A-II-1 Reverzy

Úloha má celkom jednoduché riešenie s kvadratickou časovou zložitosťou: triedenie BubbleSort. Jednoducho dokola prechádzame poľom a vždy, keď stretneme vedľa seba stojacu dvojicu čísel v poradí väčšie-menšie, použitím jedného reverzu ich vymeníme. Takéto riešenie však nie je optimálne, keďže v najhoršom prípade potrebuje až kvadratický počet reverzov.

Lineárny počet reverzov

Pomerne priamočiaro sa dá navrhnúť riešenie, ktoré si vystačí s lineárnym počtom reverzov. Základná myšlienka bude jednoduchá: nájdeme najväčší prvok v celom poli. Nech sa nachádza na pozícii i. Na pozíciu $n-1$, kam patrí, ho vieme dostať jedným reverzom: jednoducho reverzneme celý úsek od i po $n-1$. Od tejto chvíle môžeme tento prvok nechať na mieste a spokojne naň zabudnúť. Dostali sme teda pôvodný problém, ale už len s $n-1$ prvkami. Zvyšok riešenia teda bude vyzerať presne rovnako: kým sme ešte neusporiadali celé pole, nájdeme najväčší prvok v ešte neusporiadanej časti poľa a jedným reverzom ho presunieme na jej koniec.
Toto riešenie bude v najhoršom prípade potrebovať spraviť $n-1$ reverzov, ale jeho časová zložitosť je až kvadratická – v každej iterácii potrebujeme aj nájsť maximum spomedzi ešte neusporiadaných čísel, aj si na našej kópii poľa odsimulovať každý reverz, ktorý spravíme, aby sme vedeli, ktorý prvok sa kam presunul.

Kompresia súradníc

Vo zvyšku tohto vzorového riešenia si ukážeme riešenia, ktoré tiež potrebujú lineárny počet reverzov, dokážu ich ale robiť efektívnejšie.
Obe lepšie riešenia budú mať spoločný prvý krok. Začneme tým, že prvky zadaného poľa si usporiadame a v poradí od najmenšieho po najväčší ich nahradíme číslami od 0 po $n-1$. Tým sme si úlohu trochu zjednodušili: pre každé i teraz platí, že číslo i potrebujeme dostať na pozíciu i.
Tento krok vieme spraviť v čase $O(n \log n)$ – napríklad tak, že si usporiadame sadu záznamov tvaru „hodnota $A[i]$ je na pozícii i“, alebo tak, že usporiadame kópiu poľa zo vstupu a potom na prečíslovanie pôvodného poľa použijeme buď binárne vyhľadávanie v usporiadanej kópii, alebo vhodnú dátovú štruktúru.

Riešenie založené na výmenách

Keď sme prvky pôvodného poľa nahradili číslami od 0 po $n-1$, dostali sme *permutáciu*, ktorú si označíme P. K nej si môžeme v lineárnom čase zostrojiť *inverznú permutáciu* Q predpisom $\forall i : Q[P[i]] = i$. (Permutácia Q je pole, kde mám ku každej hodnote uložené, kde v poli P sa nachádza.)
Teraz si už len stačí uvedomiť, že pomocou dvoch reverzov vieme vymeniť ľubovoľnú dvojicu prvkov v našom poli: ak chceme vymeniť $P[x]$ a $P[y]$ (kde $x < y$), spravíme najskôr reverz úseku od x po y a potom (ak treba) reverz úseku od $x+1$ po $y-1$.
No a takúto zmenu v poli si už vieme odsimulovať v konštantnom čase. A čo je dôležitejšie, v konštantnom čase ju vieme odsimulovať aj v poli Q – čiže naďalej budeme vedieť, ktorú hodnotu máme na ktorej pozícii.
A toto nám už stačí na kompletné riešenie. Najskôr v čase $O(n \log n)$ zmeníme čísla v pôvodnom poli na čísla 0 až $n-1$, potom v lineárnom čase zostrojíme inverznú permutáciu, a na záver postupne pre každé x od $n-1$ po 1 umiestnime pomocou dvoch reverzov číslo x na pozíciu x. Dokopy teda dostávame riešenie s $O(n)$ reverzmi a časovou zložitosťou $O(n \log n)$.

Listing programu (Python)

```
def najdi(usporiadane_pole, prvok):
    # predpokladame ze prvok sa v poli nachadza
    # predstavime si, ze usporiadane_pole[N] = nekonecno
    # invariant: usporiadane_pole[lo] <= prvok < usporiadane_pole[hi]
    lo, hi = 0, len(usporiadane_pole)
    while hi - lo > 1:
        med = (lo + hi) // 2
        if usporiadane_pole[med] <= prvok: lo = med
        else:                              hi = med
    return lo
```

```
# nacitame vstup
N = int( input() )
A = [ int(_) for _ in input().split() ]

# usporiadame vstup a zostrojime permutacie P a Q
B = sorted(A)
P = [ najdi(B,a) for a in A ]
Q = [ None for n in range(N) ]
for n in range(N): Q[ P[n] ] = n

# postupne robime reverzy, ktore spravne umiestnia prvky N-1 az 1
for x in reversed(range(1,N)):
    kde, kam = Q[x], x
    if kde == kam: continue
    print(kde, kam)
    if kam > kde+2:
        print(kde+1, kam-1)

    P[kde], P[kam] = P[kam], P[kde]
    Q[ P[kde] ] = kde
    Q[ P[kam] ] = kam
```

Riešenie založené na posúvaní

Ukážeme si ešte druhé, rovnako dobré riešenie, avšak založené na inej myšlienke. Presnejšie, ukážeme si, ako vylepšiť a pomocou vhodných dátových štruktúr zrýchliť naše prvé riešenie, ktoré používalo $n-1$ reverzov ale potrebovalo kvadratický čas. Tentokrát namiesto výmeny dvoch prvkov budeme pomocou dvoch reverzov simulovať posunutie nejakého prvku.
Ak máme číslo $n-1$ na pozícii $i < n-1$, spravíme postupne dva reverzy: jeden na úseku od i po $n-1$ a potom druhý na úseku od i po $n-2$. Druhým reverzom dáme všetky prvky v danom úseku späť do ich pôvodného poradia. Výsledný efekt týchto dvoch reverzov je teda rovnaký, ako keby sme najväčšie číslo vybrali z poľa a presunuli ho na koniec. Všetky čísla, ktoré boli naľavo od neho, ostali na svojich pôvodných pozíciách, a všetky ostatné čísla sú teraz o pozíciu vľavo oproti tej, kde začínali.
Ako bude naše riešenie pokračovať ďalej? Predpokladajme, že už sme niekoľko najväčších čísel presunuli na správne miesta na konci poľa. V nasledujúcom kroku by sme chceli to isté spraviť s číslom x. Na to ale potrebujeme vedieť, kde sa teraz nachádza. Ako to zistiť?
Vieme, kde sa x nachádzalo na začiatku. No a po každej dvojici reverzov buď ostalo na mieste, alebo sa posunulo o pozíciu doľava. Koľkokrát sa posunulo doľava? Toľkokrát, koľko väčších čísel sa v pôvodnom poli nachádzalo naľavo od x. Ak by sme poznali túto hodnotu, vedeli by sme si vypočítať, kde sa x práve nachádza, a teda aj to, aké reverzy potrebujeme spraviť, aby sme ho dostali na správne miesto.
Celé riešenie si teda môžeme zhrnúť nasledovne:

1. Prečíslujeme prvky poľa na 0 až $n-1$. Pre každé x si zapamätáme, na ktorej pozícii sa teraz nachádza.
2. Pre každé x zistíme, koľko väčších čísel je naľavo od neho.
3. Postupne od $x = n-1$ po $x = 1$ umiestnime číslo x na správne miesto.

Už vieme, že krok 1 vieme spraviť v čase $O(n \log n)$ a krok 3 dokonca v lineárnom čase. Ostáva nám doplniť detaily o kroku 2.
Existuje viacero spôsobov ako spraviť krok 2 v čase $O(n \log n)$. Jedna možnosť je pole jednoducho prejsť zľava doprava, pričom si už spracované prvky pamätáme vo vhodnom vyvažovanom binárnom strome. Vždy, keď prečítame ďalšie číslo, pomocou stromu v logaritmickom čase zistíme, koľko skôr spracovaných čísel bolo od neho väčších. Iným riešením (bez komplikovaných dátových štruktúr) je predstaviť si, že začíname s prázdnym poľom a postupne sa v ňom zjavujú čísla v poradí od najväčšieho po najmenšie. Nad poľom si postavíme intervalový strom, v ktorom si budeme pre každý úsek pamätať, koľko čísel v ňom sa už zjavilo. Vždy, keď sa nejaké číslo zjaví, tak vieme v logaritmickom čase aj zistiť počet skôr zjavených naľavo od neho, aj následne upraviť tie vrcholy intervalového stromu, ktoré pridanie nového prvku zmenilo. Tretím riešením je upraviť triedenie MergeSort. Detaily tohto riešenia prenechávame ako cvičenie.

Bonus: menší ako lineárny počet reverzov nestačí

Vo vašich riešeniach sme toto pozorovanie nevyžadovali, ale pre úplnosť vzorového riešenia uvedieme aj stručný dôkaz toho, že riešenie s menším ako lineárnym počtom reverzov nemôže existovať.

Keď sa pozrieme na hocijaké pole, vieme si ku každému prvku zistiť, ktoré (nanajvýš) dva prvky s ním majú v usporiadanom poli susediť. Nutnou podmienkou toho, že je pole správne usporiadané, je teda skutočnosť, že každý prvok má správnych susedov.
Lenže je možné, že na začiatku vôbec nik žiadneho správneho suseda nemá – viď pole $(0, 2, 4, \ldots, 1, 3, 5, \ldots)$. No a pri každom reverze sa zmenia susedia len nanajvýš štyrom prvkom – tým na krajoch reverznutého úseku, a tým, ktoré s reverznutým úsekom susedia. Ak teda potrebujeme každému prvku v nejakej chvíli zmeniť susedov, musíme nutne spraviť aspoň lineárny počet reverzov.

A-II-2 Potrubie

Riešenie hrubou silou vieme založiť na myšlienke, že na poradí rúr nezáleží. Stačí teda vyskúšať všetky *podmnožiny* rúr a pre každú spočítať, či má potrubie správnu dĺžku a dostatočný počet filtrov.
Obe efektívnejšie riešenia, ktoré si ukážeme, budú mať podobnú hlavnú myšlienku. Začneme tým, že si všetky rúry usporiadame podľa prietoku od najväčšieho po najmenší. V tomto poradí ich budeme jednu po druhej spracúvať, až kým prvýkrát nenastane situácia, že z už spracovaných rúr vieme spraviť vyhovujúce potrubie. V tej chvíli môžeme prestať a ako odpoveď vypísať prietok poslednej spracovanej rúry.
Riešenie za 7 bodov: Budeme si postupne generovať všetky kombinácie (dĺžka potrubia, počet filtrov), ktoré sa dajú z danej sady rúr vyrobiť. Keď nám pribudne nová rúra, ostanú nám všetky staré možnosti a pribudnú nám nejaké nové možnosti. Tie nové vyrobíme tak, že ku každej starej možnosti skúsime pridať novú rúru.
Samozrejme, nemá zmysel si pamätať možnosti, ktoré sú pridlhé, a viac ako f filtrov je už to isté ako presne f filtrov. Všetkých možností, ktoré vieme zostrojiť, je teda $O(\ell f)$. Pre každú z n rúr raz prejdeme množinu aktuálnych možností. Celkovo má teda takéto riešenie časovú zložitosť $O(n\ell f)$ a pamäťovú zložitosť $O(\ell f)$.

Na riešenie za plný počet bodov už potrebujeme len jedno pozorovanie: ak vieme vyrobiť potrubie dĺžky d s tromi filtrami a iným spôsobom vieme vyrobiť potrubie tej istej dĺžky ale s piatimi filtrami, stačí si pamätať ten druhý spôsob. Totiž ak by sme v budúcnosti novými rúrami vyrobili nejaké platné riešenie z prvého potrubia, môžeme namiesto neho použiť druhé potrubie a tiež dostaneme platné riešenie (rovnakej dĺžky a s viac filtrami).
Vo všeobecnosti teda platí, že pre každú dĺžku potrubia si potrebujeme pamätať len jedno číslo: najväčší počet filtrov, s ktorými vieme túto presnú dĺžku dosiahnuť. (Ak nejakú dĺžku nevieme ešte dosiahnuť vôbec, môžeme si to značiť tak, že maximálny počet filtrov bude $-\infty$.)
V ľubovoľnom okamihu si teda potrebujeme pamätať len $O(\ell)$ čísel. Taká je aj pamäťová zložitosť nášho riešenia. Časová zložitosť je $O(n\ell)$, keďže zoznam $O(\ell)$ možností budeme n-krát prechádzať.

Listing programu (Python)

```
from sys import exit

L, F = [ int(_) for _ in input().split() ]
N = int( input() )
rury = []
for n in range(N):
    dlzka, prietok, filtre = input().split()
    rury.append( ( int(prietok), int(dlzka), 1 if filtre=='A' else 0 ) )

NEDA_SA = -(2**30)
najviac_filtrov = [ NEDA_SA for _ in range(L+1) ]
najviac_filtrov[0] = 0

for prietok, dlzka, filtre in reversed(sorted(rury)):
    for doteraz in reversed(range(0,L-dlzka+1)):
        if najviac_filtrov[doteraz] != NEDA_SA:
            najviac_filtrov[doteraz+dlzka] = max( najviac_filtrov[doteraz+dlzka], najviac_filtrov[doteraz]+filtre )
    if najviac_filtrov[L] >= F:
        print(prietok)
        exit()

print(-1)
```

A-II-3 Vírus

Je zjavné, že vírus sa nevie rozšíriť z jedného komponentu súvislosti počítačovej siete do iného. Občas sa však stane, že vírus nedokáže nakaziť ani súvislý kus siete celý. Ak napríklad máme štyri počítače zapojené do kruhu, nech by sme nakazili ľubovoľný z nich, ten nakazí protiľahlý a tým šírenie vírusu skončí. Dva zo štyroch počítačov teda vždy zostanú nenakazené.

Je tiež zjavné, že počítač x vie nakaziť počítač y práve vtedy, keď sa z x do y dá prejsť na párny počet krokov (pričom v každom kroku prejdeme z nejakého počítača na susedný). Ako ale spoznať, kedy sa takto dá postupne nakaziť všetky počítače a kedy nie?

Tvrdenie: Ak máme súvislú počítačovú sieť tvorenú viac ako jedným počítačom, vírus ju vie celú nakaziť práve vtedy, ak obsahuje cyklus nepárnej dĺžky – teda ak vieme nájsť nejakú postupnosť počítačov $p_0, \ldots, p_{2k}$ takú, že každý p_i susedí s p_{i+1} a posledný susedí s prvým.

Dôkaz prvej implikácie: Majme súvislú sieť, v ktorej je nejaký cyklus nepárnej dĺžky, a chceme ukázať, že po nakazení ľubovoľného počítača budú časom nakazené všetky.

Keďže sieť je súvislá, od ľubovoľného jej počítača sa vieme po kábloch dostať na tento cyklus (a naopak). Predpokladajme, že sme ako prvý nakazili nejaký počítač x v sieti a že y je nejaký iný počítač v tejto sieti. Uvažujme dve rôzne cesty z x ku y. Možnosť 1: prejdem z x ku p_0 (konkrétny počítač na cykle) a odtiaľ ku y. Možnosť 2: to isté ako možnosť 1, ale po príchode do p_0 najskôr raz obídem dokola celý cyklus a až potom pokračujem ďalej.

Keďže cyklus má nepárnu dĺžku, počet krokov v možnosti 2 má opačnú paritu ako počet krokov v možnosti 1. Niektorá možnosť teda určite predstavuje spôsob ako sa z x dostať ku y na párny počet krokov – a teda počítač y bude časom nakazený.

Dôkaz druhej implikácie: Dokážeme obmenenú implikáciu: ak v našej súvislej sieti (tvorenej aspoň dvoma počítačmi) žiaden cyklus nepárnej dĺžky nie je, tak ju určite nevieme nakaziť celú.

Predstavme si, že sme z ľubovoľného počítača s v našej sieti spustili prehľadávanie do šírky a pre každý iný počítač sme tak zistili jeho (minimálnu) vzdialenosť od s. Ofarbime teraz počítače podľa tejto vzdialenosti: tie, pre ktoré je párna (vrátane samotného s) budú biele, a ostatné čierne.

Navyše ofarbime aj káble: Pre každý počítač iný od s ofarbíme na zeleno kábel, ktorým sme doň prvýkrát prišli. Všetky ostatné káble budú červené.

Tvrdíme teraz, že v našej sieti nemôže existovať dvojica susediacich počítačov rovnakej farby. Prečo? Každý zelený kábel zjavne spája čierny a biely počítač. A keby sme mali červený kábel, ktorý spája dva počítače rovnakej farby, zelené káble tvoriace cestu medzi tými dvoma počítačmi a tento červený kábel by dokopy vytvorili cyklus nepárnej dĺžky.

No a teraz už je zjavné, že ak v takejto sieti nakazíme biely počítač, nikdy od neho nebude nakazený žiaden čierny, a naopak.

A tým sa už dostávame k úplnému riešeniu našej úlohy. To začneme tým, že načítame vstup, rozdelíme si ho na komponenty, a skontrolujeme, či niektorý obsahuje cyklus nepárnej dĺžky. (Algoritmus, ktorým toto vieme spraviť, sme si popísali v dôkaze druhej implikácie.)

Ak máme k komponentov súvislosti, určite potrebujeme pridať aspoň $k-1$ káblov. (Potrebujeme, aby všetko bolo prepojené, ale každým káblom vieme znížiť počet komponentov najviac o 1.)

Ak už niektorý komponent súvislosti obsahuje nepárny cyklus, presne $k-1$ káblov stačí. (Jeden konkrétny spôsob, ako ich pridať, je zvoliť si v každom komponente jeden počítač a prepojiť jeden z nich so všetkými ostatnými.)

Ak žiaden komponent súvislosti zatiaľ neobsahuje nepárny cyklus, budeme potrebovať presne k káblov. Dokážeme postupne, že $k-1$ káblov nestačí, a že k áno.

Predstavme si, že v každom komponente máme počítače ofarbené na čierno a bielo. Ak by stačilo $k-1$ káblov, musí každý kábel spojiť dva komponenty, ktoré dovtedy neboli spojené. V takejto chvíli ale vždy vieme upraviť ofarbenie počítačov tak, aby aj naďalej platilo, že každý kábel spája počítače rôznych farieb. (Ak nový kábel spája počítače rôznych farieb, netreba robiť nič, a ak chceme spojiť počítače rovnakej farby, tak najskôr celý

jeden komponent prefarbíme na opačné farby, a až potom pridáme nový kábel.) Potom ale aj po pridaní všetkých $k-1$ káblov budeme mať počítače ofarbené na čierne a biele, a teda ešte stále nebudeme mať žiaden cyklus nepárnej dĺžky.
Z vyššie uvedenej situácie však vieme ľahko riešenie dokončiť pridaním k-teho kábla: stačí prepojiť ľubovoľné dva počítače rovnakej farby. Dva takéto určite nájdeme medzi počítačmi s číslami 0, 1 a 2.
Pri dobrej implementácii má celé vyššie popísané riešenie časovú zložitosť $O(n+m)$, teda lineárnu od veľkosti vstupu. Pre poriadok ešte dodávame, že v nižšie uvedenej implementácii sme na ofarbenie komponentov použili prehľadávanie do hĺbky, nie do šírky. Rozmyslite si, že aj tento prístup funguje.

Listing programu (C++)

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

int N, M;
vector< vector<int> > susedia;
vector<int> farba;
vector<int> reprezentanti;

bool ofarbi_komponent(int v, int f) {
    // Ofarbi vsetky vrcholy komponentu striedavo ciernou a bielou, pricom ak komponent nema neparny cyklus,
    // rovnake farby nikde nebudu susedit. Vrati true ak najde neparny cyklus.
    farba[v] = f;
    bool odpoved = false;
    for (int w : susedia[v]) {
        if (farba[w] == f)   { odpoved = true; continue; }
        if (farba[w] == 1-f) {                 continue; }
        ofarbi_komponent(w,1-f);
    }
    return odpoved;
}

void spoj_dva_rovnake() {
    for (int a=0; a<3; ++a) for (int b=0; b<a; ++b) if (farba[a] == farba[b]) {
        cout << a << " " << b << endl;
        return;
    }
}

int main() {
    // nacitame vstup
    cin >> N >> M;
    susedia.resize(N);
    for (int m=0; m<M; ++m) {
        int x, y;
        cin >> x >> y;
        susedia[x].push_back(y);
        susedia[y].push_back(x);
    }

    // ofarbime komponenty, a to tak ze prvy zacneme farbou 0 a kazdy dalsi farbou 1
    farba.resize(N, -1);
    reprezentanti.push_back(0);
    bool mame_neparny_cyklus = ofarbi_komponent(0,0);
    for (int n=1; n<N; ++n) if (farba[n] == -1) {
        reprezentanti.push_back(n);
        mame_neparny_cyklus |= ofarbi_komponent(n,1);
    }

    // spojime komponenty ak mame viac ako jeden
    for (unsigned i=1; i<reprezentanti.size(); ++i) cout << reprezentanti[0] << " " << reprezentanti[i] << endl;

    // ak nemame neparny cyklus, treba este spojit lubovolne dva vrcholy rovnakej farby
    if (!mame_neparny_cyklus) spoj_dva_rovnake();
    cout << 0 << endl; // je jedno ktory pocitac nakazime ako prvy
}
```

A-I-4 Exaktné exponenciálne algoritmy

Podúloha A: poradie povinností

Začnime najskôr pár pomerne zjavnými pozorovaniami.
Nikdy sa neoplatí len tak ležať a nič nerobiť, ak ešte nemáme všetky povinnosti hotové. Totiž ak by sme dotyčný oddych vynechali, všetky neskôr splnené povinnosti by boli splnené o čosi skôr, a teda by za ne bol nižší postih.

Nikdy sa neoplatí striedavo pracovať na dvoch alebo viacerých povinnostiach, teda vždy existuje optimálne riešenie, v ktorom vždy najskôr kompletne spravíme nejakú povinnosť a až potom ideme na ďalšiu. Totiž ak sa pozrieme na hocijaké riešenie, vypíšeme si, v akom poradí sme skončili prácu na jednotlivých povinnostiach, a potom sa pozrieme na riešenie, ktoré povinnosti spraví jednu po druhej v tom istom poradí, tak ľahko nahliadneme, že toto druhé riešenie každú povinnosť dokončí vtedy keď prvé alebo skôr.

Keď teda vieme, že stačí hľadať riešenie, v ktorom povinnosti plníme jednu po druhej a bez prestávok, vieme, že vlastne hľadáme *permutáciu* povinností, ktorej zodpovedá najmenší možný celkový súčet postihov.

Algoritmus s časovou zložitosťou $\hat{O}(2^n)$ pre tento problém vyzerá napríklad nasledovne: Použijeme dynamické programovanie, pričom úlohu postupne vyriešime pre každú podmnožinu povinností.

Pre ľubovoľnú podmnožinu X našej množiny povinností označme $B(X)$ najmenší súčet postihov, ktorý by sme dostali, keby sme plnili iba povinnosti z množiny X.

Vieme, že keď budeme plniť tieto povinnosti, poslednú z nich dokončíme v čase $t_X = \sum_{x \in X} t_x$. My sa musíme rozhodnúť, ktorá z našich povinností bude tá, ktorú dokončíme ako poslednú. No a toto rozhodnutie spravíme jednoducho: vyskúšame všetky možnosti a vyberieme najlepšiu z nich. Ak si povieme, že ako poslednú robíme povinnosť x, tak v optimálnom riešení najskôr optimálne splníme ostatné povinnosti – čo prinesie celkový postih $B(X - \{x\})$, plus súčet všetkých $s_{y,x}$ za povinnosti skončené skôr ako x – a potom si splníme povinnosť x. Platí teda:

$$B(X) = \min_{x \in X} \Big(p(x, t_X) \quad + \quad B(X - \{x\}) \quad + \sum_{y \in X - \{x\}} s_{y,x} \Big)$$

Pomocou tohto vzťahu vieme teda postupne pre každú z 2^n podmnožín našej množiny povinností vypočítať v čase $O(n)$ jej optimálne riešenie.

(Na záver ešte podotkneme, že existuje aj riešenie s rovnakou časovou zložitosťou, pri ktorom skúšame, ktorú povinnosť spravíme ako *prvú*, nie *poslednú*. Pri tomto riešení však musíme namiesto $B(X)$ uvažovať iné hodnoty: $C(X)$ je optimálna suma postihov za povinnosti z množiny X, avšak za predpokladu, že začíname nie v čase 0, ale v čase, kedy sme dokončili všetky ostatné povinnosti.)

Podúloha B: vrcholové pokrytie grafu

Kľúčovým pozorovaním k riešeniu tejto úlohy je uvedomiť si, že ľubovoľné *vrcholové pokrytie* (platne rozmiestnené WC búdky zo zadania úlohy) je vždy doplnkom nejakej *nezávislej množiny* (zvieratká zo študijného textu) a naopak. Totiž:

1. Ak máme nejakú nezávislú množinu lokalít X v meste, vieme, že žiadna ulica nemá oba konce v X (keďže žiadne dve lokality v X nie sú priamo prepojené). Ak teda postavíme WC všade okrem X, bude na každej ulici aspoň jedno WC – a teda doplnok X tvorí vrcholové pokrytie.

2. Ak máme nejaké vrcholové pokrytie Y, tak každá ulica má v Y aspoň jeden svoj koniec, a teda má mimo Y najviac jeden svoj koniec. V doplnku Y teda nie sú žiadne dve lokality, ktoré by boli prepojené ulicou – čiže doplnkom Y je nezávislá množina.

Veľkosť najmenšieho vrcholového pokrytia teda vieme nájsť tak, že nájdeme i = veľkosť najväčšej nezávislej množiny a následne vypočítame a vrátime hodnotu $n - i$.

Najlepším algoritmom, s ktorým sme sa v rámci nášho seriálu zoznámili, je algoritmus z riešení domáceho kola. Jeho časová zložitosť je $\hat{O}(1.3803^n)$.

(V súčasnosti sú už známe aj lepšie algoritmy riešiace túto úlohu, napr. Xiaov-Nagamochiho algoritmus z roku 2013 má časovú zložitosť dokonca len $\hat{O}(1.1996^n)$. Tieto algoritmy však svojou komplexitou presahujú rámec nášho seriálu úloh a na plný počet bodov nebolo treba vedieť o ich existencii.)

TRIDSIATY PIATY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2020