamätať dĺžky (namiesto celých slov), čím vieme na otázku zrazu zodpovedať v konštantom čase, čím dostaneme riešenie na sadu 5 v lineárnom čase a pamäti.

Špeciálne prípady: Binárka

Nielen unárnu sústavu vieme zakódovať do jediného čísla: ak používame iba dva znaky a máme dostatočne malé slová (ako v sade 7), vieme si každé slovo "zakódovať" do čísla zmestiaceho sa do bežnej premennej (`long long` v C++). Keďže počítaču ide porovnávanie 60-bitových čísel oveľa rýchlejšie ako porovnávanie 60-znakových reťazcov, vieme sa tak tváriť že takto si vieme šikovne zakódovať (a porovnávať) sufixy a prefixy v lineárnom čase (a pamäti).

Vzorové riešenie: Hashujeme, alebo riskujeme nejednoznačné kódy

Nápad s binárkou vieme ďalej generalizovať: namiesto binárnej sústavy vieme vidieť nápisy ako čísla v 26-kovej (alebo väčšej) sústave. Poviete si možno: počkaj Paulinka, načo je nám to dobré, veď takéto veľké čísla sa mi už tobôž nezmestia ani do long longu!

Našťastie, príde na pomoc hashovanie. Totiž ak vidíme slová ako čísla v *prvočíselnej sústave* (pre dostatočne veľké prvočíslo), keď ich zmodulujeme (veľkým) prvočíslom dostaneme niečo skoro náhodné[9], a táto hodnota sa volá *hash* (a technika hashovanie). Čo nám pomôže že sú náhodné? Že je veľmi nepravdepdobné, že dve *rôzne* slová majú rovnaký hash, takže si dovolíme tvrdiť že keď dve slová majú rovnaký hash, sú rovnaké.

Hash slova $c_0c_1 \dots c_{l-1}$ vypočítame ako $c_0p^0 + c_1p^1 + \dots + c_{l-1}p^{l-1} \mod P$ (kde c_i sú znaky konvertované na čísla). Všimnite si, že pomocou prefixových súčtov si vieme vypočítať hash (vynásobený nejakým p^i) akéhoľvek prefixu, alebo sufixu v konštatnom čase. Ak chceme porovnanávať hashe, ako sa zbavíme p^i navyše? Najjednoduchšie riešenie si je všetky hashe (prefixové aj sufixové) ktoré si budeme udržovať v mape "zarovnať" na nejakú veľkú mocninu, takže namiesto $c_0p^0 + c_1p^1 + \dots + c_{k-1}p^{k-1}$ si zapamätáme $c_0p^C + c_1p^{C+1} + \dots + c_{l-1}p^{C+l-1}$ (kde C je napríklad dĺžka najväčšieho slova). Podobne aj pre sufixy.

Zvyšok riešenia je už podobné ako horeuvedené mapové riešenia: L si spočítame spočítaním "odsadeného" hashu prefixu w_i o dĺžke k, a zistíme si v mape sufixov koľko z nich má rovnaký hash, a to je naša hodnota L. R počítame analogicky.

[9] dá sa za tým pozrieť nejaká zaujímavá matika, alebo mi len verte

Takto dostaneme riešenie v čase aj pamäti $O(\sum_i |w_i|)$, teda lineárnom vo veľkosti vstupu.

Čo sa tu môže pokaziť? Pamätáte si na *skoro náhodné*? Problém je, že pri našom počte stringov sa môže stať (aj je to vcelku pravdepodobné), že nastane kolízia (a teda dve iné slová majú rovnaký hash). Ako sa tomu vyhnúť? Riešenie je zvýšiť počet "infomácie" ktoré o slovách máme (okrem hashu) tak aby sa nám stále dali jednoducho (lacno) porovnávať, ale znížili sme náhodu kolície. Jedna z možností je mať pole pre každú dĺžku prefixov (a sufixov), takže sa nám nestane kolízia medzi dvoma inak dĺhými reťazcami. Je (veľká) šanca že ani to nie je dosť, a riešenie je pridať druhý hash (teda buď iné prvočíslo P, alebo iné provočíslo p, alebo oba) a v mape si pamätať hashe dva. Prečo to by už malo stačiť? Keďže sú hashe "približne náhodné" kolícia zodpovedá tomu že dve náhodné čísla v rozsahu od 0 po $P-1$ sú rovnaké, čo je približne *narodeninový paradox*. Ten nám hovorí, aby sme mali malú pravdepodobnosť kolízie mali by sme mať P (respektívne počet možných hashov) byť približne kvadratický od počtu rôznych stringov, takže dva P približne v oblasti bilióna by malo na milión (pod)reťazcov stačiť :)

Listing programu (C++)

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

using namespace std;

#define FOR(i,n)        for(int i=0;i<(int)n;i++)
#define lli long long int
#define lili pair<lli, lli>
#define SIZE(x) int(x.size())


void increase_val(lli k1, lli k2, lli P, unordered_map<lli, int> &mapa) {
    lli key = k1 + P * k2;
    if (mapa.count(key)) mapa[key] ++;
    else mapa[key] = 1;
}

lli return_val(lli k1, lli k2, lli P, unordered_map<lli, int> &mapa) {
    lli key = k1 + P * k2;
    if (mapa.count(key)) return mapa[key];
    else return 0;
}

int main() {
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);
    int n;
    cin >> n;
    vector<string> slova(n);
    FOR(i, n) cin >> slova[i];
    int L = 0;
    FOR(i, n) L = max(L, SIZE(slova[i]));

    lli b1 = 131, b2 = 191;
    lli P = 1876956019;

    vector<vector<lili> > hashe(n, vector<lili>(1,{0, 0}));
    vector<lli> pw1(1, 1LL), pw2(1,1LL);
    FOR(i, L) pw1.push_back((pw1[i] * b1) % P);
    FOR(i, L) pw2.push_back((pw2[i] * b2) % P);
```

```
40
41      FOR(i, n) {
42          FOR(j, SIZE(slova[i])) {
43              lli c = slova[i][j];
44              hashe[i].push_back({(hashe[i][j].first + ((c * pw1[j]) % P)) % P,
45                  (hashe[i][j].second + ((c * pw2[j]) % P)) % P});
46          }
47      }
48
49      vector<unordered_map<lli, int> > suff_by_len(L), pref_by_len(L);
50      FOR(i, n) {
51          int l = SIZE(slova[i]);
52          FOR(j, l) {
53              lli hs1 = (P + hashe[i][l].first - hashe[i][l - j - 1].first) % P;
54              lli hs2 = (P + hashe[i][l].second - hashe[i][l - j - 1].second) % P;
55              lli offset1 = pw1[L - l + j + 1], offset2 = pw2[L - l + j + 1];
56              lli sh1 = (hs1 * offset1) % P, sh2 = (hs2 * offset2) % P;
57              increase_val(sh1, sh2, P, suff_by_len[j]);
58              offset1 = pw1[L]; offset2 = pw2[L];
59              increase_val((offset1 * hashe[i][j + 1].first) % P,
60                           (offset2 * hashe[i][j + 1].second) % P,
61                           P, pref_by_len[j]);
62          }
63      }
64
65      lli cnt = 0;
66      FOR(i, n) {
67          int l = SIZE(slova[i]);
68          for(int j = 0; j < l - 1; j ++) {
69              lli offset1 = pw1[L], offset2 = pw2[L];
70              lli pf_matches = return_val((offset1 * hashe[i][j + 1].first) % P,
71                                          (offset2 * hashe[i][j + 1].second) % P,
72                                          P, suff_by_len[j]);
73              if (pf_matches) {
74                  offset1 = pw1[L - (j + 1)]; offset2 = pw2[L - (j + 1)];
75                  lli hs1 = (hashe[i][l].first - hashe[i][j + 1].first + P) % P;
76                  lli hs2 = (hashe[i][l].second - hashe[i][j + 1].second + P) % P;
77                  cnt += pf_matches * return_val((offset1 * hs1) % P,
78                                                 (offset2 * hs2) % P, P,
79                                                 pref_by_len[l - j - 2]);
80              }
81          }
82      }
83
84      cout << cnt << endl;
85
86
87
88
89  }
```

Alternatívny vzorák alebo Aho-Corasick sa vracia

Iné alternatívne riešenie ktoré sa nespolieha na náhodu je využiť písmenkový strom, ktorý nám vznikne pri predpočítaní Aho-Corasicka.

Spomeňme si, že otázka ktorá nás zaujíma je: pre daný sufix, koľko existuje zhodujúcich sa prefixov? (alebo naopak, rozmyslite si, ako odpoveď na túto otázku možno počítať na počítanie zhodných sufixov pre daný prefix). V Aho-Corasickovom písmenkáči si pre každý vrchol pamätáme ktorý je najdlhší nezhodný sufix, ktorý je slovu vo vrchole prefixom.

Túto informáciu vieme využiť na spočítanie, pre každý prefix, koľko slov existuje ktorým je tento prefix sufixom a to formou dynamického programovania. Vieme si pre každý vrchol zistiť ktoré vrcholy doňho majú pointer. Potom v poradí od najhlšieho po najkratšie prefixy (teda obrátené poradie od BFS), je hodnota pre vrchol počet slov ktoré v ňom končia plus súčet hodnôt vrcholov ktoré naňho majú pointer.

Keď máme toto predpočítané, ako *rýchlo* prejdeme cez všetky miesta v strome odpovedajúce sufixom? Tu použijeme príjemnú vlastnosť stromu: slovo ktorého sufixy sledujeme je už v strome. Takže si nájdeme kde sa končí, a potom pre každý zaujímavý sufix skĺzneme o rodiča nižšie.

Takto spočítame hodnoty R, ako na L? Stačí nám nápisy obrátiť a spraviť rovnakú procedúru pre písmenkový strom sufixov (a hľadanie prefixov).

Všetko predpočítanie aj prehľadávanie vieme robiť v lineárnom čase aj pamäti a takto dostaneme ďalšie možné riešenie na plný počet bodov.

Listing programu (C++)

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

using namespace std;

#define FOR(i,n)        for(int i=0;i<(int)n;i++)
#define lli long long int
#define SIZE(x) int(x.size())

struct tria {
    int depth;
    char c;
    tria *pointer_to;
    tria *otec;
    int end_of;
    lli all_ends;
    vector<tria *> deti;

    tria (int d, char ch, tria *father) {
        depth = d;
        c = ch;
        otec = father;
        end_of = false;
        pointer_to = NULL;
        deti.resize(30, NULL);
        end_of = 0, all_ends = 0;
    }

    void add_word(string &s, int id) {
        if (id == s.size()) {
            end_of ++;
            all_ends ++;
            return;
        }
```

```
34          int it = s[id] - 'A';
35          if (deti[it] == NULL) deti[it] = new tria(depth + 1, s[id], this);
36          deti[it] -> add_word(s, id + 1);
37      }
38  };
39
40  void aho_corastic_pattern_prep(tria *root) {
41      queue<tria *> Q;
42      stack<tria *> S;
43      Q.push(root);
44      root -> pointer_to = root;
45      while (Q.size()) {
46          tria *t = Q.front();
47          S.push(t);
48          Q.pop();
49
50          for (auto c : t -> deti) {
51              if (c == NULL) continue;
52              if (t == root) c -> pointer_to = root;
53              else {
54                  tria *kde = t -> pointer_to;
55                  c -> pointer_to = root;
56                  do {
57                      if (kde -> deti[c -> c - 'A'] == NULL)
58                          kde = kde -> pointer_to;
59                      else {
60                          c -> pointer_to = kde -> deti[c -> c - 'A'];
61                          break;
62                      }
63                  } while (kde != root);
64                  if (kde == root && root -> deti[c -> c - 'A'] != NULL)
65                      c -> pointer_to = root -> deti[c -> c - 'A'];
66              }
67              Q.push(c);
68          }
69      }
70
71      while (S.size()) {
72          tria *t = S.top();
73          S.pop();
74          t -> pointer_to -> all_ends += t -> all_ends;
75      }
76  }
77
78  int main() {
79      cin.sync_with_stdio(false);
80      cout.sync_with_stdio(false);
81      int n;
82      cin >> n;
83      vector<string> slova(n);
84      FOR(i, n) cin >> slova[i];
85
86      tria *root_pf = new tria(0, '-', NULL), *root_sf = new tria(0, '-', NULL);
```

```
    FOR(i, n) {
        root_pf -> add_word(slova[i],0);
        string rev = slova[i];
        reverse(rev.begin(), rev.end());
        root_sf -> add_word(rev, 0);
    }
    aho_corastic_pattern_prep(root_pf);
    aho_corastic_pattern_prep(root_sf);

    lli cnt = 0;
    FOR(i, n) {
        tria *pf = root_pf, *sf = root_sf;
        pf = root_pf -> deti[slova[i][0] - 'A'];
        for (int j = slova[i].size() - 1; j >= 1; j --) {
            sf = sf -> deti[slova[i][j] - 'A'];
        }

        for (int j = 0; j < slova[i].size() - 1; j ++) {
            cnt += pf -> all_ends * sf -> all_ends;
            pf = pf -> deti[slova[i][j + 1] - 'A'];
            sf = sf -> otec;
        }
    }
    cout << cnt << endl;
}
```

Merlin

8. Akútna výstavba

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

Označme si najprv pre každého robotníka jeho periódu $P_i = a_i + b_i$.

Jednoduché riešenie

Jedno z priamočiarych riešení by mohlo vyzerať napríklad takto. Vytvoríme si pole `W` dĺžky N, kde hodnota `W[i]` bude počet robotníkov, ktorý pracujú v i-tej hodine. Keď nejaký robotník príde, tak zvýšime počet pracujúcich robotníkov pre každú z hodín, v ktorých bude pracovať. Naopak, ak odíde, tak znížime počet pracujúcich robotníkov v týchto hodinách.

Toto riešenie bude mať časovú zložitosť $O(N^2)$, lebo kvôli každému príchodu alebo odchodu môžeme zmeniť až $O(N)$ prvkov poľa `W`.

Ľahký robotníci

Poďme sa najprv zamerať na riešenie prvých dvoch sád. Hneď si môžeme všimnúť, že zásadný rozdiel medzi týmito sadami a ostatnými je, že $\max P_i$ je najviac 500. Rozdeľme si teda všetkých robotníkov do skupín podľa ich periódy. Teraz pre každú z týchto skupín môžeme robiť to isté, čo pri predošlom riešení. Pre každú zo skupín už ale nepotrebujeme pole veľkosti až N. Pre skupinu s periódou P nám stačí pole veľkosti P (lebo počet pracujúcich robotníkov tejto skupiny v čase t bude rovnaký, ako počet pracujúcich robotníkov v čase $t + P$).

Pre každú skupinu robotníkov si teda spravíme pole dĺžky P, kde na i-tom indexe bude počet robotníkov, ktorí pracujú v hodine i modulo P. Pri príchode alebo odchode robotníka nám stačí prejsť celé toto pole a ku každému zvyšku, v ktorom by tento robotník pracoval, pripočítame (respektíve odčitame) 1. Keď chceme zistiť, koľko robotníkov pracuje v čase t, tak sa stačí pre každú skupinu pozrieť na hodinu so správnym zvyškom (t) a tieto hodnoty sčítať.

Pridanie alebo odobratie jedného robotníka teda zaberie $O(\max P_i)$ času a zisťovanie odpovede zaberie taktiež $O(\max P_i)$ času. Toto riešenie má teda časovú zložitosť $O(N \max P_i)$.

Ľahkí a ťažkí robotníci

Už vieme pomerne efektívne riešiť situáciu, v ktorej majú všetci robotníci malú periódu. Poďme teda využiť jeden z klasických trikov, ktorý sa dá v takejto situácií využiť: rozdelíme si robotníkov na "ľahkých" a "ťažkých"

a každú z týchto skupín budeme riešiť separátne. Ľahký robotníci budú tý, ktorých perióda je menšia ako $\sqrt{N}$. Pre nich už úlohu vieme riešiť efektívne v čase $O(N\sqrt{N})$.

Ťažký robotníci

Čo ale spraviť s ťažkými robotníkmi? Môžeme využiť to, že počet periód, ktoré stihnú do konca šichty spraviť, bude malý. Presnejšie to bude najviac $\frac{N}{\sqrt{N}} = \sqrt{N}$ periód.

Pri každom príchode alebo odchode ťažkého robotníka teda potrebujeme zmeniť hodnoty $O(\sqrt{N})$ intervalov. Stačí už len vymyslieť, ako toto robiť efektívne. Mohli robiť pomocou intervalového stromu, ale dá sa to aj rýchlejšie pekným trikom s prefixovými súčtami.

Budeme si pamätať pole `A` dĺžky N. Keď chceme k intervalu $[a, b]$ pripočítať 1, tak k `A[a]` pripočítame 1 a od `A[b+1]` odčítame 1. Týmto sme dosiahli to, že sme v poli prefixových súčtov poľa `A` na intervale $[a, b]$ pripočítali 1. Ak chceme odčítavať, tak to bude fungovať rovnako, len treba zmenit znamienka.

Počet aktuálne pracujúcich ťažkých robotníkov teda môžeme počítať nasledovne. Keď príde alebo odíde nejaký ťažký robotník, tak si do poľa `A` na $O(\sqrt{N})$ pozícií poznačíme ± 1. Tiež si postupne v pomocnej premennej budeme počítať prefixový súčet tohto poľa, čo bude vlastne hľadaný počet pracujúcich ťažkých robotníkov.

Odchody a príchody ťažkých robotníkov teda vieme riešiť v čase $O(\sqrt{N})$ a dopočítať, koľko ich pracuje v aktuálnom čase vieme v konštantnom čase.

Optimálne riešenie

Na vyriešenie celej úlohy nám už len stačí všetko skonbinovať dokopy. Na začiatku hodiny vyriešime príchod alebo odchod robotníka (podľa toho, či je ľahký alebo ťažký). Potom už len stačí zvlášť dopočíť počet pracujúcich ľahkých robotníkov a dopočítať aktuálny prefixový súčet pre ťažkých robotníkov, čím dostaneme finálnu odpoveď pre túto hodinu.

Celková časová zložitosť je $O(N\sqrt{N})$, lebo vyriešiť príchod a odchod robotníka vieme v čase $O(\sqrt{N})$ (bez ohľadu na to, čí je ľahký alebo ťažký) a dopočítať počet pracujúcich ľahkých robotníkov vieme tiež v čase $O(\sqrt{N})$. Počet pracujúcich ťažkých robotníkov počitame postupne pomocou prefixových súčtov v konštantom čase.

Pamäťová zložitosť je $O(N)$, lebo pre ťažkých robotníkov si pamätáme pole dĺžky N a pre ľahkých robotníkov $\sqrt{N}$ polí dĺžky $O(\sqrt{N})$.

Listing programu (C++)

```cpp
#include<bits/stdc++.h>
using namespace std;
#define a first
#define b second

void tazky_prichod(int cas, vector<int>& tazky, pair<int,int> robotnik){
    int perioda = robotnik.a + robotnik.b;

    // zmeny z pracovania na obdivovanie
    for(int i = cas + perioda;i < tazky.size();i += perioda)
        tazky[i] -= 1;

    // zmeny z obdivovania na pracovanie
    for(int i = cas + robotnik.a;i < tazky.size();i += perioda)
        tazky[i] += 1;
}

void tazky_odchod(int zaciatok, int cas, vector<int>& tazky, pair<int,int> robotnik){
    int perioda = robotnik.a + robotnik.b;

    // ak aktualne pracuje, tak ho treba hned odstranit
    if((cas - zaciatok)%perioda >= robotnik.a){
        tazky[cas] -= 1;
```

```
    }

    // zmeny z pracovania na obdivovanie
    for(int i = zaciatok + perioda;i < tazky.size();i += perioda)if(i > cas)
        tazky[i] += 1;

    // zmeny z obdivovania na pracovanie
    for(int i = zaciatok + robotnik.a;i < tazky.size();i += perioda)if(i > cas)
        tazky[i] -= 1;
}

void lahky_prichod(int cas, vector<vector<int>>& lahky, pair<int,int> robotnik){
    int perioda = robotnik.a + robotnik.b;

    // pripocitam 1 ku vsetkym zvyskom, v ktorych tento robotnik pracuje
    for(int i = robotnik.a;i < perioda;i++)
        lahky[perioda][(cas + i)%perioda] += 1;
}

void lahky_odchod(int zaciatok, vector<vector<int>>& lahky, pair<int,int> robotnik){
    int perioda = robotnik.a + robotnik.b;

    // odcitam 1 od vsetkych zvyskov, v ktorych tento robotnik pracoval
    for(int i = robotnik.a;i < perioda;i++)
        lahky[perioda][(zaciatok + i)%perioda] -= 1;
}

// pocet lahkych robotnikov pracujucich v danom case
int pocet_lahkych(int cas, vector<vector<int>>& lahky){
    int vysledok = 0;

    // staci sa pre kazdu periodu pozriet na spravny zvysok a vsetko scitat
    for(int period = 1;period < lahky.size();period++)
        vysledok += lahky[period][cas%period];

    return vysledok;
}

const int SQRT = 450;

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

    vector<pair<int,int>> robotnici(n); // zoznam dvojic a_i, b_i
    vector<int> posledne_pridanie(n, -1); // cas, v ktorom dany robotnik posledne prisiel
    vector<vector<int>> lahky(SQRT, vector<int>(SQRT, 0));
    vector<int> tazky(m, 0);

    for(auto &robotnik : robotnici)
        cin >> robotnik.a >> robotnik.b;

    int prefix_sucet = 0; // pocet aktualne pracujucich tazkych robotnikov
```

```
77      for(int cas = 0;cas < m;cas++){
78          int op, index;
79          cin >> op >> index;
80
81          if(op == 1){
82              if(robotnici[index].a + robotnici[index].b >= SQRT)
83                  tazky_prichod(cas, tazky, robotnici[index]);
84              else
85                  lahky_prichod(cas, lahky, robotnici[index]);
86              posledne_pridanie[index] = cas;
87          }else{
88              if(robotnici[index].a + robotnici[index].b >= SQRT)
89                  tazky_odchod(posledne_pridanie[index], cas, tazky, robotnici[index]);
90              else
91                  lahky_odchod(posledne_pridanie[index], lahky, robotnici[index]);
92          }
93
94          prefix_sucet += tazky[cas];
95
96          // vysledok je pocet pracujucich tazkych + lahkych robotnikov
97          cout << prefix_sucet + pocet_lahkych(cas, lahky) << "\n";
98      }
99  }
```