# ? [UFMG]

Bruno Monteiro

## Índice

# 1 Estruturas

## 1.1 BIT

```
// BIT 1-based, v 0-based
// Para mudar o valor da posicao p para x,
// faca: poe(x - query(p, p), p)
// l_bound(x) retorna o menor p tal que
// query(1, p+1) > x    (0 based!)
//
// Complexidades:
// build - O(n)
// poe - O(log(n))
// query - O(log(n))
// l_bound - O(log(n))

int n;
int bit[MAX];
int v[MAX];

void build() {
    bit[0] = 0;
    for (int i = 1; i <= n; i++) bit[i] = v[i - 1];

    for (int i = 1; i <= n; i++) {
        int j = i + (i & -i);
        if (j <= n) bit[j] += bit[i];
    }
}

// soma x na posicao p
void poe(int x, int p) {
    for (; p <= n; p += p & -p) bit[p] += x;
}

// soma [1, p]
int pref(int p) {
    int ret = 0;
    for (; p; p -= p & -p) ret += bit[p];
    return ret;
}

// soma [a, b]
int query(int a, int b) {
    return query(b) - query(a - 1);
}
```

```
int l_bound(ll x) {
    int p = 0;
    for (int i = MAX2; i+1; i--) if (p + (1<<i) <= n
        and bit[p + (1<<i)] <= x) x -= bit[p += (1<<i)];
    return p;
}
```

## 1.2 BIT 2D

```
// BIT 1-based
// Para mudar o valor da posicao (x, y) para k,
// faca: poe(x, y, k - sum(x, y, x, y))
//
// Complexidades:
// poe - O(log^2(n))
// query - O(log^2(n))

int n;
int bit[MAX][MAX];

void poe(int x, int y, int k) {
    for (int y2 = y; x <= n; x += x & -x)
        for (y = y2; y <= n; y += y & -y)
            bit[x][y] += k;
}

int sum(int x, int y) {
    int ret = 0;
    for (int y2 = y; x; x -= x & -x)
        for (y = y2; y; y -= y & -y)
            ret += bit[x][y];

    return ret;
}

int query(int x, int y, int z, int w) {
    return sum(z, w) - sum(x-1, w)
        - sum(z, y-1) + sum(x-1, y-1);
}
```

## 1.3 Mergesort Tree

```
// query(1, 0, n-1) retorna numero de
// elementos em [a, b] <= val
// Usa O(n log(n)) de memoria
//
// Complexidades:
// build - O(n log(n))
// query - O(log^2(n))

#define ALL(x) x.begin(),x.end()

int v[MAX];
vector<vector<int> > tree(4*MAX);
int n, a, b val;

void build(int p, int l, int r) {
    if (l == r) {
        tree[p].pb(v[l]);
        return;
    }
    int m = (l+r)/2;
    build_tree(2*p, l, m);
    build_tree(2*p+1, m+1, r);
    merge(ALL(tree[2*p]), ALL(tree[2*p+1]),
        back_inserter(tree[p]));
}

int query(int p, int l, int r) {
    if (b < l or r < a) return 0; // to fora
    if (a <= l and r <= b) // to totalmente dentro
        return lower_bound(ALL(tree[p]), val+1) -
            tree[p].begin();
    int m = (l+r)/2;
    return query(2*p, l, m) + query(2*p+1, m+1, r);
}
```

## 1.4 Order Statistic Set

```
// Funciona do C++11 pra cima

#include <ext/pb_ds/assoc_container.hpp>
```

```
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template <class T>
    using ord_set = tree<T, null_type, less<T>, rb_tree_tag,
    tree_order_statistics_node_update>;

// para declarar:
ord_set<int> s;
// coisas do set normal funcionam:
for (auto i : s) cout << i << endl;
cout << s.size() << endl;
// k-esimo maior elemento O(log|s|):
// k=0: menor elemento
cout << *s.find_by_order(k) << endl;
// quantos sao menores do que k O(log|s|):
cout << s.order_of_key(k) << endl;

// Para fazer um multiset, tem que
// usar ord_set<pair<int, int> > com o
// segundo parametro sendo algo para diferenciar
// os ementos iguais.
// s.order_of_key({k, -INF}) vai retornar o
// numero de elementos < k
```

## 1.5 SQRT-decomposition

```
// 0-indexed
// MAX2 = sqrt(MAX)
//
// O bloco da posicao x eh
// sempre x/q
//
// Complexidades:
// build - O(n)
// query - O(sqrt(n))

int n, q;
int v[MAX];
int bl[MAX2];

void build() {
    q = (int) sqrt(n);

     // computa cada bloco
    for (int i = 0; i <= q; i++) {
        bl[i] = INF;
        for (int j = 0; j < q and q * i + j < n; j++)
            bl[i] = min(bl[i], v[q * i + j]);
    }
}

int query(int a, int b) {
    int ret = INF;

    // linear no bloco de a
    for (; a <= b and a % q; a++) ret = min(ret, v[a]);

    // bloco por bloco
    for (; a + q <= b; a += q) ret = min(ret, bl[a / q]);

    // linear no bloco de b
    for (; a <= b; a++) ret = min(ret, v[a]);

    return ret;
}
```

## 1.6 Seg-Tree

```
// Query: soma do range [a, b]
// Update: soma x em cada elemento do range [a, b]
//
// Complexidades:
// build - O(n)
// query - O(log(n))
// update - O(log(n))

int seg[4*MAX];
int lazy[4*MAX];
int v[MAX];
int n, a, b, x;

int build(int p, int l, int r) {
    lazy[p] = 0;
    if (l == r) return seg[p] = v[l];
```

```
    int m = (l+r)/2;
    return seg[p] = build(2*p, l, m) + build(2*p+1, m+1, r);
}

void prop(int p, int l, int r) {
    seg[p] += lazy[p] * (r-l+1);
    if (l != r) lazy[2*p] += x, lazy[2*p+1] += x;
    lazy[p] = 0;
}

int query(int p, int l, int r) {
    prop(p, l, r);
    if (a <= l and r <= b) return seg[p];
    if (b < l or r < a) return 0;

    int m = (l+r)/2;
    return query(2*p, l, m) + query(2*p+1, m+1, r);
}

int update(int p, int l, int r) {
    prop(p, l, r);
    if (a <= l and r <= b) {
        if (l != r) lazy[2*p] += x, lazy[2*p+1] += x;
        return seg[p] += x * (r-l+1);
    }
    if (b < l or r < a) return seg[p];

    int m = (l+r)/2;
    return seg[p] = update(2*p, l, m) + update(2*p+1, m+1,
       r);
}
```

## 1.7 Seg-Tree 2D

```
// Consultas 0-based
// Um valor inicial em (x, y) deve ser colocado em
   seg[x+n][y+n]
// Query: soma do retangulo ((x1, y1), (x2, y2))
// Update: muda o valor da posicao (x, y) para val
// Nao pergunte como que essa coisa funciona
//
// Para query com distancia de manhattan <= d, faca
// nx = x+y, ny = x-y
// Update em (nx, ny), query em ((nx-d, ny-d), (nx+d, ny+d))
//
// Complexidades:
// build - O(n^2)
// query - O(log^2(n))
// update - O(log^2(n))

int seg[2*MAX][2*MAX];
int n;

void build() {
    for (int x = 2*n; x; x--) for (int y = 2*n; y; y--) {
        if (x < n) seg[x][y] = seg[2*x][y] + seg[2*x+1][y];
        if (y < n) seg[x][y] = seg[x][2*y] + seg[x][2*y+1];
    }
}

int query(int x1, int y1, int x2, int y2) {
    int ret = 0, y3 = y1 + n, y4 = y2 + n;
    for (x1 += n, x2 += n; x1 <= x2; ++x1 /= 2, --x2 /= 2)
        for (y1 = y3, y2 = y4; y1 <= y2; ++y1 /= 2, --y2 /=
           2) {
            if (x1%2 == 1 and y1%2 == 1) ret += seg[x1][y1];
            if (x1%2 == 1 and y2%2 == 0) ret += seg[x1][y2];
            if (x2%2 == 0 and y1%2 == 1) ret += seg[x2][y1];
            if (x2%2 == 0 and y2%2 == 0) ret += seg[x2][y2];
        }

    return ret;
}

void update(int x, int y, int val) {
    int y2 = y += n;
    for (x += n; x; x /= 2, y = y2) {
        if (x >= n) seg[x][y] = val;
        else seg[x][y] = seg[2*x][y] + seg[2*x+1][y];

        while (y /= 2) seg[x][y] = seg[x][2*y] +
           seg[x][2*y+1];
    }
```

```
}
```

## 1.8 Seg-Tree Iterativa

```
// Consultas 0-based
// Valores iniciais devem estar em (seg[n], ... , seg[2*n-1])
// Query: soma do range [a, b]
// Update: muda o valor da posicao p para x
//
// Complexidades:
// build - O(n)
// query - O(log(n))
// update - O(log(n))

int seg[2 * MAX];
int n;

void build() {
    for (int i = n - 1; i; i--) seg[i] = seg[2*i] +
        seg[2*i+1];
}

int query(int a, int b) {
    int ret = 0;
    for(a += n, b += n; a <= b; ++a /= 2, --b /= 2) {
        if (a % 2 == 1) ret += seg[a];
        if (b % 2 == 0) ret += seg[b];
    }
    return ret;
}

void update(int p, int x) {
    seg[p += n] = x;
    while (p /= 2) seg[p] = seg[2*p] + seg[2*p+1];
}
```

## 1.9 Sparse-Table

```
// MAX2 = log(MAX)
//
// Complexidades:
// build - O(n log(n))
// query - O(1)

int n;
int v[MAX];
int m[MAX][MAX2]; // m[i][j] : posicao do minimo
                  // em [v[i], v[i + 2^j - 1]]

void build() {
    for (int i = 0; i < n; i++) m[i][0] = i;

    for (int j = 1; 1 << j <= n; j++) {
        int tam = 1 << j;
        for (int i = 0; i + tam <= n; i++) {
            if (v[m[i][j - 1]] < v[m[i + tam/2][j - 1]])
                m[i][j] = m[i][j - 1];
            else m[i][j] = m[i + tam/2][j - 1];
        }
    }
}

int query(int a, int b) {
    int j = (int) log2(b - a + 1);

    return min(v[m[a][j]], v[m[b - (1 << j) + 1][j]]);
}
```

## 1.10 Trie

```
// N deve ser maior ou igual ao numero de nos da trie
// fim indica se alguma palavra acaba nesse no
//
// Complexidade:
// Inserir e conferir string S -> O(|S|)

int trie[N][26], fim[N], nx;

void insere(string &s, int p, int l, int at){
    // se nao chegou no fim da palavra termina de inserir
    if(p != l){
        int c = s[p] - 'a';
        // se nao existe um no que representa esse prefixo +
```

```
            c
        // cria o no
        if(!trie[at][c]) trie[at][c] = nx++;
        insere(s, p+1, l, trie[at][c]);
    }
    else fim[at] = 1;
}

int check(string &s, int p, int l, int at){
    if(p != l){
        int c = s[p] - 'a';
        if(trie[at][c]) return check(s, p+1, l, trie[at][c]);
        return 0;
    }
    return fim[at];
}
```

## 1.11 Union-Find

```
// Complexidades:
// build - O(n)
// find - O(1)
// une - O(1)

int n;
int v[MAX];     // v[i] : representante do conjunto que
   contem i
int size[MAX]; // size[i] : tamanho do conjunto que tem i
   como representante

void build() {
    for (int i = 0; i < n; i++) {
        v[i] = i;
        size[i] = 1;
    }
}

int find(int k) {
    return v[k] == k ? k : v[k] = find(v[k]);
}

void une(int a, int b) {
    a = find(a);
    b = find(b);
    if (size[a] > size[b]) swap(a, b);

    size[b] += size[a];
    v[a] = b;
}
```

# 2 Grafos

## 2.1 2-SAT

```
// Retorna se eh possivel atribuir valores
// Grafo tem que caber 2n vertices
// add(x, y) adiciona implicacao x -> y
// Para adicionar uma clausula (x ou y)
// chamar add(nao(x), y)
// Se x tem que ser verdadeiro, chamar add(nao(x), x)
// O tarjan deve computar o componente conexo
// de cada vertice em comp
//
// O(|V|+|E|)

vector<vector<int> > g(MAX);
int n;

int nao(int x){ return (x + n) % (2*n); }

// x -> y  =  !x ou y
void add(int x, int y){
    g[x].pb(y);
    // contraposicao
    g[nao(y)].pb(nao(x));
}

bool doisSAT(){
    tarjan();
    for (int i = 0; i < m; i++)
        if (comp[i] == comp[nao(i)]) return 0;
    return 1;
```

```
}
```

## 2.2 Bellman-Ford

```
// Calcula a menor distancia
// entre a e todos os vertices e
// detecta ciclo negativo
// Retorna 1 se ha ciclo negativo
// Nao precisa representar o grafo,
// soh armazenar as arestas
//
// O(nm)

int n, m;
int d[MAX];
vector<pair<int, int> > ar; // vetor de arestas
vector<int> w;              // peso das arestas

bool bellman_ford(int a) {
    for (int i = 0; i < n; i++) d[i] = INF;
    d[a] = 0;

    for (int i = 0; i <= n; i++)
        for (int j = 0; j < m; j++) {
            if (d[ar[j].second] > d[ar[j].first] + w[j]) {
                if (i == n)
                    return 1;

                d[ar[j].second] = d[ar[j].first] + w[j];
            }
        }

    return 0;
}
```

## 2.3 Floyd-Warshall

```
//
// encontra o menor caminho entre todo
// par de vertices e detecta ciclo negativo
// returna 1 sse ha ciclo negativo
// d[i][i] deve ser 0
// para i != j, d[i][j] deve ser w se ha uma aresta
// (i, j) de peso w, INF caso contrario
//
// O(n^3)

int n;
int d[MAX][MAX];

bool floyd_warshall() {
    for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        d[i][j] = min(d[i][j], d[i][k] + d[k][j]);

    for (int i = 0; i < n; i++)
        if (d[i][i] < 0) return 1;

    return 0;
}
```

## 2.4 Heavy-Light Decomposition

```
// SegTree de maximo
// query_path(a, b) calcula maior aresta
// no caminho de a pra b
// query_subtree(a) calcula maior aresta
// na subarvore dos filhos de a (inclusive)
// update(p, val) muda o peso da aresta
// que vai de p para o pai de p para val
//
// SegTree pode ser facilmente modificada
//
// Complexidades:
// build - O(n)
// query_path - O(log^2 (n))
// query_subtree - O(log(n))
// update - O(log(n))

#define f first
#define s second
```

```cpp
vector<vector<pair<int, int> > > g(MAX);
int in[MAX], out[MAX], sz[MAX];
int sobe[MAX], pai[MAX];
int head[MAX], v[MAX], t;

// seg tree sobre o vetor v de tamanho t
void build_seg();
int query_seg(int a, int b);
void update_seg(int p, int x);

void hld(int k, int p = -1, int f = 1) {
    v[in[k] = t++] = sobe[k]; sz[k] = 1;
    for (auto& i : g[k]) if (i.f != p) {
        sobe[i.f] = i.s; pai[i.f] = k;
        head[i.f] = (i == g[k][0] ? head[k] : i.f);
        hld(i.f, k, f); sz[k] += sz[i.f];

        if (sz[i.f] > sz[g[k][0].f]) swap(i, g[k][0]);
    }
    out[k] = t;
    if (p*f == -1) hld(head[k] = k, -1, t = 0);
}

void build(int root = 0) {
    t = 0;
    hld(root);
    build_seg();
}

int query_path(int a, int b) {
    if (a == b) return -INF;
    if (in[a] < in[b]) swap(a, b);

    if (head[a] == head[b]) return query_seg(in[b]+1, in[a]);
    return max(query_seg(in[head[a]], in[a]),
        query_path(pai[head[a]], b));
}

int query_subtree(int a) {
    if (in[a] == out[a]-1) return -INF;;
    return query_seg(in[a]+1, out[a]-1);
}

void update(int p, int val) {
    update_seg(in[p], val);
}
```

## 2.5 LCA

```cpp
// Assume que um vertice eh ancestral dele mesmo, ou seja,
// se a eh ancestral de b, lca(a, b) = a
// MAX2 = ceil(log(MAX))
//
// Complexidades:
// build - O(n log(n))
// lca - O(log(n))

vector<vector<int> > g(MAX);
int n, p;
int pai[MAX2][MAX];
int in[MAX], out[MAX];

void dfs(int k) {
    in[k] = p++;
    for (int i = 0; i < (int) g[k].size(); i++)
        if (in[g[k][i]] == -1) {
            pai[0][g[k][i]] = k;
            dfs(g[k][i]);
        }
    out[k] = p++;
}

void build(int raiz) {
    for (int i = 0; i < n; i++) pai[0][i] = i;
    p = 0, memset(in, -1, sizeof in);
    dfs(raiz);

    // pd dos pais
    for (int k = 1; k < MAX2; k++) for (int i = 0; i < n;
       i++)
        pai[k][i] = pai[k - 1][pai[k - 1][i]];
}

bool anc(int a, int b) { // se a eh ancestral de b
```

```
    return in[a] <= in[b] and out[a] >= out[b];
}

int lca(int a, int b) {
    if (anc(a, b)) return a;
    if (anc(b, a)) return b;

    // sobe a
    for (int k = MAX2 - 1; k >= 0; k--)
        if (!anc(pai[k][a], b)) a = pai[k][a];

    return pai[0][a];
}
```

## 2.6 LCA com HLD

```
// Assume que um vertice eh ancestral dele mesmo, ou seja,
// se a eh ancestral de b, lca(a, b) = a
// Para buildar pasta chamar build(root)
//
// Complexidades:
// build - O(n)
// lca - O(log(n))

vector<vector<int> > g(MAX);
int in[MAX], h[MAX], sz[MAX];
int pai[MAX], t;

void build(int k, int p = -1, int f = 1) {
    in[k] = t++; sz[k] = 1;
    for (int& i : g[k]) if (i != p) {
        pai[i] = k;
        h[i] = (i == g[k][0] ? h[k] : i);
        build(i, k, f); sz[k] += sz[i];

        if (sz[i] > sz[g[k][0]]) swap(i, g[k][0]);
    }
    if (p*f == -1) t = 0, h[k] = k, build(k, -1, 0);
}

int lca(int a, int b) {
    if (in[a] < in[b]) swap(a, b);
    return h[a] == h[b] ? b : lca(pai[h[a]], b);
}
```

## 2.7 LCA com RMQ

```
// Assume que um vertice eh ancestral dele mesmo, ou seja,
// se a eh ancestral de b, lca(a, b) = a
//
// Complexidades:
// build - O(n) + build_RMQ
// lca - RMQ

int n;
vector<vector<int> > g(MAX);
int pos[MAX];     // pos[i] : posicao de i em v (primeira
    aparicao
int ord[2 * MAX]; // ord[i] : i-esimo vertice na ordem de
    visitacao da dfs
int v[2 * MAX];   // vetor de alturas que eh usado na RMQ
int p;

void dfs(int k, int l) {
    ord[p] = k;
    pos[k] = p;
    v[p++] = l;
    for (int i = 0; i < (int) g[k].size(); i++)
        if (pos[g[k][i]] == -1) {
            dfs(g[k][i], l + 1);
            ord[p] = k;
            v[p++] = l;
        }
}

void build(int root) {
    for (int i = 0; i < n; i++) pos[i] = -1;

    p = 0;
    dfs(root, 0);

    build_RMQ();
}
```

```
int lca(int u, int v) {
    int a = pos[u], b = pos[v];
    if (a > b) swap(a, b);
    return ord[RMQ(a, b)];
}
```

## 2.8 Tree Center

```
// Centro eh o vertice que minimiza
// a maior distancia dele pra alguem
// O centro fica no meio do diametro
// A funcao center retorna um par com
// o diametro e o centro
//
// O(n+m)

vector<vector<int> > g(MAX);
int n, vis[MAX];
int d[2][MAX];

// retorna ultimo vertice visitado
int bfs(int k, int x) {
        queue<int> q; q.push(k);
    memset(vis, 0, sizeof(vis));
    vis[k] = 1;
    d[x][k] = 0;
    int last = k;

    while (q.size()) {
        int u = q.front(); q.pop();
        last = u;
        for (int i : g[u]) if (!vis[i]) {
            vis[i] = 1;
            q.push(i);
            d[x][i] = d[x][u] + 1;
        }
    }
    return last;
}

pair<int, int> center() {
    int a = bfs(0, 0);
    int b = bfs(a, 1);
    bfs(b, 0);
    int c, mi = INF;
    for (int i = 0; i < n; i++) if (max(d[0][i], d[1][i]) <
       mi) {
        mi = max(d[0][i], d[1][i]), c = i;
    return {d[0][a], c};
}
```

## 2.9 Centroid decomposition

```
// O(n log(n))

int n;
vector<vector<int> > g(MAX);
int subsize[MAX];
int rem[MAX];
int pai[MAX];

void dfs(int k, int last) {
    subsize[k] = 1;
    for (int i = 0; i < (int) g[k].size(); i++)
        if (g[k][i] != last and !rem[g[k][i]]) {
            dfs(g[k][i], k);
            subsize[k] += subsize[g[k][i]];
        }
}

int centroid(int k, int last, int size) {
    for (int i = 0; i < (int) g[k].size(); i++) {
        int u = g[k][i];
        if (rem[u] or u == last) continue;
        if (subsize[u] > size / 2)
            return centroid(u, k, size);
    }
    // k eh o centroid
    return k;
}

void decomp(int k, int last) {
    dfs(k, k);
```

```
    // acha e tira o centroid
    int c = centroid(k, k, subsize[k]);
    rem[c] = 1;
    pai[c] = last;
    if (k == last) pai[c] = c;

    // decompoe as sub-arvores
    for (int i = 0; i < (int) g[c].size(); i++)
        if (!rem[g[c][i]]) decomp(g[c][i], c);
}

void build() {
    memset(rem, 0, sizeof rem);
    decomp(0, 0);
}
```

## 2.10 Dijkstra

```
// encontra menor distancia de a
// para todos os vertices
// se ao final do algoritmo d[i] = INF,
// entao a nao alcanca i
//
// O(m log(n))

int n;
vector<vector<int> > g(MAX);
vector<vector<int> > w(MAX); // peso das arestas
int d[MAX];

void dijsktra(int a) {
    for (int i = 0; i < n; i++) d[i] = INF;
    d[a] = 0;
    priority_queue<pair<int, int> > Q;
    Q.push(make_pair(0, a));

    while (Q.size()) {
        int u = Q.top().second, dist = -Q.top().first;
        Q.pop();
        if (dist > d[u]) continue;

        for (int i = 0; i < (int) g[u].size(); i++) {
            int v = g[u][i];
            if (d[v] > d[u] + w[u][i]) {
                d[v] = d[u] + w[u][i];
                Q.push(make_pair(-d[v], v));
            }
        }
    }
}
```

## 2.11 Dinic

```
// tem que definir o tamanho de g e de lev como o numero
// de vertices do grafo e depois char o a funcao fluxo
//
// Complexidade:
// Caso geral: O(V^2 * E)
// Grafo bipartido O(sqrt(V)*E)

#define INF 0x3f3f3f3f

struct edge{
    int p, c, id; // destino, capacidade, id
    edge() {p = c = id = 0;}
    edge(int p, int c, int id):p(p), c(c), id(id){}
};

vector<vector<edge> > g; // define o tamanho depois
vector <int> lev;

void add(int a, int b, int c){
    // de a para b com capacidade c
    edge d = {b, c, (int) g[b].size()};
    edge e = {a, 0, (int) g[a].size()};
    g[a].pb(d);
    g[b].pb(e);
}

bool bfs(int s, int t){
    // bfs de s para t construindo o level
    for(int i = 0; i < g.size(); i++)
        lev[i] = -1;
    lev[s] = 0;
```

```
    // bfs saindo de s
    queue <int> q;
    q.push(s);
    while(q.size()){
        int u = q.front(); q.pop();

        for(int i = 0; i < g[u].size(); i++){
            edge e = g[u][i];
            // se ja foi visitado ou nao tem capacidade nao
               visita
            if(lev[e.p] != -1 || !e.c) continue;
            lev[e.p] = lev[u] + 1;
            if(e.p == t) return true;
            q.push(e.p);
        }
    }

    return false;
}

int dfs(int v, int s, int f){
    if(v == s || !f) return f;

    int flu = f;
    for(int i = 0; i < g[v].size(); i++){
        edge e = g[v][i]; int u = e.p;

        // visita se tiver capacidadade e se ta no proximo
           nivel
        if(lev[u] != lev[v] + 1 || !e.c) continue;

        int tenta = dfs(u, s, min(flu, e.c));
        // se passou alguma coisa altera as capacidades
        if(tenta){
            flu -= tenta;
            g[v][i].c -= tenta;
            g[u][e.id].c += tenta;
        }
    }

    // se passou tudo tira da lista dos possiveis
    if(flu == f) lev[v] = -1;
    return f - flu;
}

int fluxo(int s, int t){
    int r = 0;
    while(bfs(s, t)) r += dfs(s, t, INF);
    return r;
}

// ja tem ate o debug
void imprime(){
    for(int i = 0; i < g.size(); i++){
        printf("%i -> ", i);
        for(int j = 0; j < g[i].size(); j++)
            printf("(%i %i)", g[i][j].p, g[i][j].c);
        printf("\n");
    }
    printf("\n");
}
```

## 2.12 Kosaraju

```
// O(n + m)

int n;
vector<vector<int> > g(MAX);
vector<vector<int> > gi(MAX); // grafo invertido
int vis[MAX];
stack<int> S;
int comp[MAX]; // componente conexo de cada vertice

void dfs(int k) {
    vis[k] = 1;
    for (int i = 0; i < (int) g[k].size(); i++)
        if (!vis[g[k][i]]) dfs(g[k][i]);

    S.push(k);
}

void scc(int k, int c) {
    vis[k] = 1;
```

```
    comp[k] = c;
    for (int i = 0; i < (int) gi[k].size(); i++)
        if (!vis[gi[k][i]]) scc(gi[k][i], c);
}

void kosaraju() {
    for (int i = 0; i < n; i++) vis[i] = 0;
    for (int i = 0; i < n; i++) if (!vis[i]) dfs(i);

    for (int i = 0; i < n; i++) vis[i] = 0;
    while (S.size()) {
        int u = S.top();
        S.pop();
        if (!vis[u]) scc(u, u);
    }
}
```

## 2.13 Kruskal

```
// Gera AGM a partir do vetor de arestas
//
// O(m log(n))

int n;
vector<pair<int, pair<int, int> > > ar; // vetor de arestas
int v[MAX];

// Union-Find em O(log(n))
void build();
int find(int k);
void une(int a, int b);

void kruskal() {
    build();

    sort(ar.begin(), ar.end());
    for (int i = 0; i < (int) ar.size(); i++) {
        int a = ar[i].s.f, b = ar[i].s.s;
        if (find(a) != find(b)) {
            une(a, b);
            // aresta faz parte da AGM
        }
    }
}
```

## 2.14 Ponte

```
// Chama zera(numDeVertices)
// Depois dfs para (0, -1) = (verticeInicial, paiDele)
// Se tiver ponte a variavel ok vai ser 0 no final
//
// Complexidade: O(n + m)

vector <vector<int> > g(N);
vector<int> di (N); // distancia do vertice inicial
vector<int> lo (N); // di do menor vertice que ele alcanca
vector<int> vi (N);
int d, ok;

void zera(int n){
    for(int i = 0; i < n; i++){
        g[i].clear();
        di[i] = -1;
        lo[i] = INF;
        vi[i] = 0;
    }
    ok = 1;
    d = 0;
}

void dfs(int v, int pai){
    vi[v] = 1;
    // ele eh o d-esimo a ser visitado e alcanca o d-esimo
        vertice
    di[v] = lo[v] = d++;

    for(int i = 0; i < g[v].size(); i++){
        int u = g[v][i];
        if(!vi[u]) dfs(u, v);

        // o filho nao alcanca ninguem menor ou igual a ele,
            eh ponte
        if(di[v] < lo[u]) ok = 0;
```

```
        // atualiza o menor que ele alcanca
        if(pai != u && lo[u] < lo[v])
            lo[v] = lo[u];
    }
}
```

## 2.15 Tarjan

```
// O(n + m)

int n;
vector<vector<int> > g(MAX);
stack<int> s;
int vis[MAX], comp[MAX];
int id[MAX], p;

int dfs(int k) {
    int lo = id[k] = p++;
    s.push(k);
    vis[k] = 2; // ta na pilha

    // calcula o menor cara q ele alcanca
    // que ainda nao esta em um scc
    for (int i = 0; i < g[k].size(); i++) {
        if (!vis[g[k][i]])
            lo = min(lo, dfs(g[k][i]));
        else if (vis[g[k][i]] == 2)
            lo = min(lo, id[g[k][i]]);
    }

    // nao alcanca ninguem menor -> comeca scc
    if (lo == id[k]) while (1) {
        int u = s.top();
        s.pop(); vis[u] = 1;
        comp[u] = k;
        if (u == k) break;
    }

    return lo;
}

void tarjan() {
    memset(vis, 0, sizeof(vis));

    p = 0;
    for (int i = 0; i < n; i++) if (!vis[i]) dfs(i);
}
```

# 3 Matemática

## 3.1 Miller-Rabin

```
// Testa se n eh primo, n <= 3 * 10^18
//
// O(log(n)), considerando multiplicacao
// e exponenciacao constantes

// multiplicacao e exponenciacao rapidas
ll mul(ll x, ll y, ll m); // x*y mod m
ll pow(ll x, ll y, ll m); // x^y mod m

bool prime(ll n) {
    if (n < 2) return 0;
    if (n <= 3) return 1;
    if (n % 2 == 0) return 0;

    ll d = n - 1;
    int r = 0;
    while (d % 2 == 0) r++, d /= 2;

    // com esses primos, o teste funciona garantido para n
       <= 3*10^18
    // funciona para n <= 3*10^24 com os primos ate 41
    int a[9] = {2, 3, 5, 7, 11, 13, 17, 19, 23};
    // outra opcao para n <= 2^64:
    // int a[7] = {2, 325, 9375, 28178, 450775, 9780504,
       1795265022};

    for (int i = 0; i < 9; i++) {
        if (a[i] >= n) break;
        ll x = pow(a[i], d, n);
        if (x == 1 or x == n - 1) continue;
```

```
        bool deu = 1;
        for (int j = 0; j < r - 1; j++) {
            x = pow(x, 2, n);
            if (x == n - 1) {
                deu = 0;
                break;
            }
        }
        if (deu) return 0;
    }
    return 1;
}
```

## 3.2 Crivo de Erastosthenes

```
// "O" crivo
//
// Encontra maior divisor primo
// Um numero eh primo sse div[x] == x
// fact fatora um numero <= lim
// A fatoracao sai ordenada
//
// crivo - O(n log(log(n)))
// fact - O(log(n))

int divi[MAX];

void crivo(int lim) {
    for (int i = 1; i <= lim; i++) divi[i] = 1;

    for (int i = 2; i <= lim; i++) if (divi[i] == 1)
        for (int j = i; j <= lim; j += i) divi[j] = i;
}

void fact(vector<int>& v, int n) {
    if (n != divi[n]) fact(v, n/divi[n]);
    v.push_back(divi[n]);
}

// Crivo de divisores
//
// Encontra numero de divisores
// ou soma dos divisores
//
// O(n log(n))

int divi[MAX];

void crivo(int lim) {
    for (int i = 1; i <= lim; i++) divi[i] = 1;

    for (int i = 2; i <= lim; i++)
        for (int j = i; j <= lim; j += i) {
            // para numero de divisores
            divi[j]++;
            // para soma dos divisores
            divi[j] += i;
        }
}

// Crivo de totiente
//
// Encontra o valor da funcao
// totiente de Euler
//
// O(n log(log(n)))

int tot[MAX];

void crivo(int lim) {
    for (int i = 1; i <= lim; i++) tot[i] = i;

    for (int i = 2; i <= lim; i++) if (tot[i] == i)
        for (int j = i; j <= lim; j += i)
            tot[j] -= tot[j] / i;
}
```

## 3.3 Exponenciação rápida

```
// (x^y mod m) em O(log(y))

typedef long long int ll;
```

```
ll pow(ll x, ll y, ll m) { // iterativo
    ll ret = 1;
    while (y) {
        if (y & 1) ret = (ret * x) % m;
        y >>= 1;
        x = (x * x) % m;
    }
    return ret;
}

ll pow(ll x, ll y, ll m) { // recursivo
    if (y == 0) return 1;

    ll ret = pow(x, y / 2, m);
    ret = (ret * ret) % m;
    if (y & 1) ret = (ret * x) % m;
    return ret;
}
```

## 3.4 Euclides

```
// O(log(min(a, b)))

int mdc(int a, int b) {
    return !b ? a : mdc(b, a % b);
}
```

## 3.5 Euclides extendido

```
// acha x e y tal que ax + by = mdc(a, b)
//
// O(log(min(a, b)))

int mdce(int a, int b, int *x, int *y){
    if(!a){
        *x = 0;
        *y = 1;
        return b;
    }

    int X, Y;
    int mdc = mdce(b % a, a, &X, &Y);
    *x = Y - (b / a) * X;
    *y = X;

    return mdc;
}
```

## 3.6 Ordem Grupo

```
// O grupo Zn eh ciclico sse n =
// 1, 2, 4, p^k ou 2 p^k, p primo impar
// Retorna -1 se nao achar
//
// O(sqrt(n) log(n))

int tot(int n); // totiente em O(sqrt(n))
int expo(int a, int b, int m); // (a^b)%m em O(log(b))

// acha todos os divisores ordenados em O(sqrt(n))
vector<int> div(int n) {
    vector<int> ret1, ret2;
    for (int i = 1; i*i <= n; i++) if (n % i == 0) {
        ret1.pb(i);
        if (i*i != n) ret2.pb(n/i);
    }

    for (int i = ret2.size()-1; i+1; i--) ret1.pb(ret2[i]);
    return ret1;
}

int ordem(int a, int n) {
    vector<int> v = div(tot(n));
    for (int i : v) if (expo(a, i, n) == 1) return i;
    return -1;
}
```

## 3.7 Pollard's Rho

```
// Codigo completo: https://pastebin.com/A2VdJ4zK
// Usa o algoritmo de deteccao de ciclo de Brent
// A fatoracao nao sai necessariamente ordenada
```

```
// O algoritmo rho encontra um fator de n,
// e funciona muito bem quando n possui um fator pequeno
// Eh recomendado chamar srand(time(NULL)) na main
// A funcao pow deve chamar mul, para nao dar overflow
//
// Complexidades (considerando mul e pow constantes):
// rho - esperado O(n^(1/4)) no pior caso
// fact - esperado menos que O(n^(1/4) log(n)) no pior caso

ll mdc(ll a, ll b);
ll mul(ll a, ll b, ll m);
ll pow(ll a, ll b, ll m);
bool prime(ll n); // Miller-Rabin O(log^2(n))

ll rho(ll n) {
    if (n == 1 or prime(n)) return n;
    if (n % 2 == 0) return 2;

    while (1) {
        ll x = 2, y = 2;
        ll ciclo = 2, i = 0;

        // tenta com essa constante
        ll c = (rand() / (double) RAND_MAX) * (n - 1) + 1;
        // divisor
        ll d = 1;

        while (d == 1) {
            // algoritmo de Brent
            if (++i == ciclo) ciclo *= 2, y = x;
            x = (pow(x, 2, n) + c) % n;

            // x = y -> ciclo
            // tenta com outra constante
            if (x == y) break;

            d = mdc(abs(x - y), n);
        }

        // sucesso -> retorna o divisor
        if (x != y) return d;
    }
}

void fact(ll n, vector<ll>& v) {
    if (n == 1) return;
    if (prime(n)) v.pb(n);
    else {
        ll d = rho(n);
        fact(d, v);
        fact(n / d, v);
    }
}
```

## 3.8 Totiente

```
// O(sqrt(n))

int tot(int n){
    int ret = n;

    for (int i = 2; i*i <= n; i++) if (n % i == 0) {
        while (n % i == 0) n /= i;
        ret -= ret / i;
    }
    if (n > 1) ret -= ret / n;

    return ret;
}
```

# 4 Problemas

## 4.1 Inversion Count

```
// O(n log(n))

int n;
int v[MAX];

// bit de soma
void poe(int p);
int query(int p);
```

```
// converte valores do array pra
// numeros de 1 a n
void conv() {
    vector<int> a;
    for (int i = 0; i < n; i++) a.push_back(v[i]);

    sort(a.begin(), a.end());

    for (int i = 0; i < n; i++)
        v[i] = 1 + (lower_bound(a.begin(), a.end(), v[i]) -
            a.begin());
}

long long inv() {
    conv();
    build();

    long long ret = 0;
    for (int i = n - 1; i >= 0; i--) {
        ret += query(v[i] - 1);
        poe(v[i]);
    }
    return ret;
}
```

## 4.2 Area Histograma

```
// Assume que todas as barras tem largura 1,
// e altura dada no vetor v
//
// O(n)

typedef long long ll;

ll area(vector<int> v) {
    ll ret = 0;
    stack<int> s;
    // valores iniciais pra dar tudo certo
    v.insert(v.begin(), -1);
    v.insert(v.end(), -1);
    s.push(0);

    for(int i = 0; i < (int) v.size(); i++) {
        while (v[s.top()] > v[i]) {
            ll h = v[s.top()]; s.pop();
            ret = max(ret, h * (i - s.top() - 1));
        }
        s.push(i);
    }

    return ret;
}
```

## 4.3 LIS

```
// Calcula uma LIS
// Para ter o tamanho basta fazer lis().size()
// Implementacao do algotitmo descrito em:
// https://goo.gl/HiFkn2
//
// O(n log(n))

const int INF = 0x3f3f3f3f;

int n, v[MAX];

vector<int> lis() {
    int I[n + 1], L[n];

    // pra BB funfar bacana
    I[0] = -INF;
    for (int i = 1; i <= n; i++) I[i] = INF;

    for (int i = 0; i < n; i++) {
        // BB
        int l = 0, r = n;
        while (l < r) {
            int m = (l + r) / 2;
            if (I[m] >= v[i]) r = m;
            else l = m + 1;
        }

        // ultimo elemento com tamanho l eh v[i]
```

```
        I[l] = v[i];
        // tamanho da LIS terminando com o
        // elemento v[i] eh 1
        L[i] = 1;
    }

    // reconstroi LIS
    vector<int> ret;
    int m = -INF, p;
    for (int i = 0; i < n; i++) if (L[i] > m) {
        m = L[i];
        p = i;
    }
    ret.push_back(v[p]);
    int last = m;
    while (p--) if (L[p] == m - 1) {
        ret.push_back(v[p]);
        m = L[p];
    }

    reverse(ret.begin(), ret.end());
    return ret;
}
```

## 4.4 Nim

```
// Calcula movimento otimo do jogo classico de Nim
// Assume que o estado atual eh perdedor
// Funcao move retorna um par com a pilha (0 indexed)
// e quanto deve ser tirado dela
// XOR deve estar armazenado em x
// Para mudar um valor, faca insere(novo_valor),
// atualize o XOR e mude o valor em v
//
// MAX2 = teto do log do maior elemento
// possivel nas pilhas
//
// O(log(n)) amortizado

int v[MAX], n, x;
stack<int> pi[MAX2];

void insere(int p) {
    for (int i = 0; i < MAX2; i++) if (v[p] & (1 << i))
        pi[i].push(p);
}

pair<int, int> move() {
    int bit = 0; while (x >> bit) bit++; bit--;

    // tira os caras invalidos
    while ((v[pi[bit].top()] & (1 << bit)) == 0)
        pi[bit].pop();

    int cara = pi[bit].top();
    int tirei = v[cara] - (x^v[cara]);
    v[cara] -= tirei;

    insere(cara);

    return make_pair(cara, tirei);
}

// Acha o movimento otimo baseado
// em v apenas
//
// O(n)

pair<int, int> move() {
    int x = 0;
    for (int i = 0; i < n; i++) x ^= v[i];

    for (int i = 0; i < n; i++) if ((v[i]^x) < v[i])
        return make_pair(i, v[i] - (v[i]^x));
}
```

# 5 String

## 5.1 KMP

```
// Primeiro chama a funcao process com o padrao
// Depois chama match com (texto, padrao)
```

```
// Vai retornar o numero de ocorrencias do padrao
//
// Complexidades:
// process - O(m)
// match - O(n + m)
// n = |texto| e m = |padrao|

int p[N];

void process(string &s){
    int i = 0, j = -1;
    p[0] = -1;
    while(i < s.size()){
        while(j >= 0 and s[i] != s[j]) j = p[j];
        i++; j++;
        p[i] = j;
    }
}

int match(string &s, string &t){
    int r = 0;
    process(t);
    int i = 0, j = 0;
    while(i < s.size()){
        while(j >= 0 and s[i] != t[j]) j = p[j];
        i++; j++;
        if(j == t.size()){
            j = p[j];
            r++;
        }
    }
    return r;
}
```

## 5.2 Hash

```
// String hashing
//
// String deve ter valores [1, x]
// p deve ser o menor primo maior que x
// Para evitar colisao: testar mais de um
// mod; so comparar strings do mesmo tamanho
//
// Complexidades:
// build - O(|s|)
// get_hash - O(1)

typedef long long ll;

ll h[MAX], power[MAX];
const int p = 31, m = 1e9+7;
int n; char s[MAX];

void build() {
    power[0] = 1;
    for (int i = 1; i < n; i++) power[i] = power[i-1]*p % m;
    h[0] = s[0];
    for (int i = 1; i < n; i++) h[i] = (h[i-1]*p + s[i]) % m;
}

ll get_hash(int i, int j) {
    if (!i) return h[j];
    return (h[j] - h[i-1]*power[j-i+1] % m + m) % m;
}
```

## 5.3 Z

```
// Complexidades:
// z - O(|s|)
// match - O(|s| + |p|)

vector<int> get_z(string s) {
    int n = s.size();
    vector<int> z(n, 0);

    // intervalo da ultima substring valida
    int l = 0, r = 0;
    for (int i = 1; i < n; i++) {
        // estimativa pra z[i]
        if (i <= r) z[i] = min(r - i + 1, z[i - l]);
        // calcula valor correto
        while (i + z[i] < n and s[z[i]] == s[i + z[i]])
            z[i]++;
        // atualiza [l, r]
```

```
        if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
    }

    return z;
}

// quantas vezes p aparece em s
int match(string s, string p) {
    int n = s.size(), m = p.size();
    vector<int> z = get_z(p + s);

    int ret = 0;
    for (int i = m; i < n + m; i++)
        if (z[i] >= m) ret++;

    return ret;
}
```

# 6 Extra

## 6.1 vimrc

```
set ts=4 si ai sw=4 number mouse=a
syntax on
```

## 6.2 Makefile

```
CXX = g++
CXXFLAGS = -O2 -Wall -Wshadow -std=c++11 -Wno-unused-result
   -Wno-sign-compare
```

## 6.3 Template

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

using namespace std;

#define sc(a) scanf("%d",&a)
#define sc2(a,b) sc(a), sc(b)
#define sc3(a,b,c) sc2(a, b), sc(c)
#define pri(x) printf("%d\n",x)
#define mp make_pair
#define pb push_back
#define f first
#define s second
#define BUFF ios::sync_with_stdio(0)

typedef long long ll;
typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<vi> vvi;
typedef vector<ii> vii;

const int INF = 0x3f3f3f3f;

int main() {

    return 0;
}
```