

# Biblioteca dos Brunos [UFMG]

Bruno Demattos & Bruno Monteiro

## Índice

<b>1 Estruturas</b>	<b>2</b>	2.8 Dinic . . . . .	10
1.1 BIT . . . . .	2	2.9 Kosaraju . . . . .	11
1.2 BIT 2D . . . . .	2	2.10 Kruskal . . . . .	12
1.3 SQRT-decomposition . . . . .	3	2.11 Ponte . . . . .	12
1.4 Seg-Tree . . . . .	4	2.12 Tarjan . . . . .	13
1.5 Sparse-Table . . . . .	4	<b>3 Matemática</b>	<b>13</b>
1.6 Trie . . . . .	5	3.1 Crivo de Erastosthenes . . . . .	13
1.7 Union-Find . . . . .	5	3.2 Exponenciação rápida . . . . .	13
<b>2 Grafos</b>	<b>6</b>	3.3 Euclides . . . . .	14
2.1 Bellman-Ford . . . . .	6	3.4 Euclides extendido . . . . .	14
2.2 Floyd-Warshall . . . . .	6	3.5 Miller-Rabin . . . . .	14
2.3 Heavy-Light decomposition . . . . .	6	3.6 Pollard's Rho . . . . .	14
2.4 LCA . . . . .	8	3.7 Totiente . . . . .	15
2.5 LCA com RMQ . . . . .	9	<b>4 Problemas</b>	<b>15</b>
2.6 Centroid decomposition . . . . .	9	4.1 Inversion Count . . . . .	15
2.7 Dijkstra . . . . .	10	4.2 LIS . . . . .	16
		4.3 Nim . . . . .	17

## 5 String

- 5.1 KMP . . . . .
- 5.2 Z . . . . .

## 6 Extra

- 6.1 Template . . . . .
- 6.2 Vimrc . . . . .

# 1 Estruturas

## 1.1 BIT

```
// BIT 1-based, v 0-based
// Para mudar o valor da posicao p para x,
// faca: poe(x - sum(p, p), p)
//
// Complexidades:
// build - O(n)
// poe - O(log(n))
// query - 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];
    }
}
```

```
17 // soma x na posicao p
17 void poe(int x, int p) {
17     while (p <= n) {
17         bit[p] += x;
17         p += p & -p;
18     }
18 }

18 // soma [1, p]
18 int query(int p) {
18     int ret = 0;
18     while (p) {
18         ret += bit[p];
18         p -= p & -p;
18     }
18     return ret;
18 }

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

## 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:
// build - O(n^2 log^2(n))
// poe - O(log^2(n))
// query - O(log^2(n))

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

void poe(int x, int y, int k) {
    int y2 = y;
    while (x <= n) {
        y = y2;
```

```

    while (y <= n) {
        bit[x][y] += k;
        y += y & -y;
    }
    x += x & -x;
}

int query(int x, int y) {
    int ret = 0;
    int y2 = y;
    while (x) {
        y = y2;
        while (y) {
            ret += bit[x][y];
            y -= y & -y;
        }
        x -= x & -x;
    }
    return ret;
}

int sum(int x, int y, int z, int w) {
    int ret = query(z, w);
    if (x > 1) ret -= query(x - 1, w);
    if (y > 1) ret -= query(z, y - 1);
    if (x > 1 and y > 1) ret += query(x - 1, y - 1);

    return ret;
}

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

    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            poe(i, j, M[i - 1][j - 1]);
}

```

### 1.3 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;
}

```

## 1.4 Seg-Tree

```
// SegTree 1-based, vetor 0-based
// 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 - 1];

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

// propagar o que ta em p para os filhos
void prop(int p, int l, int r) {
    // soma o lazy
    seg[p] += lazy[p] * (r - l + 1);

    // propaga pros filhos
    if (l != r) {
        lazy[2 * p] += lazy[p];
        lazy[2 * p + 1] += lazy[p];
    }

    // zera o lazy
    lazy[p] = 0;
}

// somar x no intervalo [l, r]
int lazy_op(int p, int l, int r) {
```

```
// soma x * (tamanho do intervalo)
seg[p] += x * (r - l + 1);

// suja os filhos
if (l != r) {
    lazy[2 * p] += x;
    lazy[2 * p + 1] += x;
}

return seg[p];
}

int query(int p, int l, int r) {
    // propaga
    prop(p, l, r);

    // to totalmente dentro
    if (a <= l and r <= b) return seg[p];
    // to fora
    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) {
    // propaga
    prop(p, l, r);

    // to totalmente dentro
    if (a <= l and r <= b) return lazy_op(p, l, r);
    // to fora
    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);
}
```

## 1.5 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]]);
}
```

## 1.6 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];
int fim[N];
int nx = 1;

void insere(string &s, int p, int l, int at){
```

```
// se nao chegou no fim da palavra termina de inserir
if(p != 1){
    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 != 1){
        int c = s[p] - 'a';
        if(trie[at][c]) return check(s, p+1, l, trie[at][c]);
        return 0;
    }
    return fim[at];
}
```

## 1.7 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| <= |b|
    a = find(a);
    b = find(b);
    if (size[a] > size[b]) swap(a, b);

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

// une sem union by size
//
// une e find ficam O(log(n))
//
// void une(int a, int b) {
//     v[find(a)] = find(b);
// }

```

## 2 Grafos

### 2.1 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;
}

```

### 2.2 Floyd-Warshall

```

// encontra o menor caminho entre todo
// par de vertices e detecta ciclo negativo
// retorna 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;
}

```

### 2.3 Heavy-Light decomposition

```

// SegTree de maximo, 0-based

```

```

// query_hld(u, v) calcula maior aresta
// no caminho de u pra v
// update_hld(p, val) muda o peso da aresta
// p para val
//
// SegTree pode ser facilmente modificada
//
// Complexidades:
// build_hld - O(n)
// query_hld - O(log^2 (n))
// update_hld - O(log(n))

int n, a, b, x; // [a, b] usado na seg tree | x : valor de
               update
vector<vector<int>> g(MAX), w(MAX);
int subsize[MAX]; // tamanho da sub-arvore
int pai[MAX], chain[MAX], head[MAX]; // cabeca de cada chain
int num[MAX]; // numeracao do vertice na segtree
int vec[MAX]; // v[i] : custo de u para pai[u], u = num[i]
vector<vector<int>> ind(MAX); // index da aresta
int ponta[MAX]; // vertice de baixo da aresta
int vis[MAX], chains, seg[4 * MAX];
int pos; // posicao atual na seg tree (na hora de montar a HLD)

// seg tree de maximo com update pontual
int build_seg(int p, int l, int r);
int query_seg(int p, int l, int r);
int update_seg(int p, int l, int r);

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

            pai[u] = k;
            subsize[k] += subsize[u];
            ponta[ind[k][i]] = u;
        }
    }
}

```

```

}

void hld(int k, int custo) {
    vis[k] = 1;
    chain[k] = chains - 1;

    num[k] = pos;
    vec[pos++] = custo;

    // acha filho pesado
    int f = -1, peso = -INF, prox_custo;
    for (int i = 0; i < (int) g[k].size(); i++) if (!vis[g[k][i]])
        if (subsize[g[k][i]] > peso) {
            f = g[k][i];
            peso = subsize[f];
            prox_custo = w[k][i];
        }

    // folha
    if (f == -1) return;

    // continua a chain
    hld(f, prox_custo);

    // começa novas chains
    for (int i = 0; i < (int) g[k].size(); i++)
        if (!vis[g[k][i]] and g[k][i] != f) {
            chains++;
            head[chains - 1] = g[k][i];
            hld(g[k][i], w[k][i]);
        }
}

void build_hld(int root) {
    for (int i = 0; i < n; i++) vis[i] = 0;

    // DFS pra calcular tamanho das sub-arvores,
    // e ponta das arestas
    dfs(root);

    for (int i = 0; i < n; i++) vis[i] = 0;
}

```

```

// começa 0 chain da root
chains = 1;
head[0] = root;
pos = 0;
hld(root, -1);

// cria seg tree
build_seg(0, 0, n - 1);
}

int query_hld(int u, int v) {
    if (u == v) return 0;

    int ret = -INF;

    while (chain[u] != chain[v]) {
        // sobe o de maior chain
        if (chain[u] < chain[v]) swap(u, v);

        a = num[head[chain[u]]], b = num[u];
        ret = max(ret, query_seg(0, 0, n - 1));

        u = head[chain[u]];
        u = pai[u];
    }

    if (u == v) return ret;

    // query final
    if (num[u] < num[v]) swap(u, v); // LCA eh v

    a = num[v] + 1, b = num[u];
    ret = max(ret, query_seg(0, 0, n - 1));

    return ret;
}

void update_hld(int p, int val) {
    x = val;
    a = b = num[ponta[p]];
    update_seg(0, 0, n - 1);
}

```

## 2.4 LCA

```

// Assume que um vertice eh ancestral dele mesmo, ou seja,
// se a eh ancestral de b, LCA(a, b) = b
// 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];
}

```

## 2.5 LCA com RMQ

```

// Assume que um vertice eh ancestral dele mesmo, ou seja,
// se a eh ancestral de b, LCA(a, b) = b
//
// 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)];
}

```

## 2.6 Centroid decomposition

```

// O(n log(n))

```

```

int n;
vvi 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);
}

```

## 2.7 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;
        if (-Q.top().first > d[u]) continue;
        Q.pop();

```

```

        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));
            }
        }
    }
}

```

## 2.8 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)

```

```

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 capacidade 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");
    }
}

```

## 2.9 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);
    }
}

```

## 2.10 Kruskal

```

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

int n;
vector<pair<int, pair<int, int>>> ar; // vetor de arestas
vector<int> agm; // agm[i] eh 1 sse a i-esima aresta ta na AGM
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);
            agm.pb(1);
        } else agm.pb(0);
    }
}

```

```

    }
}

```

## 2.11 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];
    }
}

```

## 2.12 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 < sz(g[k]); 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() {
    for (int i = 0; i < n; i++) vis[i] = 0;
}

```

```

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

```

## 3 Matemática

### 3.1 Crivo de Erastosthenes

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

```

int primo[MAX];
int n;

void crivo() {
    primo[1] = 0;
    for (int i = 2; i <= n; i++) primo[i] = 1;

    for (int i = 2; i*i <= n; i++) if (primo[i])
        for (int j = i*i; j <= n; j += i) primo[j] = 0;
}

```

### 3.2 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;
}

```

### 3.3 Euclides

```

// O(log(min(a, b)))
// Na pratica, pode ser considerado O(1)

```

```

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

```

### 3.4 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;
}

```

### 3.5 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;

```

### 3.6 Pollard's Rho

```
// Usa o algoritmo de detecção de ciclo de Brent
// A fatoração não sai necessariamente ordenada
// O algoritmo rho encontra um fator de n,
// e funciona muito bem quando n possui um fator pequeno
// É recomendado chamar srand(time(NULL)) na main
//
// 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 || 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);
    }
}
```

### 3.7 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 build();
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 LIS

```

// Calcula uma LIS
// Para ter o tamanho basta fazer lis().size()
// Implementacao do algoritmo 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 l
        L[i] = l;
    }

    // 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-- > 0) if (L[p] == last) {
        ret.push_back(v[p]);
        last = L[p];
    }

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

```



## 4.3 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
//
// MAX2 = teto do log do maior elemento
// possivel nas pilhas
//
// O(n)

int v[MAX], n;

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

    int p = -1;
    for (int i = 1 << MAX2; i; i >>= 1) if (x & i) {
        for (int j = 0; j < n; j++) if (v[j] & i) {
            p = j;
            break;
        }
        break;
    }

    x ^= v[p];
    return make_pair(p, v[p] - 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 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 - 1]);

    // 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 ret;
}

// 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 Template

```

#include <bits/stdc++.h>

using namespace std;

#define sc(a) scanf("%d",&a)
#define sc2(a,b) scanf("%d%d",&a,&b)
#define sc3(a,b,c) scanf("%d%d%d",&a,&b,&c)
#define pri(x) printf("%d\n",x)
#define prie(x) printf("%d ",x)

```

```

#define prif() printf("\n")
#define sz(x) ((int)((x).size()))
#define mp make_pair
#define pb push_back
#define f first
#define s second
#define BUFF ios::sync_with_stdio(false)

typedef long long int ll;
typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<vi> vvi;
typedef vector<ii> vii;
const int INF = 0x3f3f3f3f;
const ll LINF = 0x3f3f3f3f3f3f3f3f;

```

### 6.2 Vimrc

```

set ts=4 si ai sw=2 number mouse=a
syntax on

```