Korešpondenčný seminár z programovania
XXXIX. ročník, 2021/22
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 1. kola letnej časti

Julka

1. Zajace a mrkva

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

Myšlienka riešenia

Aby sme zistili, koľko zajacov sa skrýva za plotom, musíme najprv zistiť vzdialenosť medzi prvými dvoma ušami na vstupe. Keďže predpokladáme, že vzdialenosť medzi ušami je pre všetkych zajacov rovnaká, malo by platiť, že pre každého nasledujúceho zajaca, tj. pre každú nasledujúcu dvojicu uší na vstupe bude táto vzdialenosť rovnaká. V prípade, že sa daná vzdialenosť líši, nejedná sa o zajace, a teda na výstupe bude -1. Pokiaľ bude vzdialenosť pre každé 2 uši rovnaká, vypíšeme počet zajacov, pre ktorých sme túto vzdialenosť overovali.

Časová a pamäťová zložitosť

Keďže sa stačí na celý vstup pozrieť iba jeden raz, časová zložitosť bude lineárne závislá od dĺžky vstupu – $O(n)$. Pamäťová zložitosť bude konštantná - $O(1)$, keďže stačí ak si budeme počet zajacov uchovávať v jednej premennej a postupne počas čítania vstupu túto hodnotu zvyšovať.

Listing programu (Python)

```
def main(prvy = True, ucho = 0, vstup = input(), vzdialenost = 0):
    x = None
    pocet = 0
    for znak in vstup:
        if znak == "U":
            if ucho == 0:
                ucho +=1
                vzdialenost = 0
            elif ucho == 1:
                ucho = 0
                pocet+=1
                if prvy == True:
                    x = vzdialenost
                    prvy = False
                    vzdialenost = 0
                elif prvy == False:
                    if x != vzdialenost:
                        return -1
        elif znak !="U" and ucho==1:
            vzdialenost+=1
    return pocet

print(main())
```

Listing programu (C++)

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

int main() {
```

```
    string in;
    cin >> in;

    int sirka_zajaca = -1;
    int zaciatok_zajaca = -1;
    int pocet_zajacov = 0;

    for (int i = 0; i <= in.length(); i++) {
        if (in[i] == 'U') {
            if (zaciatok_zajaca == -1) {
                zaciatok_zajaca = i;
            } else {
                pocet_zajacov++;
                if (sirka_zajaca == -1) {
                    sirka_zajaca = i - zaciatok_zajaca;
                } else {
                    int sirka = i - zaciatok_zajaca;

                    if (sirka != sirka_zajaca) {
                        cout << "-1" << endl;
                        return 0;
                    }
                }
                zaciatok_zajaca = -1;
            }
        }
    }

    cout << pocet_zajacov << endl;
}
```

Aďo

2. Aha, psíky!

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

Už na prvý pohľad je asi celkom zrejmé, že pohyb KSPsa môžeme pomerne jednoducho odsimulovať. Stačí si pamätať pole s pozíciami KSPsíkov, našu aktuálnu pozíciu a smer. Následne sa v každom kroku posunieme v poli o jeden index doprava/doľava (podľa aktuálneho smeru KSPsa) a overíme, či vzdialenosť od pôvodnej pozície KSPsa ku KSPsíkovi, ku ktorému sme práve došli, je najviac x. V opačnom prípade sa vrátime na predchádzajúci index, zmeníme aktuálny smer KSPsa a skúsime sa posunúť sa v opačnom smere.

Ako vidíme, potrebujeme odsimulovať všetkých k krokov, teda časová zložitosť bude $O(k)$. Pamäťová zložitosť je zase $O(n)$, keďže si pamätáme celé pole s pozíciami KSPsíkov. Takéto riešenie mohlo (v závislosti od implementácie) získať na testovači približne polovicu bodov.

Listing programu (Python)

```
n, s, x, k = [int(x) for x in input().split()]
ksps = [int(x) for x in input().split()]

pos = s
direction = 1

for step in range(k):
    npos = pos + direction
    if 0 <= npos < len(ksps) and abs(ksps[npos] - ksps[pos]) <= x:
        pos = npos
    else:
        direction *= -1
        npos = pos + direction
```

```
        if 0 <= npos < len(ksps) and abs(ksps[npos] - ksps[pos]) <= x:
            pos = npos

print(ksps[pos])
```

Listing programu (C++)

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

using namespace std;

int main() {
    int n, s, x, k;
    cin >> n >> s >> x >> k;

    vector<int> ksps(n);
    for (int i = 0; i < n; i++) {
        cin >> ksps[i];
    }

    int pos = s;
    int direction = 1;

    for (int s = 0; s < k; s++) {
        int npos = pos + direction;

        if (0 <= npos && npos < n && abs(ksps[npos] - ksps[pos]) <= x) {
            pos = npos;
        } else {
            direction *= -1;
            npos = pos + direction;
            if (0 <= npos && npos < n && abs(ksps[npos] - ksps[pos]) <= x) {
                pos = npos;
            }
        }
    }

    cout << ksps[pos] << '\n';
    return 0;
}
```

Vzorák

Aby sme z priamočiarej simulácie dostali vzorové riešenie nám stačí jednoduchý trik. Mohli sme si všimnuť, že pri simulácii sa KSPes vždy otočí pri tých istých dvoch KSPsíkoch. Pri veľkom počte krokov teda KSPes strávi väčšinu času behaním medzi týmito dvomi pozíciami.

Našu simuláciu upravíme nasledovne: Najskôr necháme KSPsa ísť doprava, až kým nenavštívi posledného dosiahnuteľného KSPsíka a jeho index si zapamätáme. Podobne si zistíme index najľavejšieho dosiahnuteľného KSPsíka.

Nech je rozdiel najpravejšieho a najľavšieho dosiahnuteľného indexu l, potom KSPsovi bude trvať $2l$ krokov, kým prejde všetkých KSPsíkov. Keďže sa po takomto "kolečku" KSPes vždy vráti na pôvodnú pozíciu, môžeme túto časť simulácie jednoducho preskočiť tým, že počet zostávajúcich krokov zmodulujeme $2l$.

Nakoniec nám ostane niekoľko krokov, ktoré by sme mohli znova odsimulovať. V skutočnosti to ale ani nie je potrebné. Predpokladajme, že KSPs sa momentálne nachádza pri najľavejšom dosiahnuteľnom KSPsíkovi (tam sme ho totiž pri hľadaní najľavejšieho dosiahnuteľného KSPsíka presunuli). Teraz môže nastať jeden z dvoch prípadov: Ak je zostávajúci počet krokov k menší ako l, stačí nám posunúť KSPsa v poli s pozíciami KSPsíkov

o k indexov doprava. Naopak, ak je $k > l$, finálny index KSPsa vypočítame ak od indexu napravejšieho dosiahnuteľného KSPsíka odčítame $k - l$.

Na nájdenie najpravšieho a najľavšieho dosiahnuteľného KSPsíka budeme musieť prejsť najviac n psíkov. Následne iba vypočítame zvyšok po delení a finálnu pozíciu KSPsa v $O(1)$, čiže celková časová zložitosť bude $O(n)$. Pamäťová zložitosť ostáva stále rovnaká.

Listing programu (Python)

```
n, s, x, k = [int(x) for x in input().split()]
ksps = [int(x) for x in input().split()]
pos = s

while k > 0 and pos+1 < n and abs(ksps[pos] - ksps[pos+1]) <= x:
    pos += 1
    k -= 1
right = pos

while k > 0 and pos-1 >= 0 and abs(ksps[pos] - ksps[pos-1]) <= x:
    pos -= 1
    k -= 1
left = pos

length = right - left

if k == 0 or length == 0:
    print(ksps[pos])
    exit()

k = k % (2*length)
if k > length:
    pos = right - k + length
else:
    pos = left + k

print(ksps[pos])
```

Listing programu (C++)

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

typedef long long int lli;

int main() {
    lli n, s, x, k;
    cin >> n >> s >> x >> k;

    vector<lli> ksps(n);
    for (lli i = 0; i < n; i++) {
        cin >> ksps[i];
    }

    lli pos = s;

    while (k > 0 && pos+1 < n && abs(ksps[pos] - ksps[pos+1]) <= x) {
        pos++;
        k--;
```

```
    }
    lli right = pos;

     while (k > 0 && pos-1 >= 0 && abs(ksps[pos] - ksps[pos-1]) <= x) {
        pos--;
        k--;
    }
    lli left = pos;
    lli length = right - left;

    if (k == 0 || length == 0) {
        cout << ksps[pos] << '\n';
        return 0;
    }

    k %= 2*length;

    if (k > length) {
        pos = right - k + length;
    } else {
        pos = left + k;
    }

    cout << ksps[pos] << '\n';
    return 0;
}
```

Dávid

3. Havo sem, havo tam

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

Na začiatok si môžeme všimnúť, že na poradí, v akom sú intervaly na vstupe nezáleží. Prvá vec čo by nás mala v takomto prípade napadnúť, je zoradiť vstup.

Pomalé riešenie

Pozrime sa najprv, ako vôbec zistiť správnu odpoveď. Zo zadania si môžeme všimnúť, že v rozvrhu budú akési skupinky intervalov, ktoré sa niekde prekrývajú a teda majú všetky rovnakú šírku. Keď si intervaly zoradíme podla času začiatku, asi nás neprekvapí, že takéto skupinky budú tvoriť súvislé úseky. Stačí si uvedomiť, že ak takáto skupinka niekde končí, tak všetky ďalšie intervaly musia začínať až za koncom každého intervalu z tejto skupinky.

Ostáva nám už len tieto skupinky identifikovať a zistiť, akú majú šírku. Skupinku môžeme vytvárať tak, že budeme postupne pridávať intervaly z nášeho utriedeného poľa, a aby sme zistili či doň patrí ďalší interval, budeme si pamätať najväčšie číslo z koncov intervalov v danej skupinke.

Zistiť šírku je o niečo ťažšie, no zatiaľ sa uspokojíme s tým, že po každom pridanom intervale sa pozrieme koľko z doterajších sa s ním prekrýva – teda končí neskôr ako začína on. Na prvý pohľad sa to možno nezdá, ale takýto prístup naozaj bude fungovať. Určite nám takto nevyjde väčší výsledok ako by mal, lebo všetky intervaly, na ktoré sa pozeráme sa prekrývajú s našim začiatkom, teda sa musia prekrývať aj navzájom. Už nám stačí sa len zamyslieť, pri ktorom intervale nájdeme to hľadané maximum prekrývajúcich sa intervalov. Spomedzi tých intervalov ktoré sa v tom momente prekrívajú, niektorý začína ako posledný. Keďže začína ako posledný, všetky ostatné máme už pridané a teda ich nájdeme ako prekrývajúce sa a vyjde nám správna maximálna šírka.

Rôzne vylepšenia

Môžeme si všimnúť, že jediná problematická časť v predošlom riešení je tá, kde zisťujeme šírku. Ak máme skupinku, ktorá obsahuje veľa intervalov (napríklad všetky), dosiahli by sme až kvadratickú časovú zložitosť. Potrebujeme teda zistiť, ako to robiť šikovnejšie.

Jedna možnosť je pozrieť sa na to ako na čiastkový problém a použiť vhodnú dátovú štruktúru. Napríklad binárne vyhladávacie stromy alebo haldu. Takéto riešenie môže byť napríklad v C++ pomerne jednoduché, no Python nemá takúto štruktúru v štandardnej knižnici. Pozrime sa teda radšej na iné a krajšie riešenie.

Listing programu (C++)

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

using namespace std;

int main(){
    int n;
    cin>>n;
    vector<pair<pair<int,int>,int>> ints(n);
    for(int i=0; i<n; i++){
        cin>>ints[i].first.first>>ints[i].first.second;
        ints[i].second=i;
    }
    sort(ints.begin(),ints.end());

    multiset<int> active;
    vector<int> outs(n,0);
    vector<int>s;
    int p=0;
    int k=0;

    for (int i=0; i<n; i++){
        active.insert(ints[i].first.second);
        if (ints[i].first.first<k){
            s.push_back(ints[i].second);
            auto it = active.begin();
            while (it != active.end()  and *it <=ints[i].first.first) it =
                ↪ active.erase(it);
            p = max(p, (int)active.size());
            k = max(k,ints[i].first.second);
        }
        else{
            for(int t=0; t<s.size(); t++) outs[s[t]]=p;
            s={ints[i].second};
            k=ints[i].first.second;
            p=1;
        }
    }
    for(int t=0; t<s.size(); t++) outs[s[t]]=p;

    for(int i=0; i<n; i++) cout<<outs[i]<<endl;
}
```

Užitočný trik

Ukážeme si trik ktorý nám pomôže ku jednoduchšiemu riešeniu. Tento prístup sa niekedy nazýva aj zametanie a spočíva v tom, že sa budeme posúvať od začiatku časovej osy po koniec a počítať si kolko intervalov je v danom momente aktívnych. Aby sme však neprechádzali všetky čísla, v ktorých sa nič nedeje, najprv si do jedného poľa uložíme všetky začiatky a konce intervalov, spolu s informáciou o tom, či ide o začiatok alebo koneic a ktorému intervalu patrí. Následne toto pole usporiadame podľa času (začiatku resp. konca), a prechádzame cyklom od začiatku po koniec. Pamätáme si počet aktívnych intervalov, pričom ak nejaký skončí, zmenšíme počet o jeden a ak nejaký začne, tak zase zväčšíme o jeden.

Ku kompletnému riešeniu nám stačí si pamätať najväčší počet aktívnych intervalov a zoznam intervalov, ktoré sme už stretli. Keď narazíme na situáciu, že máme nula aktívnych intervalov, znamená to, že nejaká

skupinka skončila. Zapíšeme si aktuálne maximum pre všetky uložené intervaly, vynulujeme maximum a uložené intervaly, a pokračujeme.

Časová a pamäťová zložitosť

Takéto riešenie má časovú zložitosť $O(n \cdot \log(n))$ kvôli usporiadaniu poľa. Pamäťová zložitosť bude lineárna, pretože každé číslo zo vstupu budeme mať uložené len konštantný počet krát, a nič naviac.

Listing programu (Python)

```
n = int(input())
ints = [list(map(int,input().split()))+[i] for i in range(n)]
changes=[]
for i in ints:
  changes.append((i[0],1,i[2]))
  changes.append((i[1],-1,i[2]))
changes.sort()

outs =[0 for _ in range(n)]
ctr=0
m=0
s=[]
for n,c,i in changes:
  ctr+=c
  m=max(m,ctr)
  if c==1:
    s.append(i)
  if ctr==0:
    for t in s:
      outs[t]=m
    s=[]
    m=0

print(*outs, sep='\n')
```

danza

4. Rolka výterov

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

Na prvý pohľad môže úloha vyzerať zložito. V tejto situácii zvykne pomôcť urobiť čo najviac pozorovaní a z nich potom prísť na riešenie. Nuž, poďme teda skúsiť spraviť nejaké pozorovania.

Prvým zjavným faktom je, že dielik i má vedľa seba dieliky $i-1$ a $i+1$. Ďalším pozorovaním je, že keď toaleťák nejak skladáme, tak dieliky na párnych pozíciách pôjdu zľava doprava a dieliky na nepárnych pozíciách pôjcu sprava doľava, alebo naopak.

Pomalé riešenie

Našim cieľom je zistiť, či toaleťák niekde neprechádza sám sebou. Toaleťák sa pretína sám so sebou práve vtedy, keď medzi dielikom i a $i+1$ leží nejaké j také, že dielik $j-1$ alebo $j+1$ nie je v permutácii medzi i a $i+1$. Samozrejme, zároveň musí platiť, že i a j majú rovnakú paritu.

Ak majú i a j rovnakú paritu, znamená to, že zhyb medzi i a $i+1$ je na rovnakej strane ako zhyb medzi j a $j+1$. Preto ak by boli dieliky v permutácii v poradí i, j, $i+1$, $j+1$, zjavne by to znamenalo, že zhyb medzi i a $i+1$ pretína zhyb medzi j a $j+1$.

Stačí nám teda overiť, či naša podmienka pretínania neplatí pre žiadne i, j rovnakej parity. Toto riešenie bude mať v závislosti od implementácie časovú zložitosť $O(n^2)$ až $O(n^3)$ na permutáciu a pamäťovú zložitosť $O(n)$.

Vzorové riešenie

Permutáciu budeme konštruovať zhora nadol. Nech je najvrchnejší dielik i. Z tohto dieliku nám doľava visí dielik $i-1$ a doprava dielik $i+1$. Ak je ďalším dielikom v permutácii $i-1$, tak môžeme jednoducho podložiť

visiaci dielik $i-1$ pod i. Dieliky, ktoré visia pod $i-1$ teraz už budú visieť na pravej strane (kde už budú 2 visiace vrstvy). Analogicky by sme vedeli vyriešiť aj prípad, kde by bol ďalším očakávaným dielikom $i+1$.

Čo ale máme spraviť, ak je ďalším očakávaným dielikom niečo iné, napríklad j? Nuž, na niektorej strane nám tento dielik visí (nie navrchu, ale niekde nižšie). Podložíme ho teda pod posledný pridaný dielik. Ak je j párne (resp. nepárne, podľa toho ako sme začali), tak po podložení na ľavej strane začne visieť $j-1$ a na pravej $j+1$.

Vždy teda máme na ľavej aj na pravej strane niekoľko visiacich vrstiev. V každom kroku sa môžeme zbaviť iba jednej z vnútorných vrstiev. Tu už možno vidno, že na reprezentáciu toho, čo visí naľavo a napravo vieme použiť dva stacky.

Algoritmus následne bude vyzerať tak, že prejdeme celú permutáciu a každý jej dielik samostatne spracujeme. Vždy, keď spracovávame dielik, pozrieme sa, či zrovna nie je na vrchu niektorého stacku. Ak je, môžeme ho zo stacku odstrániť. Ak na vrchu stacku nie je, tak na jednotlivé stacky pridáme jeho susedov (podľa parity).

Časová aj pamäťová zložitosť tohto riešenia je $O(n)$ na permutáciu.

Listing programu (Python)

```
TC = int(input())

for _ in range(TC):
    n = int(input())

    left, right = [], []
    ok = lambda x: 0 < x <= n
    for x in map(int, input().split()):
        xl, xr = x - 1, x + 1
        if x % 2 == 0:
            xl, xr = xr, xl

        if left and left[-1] == xl:
            left.pop()
        elif ok(xl):
            left.append(x)

        if right and right[-1] == xr:
            right.pop()
        elif ok(xr):
            right.append(x)

    print(["Vyter nemozny", "Vyter mozny"][not left and not right])
```

Listing programu (C++)

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

using namespace std;

bool ok(int x, int n) {
    return 0 < x && x <= n;
}

int main() {
    int TC;
    cin >> TC;
    while(TC--) {
        int n;
        stack<int> left, right;
        cin >> n;
        for(int i = 0; i < n; i++) {
```

```
            int x;
            cin >> x;
            int xl = x - 1, xr = x + 1;
            if(x % 2 == 0)
                swap(xl, xr);
            if(!left.empty() && left.top() == xl)
                left.pop();
            else if(ok(xl, n))
                left.push(x);
            if(!right.empty() && right.top() == xr)
                right.pop();
            else if(ok(xr, n))
                right.push(x);
        }
        cout << "Vyter " << (left.empty() && right.empty() ? "mozny" : "nemozny"
            ↪ ) << '\n';
    }

    return 0;
}
```

Dávid

5. Ako Marianka blúdila

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

Cestu, ktorou Marianka prechádza, si môžeme predstaviť ako graf. Políčka sú vrcholy a hrana je medzi nimi práve vtedy, ak medzi nimi Marianka niekedy prešla. Formát vstupu je na grafy pomerne nezvyčajný, ale stačí si uvedomiť, že táto cesta je len akýsi zoznam hrán ktoré sa môžu opakovať. Aby sa nám lepšie pracovalo, graf si potrebujeme uložiť ako zoznam susedov. Ak na to použijeme mapu, nemusíme sa ani trápiť nejakým očíslovaním vrcholov ale môžeme si ich pamätať rovno podľa súradníc.

Teraz ešte potrebujeme zistiť, ako ďaleko je ktorý vrchol od cieľa, aby sme vedeli povedať, či išla Marianka správnym smerom. Na to nám poslúži štandardné prehľadávanie do šírky[1].

Ako posledný krok už len prejdeme cestu tak, ako bola na vstupe a pre každý krok zistíme, či bol správnym smerom, teda či vzdialenosť od ciela je pre nasledujúce políčko najmenšia z pomedzi susedných políčok.

Časová a pamäťová zložitosť

Veľmi ľahko môžeme nahliadnuť, že jediný netriviálny cyklus v tomto riešení je BFS, o ktorom však vieme že má lineárnu zložitosť, teda aj celková časová zložitosť je $O(n)$ od počtu krokov na vstupe. Pamäťová zložitosť je tiež $O(n)$, pretože si potrebujeme pamätať celý vstup, a ukladáme si ho efektívne.

Listing programu (Python)

```
import queue
from collections import defaultdict

n = int(input())

cesta = []
dx = [0,0,1,-1]
dy = [1,-1,0,0]

for _ in range(n):
    x,y = map(int, input().split())
    cesta.append((x,y))

vrcholy = set(cesta)
hrany = {}
```

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

```
for (x,y) in vrcholy:
    hrany[(x,y)]=[]

for i in range(1,n):
    hrany[cesta[i-1]].append(cesta[i])
    hrany[cesta[i]].append(cesta[i-1])

(cx,cy) = cesta[-1]
dist=defaultdict(lambda: 1000000)
q = queue.Queue()
q.put(((cx,cy),0))
while(not q.empty()):
    ((r,c), d) = q.get()
    if (r,c) in dist:
        continue
    dist[(r,c)]=d
    for s in hrany[(r,c)]:
        q.put((s,d+1))

ans=0

z= (0,0)
for u in cesta[1:]:
    m= 1000000
    for v in hrany[z]:
        m=min(m, dist[v])
    if(not dist[u]==m):
        ans+=1
    z=u

print(ans)
```

Listing programu (C++)

```
#include<iostream>
#include<vector>
#include<queue>
#include<unordered_set>
#include<unordered_map>

using namespace std;

struct pair_hash {
    inline std::size_t operator()(const std::pair<int,int> & v) const {
        return v.first*31+v.second;
    }
};

int main(){
    vector<int>dx ={0,0,1,-1};
    vector<int>dy={1,-1,0,0};
    int n;
    cin>>n;
    vector<pair<int,int>> cesta;
    unordered_set<pair<int,int>, pair_hash> vrcholy;
    unordered_map<pair<int,int>, vector<pair<int,int>>, pair_hash> hrany;
    int a,b;
    cin>>a>>b;
```

```
    cesta.push_back({a,b});
    for(int i=1; i<n; i++){
        int a,b;
        cin>>a>>b;
        cesta.push_back({a,b});
        vrcholy.insert({a,b});
        hrany[cesta[i-1]].push_back(cesta[i]);
        hrany[cesta[i]].push_back(cesta[i-1]);
    }
    int ans=0;
    unordered_map<pair<int,int>,int, pair_hash> dist;
    queue<pair<pair<int,int>,int>> q;
    pair<int,int> c = cesta[n-1];
    q.push({c,0});

    while(!q.empty()){
        pair<pair<int,int>,int> t = q.front();
        pair<int,int> v = t.first;
        q.pop();

        if(dist.count(t.first)) continue;
        dist[t.first]=t.second;
        for (int j=0; j<hrany[v].size(); j++)
            q.push({{hrany[v][j]},t.second+1 });
    }
    for(int i=1; i<n; i++){
        int m=987654321;
        for (int j=0; j<hrany[cesta[i-1]].size(); j++)
            m=min(m, dist[hrany[cesta[i-1]][j]]);
        if(dist[cesta[i]]!=m){ans++;}
    }
    cout<<ans<<endl;
}
```

6. Dlhodobý Krtkov plán

12 b za popis, 8 b za program

Prvá vec, ktorú si môžeme uvedomiť, je koľko najmenej času potrebujeme na n úloh. Najmenej času, ktorý vieme venovať najmenej dôležitej úlohe je 0. Potom najmenej času ktorý vieme venovať druhej najmenej dôležitej úlohe je 1. Vo všeobecnosti vieme povedať, že ak máme i-tu najmenej dôležitú úlohu, musíme jej venovať aspoň $i-1$ času. Každej z i-1 menej dôležitých úloh musíme totiž venovať iný čas. n úlohám teda musíme venovať aspoň $0+1+2+...+(n-1)=\frac{(n-1)\cdot n}{2}$

Keď už vieme koľko času najmenej musíme venovať ktorej úlohe, môžeme problém zo zadania mierne upraviť. Zvyšný čas, teda $m-\frac{(n-1)\cdot n}{2}$ chceme rozdeliť medzi n úloh tak, aby dôležitejšia dostala viac **alebo rovnako** ako menej dôležitá. To môže pôsobiť ako zbytočné, no povedie to ku kúsok jednoduchšej implementácií.

Riešenie

Túto úlohu budeme riešiť pomocou dynamického programovania. Cheme zistiť, ako sa dá rozdeliť čas medzi nejaké úlohy. Prvým riešením by mohlo byť, že vyskúšame všetky možnosti koľko času môžeme venovať najmenej dôležitej úlohe, a zvyšok je podobná otázka - o jednu úlohu menej a o nejaký čas menej. Možeme si všimnúť, že ak sa rozhodneme venovať najmenej dôležitej úlohe k času, musíme aj každej ďalšej úlohe venovať aspoň k času. Počet možností teda môžeme spočítať ako $f(u,c)=\sum f(u-1,\ c-u\cdot k)$, kde $f(u,c)$ je počet možností ako vieme rozdeliť medzi u úloh c času. Samozrejme využijeme memoizáciu, aby sme jednu hodnotu nepočítali viackrát.

Takéto riešenie prejde všetky možnosti ako vieme rozdeliť čas medzi úlohy. Pozrime sa na zložitosti. Máme $n\cdot m$ stavov, každý vieme spočítať v $O(m)$. Výsledná zložitosť teda bude $O(nm^2)$.

Vzorové riešenie

V predošlom riešení sme sa vždy rozhodli koľko presne času budeme venovať jednej úlohe, čo viedlo k tomu, že jeden stav sme museli počítať až v zložitosti $O(m)$. V tomto riešení to vylepšíme. Vždy sa rozhodneme, či najmenej dôležitej úlohe ešte budeme pridávať čas, alebo už nie. Presnejšie sa rozhodneme či pridáme ešte 1 čas. Opakovaným pridávaním 1 času vieme pridať ľubovoľne veľa. Ak čas pridáme, musíme ho pridať aj všetkým ostatným úlohám, podobne ako v minulom riešení. Bude teda platiť, že $f(u,c) = f(u,\ c-u)+f(u-1,c)$. Počet rôznych stavov, ktoré potrebujeme spočítať sa nezmenil, ale každý už vieme vypočítať v $O(1)$. Preto celková časová zložitosť nášho algoritmu bude $O(nm)$ a pamäťová $O(nm)$.

Táto pamäťová zložitosť sa dá ešte zlepšiť. Predstavme si, že máme iteratívne naprogramovaný náš algoritmus. Teda máme dvojrozmerné pole, a postupne ho vypĺňame hodnotami $f(u,c)$. Všimnime si, že keď dopĺňame riadok pre nejaké u, nepotrebujeme si pamätať celú tabuľku, ale stačí nám posledný a aktuálny riadok - teda tie, pre u a $u-1$. Takto vieme zmenšiť pamäťovú zložitosť na $O(m)$.

Listing programu (Python)

```
n, m = map((int), input().split())
mod = 1000000007

m -= n*(n-1)//2

if m < 0:
    print("0")
else:
    dp = []
    for i in range(n+1):
        a = []
        for j in range(m+n+1):
            a.append(0)
        dp.append(a)


    dp[0][0] = 1

    for u in range(n):
        for c in range(m+1):
            dp[u][c+n-u] += dp[u][c]
            dp[u][c+n-u] %= mod
            dp[u+1][c] += dp[u][c]
            dp[u+1][c] %= mod

    print(dp[n][m])
```

Listing programu (C++)

```
#include <bits/stdc++.h>
#define MOD 1000000007

using namespace std;

int main()
{

    long long n, m;
    cin >> n >> m;

    m -= n*(n-1)/2;

    if(m < 0 )
```

```
    {
        cout << "0\n";
        return 0;
    }

    vector<vector<long long>> DP(n+1, vector<long long>(m+n+1));
    DP[0][0] = 1;

    for(int u = 0; u < n; u++)
    {
        for(int c = 0; c <= m; c++)
        {
            DP[u][c+n-u] += DP[u][c];
            DP[u][c+n-u] %= MOD;
            DP[u+1][c] += DP[u][c];
            DP[u+1][c] %= MOD;
        }
    }

    cout << DP[n][m] << '\n';

    return 0;
}
```

Maťo

7. Krášlenie cestičiek

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

V prvom rade si treba uvedomiť, že v optimálnom riešení sa nám oplatí krášliť vždy iba jednu cestičku. Ak by sme krášlili viacero, mohli by sme miesto toho o rovnako veľa skrášliť najlacnejšiu z nich a určite by nám na to vyšli peniaze.

Môžeme teda skúšať možnosti, ktorá cestička to bude. Vždy si vyberieme jednu cestičku a skrášlime ju najviac ako vieme. S touto úpravou nám už zostáva iba vybrať cestičky s najmenšou škaredosťou. To znamená nájsť najlacnejšiu kostru grafu.

Iná možnosť je hľadať najlacnejšiu kostru, ktorá navyše obsahuje danú cestičku (bez úpravy škaredosti).

Najlacnejšia kostra sa dá nájsť v $O(m \log n)$. To musíme spraviť pre každú hranu, čím dostávame časovú zložitosť $O(m^2 \log n)$. Pamäťová zložitosť je $O(m)$.

Listing programu (C++)

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

struct Edge
{
    int a;
    int b;
    int c;
    int w;
    int id;
};

vector<vector<Edge>> kostra(vector<vector<Edge>> G)
{
    int n = G.size();

    struct GreaterWeight
    {
        bool operator()(Edge e1, Edge e2)
```

```
        {
            return e1.w > e2.w;
        }
    };
    priority_queue<Edge, vector<Edge>, GreaterWeight> Q;

    vector<vector<Edge>> kostra(n);

    vector<bool> added(n, false);
    added[0] = true;
    for (Edge e : G[0])
        Q.push(e);
    while (Q.size())
    {
        Edge e = Q.top();
        Q.pop();
        if (added[e.b])
            continue;
        kostra[e.a].push_back(e);
        added[e.b] = true;
        for (Edge e1 : G[e.b])
            Q.push(e1);
    }
    return kostra;
}

int main()
{
    int n, m;
    cin >> n >> m;
    vector<Edge> all_edges;
    for (int i = 0; i < m; i++)
    {
        int a, b, c, w;
        cin >> a >> b >> c >> w;
        all_edges.push_back({a, b, c, w, i});
    }
    int S;
    cin >> S;

    long long best_score = LONG_LONG_MAX;
    vector<Edge> best_edges;

    for (int i = 0; i < m; i++)
    {
        vector<vector<Edge>> G(n);
        for (int j = 0; j < m; j++)
        {
            Edge e = all_edges[j];
            if (i == j)
                e.w -= S / e.c;
            Edge oposite = e;
            swap(oposite.a, oposite.b);
            G[e.a].push_back(e);
            G[oposite.a].push_back(oposite);
        }
        vector<vector<Edge>> K = kostra(G);
        vector<Edge> edges;
```

```
        long long nespokojnost = 0;
        for (auto x : K)
            for (Edge e : x)
            {
                edges.push_back(e);
                nespokojnost += e.w;
            }

        if (nespokojnost < best_score)
        {
            best_score = nespokojnost;
            best_edges = edges;
        }
    }
    cout << best_score << "\n";
    for (Edge e : best_edges)
        cout << e.id << " " << e.w << "\n";
}
```

Lepšie riešenie

Povedzme, že sme si zvolili hranu ktorú skúsime skrášliť a chceme nájsť najlacnejšiu kostru ktorá ju obsahuje. Dá sa ukázať, že takúto kostru vieme dostať z najlacnejšej kostry (celkovej) vymenením najviac jednej hrany.

Najlacnejšiu kostru vieme nájsť tak, že postupne pridávame vždy najlacnejšiu hranu ktorá spojí vrcholy, ktoré ešte nie sú v jednom komponente. Najlacnejšiu kostru s nejakou konkrétnou hranou nájdeme podobne, len ešte na začiatku hneď pridáme túto hranu. Pozrime sa, ako sa tieto kostry budú líšiť po každom kroku.

Začíname s dvoma prázdnymi grafmi A a B. Do B ešte pridáme hranu medzi vrcholmi u a v. V prvých niekoľkých krokoch sa budú líšiť iba v tejto hrane. To tiež znamená, že tieto grafy budú tvorené rovnakými komponentmi, až na to, že A bude mať u, v v dvoch rôznych komponentoch a B ich bude mať spojené. Rozdiel nastane, až keď sa rozhodneme pridať hranu medzi vrcholmi, ktoré sú v jednom grafe v rôznych komponentoch a v duhom v rovnakom. To nutne musí byť hrana, ktorá v A spája komponenty obsahujúce u a v, keďže všetky ostatné komponenty sú rovnaké. Keď túto hranu pridáme do A, dosiahneme úplne rovnaké komponenty (tým aké vrcholy obsahujú) v A aj B. Preto každá ďalšia pridaná hrana do A aj B bude rovnaká. To znamená, že výsledné kostry budú rovnaké, okrem hrany, ktorú sme na začiatku pridali do B a jednej hrany ktorú sme pridali do A.

Môžeme teda použiť postup, že najprv nájdeme celkovú najlacnejšiu kostru a potom skúšame, ktorú hranu chceme skrášliť. Túto hranu pridáme ku kostre a jednu hranu z nej dáme preč. Spravíme to tak, aby sme získali najlacnejšiu kostru s touto hranou.

Hrana, ktorú odstránime, musí byť nejaká hrana na ceste medzi tými vrcholmi, kam pridávame hranu, inak by vznikol cyklus. Môže to byť ľubovoľná z nich. Najlepšie riešenie dostaneme, ak odstránime najškaredšiu z nich. Zatiaľ to spravíme tak, že prejdeme postupne všetky hrany na tejto ceste a nájdeme maximum.

Nájsť na začiatku najlacnejšiu kostru nám trvá $O(m \log n)$. Potom pre každú hranu potrebujeme prejsť cestu medzi dvoma vrcholmi v kostre, čo môže trvať $O(n)$. Z toho dostávame časovú zložitosť $O(nm)$. Pamäťová zložitosť je znova $O(m)$.

Listing programu (C++)

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

struct Edge
{
    int a;
    int b;
    int c;
    int w;
    int id;
};
```

```
vector<vector<Edge>> kostra(vector<vector<Edge>> G)
{
    int n = G.size();

    struct GreaterWeight
    {
        bool operator()(Edge e1, Edge e2)
        {
            return e1.w > e2.w;
        }
    };
    priority_queue<Edge, vector<Edge>, GreaterWeight> Q;

    vector<vector<Edge>> kostra(n);

    vector<bool> added(n, false);
    added[0] = true;
    for (Edge e : G[0])
        Q.push(e);
    while (Q.size())
    {
        Edge e = Q.top();
        Q.pop();
        if (added[e.b])
            continue;
        kostra[e.a].push_back(e);
        added[e.b] = true;
        for (Edge e1 : G[e.b])
            Q.push(e1);
    }
    return kostra;
}

struct MaxPath
{
    vector<Edge> hore;
    vector<int> depth;
    int n;

    Edge max_weight(Edge e1, Edge e2)
    {
        if (e2.w > e1.w)
            return e2;
        else
            return e1;
    }

    void dfs_depth(vector<vector<Edge>> &tree, int i, int d = 0)
    {
        depth[i] = d;
        for (Edge e : tree[i])
            dfs_depth(tree, e.b, d + 1);
    }

    MaxPath(vector<vector<Edge>> &tree)
    {
        n = tree.size();
```

```
        hore.resize(n, {-1, 0, 0, 0, 0});
        for (auto x : tree)
            for (Edge e : x)
                hore[e.b] = e;
        depth.resize(n);
        dfs_depth(tree, 0);
    }

    Edge find_max(int a, int b)
    {
        Edge max_edge = {0, 0, 0, 0, 0};
        if (depth[a] > depth[b])
            swap(a, b);

        while (depth[b] > depth[a])
        {
            max_edge = max_weight(max_edge, hore[b]);
            b = hore[b].a;
        }
        while (a != b)
        {
            max_edge = max_weight(max_edge, hore[a]);
            a = hore[a].a;
            max_edge = max_weight(max_edge, hore[b]);
            b = hore[b].a;
        }
        return max_edge;
    }
};

int main()
{
    int n, m;
    cin >> n >> m;
    vector<vector<Edge>> G(n);
    vector<Edge> all_edges;
    for (int i = 0; i < m; i++)
    {
        int a, b, c, w;
        cin >> a >> b >> c >> w;
        all_edges.push_back({a, b, c, w, i});
        G[a].push_back({a, b, c, w, i});
        G[b].push_back({b, a, c, w, i});
    }
    int S;
    cin >> S;

    vector<vector<Edge>> K = kostra(G);
    long long nespokojnost = 0;
    for (auto x : K)
        for (Edge e : x)
            nespokojnost += e.w;

    MaxPath max_path(K);
    long long best = nespokojnost;
    Edge to_remove;
    Edge to_add;
    for (Edge e : all_edges)
```

```
    {
        Edge worst = max_path.find_max(e.a, e.b);
        long long result = nespokojnost - worst.w + e.w - S / e.c;
        if (result <= best)
        {
            best = result;
            to_remove = worst;
            to_add = e;
        }
    }
    cout << best << "\n";
    for (auto x : K)
        for (Edge e : x)
        {
            if (e.id == to_remove.id)
                continue;
            cout << e.id << " " << e.w << "\n";
        }
    cout << to_add.id << " " << to_add.w - S / to_add.c << "\n";
}
```

Vzorové riešenie

Zlepšiť vieme hľadanie maximálnej hrany na ceste v strome. Začneme tým, že si ho zakoreníme v nejakom ľubovoľnom vrchole. Potom si pre každý vrchol predpočítame skoky o $1, 2, 4, 8, \ldots$ hore, tak ako keď hľadáme najnižšieho spoločného predka. Pre tieto skoky si okrem toho, na akom vrchole skončíme, pamätáme aj najlacnejšiu hranu na ceste ktorú sme preskočili. To vieme jednoducho vypočítať dynamikou od koreňa k listom: skok o $2k$ je ako skok o k a potom ešte raz o k od vrchola na ktorom pristaneme. Okrem toho si potrebujeme vypočítať hĺbku každého vrchola.

Maximálnu hranu na ceste medzi dvoma vrcholmi nájdeme tak, že najprv nájdeme ich najnižšieho spoločného predka. Na to použijeme skoky ktoré sme si predpočítali. Najprv z vrchola s väčšou hĺbkou preskáčeme na vrchol s rovnakou hĺbkou ako druhý vrchol, na čo nám postačí najviac $\log n$ skokov. Potom binárne vyhľadáme najnižšieho spoločného predka: ak skok z oboch vrcholov pristane na rovnakom vrchole sme vysoko, ak na rôznych sme nízko.

Už sa stačí pozrieť na to, ktoré skoky sme použili aby sme sa dostali od každého z vrcholov k ich spoločnému predkovi. Z týchto skokov zoberieme maximum, čím dostaneme maximálnu hranu na celej ceste.

Takto nám stačí vypočítať najlacnejšiu kostru len raz na začiatku v čase $O(m \log n)$. Potom si vypočítame skoky z každého vrchola. Každý skok vypočítame v konštantom čase a skokov je najviac $n \log n$. Následne pre každú hranu vieme nájsť maximálnu (najškaredšiu) hranu na vymenenie v čase $O(\log n)$. Celkovo má toto riešenie časovú zložitosť $O(m \log n)$. Pamäťová zložitosť je $O(m + n \log n)$ keďže si musíme navyše pamätať všetky skoky.

Listing programu (C++)

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

struct Edge
{
    int a;
    int b;
    int c;
    int w;
    int id;
};

vector<vector<Edge>> kostra(vector<vector<Edge>> G)
{
```

```
    int n = G.size();

    struct GreaterWeight
    {
        bool operator()(Edge e1, Edge e2)
        {
            return e1.w > e2.w;
        }
    };
    priority_queue<Edge, vector<Edge>, GreaterWeight> Q;

    vector<vector<Edge>> kostra(n);

    vector<bool> added(n, false);
    added[0] = true;
    for (Edge e : G[0])
        Q.push(e);
    while (Q.size())
    {
        Edge e = Q.top();
        Q.pop();
        if (added[e.b])
            continue;
        kostra[e.a].push_back(e);
        added[e.b] = true;
        for (Edge e1 : G[e.b])
            Q.push(e1);
    }
    return kostra;
}

struct Lca
{
    struct Jump
    {
        int to;
        Edge max_edge;
    };

    vector<vector<Jump>> jumps;
    vector<int> depth;
    int n;

    Edge max_weight(Edge e1, Edge e2)
    {
        if (e2.w > e1.w)
            return e2;
        else
            return e1;
    }

    void dfs_init(vector<vector<Edge>> &tree, int i, int d = 0, Jump up = {-1,
        ↪ {}})
    {
        depth[i] = d;
        if (up.to != -1)
        {
            jumps[i].push_back(up);
```

```
            for (int lvl = 0;; lvl++)
            {
                int next = jumps[i][lvl].to;
                if (lvl >= jumps[next].size())
                    break;
                Edge max_edge = max_weight(jumps[i][lvl].max_edge, jumps[next][
                    ↪ lvl].max_edge);
                jumps[i].push_back({jumps[next][lvl].to, max_edge});
            }
        }

        for (Edge e : tree[i])
            dfs_init(tree, e.b, d + 1, {i, e});
    }

    Lca(vector<vector<Edge>> &tree)
    {
        n = tree.size();
        jumps.resize(n);
        depth.resize(n);
        dfs_init(tree, 0);
    }

    Edge find_max(int a, int b)
    {
        Edge max_edge = {0, 0, 0, 0, 0};
        if (depth[a] > depth[b])
            swap(a, b);

        for (int i = jumps[b].size() - 1; i >= 0; i--)
        {
            if (i >= jumps[b].size())
                continue;
            if (depth[jumps[b][i].to] < depth[a])
                continue;
            max_edge = max_weight(max_edge, jumps[b][i].max_edge);
            b = jumps[b][i].to;
        }

        if (a == b)
            return max_edge;

        for (int i = jumps[a].size() - 1; i >= 0; i--)
        {
            if (i >= jumps[a].size())
                continue;
            if (jumps[a][i].to == jumps[b][i].to)
                continue;
            max_edge = max_weight(max_edge, jumps[a][i].max_edge);
            a = jumps[a][i].to;
            max_edge = max_weight(max_edge, jumps[b][i].max_edge);
            b = jumps[b][i].to;
        }
        max_edge = max_weight(max_edge, jumps[a][0].max_edge);
        max_edge = max_weight(max_edge, jumps[b][0].max_edge);
        return max_edge;
    }
};
```

```
int main()
{
    int n, m;
    cin >> n >> m;
    vector<vector<Edge>> G(n);
    vector<Edge> all_edges;
    for (int i = 0; i < m; i++)
    {
        int a, b, c, w;
        cin >> a >> b >> c >> w;
        all_edges.push_back({a, b, c, w, i});
        G[a].push_back({a, b, c, w, i});
        G[b].push_back({b, a, c, w, i});
    }
    int S;
    cin >> S;

    vector<vector<Edge>> K = kostra(G);
    long long nespokojnost = 0;
    for (auto x : K)
        for (Edge e : x)
            nespokojnost += e.w;

    Lca lca(K);
    long long best = nespokojnost;
    Edge to_remove;
    Edge to_add;
    for (Edge e : all_edges)
    {
        Edge worst = lca.find_max(e.a, e.b);
        long long result = nespokojnost - worst.w + e.w - S / e.c;
        if (result <= best)
        {
            best = result;
            to_remove = worst;
            to_add = e;
        }
    }
    cout << best << "\n";
    for (auto x : K)
        for (Edge e : x)
        {
            if (e.id == to_remove.id)
                continue;
            cout << e.id << " " << e.w << "\n";
        }
    cout << to_add.id << " " << to_add.w - S / to_add.c << "\n";
}
```

paulinia

8. Areál zavlažovačov

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

Hrubá sila

Keby mal Adam veľa času a trpezlivosti, mohol by na riešenie ísť hrubou silou: vždy, keď ho začne zaujímať nejaké miesto, pre každý zavlažovač samostatne skontroluje či miesto zavlažuje. To vie zistiť v konštantom čase.

Takto vie zistiť vlahu všetkých miest v $O(nm)$ čase a $O(n)$ pamäti.
Žiaľ, hrubá sila stačí len na prvú a tretiu sadu.

Malé pole

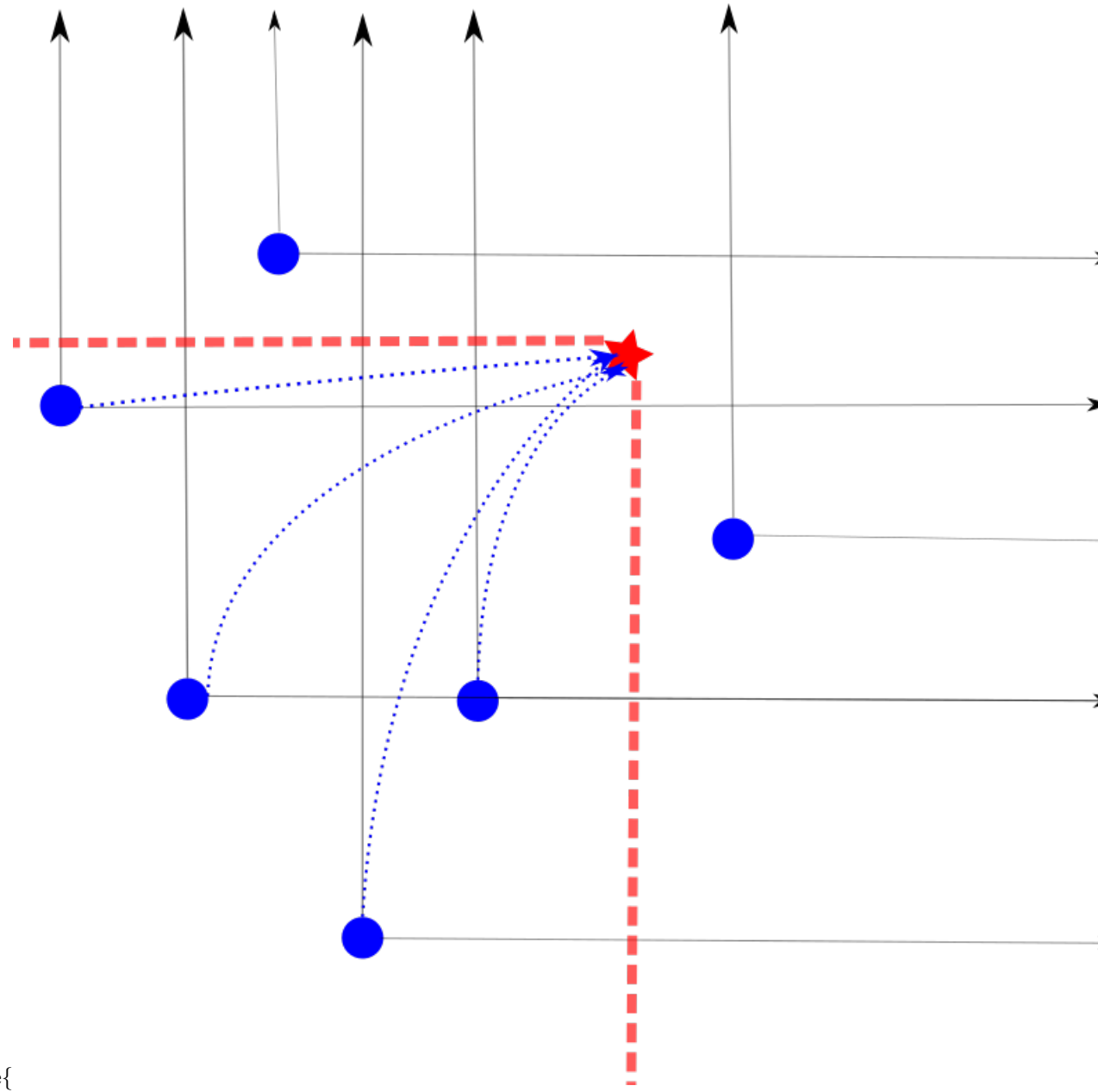
Čo ak je Adamove pole malé? Potom by si Adam mohol spočítať odpoveď pre *každé* miesto na poli, a potom vie odpovedať na otázku o akomkoľvek mieste, ktoré by ho mohlo zaujímať.

Ako to vie spočítať? Mohol by napríklad použiť 2D prefixové súčty. Viac o tom, ako ich viete použiť sa už dozviete v KSP kuchárke[2].

Presnejšie, Adam si chce spraviť prefixové súčty pre každý zo štyroch smerov, a vlahu na mieste záujmu potom dostane ako sumu vláh zo všetkých smerov. Toto vie Adam spočítať v časovej zložitosti $O(m+n+(\max(x_i, y_i))^2)$, a v pamäti $O((\max(x_i, y_i))^2)$.

K rýchlemu riešeniu

Hrubá sila očividne nestačí. Skúsme sa najskôr zamyslieť nad prípadom, že všetky zavlažovače sú orientované rovnako. Pozrime sa na miesto ktoré Adama momentálne zaujíma. Ktoré zavlažovače ho zasahujú?



\centerline{

[2] https://www.ksp.sk/kucharka/2d_prefixove_sumy/

Ako môžme vidieť na ilustračnom obrázku, vyznačené miesto (červeným) zavlažujú všetky zavlažovače v červenom rohu, takže aby sme rýchlo zistili vlahu, chceli by sme vedieť zistiť rýchlo spočítať sumu vláh zavlažovačov umiestnených v danom rohu. Skúsenejšiemu riešiteľovi už možno niečo hovorí, že by sa Adamovi zišiel intervalový strom.

Vieme miesta vopred

Na prvý pohľad by to mohlo vyzerať, že bude treba dvojrozmerný intervalový strom, ale ukáže sa, že ten vôbec netreba. Najskôr sa pozrime na prípad, že Adam vie, ktoré miesta ho zaujímajú vopred.

Zoberme si prípad, že všetky zavlažovače smerujú hore a doprava (ako na obrázku hore). Utrieďme si ich v smere x-súradnice (zľava doprava), a utrieďme si tiež všetky miesta v smere x-súradníc. Potom postupne prechádzajme miesta a zavlažovače zľava doprava a udržujme si dátovú štruktúru, ktorá nám bude hovoriť, pre každú y-súradnicu, koľko je suma vláh zo všetkých zavlažovačov, ktoré sme videli doteraz, na tejto súradnici. Všimnime si (ak sú všetky orientované hore a doprava), že pokiaľ máme spracované všetky zavlažovače naľavo alebo s rovnakou x-súradnicou ako miesto, ktoré nás zaujíma, tak potom vlaha na mieste záujmu je suma vláh všetkých spracovaných zavlažovačov s y-pozíciou menšou alebo rovnou ako miesto záujmu.

Takže nám na efektívne riešenie stačí vedieť spočítať a aktualizovať sumy na intervaloch. To ide napríklad intervalovým stromom[3].

Veľmi veľké pole

Ak by sme si chceli v intervalovom strome pamätať vlahy na všetkých y-súradniciach, potom nám riešenie zaberie $O((n+m)\log \max_i y_i + \max_i y_i)$ času a $O(n+m+\max_i y_i)$ pamäte. Pre pole veľkosti 10^9 nám to nestačí.

Ako to zlepšiť? Všimnime si, že nás zaujíma iba relatívna pozícia zavlažovačov a miest, a tých je málo. Teda si zmeňme súradnicovú sústavu pomocou *kompresie súradníc*. To znamená, utrieďme si všetky x a y súradnice na vstupe, a potom zmeňme súradnice vstupu tak, že najmenšia x-súradnica sa zmení na 0, druhá najmenšia na 1, atď. Takto sa nám zmenší počet y-súradníc ktoré nás zaujímajú na dostatočne málo a dostaneme (offline) riešenie v čase $O((n+m)\log n)$ a pamäti $O(m+n)$.

Online riešenie

Čo ak nám miesta prichádzajú postupne?

Nevieme použiť predchádzajúce riešenie: miesta nám neprichádzajú utriedene, potrebovali by sme sa vedieť "dostať" ku intervalovému stromu v rôznych *časoch*, napríklad "po pridaní všetkých zavlažovačov s x-súradnicou menšou než 5462".

Alebo by sme to mohli urobiť? Ukazuje sa že áno, riešením je *perzistentný intervalový strom*. Vysvetlenie ako na perzistentný intervaláč vám nechávam napríklad v tomto videovzoráku[4].

Riešenie pomocou perzistentného intervaláča funguje v čase $O((m+n)\log n)$ a v pamäti $O(n\log n)$ a dá vám plný počet bodov.

Čo s inými smermi?

Pozorný čitateľ ešte zbystrí: hovorím, že už máme riešenie, ale starali sme sa zatiaľ len o jeden smer? Veď zavlažovače vedia byť aj inak orientované!

Ukáže sa, že nám stačí roztriediť si zavlažovače do skupín podľa smeru zavlažovania, a pre každý mať separátny intervaláč. Riešenie vieme použiť pre každý smer rovnaké, len si je treba situáciu vhodne rotovať. Toto nám pridá do časovej a pamäťovej zložitosti iba konštantný faktor.

Listing programu (C++)

```
#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
```

[3] https://www.ksp.sk/kucharka/intervalovy_strom/
[4] https://www.youtube.com/watch?v=bmSa2HAPtE8

```
#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;
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 << "}";
    return out;
}

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

    out << "}"; return out;
}

template <class T>
ostream& operator<<(ostream& out, const vector<T>& v) {
  FOR(i, v.size()){
    if(i) out << " ";
    out << v[i];
  }
  out << endl;
  return out;
}

bool ccw(point p, point a, point b) {
  if((conj(a - p) * (b - p)).imag() <= 0) return false;
  else return true;
}
```

```
struct zavlahovac {
    int x, y;
    int vlaha;
    string dir;

    const bool operator<(const zavlahovac &z) const {
        if (dir[0] == 'H') {
            return y < z.y;
        }
        else return y > z.y;
    }
};

struct node {
    int start, range;
    int sx, ex;
    int sums;
    node *prvy, *druhy;

    node (int st, int rn, int n, vector<int> &coor) {
        sx = (st < n ? coor[st] : infinity);
        ex = (st + rn >= n ? infinity : coor[st + 1]);
        start = st;
        range = rn;
        if (rn == 1) {
            sums = 0;
            prvy = NULL;
            druhy = NULL;
        }
        else {
            prvy = new node(st, rn / 2, n, coor);
            druhy = new node(st + rn / 2, rn / 2, n, coor);
            sums = 0;
        }
    }

    node (node *old, node *newprvy, node *newdruhy, int newval) {
        start = old -> start;
        range = old -> range;
        sx = old -> sx;
        ex = old -> ex;
        sums = newval;
        prvy = newprvy;
        druhy = newdruhy;
    }

    node *update(int p, int vl) {
        if (range == 1) {
            return new node(this, NULL, NULL, sums + vl);
        }
        if (druhy -> start <= p) return new node(this, prvy, druhy->update(p, vl
            ↪ ), sums + vl);
        return new node(this, prvy->update(p, vl), druhy, sums + vl);
    }

    int query(int st, int end) {
        DBG cout << "In [" << sx << "; " << ex << "] [" << start << " range: "
            ↪ << range << "] sums = " << sums << " query(" << st << ", " << end
```

```
            ↪ << ")" << endl;
        if (start >= end) return 0;
        if (start + range <= st) return 0;
        if (start >= st && start + range <= end) return sums;
        return prvy->query(st, end) + druhy-> query(st, end);
    }

    void print(string k) {
        cout << k << "In [" << start << "; " << start + range << "] with [" <<
            ↪ sx << "; " << ex << "] sums = " << sums << endl;
        if (range != 1) {
            cout << k <<  "PRVY {\n";
            prvy -> print(k + "  ");
            cout << k << "} DRUHY {\n";
            druhy -> print(k + "  ");
            cout << k << "}" << endl;
        }
    }
};

struct intervalac {
    int N;
    int realn;
    map<int, int> xcoors, ycoors;
    vector<node *> roots;
    string dir;

    intervalac(int n, vector<zavlahovac> &zvs) {
        if (!n) {
            N = 0;
            return;
        }
        N = pow(2, ceil(log2(n)));
        realn = n;
        dir = zvs[0].dir;
        DBG cout << "intervalac " << dir << ":"  << endl;
        sort(zvs.begin(), zvs.end());
        vector<int> xunsort;
        FOR(i, n) {
            ycoors[zvs[i].y] = i;
            xunsort.push_back(zvs[i].x);
        }
        sort(xunsort.begin(), xunsort.end());
        FOR(i, n) {
            xcoors[xunsort[i]] = i;
        }
        roots.push_back(new node(0, N, n, xunsort));
        FOR(i, n) {
            DBG cout << "update " << i << " on pos " << xcoors[zvs[i].x] << endl
                ↪ ;
            roots.push_back(roots[i] -> update(xcoors[zvs[i].x], zvs[i].vlaha));
        }
        if (dir[0] == 'H') ycoors[infinity] = n;

        DBG {
            FOR(i, n + 1) {
                cout << "Iteration " << i << ":" << endl;
                roots[i] -> print("");
```

```
            }
        }

    }

    int query(int x, int y) {
        if (!N) return 0;
        if (dir[0] == 'H') {
            auto it = ycoors.upper_bound(y);
            int upto = (it == ycoors.begin() ? 0 : (--it)->second + 1);
            if (dir[1] == 'L') {
                auto itx = xcoors.lower_bound(x);
                return (itx == xcoors.end() ? 0 : roots[upto] -> query(itx->
                    ↪ second, N));
            }
            else {
                DBG cout << "[" << x << "; " << y << " ] In " << dir << " upto =
                    ↪  " << upto << " in " << ycoors << endl;
                auto itx = xcoors.upper_bound(x);
                return roots[upto] -> query(0, (itx == xcoors.end() ? N : itx ->
                    ↪  second));
            }
        }
        auto it = ycoors.lower_bound(y);
        int yid = (it == ycoors.end() ? 0 : it -> second + 1);
        if (dir[1] == 'L') {
            auto itx = xcoors.lower_bound(x);
            return (itx == xcoors.end() ? 0 : roots[yid] -> query(itx->second, N
                ↪ ));
        }
        auto itx = xcoors.upper_bound(x);
        DBG cout << "query (" << x << "; " << y << ") In " << dir << " yid = "
            ↪ << yid << " in " << ycoors << " and itx = " << (itx == xcoors.end
            ↪ () ? N : itx->second) << " from " << xcoors << endl;
        return roots[yid] -> query(0, (itx == xcoors.end() ? N : itx -> second))
            ↪ ;
    }
};

int main() {
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);
    int n = get<int>();
    vector<zavlahovac> DL, DP, HP, HL;
    FOR(i, n) {
        int x, y;
        int vl;
        string smer;
        cin >> x >> y >> vl >> smer;
        if (smer == "DL") {
            DL.push_back({x, y, vl, smer});
        }
        else if (smer == "DP") {
            DP.push_back({x, y, vl, smer});
        }
        else if (smer == "HP") {
            HP.push_back({x, y, vl, smer});
        }
```

```
        else HL.push_back({x, y, vl, smer});
    }

    intervalac iDL(DL.size(), DL);
    intervalac iDP(DP.size(), DP);
    intervalac iHL(HL.size(), HL);
    intervalac iHP(HP.size(), HP);

    int m = get<int>();
    int k = get<int>();

    int prev_ans = 0;

    FOR(i, m) {
        int x, y;
        cin >> x >> y;
        x ^= prev_ans * k;
        y ^= prev_ans * k;

        DBG cout << "Query " << i << ": " << x << " " << y << endl;
        int idl = iDL.query(x, y);
        int idp = iDP.query(x, y);
        int ihl = iHL.query(x, y);
        int ihp = iHP.query(x, y);

        DBG cout << "DL: " << idl << " DP: " << idp << " HL: " << ihl << " HP: "
            ↪  << ihp << endl;

        prev_ans = idl + idp + ihl + ihp;
        cout << prev_ans << endl;
    }
}
```