

Problema 3. Allca

Autor: Carole Șerban, Delft University of Technology

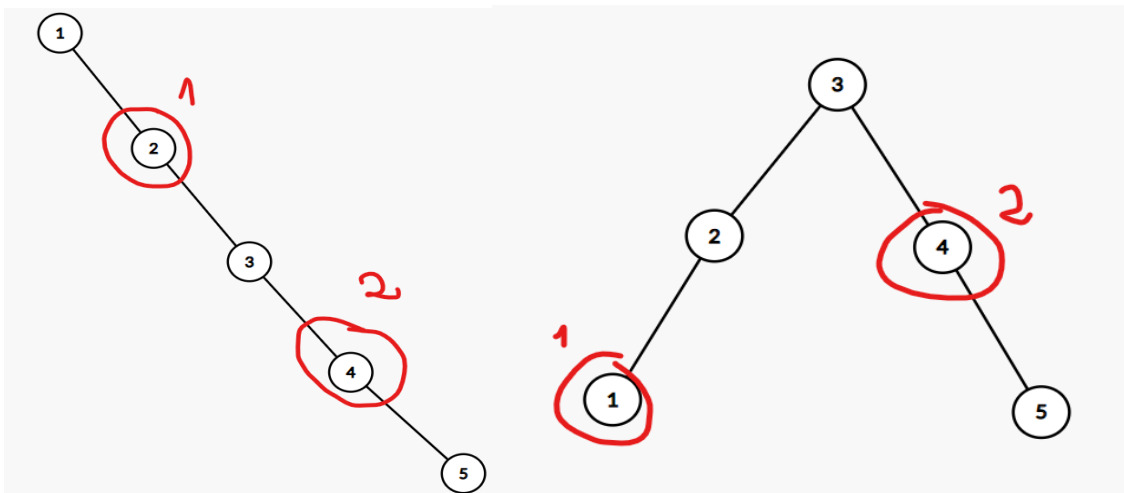
Cuvinte cheie: Lowest common ancestor (Lca), Divide and Conquer, Small-to-large

Soluție completă:

Dacă pornim de la o mulțime nevidă de noduri și adăugăm un nod, se pot întâmpla două lucruri - lca-ul va rămâne același sau va deveni strămoșul lca-ului precedent.

Să pornim de la ipoteza că avem 2 noduri. Sunt 2 situații posibile:

1. Unul dintre noduri se află pe drumul de la rădăcină la celălalt.
2. Niciunul dintre noduri nu e strămoșul celuilalt.



Observăm că în ambele cazuri, urcarea nodului 2 schimbă lca-ul doar atunci când acesta devine unul din strămoșii primului lca dintre cele două noduri. De asemenea, odată schimbat, va fi mereu egal cu nodul 2. Această observație ne duce la ideea aplicării metodei divide and conquer pentru numărare. Putem afla lca-ul dintre două noduri în $O(1)$ precalculând range minimum query pe euler tour.

Ideea cu divide and conquer este să calculăm pentru fiecare call, suma lca-urilor subsecvențelor care au capătul din stânga în $[st, mij]$, iar pe cel din dreapta în $[mij+1, dr]$, celelalte fiind calculate deja în alt call de divide. Facem apeluri succesive în care împărțim intervalele în $[st, mij]$ și $[mij+1, dr]$ și iterăm după indicii i din $[st, mij]$. Pentru fiecare, căutăm binar primul $j > mij$ cu proprietatea că lca-ul nodurilor de la i la j este egal cu lca-ul nodurilor de la $mij+1$ la j . În cazul oricărui $k \geq j$, știm din proprietatea menționată în paragraful precedent că lca-ul de la i la k este egal cu cel de la $mij+1$ la k , deci precalculăm pentru fiecare element $x > mij$ $sp[x] =$ suma după lca de la $mij+1$ la x , pentru oricare $y \geq x$, simplu prin sume parțiale. Pentru elementele mai mici ca j (cel pe care l-am căutat binar) știm că lca-ul de la i la ele este egal pentru toate. Și deci ajungem la relația pentru elementul i per call de divide ca fiind:

Suma după subsecvențele care încep în $i = sp[j] + (j-1-mij) * (lca\text{-ul de la } i \text{ la } mij+1)$.

Cum trebuie să căutăm binar pentru fiecare element în $\log$ call-uri de divide ajungem la complexitatea $O(M \cdot \log^2(M) + N \cdot \log(N))$. Această soluție nu obține punctaj maxim.

Optimizarea necesară pentru 100 de puncte este să realizăm că dacă putem urca și nodul 1, acel j căutat binar poate doar să se mute la dreapta, deoarece 2 va ajunge ori la fel de repede să depășească lca-ul, ori mai târziu.

Așa că iterăm cu i de la mij la st, pentru $i = \text{mij}$ căutăm j brut și după doar îl mutăm la dreapta când e cazul. Complexitatea finală este $O(M \log M + N \log N)$.

Soluție alternativă (Prosie Radu-Teodor, FMI Unibuc):

Ne dorim să calculăm pentru fiecare nod contribuția sa la răspuns.

Dacă notăm cu S_i = numărul de subsecvențe din A pentru care i este strămoș atunci contribuția lui i poate fi obținută după formula : $S_i = \sum_{\text{fiu al lui } i} S_F$. Pentru a calcula S_i putem folosi DSU cu tehnica small-to-large :

Facem DFS și ținem pentru fiecare nod un DSU care reprezintă subsecvențele compacte create de pozițiile nodurilor din subarboarele lui în cadrul șirului A . De asemenea, trebuie să menținem și numărul de subsecvențe ale acestor subsecvențe compacte, lucru ce poate fi recalculat ușor la fiecare unire. După ce am făcut DFS în fiul lui i , unim DSU-ul lui i și al fiului lui i inserându-l pe cel cu mai puține elemente în cel cu mai multe (eventual interschimbând, dacă fiul are DSU-ul mai mare).

A se nota că trebuie să menținem DSU-urile puțin diferit pentru a avea memorie liniară în numărul de noduri care ne interesează din subarboare, lucru pentru care putem folosi hashing. Este probabil ca soluția să nu intre dacă folosiți hashmap-ul din STL, dar intră dacă îl scrieți de mână sau folosiți `gp_hash_table` din `__gnu_pbds`.

Complexitate : $O(M \log M \cdot \alpha(M) + N)$

Soluții parțiale:

Pentru subtask-ul 1, observăm că în cazul oricărei subsecvențe, răspunsul este capătul intervalului, deci putem să calculăm contribuția pentru capete (sumă după $A[i] * i$).

Subtask-ul 2 poate fi rezolvat în $O(N + M)$, cu o stivă - calculăm vectorii $st[i]$ și $dr[i]$ = cel mai apropiat element la stânga/dreapta de i care are un nivel mai mic (la stânga) și mai mic sau egal (la dreapta). Apoi contribuția per element este egală cu $(dr[i] - i) * (i - st[i])$. Orice soluție cu un logaritm în plus ar trebui să ia punctele totuși.

Pentru subtask-urile 3 și 4, orice soluție pătratică, sau $O(M^2 \cdot \log(M))$ este îndeajuns de bună. O soluție simplă și scurtă este doar să pornim de la un nod de start (cel mai din stânga al subsecvenței), să îl punem ca lca , și odată cu adăugarea unui nod să urcăm lca -ul până când este și strămoșul noului nod. Odată ce este, oprim urcarea și îl adăugăm la sumă. Soluția aceasta este $O(M \cdot (N + M))$, deoarece urcarea pentru fiecare nod de start se amortizează la $O(H)$, unde H = distanța de la rădăcină la acest nod de început.

Soluție Divide and Conquer

```
#include <fstream>
#include <vector>
using namespace std;
ifstream cin("allca.in");
ofstream cout("allca.out");
const int nmax = 2e5 + 1;
int n , m , a , b , ind;
int v[2*nmax] , d[nmax] , p[2*nmax] , tin[nmax] , tout[nmax];
int rmq[19][2*nmax], lg[2*nmax] , l;
long long ans , sp[2*nmax];
vector <int> g[nmax];

void dfs(int x , int p)
{
    rmq[0][++ind] = x;
    tin[x] = ind;
    d[x] = d[p] + 1;
    for(auto it : g[x])
        if(it != p)
        {
            dfs(it,x);
            rmq[0][++ind] = x;
        }
    tout[x] = ind;
}

int helper(int x , int y)
{
    if(d[x] < d[y]) return x;
    else return y;
}

int lca(int x , int y)
{
    if(tin[x] > tin[y])
    {
        l = lg[tin[x]-tin[y]+1];
        return helper(rmq[l][tin[x]],rmq[l][tin[y]+(1<<l)-1]);
    }
    else
    {
        l = lg[tin[y]-tin[x]+1];
        return helper(rmq[l][tin[y]],rmq[l][tin[x]+(1<<l)-1]);
    }
}

void divide(int st , int dr)
{
    if(st == dr)
    {
        ans += v[st];
    }
    else
    {
        int mij = (st+dr)/2;
        int nod = v[mij+1];
        for(int i = mij+1 ; i <= dr ; ++i)
```

```

    {
        nod = lca(nod,v[i]);
        p[i] = nod;
        sp[i] = p[i];
    }
    for(int i = dr-1 ; i > mij ; --i) sp[i] += sp[i+1];
    int j;
    for(int i = mij ; i >= st ; --i)
    {
        if(i == mij)
        {
            j = mij+1;
            nod = v[mij];
        }
        else nod = lca(nod,v[i]);
        while(j <= dr)
        {
            if(p[j] == lca(nod,p[j])) break;
            ++j;
        }
        if(j <= dr) ans += sp[j];
        ans += 1LL*(j-mij-1)*lca(nod,v[mij+1]);
    }
    divide(st,mij);
    divide(mij+1,dr);
}

int main()
{
    cin >> n;
    for(int i = 1 ; i < n ; ++i)
    {
        cin >> a >> b;
        g[a].push_back(b);
        g[b].push_back(a);
    }
    cin >> m;
    for(int i = 1 ; i <= m ; ++i) cin >> v[i];
    dfs(1,0);
    lg[0] = -1;
    for(int i = 1 ; i <= 2*n-1 ; ++i) lg[i] = lg[i/2]+1;
    for(int i = 1 ; i <= lg[2*n-1] ; ++i)
    {
        for(int j = (1<<i) ; j <= 2*n-1 ; ++j)
        {
            rmq[i][j] = helper(rmq[i-1][j],rmq[i-1][j-(1<<(i-1))]);
        }
    }
    divide(1,m);
    cout << ans;
    return 0;
}

```

Soluție Disjoint Set Union

```
#pragma GCC optimize("Ofast,unroll-loops,inline")
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/hash_policy.hpp>
#include <fstream>
#include <utility>
#include <vector>

using namespace __gnu_pbds;
using namespace std;

ifstream cin("allca.in");
ofstream cout("allca.out");

struct custom_hash {
    size_t operator()(const int x) const noexcept {
        return static_cast<unsigned>(x);
    }
};

class lekee {
    gp_hash_table<int, int, custom_hash> *t_ptr;
    gp_hash_table<int, int, custom_hash> *sz_ptr;

    int root_impl(const int x) noexcept {
        int &parent_ref = (*t_ptr)[x];
        if (parent_ref == 0) return x;
        return parent_ref = root_impl(parent_ref);
    }

    void unite(int a, int b) noexcept {
        a = root_impl(a);
        b = root_impl(b);
        if (a == b)
            return;

        const int sa = (*sz_ptr)[a];
        const int sb = (*sz_ptr)[b];

        coef -= 111 * sa * (sa + 1);
        coef -= 111 * sb * (sb + 1);

        if (sb > sa)
            std::swap(a, b);
        (*sz_ptr)[a] += (*sz_ptr)[b];
        (*t_ptr)[b] = a;

        coef += 111 * (*sz_ptr)[a] * ((*sz_ptr)[a] + 1);
    }

public:
    long long coef;

    lekee() : t_ptr(nullptr), sz_ptr(nullptr), coef(0) {}
    ~lekee() {
        delete t_ptr;
        delete sz_ptr;
    }
}
```

```

lekee(lekee &&other) noexcept : t_ptr(nullptr), sz_ptr(nullptr), coef(0) {
    std::swap(t_ptr, other.t_ptr);
    std::swap(sz_ptr, other.sz_ptr);
    std::swap(coef, other.coef);
}

lekee &operator=(lekee &&other) noexcept {
    if (this != &other) {
        delete t_ptr;
        delete sz_ptr;
        t_ptr = nullptr;
        sz_ptr = nullptr;
        coef = 0;
        std::swap(t_ptr, other.t_ptr);
        std::swap(sz_ptr, other.sz_ptr);
        std::swap(coef, other.coef);
    }
    return *this;
}

lekee(const lekee &) = delete;
lekee &operator=(const lekee &) = delete;

void activate(const int x) {
    if (!sz_ptr)
        sz_ptr = new gp_hash_table<int, int, custom_hash>();
    if (!t_ptr)
        t_ptr = new gp_hash_table<int, int, custom_hash>();
    (*sz_ptr)[x] = 1;
    coef += 2;

    if (sz_ptr->find(x - 1) != sz_ptr->end())
        unite(x - 1, x);
    if (sz_ptr->find(x + 1) != sz_ptr->end())
        unite(x + 1, x);
}

[[nodiscard]] long long get_coef() const noexcept { return coef; }

void swap(lekee &other_leke) noexcept {
    std::swap(t_ptr, other_leke.t_ptr);
    std::swap(sz_ptr, other_leke.sz_ptr);
    std::swap(coef, other_leke.coef);
}

void clear_content() noexcept {
    delete t_ptr;
    t_ptr = nullptr;
    delete sz_ptr;
    sz_ptr = nullptr;
    coef = 0;
}
};

vector<vector<int>> g_storage, poz_storage, sub_storage;
vector<lekee> susu_storage;
vector<long long> ans_storage;

void meld(const int node_idx_a, const int node_idx_b) {
    if (sub_storage[node_idx_a].size() < sub_storage[node_idx_b].size()) {

```

```
        sub_storage[node_idx_a].swap(sub_storage[node_idx_b]);
        susu_storage[node_idx_a].swap(susu_storage[node_idx_b]);
    }
    for (auto &it : sub_storage[node_idx_b]) {
        susu_storage[node_idx_a].activate(it);
        sub_storage[node_idx_a].push_back(it);
    }
}

void dfs(const int x_node, const int p_node = 0) {
    for (int &val_idx : poz_storage[x_node]) {
        susu_storage[x_node].activate(val_idx);
        sub_storage[x_node].push_back(val_idx);
    }
    for (int child_node : g_storage[x_node]) {
        if (child_node == p_node) continue;
        dfs(child_node, x_node);
        meld(x_node, child_node);
        susu_storage[child_node].clear_content();
        sub_storage[child_node].clear();
        sub_storage[child_node].shrink_to_fit();
    }
    ans_storage[x_node] += susu_storage[x_node].get_coef();
    if (p_node != 0) {
        ans_storage[p_node] -= susu_storage[x_node].get_coef();
    }
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n_nodes, m_elements;
    cin >> n_nodes;

    g_storage.resize(n_nodes + 1);
    poz_storage.resize(n_nodes + 1);
    sub_storage.resize(n_nodes + 1);
    susu_storage.resize(n_nodes + 1);
    ans_storage.resize(n_nodes + 1, 0);

    for (int i = 1; i < n_nodes; i++) {
        int u, v;
        cin >> u >> v;
        g_storage[u].push_back(v);
        g_storage[v].push_back(u);
    }
    cin >> m_elements;
    for (int i = 1; i <= m_elements; i++) {
        int node_val_appears_at;
        cin >> node_val_appears_at;
        poz_storage[node_val_appears_at].push_back(i);
    }
    dfs(1);
    long long total = 0;
    for (int i = 1; i <= n_nodes; i++)
        total += ans_storage[i] * i;
    cout << total / 2 << "\n";
    return 0;}
```