RAUCODER 2022

INDICAȚII DE REZOLVARE

14 mai 2022

Problema 1. Ghicitoare

Propunător: Daniela Alexandra Crișan

Cuvinte cheie: operații pe biți

Indicații

Orice secvență de 4 numere naturale care începe cu un multiplu a lui 4 are suma XOR egală cu 0:
{ 0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11}, {12, 13, 14, 15}, ….

De exemplu, S = 12 ^ 13 ^ 14 ^15 =

|1100| ^

|1101| ^

|1110| ^

|1111| ^

|0000| = 0

Complexitate spațială: O(1)
Complexitate temporală: O(T)

O soluție liniară în timp O(T*n) ar obține 40 de puncte.

Soluție

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

int main()
{
    int T, n, x, S, S1, p;
    cin >> T;

    for (int t = 1; t <= T; ++t)
    {
        cin >> n >> S;
        S1 = 0;
        for (int i = n/4*4; i <= n; ++i)
            S1 ^= i;
        cout << (S ^ S1) << '\n';
    }

    return 0;
}
```

Problema 2. Graffiti

Propunător: Daniela Alexandra Crișan

Cuvinte cheie: Sume parțiale, RMQ (Range Minimum Query)

Indicații

Pentru calculul cumulativ al lațimilor se vor folosi sume parțiale.

Calculul optimizat al minimului pe intervale pentru înălțimi se poate face folosind o structură de tip RMQ care se precalculează în O(N*log2(N)), iar interogările sunt O(1).

Complexitate spațială: O(N*log2(N))
Complexitate temporală: O(N*log2(N) + Q)

O soluție cu parcurgeri repetate, de complexitate temporală O(N*Q), ar obține maxim 40 de puncte, în funcție de implementare.

O soluție cu sume parțiale care precalculează și reține o matrice NxN cu elementele Mij = min(hi,...,hj), apoi efectuează interogările în timp constant, de complexitate temporală O(N*N + Q) și de memorie O(N*N), ar obține 70 de puncte.

Solutia care implementează Șmenul lui Batog (square root decomposition) are complexitatea temporală: preprocesarea O(N), interogările se efectuează în O(sqrt(N)), în total: O(N+Q*sqrt(N)). Memorie suplimentară O(sqrt(N)). Soluția obține 70 de puncte.

Soluție

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

const int N = 1e5;
const int Log = 17;

long long S[N + 1];
int rmq[Log + 1][N + 1], lg2[N + 1];

int main()
{
    int n, m, L, x, y;
    cin >> n;
    for (int i = 1; i <= n; i++)
    {
        cin >> L >> rmq[0][i];
        S[i] = S[i - 1] + L;
    }

    for (int i = 2; i <= n; i++)
        lg2[i] = lg2[i >> 1] + 1;
    for (int i = 1; i <= Log; i++)
        for (int j = 1; j + (1 << (i - 1)) - 1 <= n; j++)
            rmq[i][j] = min(rmq[i - 1][j], rmq[i - 1][j + (1 << (i - 1))]);

    cin >> m;
    while (m--)
    {
        cin >> x >> y;
        int l_lg = lg2[y - x + 1];
        cout << (S[y] - S[x - 1]) * min(rmq[l_lg][x], rmq[l_lg][y - (1 << l_lg) + 1]) << '\n';
    }

    return 0;
}
```

Problema 3. CFR

Propunător: Alexandru Lorintz

Cuvinte cheie: arbori, păduri de mulțimi disjuncte, LCA (cel mai apropiat strămoș comun), RMQ (range minimum query), STL, arbori de intervale, combinatorică, aritmetică modulară

Indicații

Vom începe prin a face câteva observații simple (unele chiar evidente), absolut necesare în rezolvarea problemei:

- Setul de cale ferată este un arbore în care gările sunt nodurile, iar șinele muchiile. De asemenea, costul fiecărei muchii este 1.
- În fiecare zi subgraful obținut prin utilizarea doar a muchiilor permise în ziua respectivă este o pădure de arbori.
- Dacă RAU-Gigel alege un lanț dintr-un oraș (arbore din pădurea de arbori obținută într-o zi) mereu va alege unul cât mai lung pentru a maximiza mai apoi lungimea totală a lanțului pe care îl construiește, deci din fiecare arbore putem considera că alege diametrul (cel mai lung lanț).
- Dacă într-o zi avem la dispoziție K_i șine în plus, putem "lega" cu acestea $K_i + 1$ diametre (dacă există $K_i + 1$ diametre, dacă nu, le legăm pe toate și rămânem cu câteva șine în plus cu care nu facem nimic). Să notăm în continuare cu D_i numărul de diametre pe care le legăm defapt în ziua *i*.
- În fiecare zi este în mod evident optim să legăm cele mai mari D_i diametre pentru a maximiza lungimea totală.
- Răspunsul primei cerințe este dat de suma lungimilor celor mai mari D_i diametre la care se mai adaugă $D_i - 1$ (șinele adăugate în plus de RAU-Gigel pentru a lega diametrele respective).
- Se observă că răspunsul la a doua cerință este dat doar de diametrul de lungime minimă dintre cele D_i pentru că suntem obligați să le "luăm" în soluție pe **toate** acelea de lungime strict mai mare ca el (dacă există) pentru a maximiza suma. Astfel, dacă lungimea minimă din cele D_i apare de *q* ori între acestea și de *p* ori în toate diametrele arborilor din ziua curentă, răspunsul celei de-a doua cerințe în ziua respectivă va fi *C(p, q)* (numărul de submulțimi distincte de orașe care au lungimea diametrului egală cu cea a diametrului minim dintre cele D_i), unde am notat cu *C(N, K)* - combinări de *N* luate câte *K*.

Având în vedere cele menționate anterior și faptul că diametrul unui arbore se poate afla ușor în timp liniar cu programare dinamică sau cu 2 dfs-uri (https://infoarena.ro/problema/darb), deja este evidentă o soluție de tip "brute-force" de complexitate O(Q * N * $\log$ N), în care rezolvăm independent fiecare zi în același mod: aflăm toate diametrele din ziua respectivă, le sortăm și adunăm cele mai mari D_i pentru prima cerință, respectiv calculăm ocurențele necesare ale minimului din cele D_i la a doua cerință. Valorile combinărilor necesare pentru a doua cerință se pot precalcula în O(N^2) folosind recurența *C(N, K) = C(N – 1, K – 1) + C(N – 1, K)* la acest subtask (calculele trebuie făcute modulo $10^9 + 7$).

Pentru subtask-urile în care toate valorile M_i sunt egale cu 10^9 observăm că avem la dispoziție toate muchiile în fiecare zi, deci răspunsul primei cerințe va fi mereu diametrul arborelui dat inițial, iar la cerința 2 răspunsul va fi mereu 1 (avem la dispoziție mereu un singur diametru care e și unicul minim). Diametrul se poate determina la fel ca în soluția anterioară, deci complexitatea pentru aceste subtask-uri este O(N). De asemenea, se observă că pentru aceste subtask-uri nu este necesară calcularea combinărilor.

În continuare vom mai face o observație importantă pentru a încerca să rezolvăm problema pentru cazuri mai generale: poate fi avantajos să considerăm zilele în ordine crescătoare după M_i, pentru că astfel ar trebui pe parcurs doar să adăugăm noi muchii la pădurea de arbori curentă (inițial se pleacă de la *N* arbori fiecare cu câte

un nod), deci vom rezolva problema **offline**. Astfel avem de fapt nevoie de o structură de date (sau mai multe) cu care să realizăm următoarele operații:

- Unește doi arbori printr-o muchie și află (rapid) diametrul noului arbore obținut.
- Calculează suma celor mai mari *X* diametre, respectiv ce frecvență are o anumită lungime de diametru în mulțimea tuturor diametrelor.

Să ne ocupăm mai întâi de prima operație. Pentru a uni 2 arbori este evident că putem folosi **păduri de mulțimi disjuncte**, dar avem nevoie și de un mod eficient pentru a calcula diametrul noului arbore obținut. Pentru subtask-urile în care arborele inițial dat este un singur lanț lungimea diametrului este de fapt egală cu numărul de muchii din componentă, adică numărul de noduri - 1. Pentru cazul general, să zicem că vrem să unim 2 arbori ale căror diametre (sau unul dintre diametre dacă au mai multe) au capetele în nodurile *(u, v)*, respectiv *(p, q)* (o pereche poate avea și noduri egale, când arborele are doar un singur nod). O observație intuitivă este că diametrul noului arbore obținut va avea capetele în mulțimea {u, v, p, q}, deci putem doar încerca fiecare variantă de nou diametru candidat, numărul de candidați fiind constant. Demonstrația acestui fapt este lăsată ca exercițiu cititorului pentru că aceasta reiese imediat dintr-o reprezentare vizuală a celor 2 arbori "de-a lungul" diametrului fiecăruia și considerarea câtorva cazuri de discuție și a unor proprietăți ale distanțelor dintre noduri în reprezentarea respectivă. Astfel, pentru a finaliza această parte a soluției mai avem nevoie doar de un mod eficient pentru a calcula distanța dintre două noduri pentru a putea afla care este candidatul optim pentru noul diametru al arborelui obținut în urma adăugării unei noi muchii, ceea ce se poate face ușor aflând eficient distanțele de la rădăcină (care poate fi aleasă arbitrar) la fiecare nod și **LCA**-ul a două noduri (cel mai apropiat strămoș comun al lor în arborele inițial dat) prin formula: *d[u] + d[v] – 2 * d[lca(u, v)]*, unde am notat cu u, respectiv v cele două noduri între care vrem să aflăm distanța, cu *lca(u, v)* - cel mai apropiat strămoș comun al lor, iar *d[nod]* reprezintă distanța de la rădăcină la *nod*. Pentru a calcula eficient LCA-urile se recomandă folosirea algoritmului cu **RMQ** (Range Minimum Query) pe parcurgerea Euler a arborelui (https://infoarena.ro/problema/lca), query-urile fiind rezolvate astfel în timp constant, dar nu este exclus ca și algoritmul de aflare cu **binary lifting** să se încadreze în timp.

În cazul celei de-a doua operații, pentru subtask-urile cu $K_i = 1$ avem nevoie mereu doar de cele mai mari două diametre (dacă există două, dacă nu, avem nevoie doar de maxim) și de un vector de frecvență pentru lungimile diametrelor. De asemenea, structura de date pe care o folosim pentru aflarea celor mai mari diametre trebuie să suporte inserări și ștergeri. Astfel, un **set** din STL pare a fi ideal în rezolvarea acestor subtask-uri. La ștergeri și inserări vom actualiza întâi vectorul de frecvență și vom modifica set-ul doar dacă este necesar (astfel este necesar doar set, nu **multiset**, pentru că reținem doar o singură dată fiecare lungime, iar frecvența sa o reținem separat). De asemenea, nici în acest caz nu este necesară o calculare a combinărilor pentru că la cerința doi, în funcție de caz, numărul de variante este dat de câte o formulă destul de simplă. Astfel, complexitatea pentru aceste subtask-uri este $O(N * \log N)$.

Rămâne doar de rezolvat problema pe cazul general, adică pentru K_i arbitrar dat. Tot ce rămâne de făcut este să înlocuim structura de date set cu un **arbore de intervale pe lungimile diametrelor** (lungimea unui diametru ia valori în intervalul [0, N – 1]), care calculează suma lungimilor **tuturor** diametrelor dintr-un interval de lungimi distincte. Astfel, inserările și ștergerile devin un update pe poziție, iar aflarea sumei celor mai mari K_i diametre devine o "căutare binară" pe acest arbore. De asemenea, este evident și că aflarea ocurențelor necesare diametrului de lungime minimă dintre cele D_i la cerința 2 se poate face simplu cu un query pe poziție în arborele de intervale și niște calcule. Mai este necesară doar precalcularea factorialelor și a inverselor lor modulare (care se poate face în timp liniar sau în $O(N * \log(10^9 + 7))$) pentru a putea calcula valorile combinărilor necesare la cerința 2 prin formula cunoscută cu factoriale (https://en.wikipedia.org/wiki/Combination). Astfel, complexitatea temporală a algoritmului final este $O(N * \log N)$.

Bonus: Problema poate fi rezolvată și dacă muchiile arborelui au costuri numere naturale nenule mai mici sau egale decât 10^9 (adică să nu fie toate egale cu 1 ca în versiunea din concurs), exact pe aceeași idee, dar este

necesară reținerea doar a lungimilor **distincte** relevante pe parcurs prin rularea întâi a tuturor operațiilor de adăugare de muchii pentru fiecare zi (zilele fiind considerate în ordine crescătoare după M_i, în aceeași manieră ca în soluția descrisă), fără a rezolva cerințele, apoi **normalizându-se** lungimile obținute. Astfel, arborele de intervale se face pe lungimile normalizate și se rezolvă cerințele ca în versiunea inițială a problemei. O altă variantă care nu necesită normalizarea lungimilor în prealabil este folosirea structurii de date **treap** (sau a unui alt arbore binar de căutare echilibrat care suportă operațiile necesare) în locul arborelui de intervale, logica operațiilor fiind apoi similară cu cea din soluția cu arbore de intervale. Totuși, aceste lucruri nu aduc nimic interesant în plus problemei pentru că doar ar mai complica puțin implementarea (care oricum nu este una trivială nici în variantă dată la concurs), ideea de rezolvare a problemei fiind păstrată în totalitate, deci nu au fost considerate necesare.

Soluție

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

const int kN = 1e5;
const int kM = 2e5;
const int kLog = 17;
const int mod = 1e9 + 7;
int m, f[1 + kN], invf[1 + kN], sol[1 + kN], first[1 + kN], tour[1 + kM], dep[1 + kM], lg2[1 + kM], rmq[1 + kLog][1 + kM];
vector<int> g[1 + kN];

struct edge_t {
    int u, v, w;

    void read() {
        cin >> u >> v >> w;
    }

    void addEdge() {
        g[u].emplace_back(v);
        g[v].emplace_back(u);
    }

    bool operator < (const edge_t& rhs) const {
        return w < rhs.w;
    }
} edges[kN];

struct query_t {
    int m, k, index;

    void read() {
        cin >> m >> k;
    }

    bool operator < (const query_t& rhs) const {
        return m < rhs.m;
    }
} q[kN];

struct node {
    int sum, cnt;

    node operator + (const node& rhs) const {
        return { sum + rhs.sum, cnt + rhs.cnt };
```

```
    }
} NIL{ 0, 0 };

struct ST {
    int n;
    vector<node> t;

    void init(int N) {
        n = N;
        int dim = 1;
        while (dim < n) {
            dim *= 2;
        }
        t.assign(dim * 2, NIL);
    }

    void update(int x, int lx, int rx, int pos, int v) {
        if (lx == rx) {
            t[x].sum += v * lx;
            t[x].cnt += v;
            return;
        }
        int mid = (lx + rx) / 2;
        if (pos <= mid) {
            update(x * 2, lx, mid, pos, v);
        }
        else {
            update(x * 2 + 1, mid + 1, rx, pos, v);
        }
        t[x] = t[x * 2] + t[x * 2 + 1];
    }

    void update(int pos, int v) {
        update(1, 0, n - 1, pos, v);
    }

    int queryPos(int x, int lx, int rx, int req) {
        if (lx == rx) {
            if (t[x].cnt < req) {
                return lx;
            }
            return lx + 1;
        }
        int mid = (lx + rx) / 2;
        if (t[x * 2 + 1].cnt < req) {
            return queryPos(x * 2, lx, mid, req - t[x * 2 + 1].cnt);
        }
        return queryPos(x * 2 + 1, mid + 1, rx, req);
    }

    node query(int x, int lx, int rx, int st, int dr) {
        if (st <= lx && rx <= dr) {
            return t[x];
        }
        int mid = (lx + rx) / 2;
        node left = NIL, right = NIL;
        if (st <= mid) {
            left = query(x * 2, lx, mid, st, dr);
        }
        if (mid < dr) {
            right = query(x * 2 + 1, mid + 1, rx, st, dr);
        }
        return left + right;
```

```
    }

    node query(int st, int dr) {
        return query(1, 0, n - 1, st, dr);
    }
} t;

void multSelf(int& x, const int& y) {
    x = (int64_t)x * y % mod;
}

int mult(int x, const int& y) {
    multSelf(x, y);
    return x;
}

int Pow(int x, int n) {
    int ans = 1;
    while (n != 0) {
        if (n % 2 == 1) {
            multSelf(ans, x);
        }
        multSelf(x, x);
        n /= 2;
    }
    return ans;
}

int invers(int x) {
    return Pow(x, mod - 2);
}

void precalc() {
    f[0] = f[1] = 1;
    for (int i = 2; i <= kN; ++i) {
        f[i] = mult(f[i - 1], i);
    }
    invf[kN] = invers(f[kN]);
    for (int i = kN - 1; i >= 0; --i) {
        invf[i] = mult(invf[i + 1], i + 1);
    }
    for (int i = 2; i <= kM; ++i) {
        lg2[i] = lg2[i / 2] + 1;
    }
    for (int i = 1; i <= kM; ++i) {
        rmq[0][i] = i;
    }
}

int nck(int n, int k) {
    return mult(f[n], mult(invf[k], invf[n - k]));
}

void dfs(int u, int par, int level) {
    tour[++m] = u;
    dep[m] = level;
    first[u] = m;
    for (int v : g[u]) {
        if (v != par) {
            dfs(v, u, level + 1);
            tour[++m] = u;
            dep[m] = level;
        }
```

```
    }
}

void computeRmq() {
    for (int i = 1; (1 << i) <= m; ++i) {
        for (int j = 1; j <= m - (1 << i) + 1; ++j) {
            int l = (1 << (i - 1));
            rmq[i][j] = rmq[i - 1][j];
            if (dep[rmq[i - 1][j + l]] < dep[rmq[i][j]]) {
                rmq[i][j] = rmq[i - 1][j + l];
            }
        }
    }
}

int lca(int u, int v, bool flag = false) {
    int x = first[u], y = first[v];
    if (y < x) {
        swap(x, y);
    }
    int len = y - x + 1;
    int l = lg2[len];
    int index = rmq[l][x];
    int shift = len - (1 << l);
    if (dep[rmq[l][x + shift]] < dep[index]) {
        index = rmq[l][x + shift];
    }
    return tour[index];
}

int dist(int u, int v) {
    return dep[first[u]] + dep[first[v]] - 2 * dep[first[lca(u, v)]];
}

struct DSU {
    vector<int> p, sz;
    vector<array<int, 2>> ends;

    DSU(int n) {
        p.resize(n + 1);
        iota(p.begin(), p.end(), 0);
        sz.assign(n + 1, 1);
        ends.resize(n + 1);
        for (int i = 1; i <= n; ++i) {
            ends[i] = { i, i };
        }
    }

    int root(int x) {
        if (x == p[x]) {
            return x;
        }
        return p[x] = root(p[x]);
    }

    bool unite(int u, int v) {
        int x = root(u), y = root(v);
        if (x == y) {
            return false;
        }
        if (sz[y] < sz[x]) {
            swap(x, y);
        }
```

```
        array<int, 2> nodes;
        int dX = dist(ends[x][0], ends[x][1]), dY = dist(ends[y][0], ends[y][1]);
        if (dX < dY) {
            nodes = ends[y];
        }
        else {
            nodes = ends[x];
        }
        int best = dist(nodes[0], nodes[1]);
        for (const int& st : ends[x]) {
            for (const int& dr : ends[y]) {
                int ret = dist(st, dr);
                if (best < ret) {
                    best = ret;
                    nodes = { st, dr };
                }
            }
        }
        t.update(dX, -1);
        t.update(dY, -1);
        t.update(best, 1);
        ends[y] = nodes;
        p[x] = y;
        sz[y] += sz[x];
        return true;
    }
};

void testCase() {
    int task, n;
    cin >> task >> n;
    for (int i = 1; i <= n; ++i) {
        g[i].clear();
    }
    for (int i = 0; i < n - 1; ++i) {
        edges[i].read();
        edges[i].addEdge();
    }
    sort(edges, edges + n - 1);
    int Q;
    cin >> Q;
    for (int i = 0; i < Q; ++i) {
        q[i].read();
        q[i].k += 1;
        q[i].index = i;
    }
    m = 0;
    dfs(1, 0, 0);
    computeRmq();
    sort(q, q + Q);
    DSU dsu(n);
    t.init(n);
    t.update(0, n);
    int ptr = 0;
    for (int i = 0; i < Q; ++i) {
        while (ptr < n - 1 && edges[ptr].w <= q[i].m) {
            dsu.unite(edges[ptr].u, edges[ptr].v);
            ptr += 1;
        }
        int minD = t.queryPos(1, 0, n - 1, q[i].k);
        if (minD) {
            minD -= 1;
        }
```

```
            node ret = t.query(minD, n - 1);
            int ways = 1;
            if (ret.cnt <= q[i].k) {
                q[i].k = ret.cnt;
            }
            else {
                node last = t.query(minD, minD);
                int rem = ret.cnt - q[i].k;
                ret.sum -= rem * minD;
                ways = nck(last.cnt, rem);
            }
            if (task == 1) {
                sol[q[i].index] = ret.sum + q[i].k - 1;
            }
            else {
                sol[q[i].index] = ways;
            }
        }
        for (int i = 0; i < Q; ++i) {
            cout << sol[i] << '\n';
        }
    }

    int main() {
        precalc();
        int tests;
        cin >> tests;
        for (int tc = 0; tc < tests; ++tc) {
            testCase();
        }
        return 0;
    }
```