A-III-4 Štvorec

Efektívne hľadanie štvorca vieme spraviť na základe nasledovného pozorovania: hľadáme vlastne dve lokality a a c, ktoré majú aspoň dvoch spoločných susedov.

Spravíme si dvojrozmerné pole S, ktorého všetky prvky nastavíme na nulu. Význam premennej $S[i,j]$ bude nasledovný: Ak $S[i,j] = 0$, ešte nepoznáme žiadnu lokalitu, ktorá by susedila s i aj j. V opačnom prípade je $S[i,j]$ číslo tejto lokality.

Teraz budeme postupne prechádzať cez všetky lokality. Pre každú lokalitu v spravíme nasledovné: Postupne vygenerujeme všetky dvojice (i,j) susedov lokality v. Pre každú z nich sa pozrieme, či sme ju už niekedy skôr videli. Ak nie, nastavíme $S[i,j]$ na v. Ak áno, našli sme práve štvorec: $v \to i \to S[i,j] \to j \to v$. Vypíšeme ho a skončíme.

Ak už celý vyššie popísaný postup skončil, zjavne neexistujú žiadne dve lokality, ktoré by mali dvoch spoločných susedov, a teda môžeme spokojne vypísať, že štvorec neexistuje.

Časová zložitosť je $O(n^2)$, lebo každé políčko poľa S počas behu algoritmu raz inicializujeme a nanajvýš raz vyplníme nenulovou hodnotou. Pamäťová zložitosť je zjavne tiež $O(n^2)$.

Listing programu (C++)

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

void check_and_note(vector< vector<int> > &S, int v, int i, int j) {
    if (i > j) swap(i,j);
    if (S[i][j]) {
        cout << (v+1) << "␣" << (i+1) << "␣" << (S[i][j]+1) << "␣" << (j+1) << "\n";
        exit(0);
    } else {
        S[i][j] = v;
    }
}

int main() {
    int N, M;
    cin >> N >> M;
    vector< vector<int> > susedia(N);
    while (M--) {
        int x, y;
        cin >> x >> y;
        --x; --y;
        susedia[x].push_back(y);
        susedia[y].push_back(x);
    }
    vector< vector<int> > S(N, vector<int>(N,0) );
    for (int v=0; v<N; ++v) {
        for (int i : susedia[v]) {
            for (int j : susedia[v]) {
                if (i == j) break;
                check_and_note(S,v,i,j);
            }
        }
    }
    cout << "0\n";
}
```

A-III-5 Jaskyňa

Ak si každý vstup do jaskyne zapíšeme znakom `(` a každý výstup znakom `)`, je zjavné, že každá platná postupnosť vstupov a výstupov bude zodpovedať nejakému dobre uzátvorkovanému výrazu. Opačná implikácia však nemusí platiť – niektorým dobre uzátvorkovaným výrazom totiž budú prislúchať také scenáre, pri ktorých by niekto zostal v jaskyni dlhšie ako môže.

Niekoľko bodov sa dalo získať za vstupy, v ktorých platilo $m = 1\,000\,000$, a teda každý mohol ostať v jaskyni ľubovoľne dlho. Pre tieto vstupy je správnou odpoveďou presne počet dobre uzátvorkovaných výrazov tvorených n pármi zátvoriek.

Označme c_k počet dobre uzátvorkovaných výrazov tvorených presne k zátvorkami.[1] Ako tento počet vypočítať?

[1] Tieto hodnoty zvykneme volať Catalanove čísla.

Prázdny dobre uzátvorkovaný výraz je práve jeden.
Neprázdny dobre uzátvorkovaný výraz musí zjavne mať tvar `(A)B`, kde `A` a `B` sú nejaké (možno aj prázdne) dobre uzátvorkované výrazy. Toto si ľahko dokážeme, stačí si všimnúť, že prvý znak musí byť `(` a táto zátvorka musí mať niekde vo výraze zodpovedajúcu `)`.
Ak máme mať dokopy k zátvoriek, musíme sa rozhodnúť, koľko zátvoriek (i) použijeme vo výraze `A` a koľko (všetky zvyšné, teda $k-1-i$) vo výraze `B`. Rôzne voľby i nám zjavne vyrobia rôzne výsledné výrazy.
Ku pevne zvolenému k a i prislúcha presne $c_i c_{k-1-i}$ dobre uzátvorkovaných výrazov tvorených k pármi zátvoriek. Totiž ľubovoľný výraz `A` s i zátvorkami vieme skombinovať s ľubovoľným výrazom `B` s $k-1-i$ zátvorkami.
Dostávame teda, že platí: $c_0 = 1$ a $\forall k > 0 : c_k = \sum_{i=0}^{k-1} c_i c_{k-1-i}$.
Podľa tohto vzťahu vieme vypočítať hodnotu c_n z predchádzajúcich hodnôt v čase $O(n)$, dokopy teda výpočet všetkých hodnôt $c_0, c_1, \ldots, c_n$ bude trvať čas $O(n^2)$.

Ako teraz vyššie popísanú úvahu upraviť tak, aby sme brali do úvahy aj časy z fotobunky?
Označme u_k počet dobre uzátvorkovaných výrazov, ktoré zodpovedajú platným riešeniam pre posledných $2k$ čísel zo vstupu. Zdôrazňujeme, že do u_k rátame len tie uzátvorkované výrazy, ktoré zodpovedajú scenárom, pri ktorých sa každý jaskyniar vynoril dostatočne skoro. Hodnota u_n je zjavne riešením pôvodnej úlohy, tú teda chceme vypočítať.
Zjavne platí $u_0 = 1$. Zvoľme teraz nejaké $k > 0$ a zamyslime sa, ako vypočítame u_k.
Podobne ako vyššie, vieme, že každý platný výraz musí mať tvar `(A)B`, kde `A` a `B` sú nejaké (možno aj prázdne) dobre uzátvorkované výrazy. Tentokrát už ale nebudeme skúšať všetky možnosti pre i (čiže pre počet párov zátvoriek vo výraze `A`), ale len tie, pri ktorých sa prvý človek vynorí dostatočne rýchlo.
Predstavme si, že sme si už zvolili konkrétne takéto i. Čo teraz vieme povedať? Pre výraz `A` nemáme už vôbec žiadne obmedzenia – totiž vieme, že niekto sa ponoril skôr a vynoril neskôr ako prebehlo všetko zodpovedajúce tomuto výrazu. A teda môžeme ako výraz `A` použiť ľubovoľný z c_i dobre uzátvorkovaných výrazov dĺžky $2i$.
Pre výraz `B` máme to isté obmedzenie ako pre pôvodný výraz: musíme zabezpečiť, aby zodpovedal platnému riešeniu. Výraz `B` však zodpovedá nejakému (kratšiemu) sufixu vstupu, a teda vieme povedať, že preň máme presne u_{k-1-i} možností.
Dostávame teda, že platí: $u_0 = 1$ a $\forall k > 0 : u_k = \sum_{i=0}^{???} c_i u_{k-1-i}$.
Vo vyššie uvedenej sume sčitujeme len cez tie i, pre ktoré je rozdiel hodnôt časov zodpovedajúcich úvodnej `(` a jej zodpovedajúcej `)` nanajvýš rovný m.
Po tom, ako už poznáme hodnoty c_k, si teda rovnako v čase $O(n^2)$ vieme postupne dopočítať hodnoty u_k a na záver vypísať hodnotu u_n na výstup.
(Skutočné hodnoty samozrejme môžu byť veľké, všetky výpočty preto robíme modulo prvočíslo zo zadania.)

Listing programu (C++)

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

const int MOD = 1000000007;

int main() {
    int N, M;
    cin >> N >> M;
    vector<int> T(2*N);
    for (int &t : T) cin >> t;

    vector<long long> C(N+1);
    C[0] = 1;
    for (int n=1; n<=N; ++n) for (int i=0; i<n; ++i) C[n] = (C[n] + C[i]*C[n-1-i]) % MOD;

    vector<long long> U(N+1);
    U[0] = 1;
    for (int n=1; n<=N; ++n) {
        for (int i=0; i<n; ++i) {
            if (T[2*N-2*n+2*i+1] - T[2*N-2*n] > M) break;
            U[n] = (U[n] + C[i]*U[n-1-i]) % MOD;
        }
    }
    cout << U[N] << endl;
}
```

A-III-6 Wienerov index

Odpoveď, ktorú máme vypočítať, je definovaná ako súčet dĺžok všetkých ciest medzi dvoma vrcholmi. Inými slovami, pre každú cestu máme zarátať 1 za každú hranu, ktorá na nej leží.
Túto istú sumu si môžeme vyjadriť aj obrátene: pre každú hranu zarátame 1 za každú cestu, ktorá ňou prechádza. Keď zo stromu odstránime hranu uv, rozpadne sa nám na dve časti. Označme ich vrcholy U a V. Kedy cesta z vrcholu x do vrcholu y prechádza hranou uv? Práve vtedy, keď jeden z vrcholov x a y leží v U a druhý vo V. Touto hranou uv preto prechádza presne $|U| \times |V|$ rôznych ciest.
Na výpočet správnej odpovede stačí strom zakoreniť v ľubovoľnom jednom vrchole, prehľadať ho do hĺbky a vypočítať si veľkosti jednotlivých podstromov. Pre ľubovoľnú hranu stromu potom vieme, koľko vrcholov leží na jednej jej strane – no a na druhej nutne ležia všetky zvyšné.
Vďaka formátu vstupu je toto ešte jednoduchšie: Strom sme už dostali zakorenený, teraz nám stačí vstup prejsť v obyčajnom cykle od konca ku začiatku. Tento cyklus zodpovedá prechodu stromom „zdola hore“: každý vrchol spracujeme až po tom ako boli spracované všetky jeho deti.
Algoritmus má lineárnu časovú aj pamäťovú zložitosť.

Listing programu (C++)

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

int main() {
    int N; cin >> N;
    vector<int> parent(N);
    for (int n=1; n<N; ++n) { cin >> parent[n]; --parent[n]; }
    vector<int> subtree_size(N,1);
    for (int n=N-1; n>=1; --n) subtree_size[parent[n]] += subtree_size[n];
    long long answer = 0;
    for (int n=1; n<N; ++n) answer += 1LL * subtree_size[n] * (N - subtree_size[n]);
    cout << answer << endl;
}
```

TRIDSIATY ŠTVRTÝ ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Bui Truc Lam, Michal Forišek, Samuel Sládek
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2019