BRUTE
UDESC

Eliton Machado da Silva, Enzo de Almeida Rodrigues, Eric Grochowicz, Igor Froehner
João Vitor Frölich, João Marcos de Oliveira e Rafael Granza de Mello

16 de janeiro de 2024

Índice

Capítulo 1

Estruturas de Dados

1.1 Disjoint Set Union

1.1.1 DSU

Estrutura que mantém uma coleção de conjuntos e permite as operações de unir dois conjuntos e verificar em qual conjunto um elemento está, ambas em $O(1)$ amortizado. O método *find* retorna o representante do conjunto que contém o elemento, e o método *unite* une os conjuntos que contém os elementos dados, retornando *true* se eles estavam em conjuntos diferentes e *false* caso contrário.

```
struct DSU {
    vector<int> par, sz;
    int number_of_sets;
    DSU(int n = 0) : par(n), sz(n, 1), number_of_sets(n) {
        iota(par.begin(), par.end(), 0);
    }
    int find(int a) {
        return a == par[a] ? a : par[a] = find(par[a]);
    }
    bool unite(int a, int b) {
        a = find(a), b = find(b);
        if (a == b) {
            return false;
        }
        number_of_sets--;
        if (sz[a] < sz[b]) {
            swap(a, b);
        }
        par[b] = a;
        sz[a] += sz[b];
```

```
        return true;
    }
};
```

1.1.2 DSU Bipartido

DSU que mantém se um conjunto é bipartido (visualize os conjuntos como componentes conexas de um grafo e os elementos como vértices). O método *unite* adiciona uma aresta entre os dois elementos dados, e retorna *true* se os elementos estavam em conjuntos diferentes (componentes conexas diferentes) e *false* caso contrário. O método *bipartite* retorna *true* se o conjunto (componente conexa) que contém o elemento dado é bipartido e *false* caso contrário. Todas as operações são $O(\log n)$.

```
struct Bipartite_DSU {
    vector<int> par, sz, c, bip;
    int number_of_sets, all_bipartite;
    Bipartite_DSU(int n = 0)
        : par(n), sz(n, 1), c(n), bip(n, 1), number_of_sets(n),
          all_bipartite(1) {
        iota(par.begin(), par.end(), 0);
    }
    int find(int a) {
        return a == par[a] ? a : find(par[a]);
    }
    int color(int a) {
        return a == par[a] ? c[a] : c[a] ^ color(par[a]);
    }
    bool bipartite(int a) {
        return bip[find(a)];
    }
    bool unite(int a, int b) {
        bool equal_color = color(a) == color(b);
        a = find(a), b = find(b);
        if (a == b) {
            if (equal_color) {
                bip[a] = 0;
                all_bipartite = 0;
            }
            return false;
        }
        if (sz[a] < sz[b]) {
            swap(a, b);
        }
        number_of_sets--;
        par[b] = a;
        sz[a] += sz[b];
        if (equal_color) {
            c[b] = 1;
        }
        bip[a] &= bip[b];
        all_bipartite &= bip[a];
        return true;
    }
};
```

1.1.3 DSU Rollback

DSU que desfaz as últimas operações. O método *checkpoint* salva o estado atual da estrutura, e o método *rollback* desfaz as últimas operações até o último checkpoint. As operações de unir dois conjuntos e verificar em qual conjunto um elemento está agora são $O(\log n)$, e o rollback é $O(k)$, onde k é o número de alterações a serem desfeitas. Importante notar que o rollback não altera a complexidade, uma vez que $\sum k = O(q)$, onde q é o número de operações realizadas.

```
struct Rollback_DSU {
    vector<int> par, sz;
    int number_of_sets;
    stack<stack<pair<int &, int>>> changes;
    Rollback_DSU(int n = 0) : par(n), sz(n, 1), number_of_sets(n) {
        iota(par.begin(), par.end(), 0);
        changes.emplace();
    }
    int find(int a) {
        return a == par[a] ? a : find(par[a]);
    }
    void checkpoint() {
        changes.emplace();
    }
    void save(int &a) {
        changes.top().emplace(a, a);
    }
    bool unite(int a, int b) {
        a = find(a), b = find(b);
        if (a == b) {
            return false;
        }
        if (sz[a] < sz[b]) {
            swap(a, b);
        }
        save(number_of_sets);
        save(par[b]);
        save(sz[a]);
        number_of_sets--;
        par[b] = a;
        sz[a] += sz[b];
        return true;
    }
    void rollback() {
        while (changes.top().size()) {
            auto [a, b] = changes.top().top();
            a = b;
            changes.top().pop();
        }
        changes.pop();
    }
};
```

1.1.4 DSU Rollback Bipartido

DSU com rollback e bipartido.

```
struct Full_DSU {
    vector<int> par, sz, c, bip;
    int number_of_sets, all_bipartite;
    stack<stack<pair<int &, int>>> changes;
    Full_DSU(int n = 0)
        : par(n), sz(n, 1), c(n), bip(n, 1), number_of_sets(n),
            all_bipartite(1) {
        iota(par.begin(), par.end(), 0);
        changes.emplace();
    }
    int find(int a) {
        return a == par[a] ? a : find(par[a]);
    }
    int color(int a) {
        return a == par[a] ? c[a] : c[a] ^ color(par[a]);
    }
    bool bipartite(int a) {
        return bip[find(a)];
    }
    void checkpoint() {
        changes.emplace();
    }
    void save(int &a) {
        changes.top().emplace(a, a);
    }
    bool unite(int a, int b) {
        bool equal_color = color(a) == color(b);
        a = find(a), b = find(b);
        if (a == b) {
            if (equal_color) {
                save(bip[a]);
                save(all_bipartite);
                bip[a] = 0;
                all_bipartite = 0;
            }
            return false;
        }
        if (sz[a] < sz[b]) {
            swap(a, b);
        }
        save(number_of_sets);
        save(par[b]);
        save(sz[a]);
        save(c[b]);
        save(bip[a]);
        save(all_bipartite);
        number_of_sets--;
        par[b] = a;
        sz[a] += sz[b];
        if (equal_color) {
            c[b] = 1;
        }
        bip[a] &= bip[b];
        all_bipartite &= bip[a];
        return true;
    }
    void rollback() {
        while (changes.top().size()) {
            auto [a, b] = changes.top().top();
            a = b;
            changes.top().pop();
        }
        changes.pop();
    }
};
```

1.1.5 Offline DSU

Algoritmo que utiliza o Full DSU (DSU com Rollback e Bipartido) que permite adição e **remoção** de arestas. O algoritmo funciona de maneira offline, recebendo previamente todas as operações de adição e remoção de arestas, bem como todas as perguntas (de qualquer tipo, conectividade, bipartição, etc), e retornando as respostas para cada pergunta no retorno do método *solve*. Complexidade total $O(q(\log q + \log n))$, onde q é o número de operações realizadas e n é o número de nodos.

```
struct Offline_DSU : Full_DSU {
    int time;
    Offline_DSU(int n = 0) : Full_DSU(n), time(0) {
    }
    struct query {
        int type, a, b;
    };
    vector<query> queries;
    void askConnect(int a, int b) {
        if (a > b) {
            swap(a, b);
        }
        queries.push_back({0, a, b});
        time++;
    }
    void askBipartite(int a) {
        queries.push_back({1, a, -1});
        time++;
    }
    void askAllBipartite() {
        queries.push_back({2, -1, -1});
        time++;
    }
    void addEdge(int a, int b) {
        if (a > b) {
            swap(a, b);
        }
        queries.push_back({3, a, b});
        time++;
    }
    void removeEdge(int a, int b) {
        if (a > b) {
            swap(a, b);
        }
        queries.push_back({4, a, b});
        time++;
    }
    vector<vector<pair<int, int>>> lazy;
    void update(int l, int r, pair<int, int> edge, int u, int L, int
        R) {
        if (R < l || L > r) {
            return;
        }
        if (L >= l && R <= r) {
            lazy[u].push_back(edge);
            return;
        }
        int mid = (L + R) / 2;
        update(l, r, edge, 2 * u, L, mid);
        update(l, r, edge, 2 * u + 1, mid + 1, R);
    }
    void dfs(int u, int L, int R, vector<int> &ans) {
        if (L > R) {
            return;
        }
        checkpoint();
        for (auto [a, b] : lazy[u]) {
            unite(a, b);
        }
        if (L == R) {
            auto [type, a, b] = queries[L];
            if (type == 0) {
                ans.push_back(find(a) == find(b));
            } else if (type == 1) {
```

```
                ans.push_back(bipartite(a));
            } else if (type == 2) {
                ans.push_back(all_bipartite);
            }
        } else {
            int mid = (L + R) / 2;
            dfs(2 * u, L, mid, ans);
            dfs(2 * u + 1, mid + 1, R, ans);
        }
        rollback();
    }
    vector<int> solve() {
        lazy.assign(4 * time, {});
        map<pair<int, int>, int> edges;
        for (int i = 0; i < time; i++) {
            auto [type, a, b] = queries[i];
            if (type == 3) {
                edges[{a, b}] = i;
            } else if (type == 4) {
                update(edges[{a, b}], i, {a, b}, 1, 0, time - 1);
                edges.erase({a, b});
            }
        }
        for (auto [k, v] : edges) {
            update(v, time - 1, k, 1, 0, time - 1);
        }
        vector<int> ans;
        dfs(1, 0, time - 1, ans);
        return ans;
    }
};
```

1.2 Fenwick Tree

Árvore de Fenwick (ou BIT) é uma estrutura de dados que permite atualizações pontuais e consultas de prefixos em um vetor em $O(\log n)$. A implementação abaixo é 0-indexada (é mais comum encontrar a implementação 1-indexada). A consulta em ranges arbitrários com o método *query* é possível para qualquer operação inversível, como soma, XOR, multiplicação, etc. A implementação abaixo é para soma, mas é fácil adaptar para outras operações. O método *update* soma d à posição i do vetor, enquanto o método *updateSet* substitue o valor da posição i do vetor por d.

```
template <typename T> struct FenwickTree {
    vector<T> bit;
    FenwickTree(int n = 0) : bit(n, T()) {
    }
    FenwickTree(vector<T> &v) : bit(v.size(), T()) {
        int n = v.size();
        for (int i = 0; i < n; i++)
            bit[i] = v[i];
        for (int i = 0; i < n; i++) {
            int j = i | (i + 1);
            if (j < n)
                bit[j] = bit[j] + bit[i];
        }
    }
    T pref(int i) {
        T res = T();
        while (i >= 0) {
            res = res + bit[i];
            i &= i + 1;
            i--;
```

```
        }
        return res;
    }
    T query(int l, int r) {
        if (l == 0)
            return pref(r);
        return pref(r) - pref(l - 1);
    }
    void update(int i, T d) {
        while (i < (int)bit.size()) {
            bit[i] = bit[i] + d;
            i |= i + 1;
        }
    }
    void updateSet(int i, T d) {
        // funciona pra fenwick de soma
        T now = query(i, i);
        update(i, now * -1);
        update(i, d);
    }
};
```

1.3 Interval Tree

Por Rafael Granza de Mello

Estrutura que trata intersecções de intervalos.

Capaz de retornar todos os intervalos que intersectam $[L, R]$. Contém métodos $insert(L, R, ID)$, $erase(L, R, ID)$, $overlaps(L, R)$ e $find(L, R, ID)$. É necessário inserir e apagar indicando tanto os limites quanto o ID do intervalo. Todas as operações são $O(\log n)$, exceto *overlaps* que é $O(k + \log n)$, onde k é o número de intervalos que intersectam $[L, R]$. Também podem ser usadas as operações padrões de um *std* :: *set*.

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

struct interval {
    long long lo, hi, id;
    bool operator<(const interval &i) const {
        return tuple(lo, hi, id) < tuple(i.lo, i.hi, i.id);
    }
};
const long long INF = 1e18;

template <class CNI, class NI, class Cmp_Fn, class Allocator>
struct intervals_node_update {
    typedef long long metadata_type;
    int sz = 0;
    virtual CNI node_begin() const = 0;
    virtual CNI node_end() const = 0;
    inline vector<int> overlaps(const long long l, const long long r)
        {
        queue<CNI> q;
```

```
        q.push(node_begin());
        vector<int> vec;
        while (!q.empty()) {
            CNI it = q.front();
            q.pop();
            if (it == node_end()) {
                continue;
            }
            if (r >= (*it)->lo && l <= (*it)->hi) {
                vec.push_back((*it)->id);
            }
            CNI l_it = it.get_l_child();
            long long l_max = (l_it == node_end()) ? -INF :
                l_it.get_metadata();
            if (l_max >= l) {
                q.push(l_it);
            }
            if ((*it)->lo <= r) {
                q.push(it.get_r_child());
            }
        }
        return vec;
    }
    inline void operator()(NI it, CNI end_it) {
        const long long l_max =
            (it.get_l_child() == end_it) ? -INF :
                it.get_l_child().get_metadata();
        const long long r_max =
            (it.get_r_child() == end_it) ? -INF :
                it.get_r_child().get_metadata();
        const_cast<long long &>(it.get_metadata()) = max((*it)->hi,
            max(l_max, r_max));
    }
};
typedef tree<interval, null_type, less<interval>, rb_tree_tag,
    intervals_node_update>
    interval_tree;
```

1.4 Kd Fenwick Tree

KD Fenwick Tree

Fenwick Tree em K dimensoes.

* Complexidade de update: $O(log^k(N))$.

* Complexidade de query: $O(log^k(N))$.

```
const int MAX = 10;
ll tree[MAX][MAX][MAX][MAX][MAX][MAX][MAX][MAX]; // insira a
    quantidade
                                                  // necessaria de
                                                     dimensoes

int lsONE(int x) {
    return x & (-x);
}
```

```
ll query(vector<int> s, int pos) {
    ll sum = 0;
    while (s[pos] > 0) {
        if (pos < s.size() - 1) {
            sum += query(s, pos + 1);
        } else {
            sum +=
                tree[s[0]][s[1]][s[2]][s[3]][s[4]][s[5]][s[6]][s[7]];
        }
        s[pos] -= lsONE(s[pos]);
    }
    return sum;
}
```

```
void update(vector<int> s, int pos, int v) {
    while (s[pos] < MAX + 1) {
        if (pos < s.size() - 1) {
            update(s, pos + 1, v);
        } else {
            tree[s[0]][s[1]][s[2]][s[3]][s[4]][s[5]][s[6]][s[7]] += v;
        }

        s[pos] += lsONE(s[pos]);
    }
}
```

1.5 LiChao Tree

Uma árvore de Funções. Retorna o F(x) máximo em um ponto X.

Para retornar o minimo deve-se inserir o negativo da função e pegar o negativo do resultado.

Está pronta para usar função linear do tipo F(x) = mx + b.

Funciona para funções com a seguinte propriedade, sejam duas funções f(x) e g(x), uma vez que f(x) ganha/perde de g(x), f(x) vai continuar ganhando/perdendo de g(x),

ou seja f(x) e g(x) se intersectam apenas uma vez.

* Complexidade de consulta : O(log(N))

* Complexidade de update: O(log(N))

LiChao Tree Sparse

O mesmo que a superior, no entanto suporta consultas com |x| <= 1e18.

* Complexidade de consulta : O(log(tamanho do intervalo))

* Complexidade de update: O(log(tamanho do intervalo))

```
typedef long long ll;

const ll MAXN = 1e5 + 5, INF = 1e18 + 9;

struct Line {
    ll a, b = -INF;
    ll operator()(ll x) {
        return a * x + b;
    }
} tree[4 * MAXN];

int le(int n) {
    return 2 * n + 1;
}
int ri(int n) {
    return 2 * n + 2;
}

void insert(Line line, int n = 0, int l = 0, int r = MAXN) {
    int mid = (l + r) / 2;
    bool bl = line(l) < tree[n](l);
    bool bm = line(mid) < tree[n](mid);
    if (!bm) {
        swap(tree[n], line);
    }
    if (l == r) {
        return;
    }
    if (bl != bm) {
        insert(line, le(n), l, mid);
    } else {
        insert(line, ri(n), mid + 1, r);
    }
}

ll query(int x, int n = 0, int l = 0, int r = MAXN) {
    if (l == r) {
        return tree[n](x);
    }
    int mid = (l + r) / 2;
    if (x < mid) {
        return max(tree[n](x), query(x, le(n), l, mid));
    } else {
        return max(tree[n](x), query(x, ri(n), mid + 1, r));
    }
}
```

```
typedef long long ll;

const ll MAXN = 1e5 + 5, INF = 1e18 + 9, MAXR = 1e18;

struct Line {
    ll a, b = -INF;
    __int128 operator()(ll x) {
        return (__int128)a * x + b;
    }
} tree[4 * MAXN];
int idx = 0, L[4 * MAXN], R[4 * MAXN];

int le(int n) {
    if (!L[n]) {
        L[n] = ++idx;
    }
```

```
    return L[n];
}
int ri(int n) {
    if (!R[n]) {
        R[n] = ++idx;
    }
    return R[n];
}

void insert(Line line, int n = 0, ll l = -MAXR, ll r = MAXR) {
    ll mid = (l + r) / 2;
    bool bl = line(l) < tree[n](l);
    bool bm = line(mid) < tree[n](mid);
    if (!bm) {
        swap(tree[n], line);
    }
    if (l == r) {
        return;
    }
    if (bl != bm) {
        insert(line, le(n), l, mid);
    } else {
        insert(line, ri(n), mid + 1, r);
    }
}

__int128 query(int x, int n = 0, ll l = -MAXR, ll r = MAXR) {
    if (l == r) {
        return tree[n](x);
    }
    ll mid = (l + r) / 2;
    if (x < mid) {
        return max(tree[n](x), query(x, le(n), l, mid));
    } else {
        return max(tree[n](x), query(x, ri(n), mid + 1, r));
    }
}
```

1.6 MergeSort Tree

Árvore que resolve queries que envolvam ordenação em range.

- Complexidade de construção : $O(N * log(N))$
- Complexidade de consulta : $O(log^2(N))$

MergeSort Tree com Update Pontual

Resolve Queries que envolvam ordenação em Range. (**COM UPDATE**)

1 segundo para vetores de tamanho $3 * 10^5$

- Complexidade de construção : $O(N * log^2(N))$
- Complexidade de consulta : $O(log^2(N))$
- Complexidade de update : $O(log^2(N))$

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

using namespace __gnu_pbds;

namespace mergesort {
    typedef tree<ii, null_type, less<ii>, rb_tree_tag,
        tree_order_statistics_node_update>
        ordered_set;
    const int MAX = 1e5 + 5;

    int n;
    ordered_set mgtree[4 * MAX];
    vi values;

    int le(int n) {
        return 2 * n + 1;
    }
    int ri(int n) {
        return 2 * n + 2;
    }

    ordered_set join(ordered_set set_l, ordered_set set_r) {
        for (auto v : set_r) {
            set_l.insert(v);
        }
        return set_l;
    }

    void build(int n, int esq, int dir) {
        if (esq == dir) {
            mgtree[n].insert(ii(values[esq], esq));
        } else {
            int mid = (esq + dir) / 2;
            build(le(n), esq, mid);
            build(ri(n), mid + 1, dir);
            mgtree[n] = join(mgtree[le(n)], mgtree[ri(n)]);
        }
    }
    void build(vi &v) {
        n = v.size();
        values = v;
        build(0, 0, n - 1);
    }

    int less(int n, int esq, int dir, int l, int r, int k) {
        if (esq > r || dir < l) {
            return 0;
        }
        if (l <= esq && dir <= r) {
            return mgtree[n].order_of_key({k, -1});
        }
        int mid = (esq + dir) / 2;
        return less(le(n), esq, mid, l, r, k) + less(ri(n), mid + 1,
            dir, l, r, k);
    }
    int less(int l, int r, int k) {
        return less(0, 0, n - 1, l, r, k);
    }

    void update(int n, int esq, int dir, int x, int v) {
        if (esq > x || dir < x) {
```

```
            return;
        }
        if (esq == dir) {
            mgtree[n].clear(), mgtree[n].insert(ii(v, x));
        } else {
            int mid = (esq + dir) / 2;
            if (x <= mid) {
                update(le(n), esq, mid, x, v);
            } else {
                update(ri(n), mid + 1, dir, x, v);
            }
            mgtree[n].erase(ii(values[x], x));
            mgtree[n].insert(ii(v, x));
        }
    }
    void update(int x, int v) {
        update(0, 0, n - 1, x, v);
        values[x] = v;
    }

    // ordered_set debug_query(int n, int esq, int
    // dir, int l, int r) {
    //     if (esq > r || dir < l) return
    //     ordered_set(); if (l <= esq && dir <=
    //     r) return mgtree[n]; int mid = (esq +
    //     dir) / 2; return
    //     join(debug_query(le(n), esq, mid, l,
    //     r), debug_query(ri(n), mid+1, dir, l,
    //     r));
    // }
    // ordered_set debug_query(int l, int r)
    // {return debug_query(0, 0, n-1, l, r);}

    // int greater(int n, int esq, int dir, int l,
    // int r, int k) {
    //     if (esq > r || dir < l) return 0;
    //     if (l <= esq && dir <= r) return
    //     (r-l+1) - mgtree[n].order_of_key({k,
    //     1e8}); int mid = (esq + dir) / 2;
    //     return greater(le(n), esq, mid, l, r,
    //     k) + greater(ri(n), mid+1, dir, l, r,
    //     k);
    // }
    // int greater(int l, int r, int k) {return
    // greater(0, 0, n-1, l, r, k);}
};
```

```
namespace mergesort {
    const int MAX = 1e5 + 5;

    int n;
    vi mgtree[4 * MAX];

    int le(int n) {
        return 2 * n + 1;
    }
    int ri(int n) {
        return 2 * n + 2;
    }

    void build(int n, int esq, int dir, vi &v) {
        mgtree[n] = vi(dir - esq + 1, 0);
        if (esq == dir) {
            mgtree[n][0] = v[esq];
        } else {
            int mid = (esq + dir) / 2;
            build(le(n), esq, mid, v);
            build(ri(n), mid + 1, dir, v);
            merge(mgtree[le(n)].begin(),
                  mgtree[le(n)].end(),
                  mgtree[ri(n)].begin(),
```

```
                mgtree[ri(n)].end(),
                mgtree[n].begin());
        }
    }
    void build(vi &v) {
        n = v.size();
        build(0, 0, n - 1, v);
    }

    int less(int n, int esq, int dir, int l, int r, int k) {
        if (esq > r || dir < l) {
            return 0;
        }
        if (l <= esq && dir <= r) {
            return lower_bound(mgtree[n].begin(), mgtree[n].end(), k)
                - mgtree[n].begin();
        }
        int mid = (esq + dir) / 2;
        return less(le(n), esq, mid, l, r, k) + less(ri(n), mid + 1,
            dir, l, r, k);
    }
    int less(int l, int r, int k) {
        return less(0, 0, n - 1, l, r, k);
    }

    // vi debug_query(int n, int esq, int dir, int
    // l, int r) {
    //     if (esq > r || dir < l) return vi();
    //     if (l <= esq && dir <= r) return
    //     mgtree[n]; int mid = (esq + dir) / 2;
    //     auto vl = debug_query(le(n), esq, mid,
    //     l, r); auto vr = debug_query(ri(n),
    //     mid+1, dir, l, r); vi ans =
    //     vi(vl.size() + vr.size());
    //     merge(vl.begin(), vl.end(),
    //         vr.begin(), vr.end(),
    //         ans.begin());
    //     return ans;
    // }
    // vi debug_query(int l, int r) {return
    // debug_query(0, 0, n-1, l, r);}
};
```

1.7 Operation Queue

Fila que armazena o resultado do operatório dos itens (ou seja, dado uma fila, responde qual é o elemento mínimo, por exemplo). É uma extensão da *std* :: *queue* que permite as operações *push*, *pop*, *front* e *back* em $O(1)$ amortizado, agora permitindo também a operação *get* que retorna o resultado do operatório dos itens da fila em $O(1)$ amortizado.

Obs: usa a estrutura Operation Stack.

```
template <typename T> struct op_queue : queue<T> {
    op_stack<T> st1, st2;
    T get() {
        if (st1.empty())
            return st2.get();
        if (st2.empty())
            return st1.get();
        return st1.op(st1.get(), st2.get());
    }
    void add(T element) {
        this->push(element);
        st1.add(element);
    }
    void remove() {
        if (st2.empty()) {
            while (!st1.empty()) {
                st2.add(st1.get());
                st1.remove();
            }
        }
        st2.remove();
        this->pop();
    }
};
```

1.8 Operation Stack

Pilha que armazena o resultado do operatório dos itens (ou seja, dado uma pilha, responde qual é o elemento mínimo, por exemplo). É uma extensão da *std* :: *stack* que permite as operações *push*, *pop* e *top* em $O(1)$ amortizado, agora permitindo também a operação *get* que retorna o resultado do operatório dos itens da pilha em $O(1)$ amortizado.

```
template <typename T> struct op_stack : stack<T> {
    stack<T> st;
    T op(T a, T b) {
        return min(a, b); // TODO: Operacao, pode ser qualquer uma
    }
    T get() {
        return st.top();
    }
    void add(T element) {
        this->push(element);
        st.push(st.empty() ? element : op(element, st.top()));
    }
    void remove() {
        st.pop();
        this->pop();
    }
};
```

1.9 Ordered Set

Set com operações de busca por ordem e índice.

Pode ser usado como um set normal, a principal diferença são duas novas operações possíveis:

- find_by_order(x): retorna o item na posição x.
- order_of_key(k): retorna o número de elementos menores que k. (o índice de k)

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

using namespace __gnu_pbds;
typedef tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update> ordered_set;

ordered_set X;
X.insert(1);
X.insert(2);
X.insert(4);
X.insert(8);
X.insert(16);

cout<<*X.find_by_order(1)<<endl; // 2
cout<<*X.find_by_order(2)<<endl; // 4
cout<<*X.find_by_order(4)<<endl; // 16
cout<<(end(X)==X.find_by_order(6))<<endl; // true

cout<<X.order_of_key(-5)<<endl;  // 0
cout<<X.order_of_key(1)<<endl;   // 0
cout<<X.order_of_key(3)<<endl;   // 2
cout<<X.order_of_key(4)<<endl;   // 2
cout<<X.order_of_key(400)<<endl; // 5
```

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

using namespace __gnu_pbds;

template <typename T>
using ordered_set =
    tree<T, null_type, less<T>, rb_tree_tag,
        tree_order_statistics_node_update>;
```

1.10 Segment Tree

1.10.1 Segment Tree

Implementação padrão de Seg Tree

- Complexidade de tempo (Pré-processamento): O(N)
- Complexidade de tempo (Consulta em intervalo): O(log(N))
- Complexidade de tempo (Update em ponto): O(log(N))
- Complexidade de espaço: 4 *N = O(N)

```
namespace seg {
    const int MAX = 2e5 + 5;
    int n;
    ll tree[4 * MAX];
    ll merge(ll a, ll b) {
        return a + b;
    }
    int le(int n) {
        return 2 * n + 1;
    }
    int ri(int n) {
        return 2 * n + 2;
    }
    void build(int n, int esq, int dir, const vector<ll> &v) {
        if (esq == dir) {
            tree[n] = v[esq];
        } else {
            int mid = (esq + dir) / 2;
            build(le(n), esq, mid, v);
            build(ri(n), mid + 1, dir, v);
```

```
            tree[n] = merge(tree[le(n)], tree[ri(n)]);
        }
    }
    void build(const vector<ll> &v) {
        n = v.size();
        build(0, 0, n - 1, v);
    }
    ll query(int n, int esq, int dir, int l, int r) {
        if (esq > r || dir < l) {
            return 0;
        }
        if (l <= esq && dir <= r) {
            return tree[n];
        }
        int mid = (esq + dir) / 2;
        return merge(query(le(n), esq, mid, l, r), query(ri(n), mid +
            1, dir, l, r));
    }
    ll query(int l, int r) {
        return query(0, 0, n - 1, l, r);
    }
    void update(int n, int esq, int dir, int x, ll v) {
        if (esq > x || dir < x) {
            return;
        }
        if (esq == dir) {
            tree[n] = v;
        } else {
            int mid = (esq + dir) / 2;
            if (x <= mid) {
                update(le(n), esq, mid, x, v);
            } else {
                update(ri(n), mid + 1, dir, x, v);
            }
            tree[n] = merge(tree[le(n)], tree[ri(n)]);
        }
    }
    void update(int x, ll v) {
        update(0, 0, n - 1, x, v);
    }
}
```

1.10.2 Segment Tree 2D

Segment Tree em 2 dimensões.

- Complexidade de tempo (Pré-processamento): O(N*M)
- Complexidade de tempo (Consulta em intervalo): O(log(N)*log(M))
- Complexidade de tempo (Update em ponto): O(log(N)*log(M))
- Complexidade de espaço: 4 * N * 4 * M = O(N*M)

```
const int MAX = 2505;

int n, m, mat[MAX][MAX], tree[4 * MAX][4 * MAX];

int le(int x) {
    return 2 * x + 1;
}
int ri(int x) {
    return 2 * x + 2;
}

void build_y(int nx, int lx, int rx, int ny, int ly, int ry) {
    if (ly == ry) {
        if (lx == rx) {
            tree[nx][ny] = mat[lx][ly];
        } else {
            tree[nx][ny] = tree[le(nx)][ny] + tree[ri(nx)][ny];
        }
    } else {
        int my = (ly + ry) / 2;
        build_y(nx, lx, rx, le(ny), ly, my);
        build_y(nx, lx, rx, ri(ny), my + 1, ry);
        tree[nx][ny] = tree[nx][le(ny)] + tree[nx][ri(ny)];
    }
}
void build_x(int nx, int lx, int rx) {
    if (lx != rx) {
        int mx = (lx + rx) / 2;
        build_x(le(nx), lx, mx);
        build_x(ri(nx), mx + 1, rx);
    }
    build_y(nx, lx, rx, 0, 0, m - 1);
}
void build() {
    build_x(0, 0, n - 1);
}

void update_y(int nx, int lx, int rx, int ny, int ly, int ry, int x,
    int y, int v) {
    if (ly == ry) {
        if (lx == rx) {
            tree[nx][ny] = v;
        } else {
            tree[nx][ny] = tree[le(nx)][ny] + tree[ri(nx)][ny];
        }
    } else {
        int my = (ly + ry) / 2;
        if (y <= my) {
            update_y(nx, lx, rx, le(ny), ly, my, x, y, v);
        } else {
            update_y(nx, lx, rx, ri(ny), my + 1, ry, x, y, v);
        }
        tree[nx][ny] = tree[nx][le(ny)] + tree[nx][ri(ny)];
    }
}
void update_x(int nx, int lx, int rx, int x, int y, int v) {
    if (lx != rx) {
        int mx = (lx + rx) / 2;
        if (x <= mx) {
            update_x(le(nx), lx, mx, x, y, v);
        } else {
            update_x(ri(nx), mx + 1, rx, x, y, v);
        }
    }
    update_y(nx, lx, rx, 0, 0, m - 1, x, y, v);
}
void update(int x, int y, int v) {
    update_x(0, 0, n - 1, x, y, v);
}

int sum_y(int nx, int ny, int ly, int ry, int qly, int qry) {
    if (ry < qly || ly > qry) {
        return 0;
    }
    if (qly <= ly && ry <= qry) {
        return tree[nx][ny];
    }
    int my = (ly + ry) / 2;
    return sum_y(nx, le(ny), ly, my, qly, qry) + sum_y(nx, ri(ny), my
        + 1, ry, qly, qry);
}
int sum_x(int nx, int lx, int rx, int qlx, int qrx, int qly, int qry)
```

```
    {
    if (rx < qlx || lx > qrx) {
        return 0;
    }
    if (qlx <= lx && rx <= qrx) {
        return sum_y(nx, 0, 0, m - 1, qly, qry);
    }
    int mx = (lx + rx) / 2;
    return sum_x(le(nx), lx, mx, qlx, qrx, qly, qry) +
           sum_x(ri(nx), mx + 1, rx, qlx, qrx, qly, qry);
}
int sum(int lx, int rx, int ly, int ry) {
    return sum_x(0, 0, n - 1, lx, rx, ly, ry);
}
```

1.10.3 Segment Tree Beats Max And Sum Update

Seg Tree que suporta update de maximo, update de soma e query de soma.

Utiliza uma fila de lazy para diferenciar os updates

- Complexidade de tempo (Pré-processamento): O(N)
- Complexidade de tempo (Consulta em intervalo): O(log(N))
- Complexidade de tempo (Update em ponto): O(log(N))
- Complexidade de tempo (Update em intervalo): O(log(N))
- Complexidade de espaço: 2 *4 *N = O(N)

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

#define ll long long
#define INF 1e9
#define fi first
#define se second

typedef pair<int, int> ii;

struct Node {
    int m1 = INF, m2 = INF, cont = 0;
    ll soma = 0;
    queue<ii> lazy;
```

```
    void set(int v) {
        m1 = v;
        cont = 1;
        soma = v;
    }

    void merge(Node a, Node b) {
        m1 = min(a.m1, b.m1);
        m2 = INF;
        if (a.m1 != b.m1) {
            m2 = min(m2, max(a.m1, b.m1));
        }
        if (a.m2 != m1) {
            m2 = min(m2, a.m2);
        }
        if (b.m2 != m1) {
            m2 = min(m2, b.m2);
        }
        cont = (a.m1 == m1 ? a.cont : 0) + (b.m1 == m1 ? b.cont : 0);
        soma = a.soma + b.soma;
    }

    void print() {
        printf("%d %d %d %lld\n", m1, m2, cont, soma);
    }
};

int n, q;
vector<Node> tree;

int le(int n) {
    return 2 * n + 1;
}
int ri(int n) {
    return 2 * n + 2;
}

void push(int n, int esq, int dir) {
    while (!tree[n].lazy.empty()) {
        ii p = tree[n].lazy.front();
        tree[n].lazy.pop();
        int op = p.fi, v = p.se;
        if (op == 0) {
            if (v <= tree[n].m1) {
                continue;
            }
            tree[n].soma += (ll)abs(tree[n].m1 - v) * tree[n].cont;
            tree[n].m1 = v;
            if (esq != dir) {
                tree[le(n)].lazy.push({0, v});
                tree[ri(n)].lazy.push({0, v});
            }
        } else if (op == 1) {
            tree[n].soma += v * (dir - esq + 1);
            tree[n].m1 += v;
            tree[n].m2 += v;
            if (esq != dir) {
                tree[le(n)].lazy.push({1, v});
                tree[ri(n)].lazy.push({1, v});
            }
        }
    }
}

void build(int n, int esq, int dir, vector<int> &v) {
    if (esq == dir) {
        tree[n].set(v[esq]);
    } else {
        int mid = (esq + dir) / 2;
        build(le(n), esq, mid, v);
        build(ri(n), mid + 1, dir, v);
        tree[n].merge(tree[le(n)], tree[ri(n)]);
    }
}
void build(vector<int> &v) {
    build(0, 0, n - 1, v);
}

// ai = max(ai, mi) em [l, r]
void update(int n, int esq, int dir, int l, int r, int mi) {
    push(n, esq, dir);
```

```
    if (esq > r || dir < l || mi <= tree[n].m1) {
        return;
    }
    if (l <= esq && dir <= r && mi < tree[n].m2) {
        tree[n].soma += (ll)abs(tree[n].m1 - mi) * tree[n].cont;
        tree[n].m1 = mi;
        if (esq != dir) {
            tree[le(n)].lazy.push({0, mi});
            tree[ri(n)].lazy.push({0, mi});
        }
    } else {
        int mid = (esq + dir) / 2;
        update(le(n), esq, mid, l, r, mi);
        update(ri(n), mid + 1, dir, l, r, mi);
        tree[n].merge(tree[le(n)], tree[ri(n)]);
    }
}
void update(int l, int r, int mi) {
    update(0, 0, n - 1, l, r, mi);
}

// soma v em [l, r]
void upsoma(int n, int esq, int dir, int l, int r, int v) {
    push(n, esq, dir);
    if (esq > r || dir < l) {
        return;
    }
    if (l <= esq && dir <= r) {
        tree[n].soma += v * (dir - esq + 1);
        tree[n].m1 += v;
        tree[n].m2 += v;
        if (esq != dir) {
            tree[le(n)].lazy.push({1, v});
            tree[ri(n)].lazy.push({1, v});
        }
    } else {
        int mid = (esq + dir) / 2;
        upsoma(le(n), esq, mid, l, r, v);
        upsoma(ri(n), mid + 1, dir, l, r, v);
        tree[n].merge(tree[le(n)], tree[ri(n)]);
    }
}
void upsoma(int l, int r, int v) {
    upsoma(0, 0, n - 1, l, r, v);
}

// soma de [l, r]
int query(int n, int esq, int dir, int l, int r) {
    push(n, esq, dir);
    if (esq > r || dir < l) {
        return 0;
    }
    if (l <= esq && dir <= r) {
        return tree[n].soma;
    }
    int mid = (esq + dir) / 2;
    return query(le(n), esq, mid, l, r) + query(ri(n), mid + 1, dir,
        l, r);
}
int query(int l, int r) {
    return query(0, 0, n - 1, l, r);
}

int main() {
    cin >> n;
    tree.assign(4 * n, Node());
    build(v);
}
```

1.10.4 Segment Tree Beats Max Update

Seg Tree que suporta update de maximo e query de soma

- Complexidade de tempo (Pré-processamento): O(N)
- Complexidade de tempo (Consulta em intervalo): O(log(N))
- Complexidade de tempo (Update em ponto): O(log(N))
- Complexidade de tempo (Update em intervalo): O(log(N))
- Complexidade de espaço: 2 *4 *N = O(N)

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

#define ll long long
#define INF 1e9

struct Node {
    int m1 = INF, m2 = INF, cont = 0, lazy = 0;
    ll soma = 0;

    void set(int v) {
        m1 = v;
        cont = 1;
        soma = v;
    }

    void merge(Node a, Node b) {
        m1 = min(a.m1, b.m1);
        m2 = INF;
        if (a.m1 != b.m1) {
            m2 = min(m2, max(a.m1, b.m1));
        }
        if (a.m2 != m1) {
            m2 = min(m2, a.m2);
        }
        if (b.m2 != m1) {
            m2 = min(m2, b.m2);
        }
        cont = (a.m1 == m1 ? a.cont : 0) + (b.m1 == m1 ? b.cont : 0);
        soma = a.soma + b.soma;
    }

    void print() {
        printf("%d %d %d %lld %d\n", m1, m2, cont, soma, lazy);
    }
};

int n, q;
vector<Node> tree;

int le(int n) {
    return 2 * n + 1;
}
int ri(int n) {
```

```
    return 2 * n + 2;
}

void push(int n, int esq, int dir) {
    if (tree[n].lazy <= tree[n].m1) {
        return;
    }
    tree[n].soma += (ll)abs(tree[n].m1 - tree[n].lazy) * tree[n].cont;
    tree[n].m1 = tree[n].lazy;
    if (esq != dir) {
        tree[le(n)].lazy = max(tree[le(n)].lazy, tree[n].lazy);
        tree[ri(n)].lazy = max(tree[ri(n)].lazy, tree[n].lazy);
    }
    tree[n].lazy = 0;
}

void build(int n, int esq, int dir, vector<int> &v) {
    if (esq == dir) {
        tree[n].set(v[esq]);
    } else {
        int mid = (esq + dir) / 2;
        build(le(n), esq, mid, v);
        build(ri(n), mid + 1, dir, v);
        tree[n].merge(tree[le(n)], tree[ri(n)]);
    }
}
void build(vector<int> &v) {
    build(0, 0, n - 1, v);
}

// ai = max(ai, mi) em [l, r]
void update(int n, int esq, int dir, int l, int r, int mi) {
    push(n, esq, dir);
    if (esq > r || dir < l || mi <= tree[n].m1) {
        return;
    }
    if (l <= esq && dir <= r && mi < tree[n].m2) {
        tree[n].lazy = mi;
        push(n, esq, dir);
    } else {
        int mid = (esq + dir) / 2;
        update(le(n), esq, mid, l, r, mi);
        update(ri(n), mid + 1, dir, l, r, mi);
        tree[n].merge(tree[le(n)], tree[ri(n)]);
    }
}
void update(int l, int r, int mi) {
    update(0, 0, n - 1, l, r, mi);
}

// soma de [l, r]
int query(int n, int esq, int dir, int l, int r) {
    push(n, esq, dir);
    if (esq > r || dir < l) {
        return 0;
    }
    if (l <= esq && dir <= r) {
        return tree[n].soma;
    }
    int mid = (esq + dir) / 2;
    return query(le(n), esq, mid, l, r) + query(ri(n), mid + 1, dir,
        l, r);
}
int query(int l, int r) {
    return query(0, 0, n - 1, l, r);
}

int main() {
    cin >> n;
    tree.assign(4 * n, Node());
}
```

1.10.5 Segment Tree Esparsa

Seg Tree Esparsa, ou seja, uma seg tree que não guarda todos os nós, mas apenas os nós que são necessários para responder as queries, permitindo fazer queries em intervalos de tamanho arbitrário.

Seja LEN o tamanho do intervalo em que a Seg Tree foi construída:

- Complexidade de tempo (Pré-processamento): $O(1)$
- Complexidade de tempo (Consulta em intervalo): $O(log(LEN))$
- Complexidade de tempo (Update em ponto): $O(log(LEN))$

```
const int SEGMAX = 8e6 + 5; // should be Q * log(DIR-ESQ+1)
const ll ESQ = 0, DIR = 1e9 + 7;

struct seg {
    ll tree[SEGMAX];
    int R[SEGMAX], L[SEGMAX],
        ptr = 2; // 0 is NULL; 1 is First Root
    ll op(ll a, ll b) {
        return (a + b) % MOD;
    }
    int le(int i) {
        if (L[i] == 0) {
            L[i] = ptr++;
        }
        return L[i];
    }
    int ri(int i) {
        if (R[i] == 0) {
            R[i] = ptr++;
        }
        return R[i];
    }
    ll query(ll l, ll r, int n = 1, ll esq = ESQ, ll dir = DIR) {
        if (r < esq || dir < l) {
            return 0;
        }
        if (l <= esq && dir <= r) {
            return tree[n];
        }
        ll mid = (esq + dir) / 2;
        return op(query(l, r, le(n), esq, mid), query(l, r, ri(n),
           mid + 1, dir));
    }
    void update(ll x, ll v, int n = 1, ll esq = ESQ, ll dir = DIR) {
        if (esq == dir) {
            tree[n] = (tree[n] + v) % MOD;
        } else {
            ll mid = (esq + dir) / 2;
            if (x <= mid) {
                update(x, v, le(n), esq, mid);
            } else {
                update(x, v, ri(n), mid + 1, dir);
            }
            tree[n] = op(tree[le(n)], tree[ri(n)]);
        }
    }
};
```

1.10.6 Segment Tree Kadani

Implementação de uma Seg Tree que suporta update de soma e query de soma máxima em intervalo.

- Complexidade de tempo (Pré-processamento): O(N)
- Complexidade de tempo (Consulta em intervalo): O(log(N))
- Complexidade de tempo (Update em ponto): O(log(N))
- Complexidade de espaço: 4 * N = O(N)

```
namespace seg {
    const int MAX = 1e5 + 5;
    struct node {
        ll pref, suff, sum, best;
    };
    node new_node(ll v) {
        return node{v, v, v, v};
    }
    const node NEUTRAL = {0, 0, 0, 0};
    node tree[4 * MAX];
    node merge(node a, node b) {
        ll pref = max(a.pref, a.sum + b.pref);
        ll suff = max(b.suff, b.sum + a.suff);
        ll sum = a.sum + b.sum;
        ll best = max(a.suff + b.pref, max(a.best, b.best));
        return node{pref, suff, sum, best};
    }

    int n;
    int le(int n) {
        return 2 * n + 1;
    }
    int ri(int n) {
        return 2 * n + 2;
    }
    void build(int n, int esq, int dir, const vector<ll> &v) {
        if (esq == dir) {
            tree[n] = new_node(v[esq]);
        } else {
            int mid = (esq + dir) / 2;
            build(le(n), esq, mid, v);
            build(ri(n), mid + 1, dir, v);
            tree[n] = merge(tree[le(n)], tree[ri(n)]);
        }
    }
    void build(const vector<ll> &v) {
        n = v.size();
        build(0, 0, n - 1, v);
    }
    node query(int n, int esq, int dir, int l, int r) {
        if (esq > r || dir < l) {
            return NEUTRAL;
```

```
        }
        if (l <= esq && dir <= r) {
            return tree[n];
        }
        int mid = (esq + dir) / 2;
        return merge(query(le(n), esq, mid, l, r), query(ri(n), mid +
            1, dir, l, r));
    }
    ll query(int l, int r) {
        return query(0, 0, n - 1, l, r).best;
    }
    void update(int n, int esq, int dir, int x, ll v) {
        if (esq > x || dir < x) {
            return;
        }
        if (esq == dir) {
            tree[n] = new_node(v);
        } else {
            int mid = (esq + dir) / 2;
            if (x <= mid) {
                update(le(n), esq, mid, x, v);
            } else {
                update(ri(n), mid + 1, dir, x, v);
            }
            tree[n] = merge(tree[le(n)], tree[ri(n)]);
        }
    }
    void update(int x, ll v) {
        update(0, 0, n - 1, x, v);
    }
}
```

1.10.7 Segment Tree Lazy

Implementação padrão de Seg Tree com lazy update

- Complexidade de tempo (Pré-processamento): O(N)
- Complexidade de tempo (Consulta em intervalo): O(log(N))
- Complexidade de tempo (Update em ponto): O(log(N))
- Complexidade de tempo (Update em intervalo): O(log(N))
- Complexidade de espaço: 2 * 4 * N = O(N)

```
namespace seg {
    const int MAX = 2e5 + 5;
    const ll NEUTRAL = 0; // merge(a, neutral) = a
    ll merge(ll a, ll b) {
        return a + b;
    }
    int sz; // size of the array
    ll tree[4 * MAX], lazy[4 * MAX];
    int le(int n) {
        return 2 * n + 1;
    }
    int ri(int n) {
        return 2 * n + 2;
    }
    void push(int n, int esq, int dir) {
        if (lazy[n] == 0) {
            return;
        }
        tree[n] += lazy[n] * (dir - esq + 1);
        if (esq != dir) {
            lazy[le(n)] += lazy[n];
            lazy[ri(n)] += lazy[n];
        }
        lazy[n] = 0;
    }
    void build(span<const ll> v, int n, int esq, int dir) {
        if (esq == dir) {
            tree[n] = v[esq];
        } else {
            int mid = (esq + dir) / 2;
            build(v, le(n), esq, mid);
            build(v, ri(n), mid + 1, dir);
            tree[n] = merge(tree[le(n)], tree[ri(n)]);
        }
    }
    void build(span<const ll> v) {
        sz = v.size();
        build(v, 0, 0, sz - 1);
    }
    ll query(int l, int r, int n = 0, int esq = 0, int dir = sz - 1) {
        push(n, esq, dir);
        if (esq > r || dir < l) {
            return NEUTRAL;
        }
        if (l <= esq && dir <= r) {
            return tree[n];
        }
        int mid = (esq + dir) / 2;
        return merge(query(l, r, le(n), esq, mid), query(l, r, ri(n),
           mid + 1, dir));
    }
    void update(int l, int r, ll v, int n = 0, int esq = 0, int dir =
        sz - 1) {
        push(n, esq, dir);
        if (esq > r || dir < l) {
            return;
        }
        if (l <= esq && dir <= r) {
            lazy[n] += v;
            push(n, esq, dir);
        } else {
            int mid = (esq + dir) / 2;
            update(l, r, v, le(n), esq, mid);
            update(l, r, v, ri(n), mid + 1, dir);
            tree[n] = merge(tree[le(n)], tree[ri(n)]);
        }
    }
}
```

1.10.8 Segment Tree Persisente

Seg Tree Esparsa com histórico de Updates:

- Complexidade de tempo (Pré-processamento): O(N *log(N))
- Complexidade de tempo (Consulta em intervalo): O(log(N))
- Complexidade de tempo (Update em ponto): O(log(N))
- **Para fazer consulta em um tempo específico basta indicar o tempo na query**

```
namespace seg {
    const ll ESQ = 0, DIR = 1e9 + 7;
    struct node {
        ll v = 0;
        node *l = NULL, *r = NULL;
        node() {
        }
        node(ll v) : v(v) {
        }
        node(node *l, node *r) : l(l), r(r) {
            v = l->v + r->v;
        }
        void apply() {
            if (l == NULL) {
                l = new node();
            }
            if (r == NULL) {
                r = new node();
            }
        }
    };
    vector<node *> roots;
    void build() {
        roots.push_back(new node());
    }
    void push(node *n, int esq, int dir) {
        if (esq != dir) {
            n->apply();
        }
    }
    // sum v on x
    node *update(node *n, int esq, int dir, int x, int v) {
        push(n, esq, dir);
        if (esq == dir) {
            return new node(n->v + v);
        }
        int mid = (esq + dir) / 2;
        if (x <= mid) {
            return new node(update(n->l, esq, mid, x, v), n->r);
        } else {
            return new node(n->l, update(n->r, mid + 1, dir, x, v));
        }
    }
    int update(int root, int pos, int val) {
        node *novo = update(roots[root], ESQ, DIR, pos, val);
        roots.push_back(novo);
        return roots.size() - 1;
    }
```

```
    // sum in [L, R]
    ll query(node *n, int esq, int dir, int l, int r) {
        push(n, esq, dir);
        if (esq > r || dir < l) {
            return 0;
        }
        if (l <= esq && dir <= r) {
            return n->v;
        }
        int mid = (esq + dir) / 2;
        return query(n->l, esq, mid, l, r) + query(n->r, mid + 1,
            dir, l, r);
    }
    ll query(int root, int l, int r) {
        return query(roots[root], ESQ, DIR, l, r);
    }
    // kth min number in [L, R] (l_root can not be
    // 0)
    int kth(node *L, node *R, int esq, int dir, int k) {
        push(L, esq, dir);
        push(R, esq, dir);
        if (esq == dir) {
            return esq;
        }
        int mid = (esq + dir) / 2;
        int cont = R->l->v - L->l->v;
        if (cont >= k) {
            return kth(L->l, R->l, esq, mid, k);
        } else {
            return kth(L->r, R->r, mid + 1, dir, k - cont);
        }
    }
    int kth(int l_root, int r_root, int k) {
        return kth(roots[l_root - 1], roots[r_root], ESQ, DIR, k);
    }
};
```

1.11 Sparse Table

1.11.1 Disjoint Sparse Table

Resolve query de range para qualquer operação associativa em $O(1)$.

Pré-processamento em $O(n \log n)$.

```
struct dst {
    const int neutral = 1;
#define comp(a, b) (a | b)
    vector<vector<int>> t;
    dst(vector<int> v) {
        int n, k, sz = v.size();
        for (n = 1, k = 0; n < sz; n <<= 1, k++)
            ;
        t.assign(k, vector<int>(n));
        for (int i = 0; i < n; i++) {
            t[0][i] = i < sz ? v[i] : neutral;
        }
        for (int j = 0, len = 1; j <= k; j++, len <<= 1) {
            for (int s = len; s < n; s += (len << 1)) {
```

```
                t[j][s] = v[s];
                t[j][s - 1] = v[s - 1];
                for (int i = 1; i < len; i++) {
                    t[j][s + i] = comp(t[j][s + i - 1], v[s + i]);
                    t[j][s - 1 - i] = comp(v[s - 1 - i], t[j][s - i]);
                }
            }
        }
    }
```

```
    int query(int l, int r) {
        if (l == r) {
            return t[0][r];
        }
        int i = 31 - __builtin_clz(l ^ r);
        return comp(t[i][l], t[i][r]);
    }
};
```

1.11.2 Sparse Table

Read in [English](README.en.md)

Responde consultas de maneira eficiente em um conjunto de dados estáticos.

Realiza um pré-processamento para diminuir o tempo de cada consulta.

- Complexidade de tempo (Pré-processamento): O(N * log(N))
- Complexidade de tempo (Consulta para operações sem sobreposição amigável): O(N * log(N))
- Complexidade de tempo (Consulta para operações com sobreposição amigável): O(1)
- Complexidade de espaço: O(N * log(N))

Exemplo de operações com sobreposição amigável: max(), min(), gcd(), f(x, y) = x

```
struct SparseTable {
    int n, e;
    vector<vector<int>> st;
    SparseTable(vector<int> &v) : n(v.size()), e(floor(log2(n))) {
```

```
        st.assign(e + 1, vector<int>(n));
        for (int i = 0; i < n; i++) {
            st[0][i] = v[i];
        }
```

```
        for (int i = 1; i <= e; i++) {
            for (int j = 0; j + (1 << i) <= n; j++) {
                st[i][j] = min(st[i - 1][j], st[i - 1][j + (1 << (i -
                    1))]);
            }
        }
    }
    // O(log(N)) Query for non overlap friendly
    // operations
    int logquery(int l, int r) {
        int res = 2e9;
        for (int i = e; i >= 0; i--) {
            if ((1 << i) <= r - l + 1) {
                res = min(res, st[i][l]);
                l += 1 << i;
            }
        }
        return res;
    }
    // O(1) Query for overlab friendly operations
    // ex: max(), min(), gcd(), f(x, y) = x
    int query(int l, int r) {
        // if (l > r) return 2e9;
        int i = ilogb(r - l + 1);
        return min(st[i][l], st[i][r - (1 << i) + 1]);
    }
};
```

Capítulo 2

Grafos

2.1 2 SAT

Resolve problema do 2-SAT.

- Complexidade de tempo (caso médio): O(N + M)

N é o número de variáveis e M é o número de cláusulas.

A configuração da solução fica guardada no vetor *assignment*.

Em relaçõa ao sinal, tanto faz se 0 liga ou desliga, apenas siga o mesmo padrão.

```
struct sat2 {
    int n;
    vector<vector<int>> g, gt;
    vector<bool> used;
    vector<int> order, comp;
    vector<bool> assignment;

    // number of variables
    sat2(int _n) {
        n = 2 * (_n + 5);
        g.assign(n, vector<int>());
        gt.assign(n, vector<int>());
```

```
    }
    void add_edge(int v, int u, bool v_sign, bool u_sign) {
        g[2 * v + v_sign].push_back(2 * u + !u_sign);
        g[2 * u + u_sign].push_back(2 * v + !v_sign);
        gt[2 * u + !u_sign].push_back(2 * v + v_sign);
        gt[2 * v + !v_sign].push_back(2 * u + u_sign);
    }
    void dfs1(int v) {
        used[v] = true;
        for (int u : g[v]) {
            if (!used[u]) {
                dfs1(u);
            }
        }
        order.push_back(v);
    }
    void dfs2(int v, int cl) {
        comp[v] = cl;
        for (int u : gt[v]) {
            if (comp[u] == -1) {
                dfs2(u, cl);
            }
        }
    }
    bool solve() {
        order.clear();
        used.assign(n, false);
        for (int i = 0; i < n; ++i) {
            if (!used[i]) {
                dfs1(i);
            }
        }

        comp.assign(n, -1);
        for (int i = 0, j = 0; i < n; ++i) {
            int v = order[n - i - 1];
            if (comp[v] == -1) {
                dfs2(v, j++);
            }
        }

        assignment.assign(n / 2, false);
        for (int i = 0; i < n; i += 2) {
            if (comp[i] == comp[i + 1]) {
                return false;
            }
            assignment[i / 2] = comp[i] > comp[i + 1];
        }
        return true;
    }
};
```

2.2 Binary Lifting

Usa uma sparse table para calcular o k-ésimo ancestral de u.

Pode ser usada com o algoritmo de EulerTour para calcular o LCA.

Complexidade de tempo:

- Pré-processamento: O(N * log(N))
- Consulta do k-ésimo ancestral de u: O(log(N))
- LCA: O(log(N))

Complexidade de espaço: O(Nlog(N))

```cpp
namespace st {
    int n, me, timer;
    vector<int> tin, tout;
    vector<vector<int>> st;
    void et_dfs(int u, int p) {
        tin[u] = ++timer;
        st[u][0] = p;
        for (int i = 1; i <= me; i++) {
            st[u][i] = st[st[u][i - 1]][i - 1];
        }
        for (int v : adj[u]) {
            if (v != p) {
                et_dfs(v, u);
            }
        }
        tout[u] = ++timer;
    }
    void build(int _n, int root = 0) {
        n = _n;
        tin.assign(n, 0);
        tout.assign(n, 0);
        timer = 0;
        me = floor(log2(n));
        st.assign(n, vector<int>(me + 1, 0));
        et_dfs(root, root);
    }
    bool is_ancestor(int u, int v) {
        return tin[u] <= tin[v] && tout[u] >= tout[v];
    }
    int lca(int u, int v) {
        if (is_ancestor(u, v)) {
            return u;
        }
        if (is_ancestor(v, u)) {
            return v;
        }
        for (int i = me; i >= 0; i--) {
            if (!is_ancestor(st[u][i], v)) {
                u = st[u][i];
            }
        }
        return st[u][0];
    }
    int ancestor(int u,
                 int k) { // k-th ancestor of u
        for (int i = me; i >= 0; i--) {
            if ((1 << i) & k) {
                u = st[u][i];
            }
        }
        return u;
    }
}
```

```cpp
namespace st {
    int n, me;
    vector<vector<int>> st;
    void bl_dfs(int u, int p) {
        st[u][0] = p;
        for (int i = 1; i <= me; i++) {
            st[u][i] = st[st[u][i - 1]][i - 1];
        }
        for (int v : adj[u]) {
            if (v != p) {
                bl_dfs(v, u);
            }
        }
    }
    void build(int _n, int root = 0) {
        n = _n;
        me = floor(log2(n));
        st.assign(n, vector<int>(me + 1, 0));
        bl_dfs(root, root);
    }
    int ancestor(int u,
                 int k) { // k-th ancestor of u
        for (int i = me; i >= 0; i--) {
            if ((1 << i) & k) {
                u = st[u][i];
            }
        }
        return u;
    }
}
```

2.3 Bridge

Algoritmo que acha pontes utilizando uma dfs

Complexidade de tempo: O(N + M)

```cpp
int n;                        // number of nodes
vector<vector<int>> adj; // adjacency list of graph

vector<bool> visited;
vector<int> tin, low;
int timer;

void dfs(int u, int p = -1) {
    visited[u] = true;
    tin[u] = low[u] = timer++;
    for (int v : adj[u]) {
        if (v == p) {
            continue;
        }
        if (visited[v]) {
            low[u] = min(low[u], tin[v]);
        } else {
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            if (low[v] > tin[u]) {
                // edge UV is a bridge
                // do_something(u, v)
            }
        }
```

```
    }
}

void find_bridges() {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i]) {
            dfs(i);
        }
    }
}
```

2.4 Fluxo

Conjunto de algoritmos para calcular o fluxo máximo em redes de fluxo.

Muito útil para grafos bipartidos e para grafos com muitas arestas

Complexidade de tempo: O(V^2 * E), mas em grafo bipartido a complexidade é O(sqrt(V) * E)

Útil para grafos com poucas arestas

Complexidade de tempo: O(V * E^2)

Computa o fluxo máximo com custo mínimo

Complexidade de tempo: O(V^2 * E^2)

```
const long long INF = 1e18;

struct FlowEdge {
    int u, v;
    long long cap, flow = 0;
    FlowEdge(int u, int v, long long cap) : u(u), v(v), cap(cap) {
```

```
    }
};

struct EdmondsKarp {
    int n, s, t, m = 0, vistoken = 0;
    vector<FlowEdge> edges;
    vector<vector<int>> adj;
    vector<int> visto;

    EdmondsKarp(int n, int s, int t) : n(n), s(s), t(t) {
        adj.resize(n);
        visto.resize(n);
    }

    void add_edge(int u, int v, long long cap) {
        edges.emplace_back(u, v, cap);
        edges.emplace_back(v, u, 0);
        adj[u].push_back(m);
        adj[v].push_back(m + 1);
        m += 2;
    }

    int bfs() {
        vistoken++;
        queue<int> fila;
        fila.push(s);
        vector<int> pego(n, -1);
        while (!fila.empty()) {
            int u = fila.front();
            if (u == t) {
                break;
            }
            fila.pop();
            visto[u] = vistoken;
            for (int id : adj[u]) {
                if (edges[id].cap - edges[id].flow < 1) {
                    continue;
                }
                int v = edges[id].v;
                if (visto[v] == -1) {
                    continue;
                }
                fila.push(v);
                pego[v] = id;
            }
        }
        if (pego[t] == -1) {
            return 0;
        }
        long long f = INF;
        for (int id = pego[t]; id != -1; id = pego[edges[id].u]) {
            f = min(f, edges[id].cap - edges[id].flow);
        }
        for (int id = pego[t]; id != -1; id = pego[edges[id].u]) {
            edges[id].flow += f;
            edges[id ^ 1].flow -= f;
        }
        return f;
    }

    long long flow() {
        long long maxflow = 0;
        while (long long f = bfs()) {
            maxflow += f;
        }
        return maxflow;
    }
};
```

```
struct MinCostMaxFlow {
    int n, s, t, m = 0;
    ll maxflow = 0, mincost = 0;
    vector<FlowEdge> edges;
    vector<vector<int>> adj;

    MinCostMaxFlow(int n, int s, int t) : n(n), s(s), t(t) {
        adj.resize(n);
    }

    void add_edge(int u, int v, ll cap, ll cost) {
        edges.emplace_back(u, v, cap, cost);
        edges.emplace_back(v, u, 0, -cost);
        adj[u].push_back(m);
        adj[v].push_back(m + 1);
        m += 2;
    }

    bool spfa() {
        vector<int> pego(n, -1);
        vector<ll> dis(n, INF);
        vector<bool> inq(n, false);
        queue<int> fila;
        fila.push(s);
        dis[s] = 0;
        inq[s] = 1;
        while (!fila.empty()) {
            int u = fila.front();
            fila.pop();
            inq[u] = false;
            for (int id : adj[u]) {
                if (edges[id].cap - edges[id].flow < 1) {
                    continue;
                }
                int v = edges[id].v;
                if (dis[v] > dis[u] + edges[id].cost) {
                    dis[v] = dis[u] + edges[id].cost;
                    pego[v] = id;
                    if (!inq[v]) {
                        inq[v] = true;
                        fila.push(v);
                    }
                }
            }
        }

        if (pego[t] == -1) {
            return 0;
        }
        ll f = INF;
        for (int id = pego[t]; id != -1; id = pego[edges[id].u]) {
            f = min(f, edges[id].cap - edges[id].flow);
            mincost += edges[id].cost;
        }
        for (int id = pego[t]; id != -1; id = pego[edges[id].u]) {
            edges[id].flow += f;
            edges[id ^ 1].flow -= f;
        }
        maxflow += f;
        return 1;
    }

    ll flow() {
        while (spfa())
            ;
        return maxflow;
    }
};
```

```
typedef long long ll;

const ll INF = 1e18;

struct FlowEdge {
    int u, v;
    ll cap, flow = 0;
    FlowEdge(int u, int v, ll cap) : u(u), v(v), cap(cap) {
    }
};

struct Dinic {
    vector<FlowEdge> edges;
    vector<vector<int>> adj;
    int n, s, t, m = 0;
    vector<int> level, ptr;
    queue<int> q;

    Dinic(int n, int s, int t) : n(n), s(s), t(t) {
        adj.resize(n);
        level.resize(n);
        ptr.resize(n);
    }

    void add_edge(int u, int v, ll cap) {
        edges.emplace_back(u, v, cap);
        edges.emplace_back(v, u, 0);
        adj[u].push_back(m);
        adj[v].push_back(m + 1);
        m += 2;
    }

    bool bfs() {
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (int id : adj[u]) {
                if (edges[id].cap - edges[id].flow < 1) {
                    continue;
                }
                int v = edges[id].v;
                if (level[v] != -1) {
                    continue;
                }
                level[v] = level[u] + 1;
                q.push(v);
            }
        }
        return level[t] != -1;
    }

    ll dfs(int u, ll f) {
        if (f == 0) {
            return 0;
        }
        if (u == t) {
            return f;
        }
        for (int &cid = ptr[u]; cid < (int)adj[u].size(); cid++) {
            int id = adj[u][cid];
            int v = edges[id].v;
            if (level[u] + 1 != level[v] || edges[id].cap -
               edges[id].flow < 1) {
                continue;
            }
            ll tr = dfs(v, min(f, edges[id].cap - edges[id].flow));
            if (tr == 0) {
                continue;
            }
            edges[id].flow += tr;
            edges[id ^ 1].flow -= tr;
            return tr;
        }
        return 0;
    }

    ll flow() {
        ll maxflow = 0;
        while (true) {
            fill(level.begin(), level.end(), -1);
            level[s] = 0;
            q.push(s);
```

```
            if (!bfs()) {
                break;
            }
            fill(ptr.begin(), ptr.end(), 0);
            while (ll f = dfs(s, INF)) {
                maxflow += f;
            }
        }
        return maxflow;
    }
};
```

2.5 Graph Center

Encontra o centro e o diâmetro de um grafo

Complexidade de tempo: O(N)

```
const int INF = 1e9 + 9;

vector<vector<int>> adj;

struct GraphCenter {
    int n, diam = 0;
    vector<int> centros, dist, pai;
    int bfs(int s) {
        queue<int> q;
        q.push(s);
        dist.assign(n + 5, INF);
        pai.assign(n + 5, -1);
        dist[s] = 0;
        int maxidist = 0, maxinode = 0;
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            if (dist[u] >= maxidist) {
                maxidist = dist[u], maxinode = u;
            }
            for (int v : adj[u]) {
                if (dist[u] + 1 < dist[v]) {
                    dist[v] = dist[u] + 1;
                    pai[v] = u;
                    q.push(v);
                }
            }
        }
        diam = max(diam, maxidist);
        return maxinode;
    }
    GraphCenter(int st = 0) : n(adj.size()) {
        int d1 = bfs(st);
        int d2 = bfs(d1);
        vector<int> path;
        for (int u = d2; u != -1; u = pai[u]) {
            path.push_back(u);
        }
        int len = path.size();
        if (len % 2 == 1) {
```

```
            centros.push_back(path[len / 2]);
        } else {
            centros.push_back(path[len / 2]);
            centros.push_back(path[len / 2 - 1]);
                }
            }
        };
```

2.6 HLD

Técnica usada para otimizar a execução de operações em árvores.

- Pré-Processamento: O(N)
- Range Query/Update: O(Log(N)) * O(Complexidade de query da estrutura)
- Point Query/Update: O(Complexidade de query da estrutura)
- LCA: O(Log(N))
- Subtree Query: O(Complexidade de query da estrutura)
- Complexidade de espaço: O(N)

```
namespace hld {
    const int MAX = 2e5 + 5;
    int t, sz[MAX], pos[MAX], pai[MAX], head[MAX];
    bool e = 0;
    ll merge(ll a, ll b) {
        return max(a, b); // how to merge paths
    }
    void dfs_sz(int u, int p = -1) {
        sz[u] = 1;
        for (int &v : adj[u]) {
            if (v != p) {
                dfs_sz(v, u);
                sz[u] += sz[v];
                if (sz[v] > sz[adj[u][0]] || adj[u][0] == p) {
                    swap(v, adj[u][0]);
                }
            }
        }
    }
    void dfs_hld(int u, int p = -1) {
```

```
        pos[u] = t++;
        for (int v : adj[u]) {
            if (v != p) {
                pai[v] = u;
                head[v] = (v == adj[u][0] ? head[u] : v);
                dfs_hld(v, u);
            }
        }
    }
    void build(int root) {
        dfs_sz(root);
        t = 0;
        pai[root] = root;
        head[root] = root;
        dfs_hld(root);
    }
    void build(int root, vector<ll> &v) {
        build(root);
        vector<ll> aux(v.size());
        for (int i = 0; i < (int)v.size(); i++) {
            aux[pos[i]] = v[i];
        }
        seg::build(aux);
    }
    void build(int root,
               vector<i3> &edges) { // use this if
                                    // weighted edges
        build(root);
        e = 1;
        vector<ll> aux(edges.size() + 1);
        for (auto [u, v, w] : edges) {
            if (pos[u] > pos[v]) {
                swap(u, v);
            }
            aux[pos[v]] = w;
        }
        seg::build(aux);
    }
    ll query(int u, int v) {
        if (pos[u] > pos[v]) {
            swap(u, v);
        }
        if (head[u] == head[v]) {
            return seg::query(pos[u] + e, pos[v]);
        } else {
            ll qv = seg::query(pos[head[v]], pos[v]);
            ll qu = query(u, pai[head[v]]);
            return merge(qu, qv);
        }
    }
    void update(int u, int v, ll k) {
        if (pos[u] > pos[v]) {
            swap(u, v);
        }
        if (head[u] == head[v]) {
            seg::update(pos[u] + e, pos[v], k);
        } else {
            seg::update(pos[head[v]], pos[v], k);
            update(u, pai[head[v]], k);
        }
    }
    int lca(int u, int v) {
        if (pos[u] > pos[v]) {
            swap(u, v);
        }
        return (head[u] == head[v] ? u : lca(u, pai[head[v]]));
    }
    ll query_subtree(int u) {
        return seg::query(pos[u], pos[u] + sz[u] - 1);
    }
}
```

2.7 Inverse Graph

Algoritmo que encontra as componentes conexas quando se é dado o grafo complemento.

Resolve problemas em que se deseja encontrar as componentes conexas quando são dadas as arestas que não pertencem ao grafo

- Complexidade de tempo: O(N log N + N log M)

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

set<int> nodes;
vector<set<int>> adj;

void bfs(int s) {
    queue<int> f;
    f.push(s);
    nodes.erase(s);
    set<int> aux;
    while (!f.empty()) {
        int x = f.front();
        f.pop();
        for (int y : nodes) {
            if (adj[x].count(y) == 0) {
                aux.insert(y);
            }
        }
        for (int y : aux) {
            f.push(y);
            nodes.erase(y);
        }
        aux.clear();
    }
}
```

2.8 Kruskal

Algoritimo para encontrar a MST (minimum spanning tree) de um grafo.

Utiliza [DSU](../../Estruturas%20de%20Dados/DSU/dsu.cpp) - (disjoint set union) - para construir MST - (minimum spanning tree)

- Complexidade de tempo (Construção): O(M log N)

```
struct Edge {
    int u, v, w;
    bool operator<(Edge const &other) {
        return w < other.w;
    }
};

vector<Edge> edges, result;
int cost;

struct DSU {
    vector<int> pa, sz;
    DSU(int n) {
        sz.assign(n + 5, 1);
        for (int i = 0; i < n + 5; i++) {
            pa.push_back(i);
        }
    }
    int root(int a) {
        return pa[a] = (a == pa[a] ? a : root(pa[a]));
    }
    bool find(int a, int b) {
        return root(a) == root(b);
    }
    void uni(int a, int b) {
        int ra = root(a), rb = root(b);
        if (ra == rb) {
            return;
        }
        if (sz[ra] > sz[rb]) {
            swap(ra, rb);
        }
        pa[ra] = rb;
        sz[rb] += sz[ra];
    }
};

void kruskal(int m, int n) {
    DSU dsu(n);

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

    for (Edge e : edges) {
        if (!dsu.find(e.u, e.v)) {
            cost += e.w;
            result.push_back(e); // remove if need only cost
            dsu.uni(e.u, e.v);
        }
    }
}
```

2.9 LCA

Algoritmo de Lowest Common Ancestor usando EulerTour e Sparse Table

Complexidade de tempo:

- O(Nlog(N)) Preprocessing
- O(1) Query LCA

Complexidade de espaço: O(Nlog(N))

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

#define INF 1e9
#define fi first
#define se second

typedef pair<int, int> ii;

vector<int> tin, tout;
vector<vector<int>> adj;
vector<ii> prof;
vector<vector<ii>> st;

int n, timer;

void SparseTable(vector<ii> &v) {
    int n = v.size();
    int e = floor(log2(n));
    st.assign(e + 1, vector<ii>(n));
    for (int i = 0; i < n; i++) {
        st[0][i] = v[i];
    }
    for (int i = 1; i <= e; i++) {
        for (int j = 0; j + (1 << i) <= n; j++) {
            st[i][j] = min(st[i - 1][j], st[i - 1][j + (1 << (i -
                1))]);
        }
    }
}
```

```
void et_dfs(int u, int p, int h) {
    tin[u] = timer++;
    prof.emplace_back(h, u);
    for (int v : adj[u]) {
        if (v != p) {
            et_dfs(v, u, h + 1);
            prof.emplace_back(h, u);
        }
    }
    tout[u] = timer++;
}

void build(int root = 0) {
    tin.assign(n, 0);
    tout.assign(n, 0);
    prof.clear();
    timer = 0;
    et_dfs(root, root, 0);
    SparseTable(prof);
}

int lca(int u, int v) {
    int l = tout[u], r = tin[v];
    if (l > r) {
        swap(l, r);
    }
    int i = floor(log2(r - l + 1));
    return min(st[i][l], st[i][r - (1 << i) + 1]).se;
}

int main() {
```

```
    cin >> n;

    adj.assign(n, vector<int>(0));

    for (int i = 0; i < n - 1; i++) {
        int a, b;
        cin >> a >> b;
        adj[a].push_back(b);
        adj[b].push_back(a);
    }

    build();
}
```

2.10 Matching

2.10.1 Hungaro

Resolve o problema de Matching para uma matriz A[n][m], onde $n \leq m$.

A implementação minimiza os custos, para maximizar basta multiplicar os pesos por -1.

A matriz de entrada precisa ser indexada em 1 !!!

O vetor result guarda os pares do matching.

Complexidade de tempo: $O(n^2 * m)$

```
const ll INF = 1e18 + 18;

vector<pair<int, int>> result;

ll hungarian(int n, int m, vector<vector<int>> &A) {
    vector<int> u(n + 1), v(m + 1), p(m + 1), way(m + 1);
    for (int i = 1; i <= n; i++) {
        p[0] = i;
        int j0 = 0;
        vector<int> minv(m + 1, INF);
        vector<char> used(m + 1, false);
        do {
            used[j0] = true;
            ll i0 = p[j0], delta = INF, j1;
            for (int j = 1; j <= m; j++) {
                if (!used[j]) {
                    int cur = A[i0][j] - u[i0] - v[j];
                    if (cur < minv[j]) {
                        minv[j] = cur, way[j] = j0;
                    }
                    if (minv[j] < delta) {
                        delta = minv[j], j1 = j;
                    }
                }
            }
            for (int j = 0; j <= m; j++) {
```

```
                if (used[j]) {
                    u[p[j]] += delta, v[j] -= delta;
                } else {
                    minv[j] -= delta;
                }
            }
            j0 = j1;
        } while (p[j0] != 0);
        do {
            int j1 = way[j0];
            p[j0] = p[j1];
            j0 = j1;
        } while (j0);
    }
    for (int i = 1; i <= m; i++) {
        result.emplace_back(p[i], i);
    }
    return -v[0];
}
```

2.11 Shortest Paths

2.11.1 Dijkstra

Computa o menor caminho entre nós de um grafo.

Dado dois nós u e v, computa o menor caminho de u para v.

Complexidade de tempo: O((E + V) * log(E))

Dado um nó u, computa o menor caminho de u para todos os nós.

Complexidade de tempo: O((E + V) * log(E))

Computa o menor caminho de todos os nós para todos os nós

Complexidade de tempo: O(V * ((E + V) * log(E)))

```
const int MAX = 505, INF = 1e9 + 9;

vector<ii> adj[MAX];
int dist[MAX][MAX];

void dk(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            dist[i][j] = INF;
        }
    }
    for (int s = 0; s < n; s++) {
        priority_queue<ii, vector<ii>, greater<ii>> fila;
        dist[s][s] = 0;
        fila.emplace(dist[s][s], s);
        while (!fila.empty()) {
            auto [d, u] = fila.top();
            fila.pop();
            if (d != dist[s][u]) {
                continue;
            }
            for (auto [w, v] : adj[u]) {
                if (dist[s][v] > d + w) {
                    dist[s][v] = d + w;
                    fila.emplace(dist[s][v], v);
                }
            }
        }
    }
}
```

```
const int MAX = 1e5 + 5, INF = 1e9 + 9;

vector<ii> adj[MAX];
int dist[MAX];

void dk(int s) {
    priority_queue<ii, vector<ii>, greater<ii>> fila;
    fill(begin(dist), end(dist), INF);
    dist[s] = 0;
    fila.emplace(dist[s], s);
    while (!fila.empty()) {
        auto [d, u] = fila.top();
        fila.pop();
        if (d != dist[u]) {
            continue;
        }
        for (auto [w, v] : adj[u]) {
            if (dist[v] > d + w) {
                dist[v] = d + w;
                fila.emplace(dist[v], v);
            }
        }
    }
}
```

```
const int MAX = 1e5 + 5, INF = 1e9 + 9;

vector<ii> adj[MAX];
int dist[MAX];

int dk(int s, int t) {
```

```
    priority_queue<ii, vector<ii>, greater<ii>> fila;
    fill(begin(dist), end(dist), INF);
    dist[s] = 0;
    fila.emplace(dist[s], s);
    while (!fila.empty()) {
        auto [d, u] = fila.top();
        fila.pop();
        if (u == t) {
            return dist[t];
        }
        if (d != dist[u]) {
            continue;
        }
        for (auto [w, v] : adj[u]) {
            if (dist[v] > d + w) {
                dist[v] = d + w;
                fila.emplace(dist[v], v);
            }
        }
    }
    return -1;
}
```

2.11.2 SPFA

Encontra o caminho mais curto entre um vértice e todos os outros vértices de um grafo.

Detecta ciclos negativos.

Complexidade de tempo: O(|V| * |E|)

```
const int MAX = 1e4 + 4;
const ll INF = 1e18 + 18;

vector<ii> adj[MAX];
ll dist[MAX];

void spfa(int s, int n) {
    fill(dist, dist + n, INF);
    vector<int> cnt(n, 0);
    vector<bool> inq(n, false);
    queue<int> fila;
    fila.push(s);
    inq[s] = true;
    dist[s] = 0;
    while (!fila.empty()) {
        int u = fila.front();
        fila.pop();
        inq[u] = false;
        for (auto [w, v] : adj[u]) {
            ll newd = (dist[u] == -INF ? -INF : max(w + dist[u],
                -INF));
            if (newd < dist[v]) {
                dist[v] = newd;
                if (!inq[v]) {
                    fila.push(v);
                    inq[v] = true;
```

```
                    cnt[v]++;
                    if (cnt[v] > n) { // negative cycle
                        dist[v] = -INF;
                    }
                }
            }
        }
    }
}
```

2.12 Stoer–Wagner Min Cut

Algortimo de Stoer-Wagner para encontrar o corte mínimo de um grafo.

O algoritmo de Stoer-Wagner é um algoritmo para resolver o problema de corte mínimo em grafos não direcionados com pesos não negativos. A ideia essencial deste algoritmo é encolher o grafo mesclando os vértices mais intensos até que o grafo contenha apenas dois conjuntos de vértices combinados

Complexidade de tempo: $O(V^3)$

```
const int MAXN = 555, INF = 1e9 + 7;

int n, e, adj[MAXN][MAXN];
vector<int> bestCut;

int mincut() {
    int bestCost = INF;
    vector<int> v[MAXN];
    for (int i = 0; i < n; i++) {
        v[i].assign(1, i);
    }
    int w[MAXN], sel;
    bool exist[MAXN], added[MAXN];
    memset(exist, true, sizeof(exist));
    for (int phase = 0; phase < n - 1; phase++) {
        memset(added, false, sizeof(added));
        memset(w, 0, sizeof(w));
        for (int j = 0, prev; j < n - phase; j++) {
            sel = -1;
            for (int i = 0; i < n; i++) {
                if (exist[i] && !added[i] && (sel == -1 || w[i] >
                  w[sel])) {
                    sel = i;
                }
            }
            if (j == n - phase - 1) {
                if (w[sel] < bestCost) {
                    bestCost = w[sel];
                    bestCut = v[sel];
                }
                v[prev].insert(v[prev].end(), v[sel].begin(),
                    v[sel].end());
                for (int i = 0; i < n; i++) {
                    adj[prev][i] = adj[i][prev] += adj[sel][i];
                }
                exist[sel] = false;
```

```
            } else {
                added[sel] = true;
                for (int i = 0; i < n; i++) {
                    w[i] += adj[sel][i];
                }
                prev = sel;
            }
        }
    }
    return bestCost;
}
```

Capítulo 3

String

3.1 Aho Corasick

Constrói uma estrutura de dados semelhante a um trie com links adicionais e, em seguida, constrói uma máquina de estados finitos (autômato). Útil para pattern matching de um set de strings em um texto.

Complexidade de tempo: O(|S|+|T|), onde |S| é o somatório do tamanho das strings e |T| é o tamanho do texto

```
const int K = 26;

struct Vertex {
    int next[K], p = -1, link = -1, exi = -1, go[K], cont = 0;
    bool term = false;
    vector<int> idxs;
    char pch;
    Vertex(int p = -1, char ch = '$') : p(p), pch(ch) {
        fill(begin(next), end(next), -1);
        fill(begin(go), end(go), -1);
    }
};
```

```
vector<Vertex> aho(1);
void add_string(const string &s, int idx) {
    int v = 0;
    for (char ch : s) {
        int c = ch - 'a';
        if (aho[v].next[c] == -1) {
            aho[v].next[c] = aho.size();
            aho.emplace_back(v, ch);
        }
        v = aho[v].next[c];
    }
    aho[v].term = true;
```

```
    aho[v].idxs.push_back(idx);
}
int go(int u, char ch);
int get_link(int u) {
    if (aho[u].link == -1) {
        if (u == 0 || aho[u].p == 0) {
            aho[u].link = 0;
        } else {
            aho[u].link = go(get_link(aho[u].p), aho[u].pch);
        }
    }
    return aho[u].link;
}
int go(int u, char ch) {
    int c = ch - 'a';
    if (aho[u].go[c] == -1) {
        if (aho[u].next[c] != -1) {
            aho[u].go[c] = aho[u].next[c];
        } else {
            aho[u].go[c] = u == 0 ? 0 : go(get_link(u), ch);
        }
    }
    return aho[u].go[c];
}
int exi(int u) {
    if (aho[u].exi != -1) {
        return aho[u].exi;
    }
    int v = get_link(u);
    return aho[u].exi = (v == 0 || aho[v].term ? v : exi(v));
}
void process(const string &s) {
    int st = 0;
    for (char c : s) {
        st = go(st, c);
        for (int aux = st; aux; aux = exi(aux)) {
            aho[aux].cont++;
        }
    }
    for (int st = 1; st < aho_sz; st++) {
        if (!aho[st].term) {
            continue;
        }
        for (int i : aho[st].idxs) {
            // Do something here
            // idx i ocurs + aho[st].cont times
            h[i] += aho[st].cont;
        }
    }
}
```

3.2 Hashing

Hashing para testar igualdade de duas strings.

A função ***range(i, j)*** retorna o hash da substring nesse range.

Pode ser necessário usar pares de hash para evitar colisões.

* Complexidade de tempo (Construção): O(N)

* Complexidade de tempo (Consulta de range): O(1)

```
struct hashing {
    const long long LIM = 1000006;
    long long p, m;
    vector<long long> pw, hsh;
    hashing(long long _p, long long _m) : p(_p), m(_m) {
        pw.resize(LIM);
        hsh.resize(LIM);
        pw[0] = 1;
        for (int i = 1; i < LIM; i++) {
            pw[i] = (pw[i - 1] * p) % m;
        }
    }
    void set_string(string &s) {
        hsh[0] = s[0];
        for (int i = 1; i < s.size(); i++) {
            hsh[i] = (hsh[i - 1] * p + s[i]) % m;
        }
    }
    long long range(int esq, int dir) {
        long long ans = hsh[dir];
        if (esq > 0) {
            ans = (ans - (hsh[esq - 1] * pw[dir - esq + 1] % m) + m)
                % m;
        }
        return ans;
    }
};
```

3.3 Lyndon

Strings em decomposição única em subcadeias que são ordenadas lexicograficamente e não podem ser mais reduzidas.

Duval

Gera a Lyndon Factorization de uma string

* Complexidade de tempo: O(N)

Min Cyclic Shift

Gera a menor rotação circular da string original que pode ser obtida por meio de deslocamentos cíclicos dos caracteres.

* Complexidade de tempo: O(N)

```
string min_cyclic_shift(string s) {
    s += s;
    int n = s.size();
    int i = 0, ans = 0;
    while (i < n / 2) {
        ans = i;
        int j = i + 1, k = i;
        while (j < n && s[k] <= s[j]) {
            if (s[k] < s[j]) {
                k = i;
            } else {
                k++;
            }
            j++;
        }
        while (i <= k) {
            i += j - k;
        }
    }
    return s.substr(ans, n / 2);
}
```

```
vector<string> duval(string const &s) {
    int n = s.size();
    int i = 0;
    vector<string> factorization;
    while (i < n) {
        int j = i + 1, k = i;
        while (j < n && s[k] <= s[j]) {
            if (s[k] < s[j]) {
                k = i;
            } else {
                k++;
            }
            j++;
        }
        while (i <= k) {
            factorization.push_back(s.substr(i, j - k));
            i += j - k;
        }
    }
    return factorization;
}
```

3.4 Manacher

Encontra todos os palindromos de uma string.

Dada uma string s de tamanho n, encontra todos os pares (i,j) tal que a substring $s[i...j]$ seja um palindromo.

* Complexidade de tempo: O(N)

```
struct manacher {
    long long n, count;
    vector<int> d1, d2;
    long long solve(string &s) {
        n = s.size(), count = 0;
        solve_odd(s);
        solve_even(s);
        return count;
    }
    void solve_odd(string &s) {
        d1.resize(n);
        for (int i = 0, l = 0, r = -1; i < n; i++) {
            int k = (i > r) ? 1 : min(d1[l + r - i], r - i + 1);
            while (0 <= i - k && i + k < n && s[i - k] == s[i + k]) {
                k++;
            }
            count += d1[i] = k--;
            if (i + k > r) {
                l = i - k;
                r = i + k;
            }
        }
    }
    void solve_even(string &s) {
        d2.resize(n);
        for (int i = 0, l = 0, r = -1; i < n; i++) {
            int k = (i > r) ? 0 : min(d2[l + r - i + 1], r - i + 1);
            while (0 <= i - k - 1 && i + k < n && s[i - k - 1] == s[i
                + k]) {
                k++;
            }
            count += d2[i] = k--;
            if (i + k > r) {
                l = i - k - 1;
                r = i + k;
            }
        }
    }
} mana;
```

3.5 Patricia Tree

Estrutura de dados que armazena strings e permite consultas por prefixo.

Implementação PB-DS, extremamente curta e confusa:

- Criar: `patricia_tree pat;`
- Inserir: `pat.insert("sei la");`
- Remover: `pat.erase("sei la");`
- Verificar existência: `pat.find("sei la") != pat.end();`

- Pegar palavras que começam com um prefixo: **auto** match = pat.prefix_range("sei");
- Percorrer *match* : **for**(**auto** it = match.first; it != match.second; ++it);
- Pegar menor elemento lexicográfico *maior ou igual* ao prefixo: *pat.lower_bound("sei");
- Pegar menor elemento lexicográfico *maior* ao prefixo: *pat.upper_bound("sei");

TODAS AS OPERAÇÕES EM O(|S|)

NÃO ACEITA ELEMENTOS REPETIDOS

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

using namespace __gnu_pbds;
typedef trie<string, null_type, trie_string_access_traits<>,
    pat_trie_tag,
              trie_prefix_search_node_update>
     patricia_tree;
```

3.6 Prefix Function

Para cada prefixo k de uma dada string s, calcula o maior prefixo que tambem é sufixo de k.

Seja n o tamanho do texto e m o tamanho do padrão.

KMP

String matching em O(n + m).

Autômato de KMP

String matching em O(n) com O(m) de pré-processamento.

Prefix Count

Dada uma string s, calcula quantas vezes cada prefixo de s aparece em s com complexidade de tempo de O(n).

```
vector<int> pi(string &s) {
    vector<int> p(s.size());
    for (int i = 1, j = 0; i < s.size(); i++) {
        while (j > 0 && s[i] != s[j]) {
            j = p[j - 1];
        }
        if (s[i] == s[j]) {
            j++;
        }
        p[i] = j;
    }
    return p;
}
```

```
vector<int> pi(string &s) {
    vector<int> p(s.size());
    for (int i = 1, j = 0; i < s.size(); i++) {
        while (j > 0 && s[i] != s[j]) {
            j = p[j - 1];
        }
        if (s[i] == s[j]) {
            j++;
        }
        p[i] = j;
    }
    return p;
}

vector<int> kmp(string &s, string t) {
    t += '$';
    vector<int> p = pi(t), match;
    for (int i = 0, j = 0; i < s.size(); i++) {
        while (j > 0 && s[i] != t[j]) {
            j = p[j - 1];
        }
        if (s[i] == t[j]) {
            j++;
        }
        if (j == t.size() - 1) {
            match.push_back(i - j + 1);
        }
    }
    return match;
}
```

```
vector<int> pi(string s) {
    vector<int> p(s.size());
    for (int i = 1, j = 0; i < s.size(); i++) {
        while (j > 0 && s[i] != s[j]) {
            j = p[j - 1];
        }
        if (s[i] == s[j]) {
            j++;
        }
        p[i] = j;
```

```
    }
    return p;
}

vector<int> prefixCount(string s) {
    vector<int> p = pi(s + '#');
    int n = s.size();
    vector<int> cnt(n + 1, 0);
    for (int i = 0; i < n; i++) {
        cnt[p[i]]++;
    }
    for (int i = n - 1; i > 0; i--) {
        cnt[p[i - 1]] += cnt[i];
    }
    for (int i = 0; i <= n; i++) {
        cnt[i]++;
    }
    return cnt;
}
```

```
struct AutKMP {
    vector<vector<int>> nxt;

    vector<int> pi(string &s) {
        vector<int> p(s.size());
        for (int i = 1, j = 0; i < s.size(); i++) {
            while (j > 0 && s[i] != s[j]) {
                j = p[j - 1];
            }
            if (s[i] == s[j]) {
                j++;
            }
            p[i] = j;
        }
        return p;
    }

    void setString(string s) {
        s += '#';
        nxt.assign(s.size(), vector<int>(26));
        vector<int> p = pi(s);
        for (int c = 0; c < 26; c++) {
            nxt[0][c] = ('a' + c == s[0]);
        }
        for (int i = 1; i < s.size(); i++) {
            for (int c = 0; c < 26; c++) {
                nxt[i][c] = ('a' + c == s[i]) ? i + 1 : nxt[p[i -
                    1]][c];
            }
        }
    }

    vector<int> kmp(string &s, string &t) {
        vector<int> match;
        for (int i = 0, j = 0; i < s.size(); i++) {
            j = nxt[j][s[i] - 'a'];
            if (j == t.size()) {
                match.push_back(i - j + 1);
            }
        }
        return match;
    }
} aut;
```

3.7 Suffix Array

Estrutura que conterá inteiros que representam os índices iniciais de todos os sufixos ordenados de uma determinada string.

Tambem Constroi a tabela LCP(Longest common prefix).

* Complexidade de tempo (Pré-Processamento): O(|S|*log(|S|))

* Complexidade de tempo (Contar ocorrencias de S em T): O(|S|*log(|T|))

```
pair<int, int> busca(string &t, int i, pair<int, int> &range) {
    int esq = range.first, dir = range.second, L = -1, R = -1;
    while (esq <= dir) {
        int mid = (esq + dir) / 2;
        if (s[sa[mid] + i] == t[i]) {
            L = mid;
        }
        if (s[sa[mid] + i] < t[i]) {
            esq = mid + 1;
        } else {
            dir = mid - 1;
        }
    }
    esq = range.first, dir = range.second;
    while (esq <= dir) {
        int mid = (esq + dir) / 2;
        if (s[sa[mid] + i] == t[i]) {
            R = mid;
        }
        if (s[sa[mid] + i] <= t[i]) {
            esq = mid + 1;
        } else {
            dir = mid - 1;
        }
    }
    return {L, R};
}
// count ocurences of s on t
int busca_string(string &t) {
    pair<int, int> range = {0, n - 1};
    for (int i = 0; i < t.size(); i++) {
        range = busca(t, i, range);
        if (range.first == -1) {
            return 0;
        }
    }
    return range.second - range.first + 1;
}
```

```
const int MAX_N = 5e5 + 5;

struct suffix_array {
    string s;
    int n, sum, r, ra[MAX_N], sa[MAX_N], auxra[MAX_N], auxsa[MAX_N],
        c[MAX_N], lcp[MAX_N];
    void counting_sort(int k) {
        memset(c, 0, sizeof(c));
        for (int i = 0; i < n; i++) {
            c[(i + k < n) ? ra[i + k] : 0]++;
        }
        for (int i = sum = 0; i < max(256, n); i++) {
```

```
            sum += c[i], c[i] = sum - c[i];
        }
        for (int i = 0; i < n; i++) {
            auxsa[c[sa[i] + k < n ? ra[sa[i] + k] : 0]++] = sa[i];
        }
        for (int i = 0; i < n; i++) {
            sa[i] = auxsa[i];
        }
    }
    void build_sa() {
        for (int k = 1; k < n; k <<= 1) {
            counting_sort(k);
            counting_sort(0);
            auxra[sa[0]] = r = 0;
            for (int i = 1; i < n; i++) {
                auxra[sa[i]] =
                    (ra[sa[i]] == ra[sa[i - 1]] && ra[sa[i] + k] ==
                        ra[sa[i - 1] + k])
                        ? r
                        : ++r;
            }
            for (int i = 0; i < n; i++) {
                ra[i] = auxra[i];
            }
            if (ra[sa[n - 1]] == n - 1) {
                break;
            }
        }
    }
    void build_lcp() {
        for (int i = 0, k = 0; i < n - 1; i++) {
            int j = sa[ra[i] - 1];
            while (s[i + k] == s[j + k]) {
                k++;
            }
            lcp[ra[i]] = k;
            if (k) {
                k--;
            }
        }
    }
    void set_string(string _s) {
        s = _s + '$';
        n = s.size();
        for (int i = 0; i < n; i++) {
            ra[i] = s[i], sa[i] = i;
        }
        build_sa();
        build_lcp();
        // for (int i = 0; i < n; i++)
        // printf("%2d: %s\n", sa[i], s.c_str() +
        // sa[i]);
    }
    int operator[](int i) {
        return sa[i];
    }
} sa;
```

3.8 Trie

Estrutura que guarda informações indexadas por palavra.

Útil encontrar todos os prefixos inseridos anteriormente de uma palavra específica.

* Complexidade de tempo (Update): O(|S|)

* Complexidade de tempo (Consulta de palavra): O(|S|)

```
struct trie {
    map<char, int> trie[100005];
    int value[100005];
    int n_nodes = 0;
    void insert(string &s, int v) {
        int id = 0;
        for (char c : s) {
            if (!trie[id].count(c)) {
                trie[id][c] = ++n_nodes;
            }
            id = trie[id][c];
        }
        value[id] = v;
    }
    int get_value(string &s) {
        int id = 0;
        for (char c : s) {
            if (!trie[id].count(c)) {
                return -1;
            }
            id = trie[id][c];
        }
        return value[id];
    }
};
```

Capítulo 4

Paradigmas

4.1 All Submasks

Percorre todas as submáscaras de uma máscara.

* Complexidade de tempo: $O(3^N)$

```
int mask;
for (int sub = mask; sub; sub = (sub - 1) & mask) { }
```

4.2 Busca Binaria Paralela

Faz a busca binária para múltiplas consultas quando a busca binária é muito pesada.

- Complexidade de tempo: O((N+Q)log(N) * O(F)), onde N é o tamanho do espaço de busca, Q é o número de consultas e O(F), o custo de avaliação da função.

```
namespace parallel_binary_search {
    typedef tuple<int, int, long long, long long> query; //{value,
        id, l, r}
    vector<query> queries[1123456];                  // pode ser
        um mapa se
                                                     // for muito
                                                        esparso
    long long ans[1123456];                          // definir
        pro tamanho
                                                     // das
                                                        queries
    long long l, r, mid;
    int id = 0;
    void set_lim_search(long long n) {
        l = 0;
        r = n;
        mid = (l + r) / 2;
    }

    void add_query(long long v) {
        queries[mid].push_back({v, id++, l, r});
    }

    void advance_search(long long v) {
        // advance search
    }

    bool satisfies(long long mid, int v, long long l, long long r) {
        // implement the evaluation
    }

    bool get_ans() {
        // implement the get ans
    }

    void parallel_binary_search(long long l, long long r) {

        bool go = 1;
        while (go) {
            go = 0;
            int i = 0; // outra logica se for usar
                    // um mapa
            for (auto &vec : queries) {
                advance_search(i++);
                for (auto q : vec) {
                    auto [v, id, l, r] = q;
                    if (l > r) {
                        continue;
                    }
                    go = 1;
                    // return while satisfies
                    if (satisfies(i, v, l, r)) {
                        ans[i] = get_ans();
                        long long mid = (i + l) / 2;
                        queries[mid] = query(v, id, l, i - 1);
                    } else {
                        long long mid = (i + r) / 2;
                        queries[mid] = query(v, id, i + 1, r);
                    }
                }
                vec.clear();
            }
        }
    }

} // namespace name
```

4.3 Busca Ternaria

Encontra um ponto ótimo em uma função que pode ser separada em duas funções estritamente monotônicas (e.g. parábolas).

- Complexidade de tempo: O(log(N) * O(eval)). Onde N é o tamanho do espaço de busca e O(eval) o custo de avaliação da função.

Busca Ternária em Espaço Discreto

Encontra um ponto ótimo em uma função que pode ser separada em duas funções estritamente monotônicas (e.g. parábolas).

Versão para espaços discretos.

- Complexidade de tempo: O(log(N) * O(eval)). Onde N é o tamanho do espaço de busca e O(eval) o custo de avaliação da função.

```
double eval(double mid) {
    // implement the evaluation
}

double ternary_search(double l, double r) {
    int k = 100;
    while (k--) {
        double step = (l + r) / 3;
        double mid_1 = l + step;
        double mid_2 = r - step;

        // minimizing. To maximize use >= to
        // compare
        if (eval(mid_1) <= eval(mid_2)) {
            r = mid_2;
        } else {
            l = mid_1;
        }
    }
    return l;
}
```

```
long long eval(long long mid) {
    // implement the evaluation
}

long long discrete_ternary_search(long long l, long long r) {
    long long ans = -1;
    r--; // to not space r
```

```
    while (l <= r) {
        long long mid = (l + r) / 2;

        // minimizing. To maximize use >= to
        // compare
        if (eval(mid) <= eval(mid + 1)) {
            ans = mid;
            r = mid - 1;
        } else {
            l = mid + 1;
        }
    }
    return ans;
}
```

4.4 Convex Hull Trick

Otimização de DP onde se mantém as retas que formam um Convex Hull em uma estrutura que permite consultar qual o melhor valor para um determinado x.

Só funciona quando as retas são monotônicas. Caso não forem, usar LiChao Tree para guardar as retas

Complexidade de tempo:

- Inserir reta: O(1) amortizado
- Consultar x: O(log(N))
- Consultar x quando x tem crescimento monotônico: O(1)

```
const ll INF = 1e18 + 18;
bool op(ll a, ll b) {
    return a >= b; // either >= or <=
}
struct line {
    ll a, b;
    ll get(ll x) {
        return a * x + b;
    }
    ll intersect(line l) {
        return (l.b - b + a - l.a) / (a - l.a); // rounds up for
            integer
                                                 // only
    }
};
deque<pair<line, ll>> fila;
```

```cpp
void add_line(ll a, ll b) {
    line nova = {a, b};
    if (!fila.empty() && fila.back().first.a == a &&
        fila.back().first.b == b) {
        return;
    }
    while (!fila.empty() && op(fila.back().second,
        nova.intersect(fila.back().first))) {
        fila.pop_back();
    }
    ll x = fila.empty() ? -INF : nova.intersect(fila.back().first);
    fila.emplace_back(nova, x);
}
ll get_binary_search(ll x) {
    int esq = 0, dir = fila.size() - 1, r = -1;
    while (esq <= dir) {
        int mid = (esq + dir) / 2;
        if (op(x, fila[mid].second)) {
            esq = mid + 1;
            r = mid;
        } else {
            dir = mid - 1;
        }
    }
    return fila[r].first.get(x);
}
// O(1), use only when QUERIES are monotonic!
ll get(ll x) {
    while (fila.size() >= 2 && op(x, fila[1].second)) {
        fila.pop_front();
    }
    return fila.front().first.get(x);
}
```

4.5 DP de Permutacao

Otimização do problema do Caixeiro Viajante

* Complexidade de tempo: $O(n^2 * 2^n)$

Para rodar a função basta setar a matriz de adjacência 'dist' e chamar solve(0,0,n).

```cpp
const int lim = 17;          // setar para o maximo de itens
long double dist[lim][lim];  // eh preciso dar as
                             // distancias de n para n
long double dp[lim][1 << lim];

int limMask = (1 << lim) - 1; // 2**(maximo de itens) - 1
long double solve(int atual, int mask, int n) {
    if (dp[atual][mask] != 0) {
        return dp[atual][mask];
    }
    if (mask == (1 << n) - 1) {
        return dp[atual][mask] = 0; // o que fazer quando
                                    // chega no final
    }
```

```
    long double res = 1e13; // pode ser maior se precisar
    for (int i = 0; i < n; i++) {
        if (!(mask & (1 << i))) {
            long double aux = solve(i, mask | (1 << i), n);
            if (mask) {
                aux += dist[atual][i];
            }
            res = min(res, aux);
        }
    }
    return dp[atual][mask] = res;
}
```

4.6 Divide and Conquer

Otimização para DP de prefixo quando se pretende separar o vetor em K subgrupos.

É preciso fazer a função query(i, j) que computa o custo do subgrupo

$$i, j$$

.

* Complexidade de tempo: O(n * k * log(n) * O(query))

Divide and Conquer com Query on demand

<!– *Read in [English](README.en.md)* –>

Usado para evitar queries pesadas ou o custo de pré-processamento.

É preciso fazer as funções da estrutura **janela**, eles adicionam e removem itens um a um como uma janela flutuante.

* Complexidade de tempo: O(n * k * log(n) * O(update da janela))

```
namespace DC {
    vi dp_before, dp_cur;
    void compute(int l, int r, int optl, int optr) {
        if (l > r) {
            return;
        }
        int mid = (l + r) >> 1;
        pair<ll, int> best = {0, -1}; // {INF, -1} se quiser minimizar
        for (int i = optl; i <= min(mid, optr); i++) {
            best = max(best,
```

```
                    {(i ? dp_before[i - 1] : 0) + query(i, mid),
                     i}); // min() se quiser minimizar
        }
        dp_cur[mid] = best.first;
        int opt = best.second;
        compute(l, mid - 1, optl, opt);
        compute(mid + 1, r, opt, optr);
    }

    ll solve(int n, int k) {
        dp_before.assign(n + 5, 0);
        dp_cur.assign(n + 5, 0);
        for (int i = 0; i < n; i++) {
            dp_before[i] = query(0, i);
        }
        for (int i = 1; i < k; i++) {
            compute(0, n - 1, 0, n - 1);
            dp_before = dp_cur;
        }
        return dp_before[n - 1];
    }
};
```

```
namespace DC {
    struct range { // eh preciso definir a forma
                   // de calcular o range
        vi freq;
        ll sum = 0;
        int l = 0, r = -1;
        void back_l(int v) { // Mover o 'l' do range
                             // para a esquerda
            sum += freq[v];
            freq[v]++;
            l--;
        }
        void advance_r(int v) { // Mover o 'r' do range
                                // para a direita
            sum += freq[v];
            freq[v]++;
            r++;
        }
        void advance_l(int v) { // Mover o 'l' do range
                                // para a direita
            freq[v]--;
            sum -= freq[v];
            l++;
        }
        void back_r(int v) { // Mover o 'r' do range
                             // para a esquerda
            freq[v]--;
            sum -= freq[v];
            r--;
        }
        void clear(int n) { // Limpar range
            l = 0;
            r = -1;
            sum = 0;
            freq.assign(n + 5, 0);
        }
    } s;

    vi dp_before, dp_cur;
    void compute(int l, int r, int optl, int optr) {
        if (l > r) {
            return;
        }
        int mid = (l + r) >> 1;
        pair<ll, int> best = {0, -1}; // {INF, -1} se quiser minimizar

        while (s.l < optl) {
            s.advance_l(v[s.l]);
```

```
        }
        while (s.l > optl) {
            s.back_l(v[s.l - 1]);
        }
        while (s.r < mid) {
            s.advance_r(v[s.r + 1]);
        }
        while (s.r > mid) {
            s.back_r(v[s.r]);
        }

        vi removed;
        for (int i = optl; i <= min(mid, optr); i++) {
            best =
                min(best,
                    {(i ? dp_before[i - 1] : 0) + s.sum, i}); //
                        min() se quiser minimizar
            removed.push_back(v[s.l]);
            s.advance_l(v[s.l]);
        }
        for (int rem : removed) {
            s.back_l(v[s.l - 1]);
        }
        dp_cur[mid] = best.first;
        int opt = best.second;
        compute(l, mid - 1, optl, opt);
        compute(mid + 1, r, opt, optr);
    }

    ll solve(int n, int k) {
        dp_before.assign(n, 0);
        dp_cur.assign(n, 0);
        s.clear(n);
        for (int i = 0; i < n; i++) {
            s.advance_r(v[i]);
            dp_before[i] = s.sum;
        }
        for (int i = 1; i < k; i++) {
            s.clear(n);
            compute(0, n - 1, 0, n - 1);
            dp_before = dp_cur;
        }
        return dp_before[n - 1];
    }
};
```

4.7 Exponenciação de Matriz

Otimização para DP de prefixo quando o valor atual está em função dos últimos K valores já calculados.

* Complexidade de tempo: $O(log(n) * k^3)$

É preciso mapear a DP para uma exponenciação de matriz.

DP:

$$dp[n] = \sum_{i=1}^{k} c[i] \cdot dp[n-i]$$

Mapeamento:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & ... & 0 \\ 0 & 0 & 1 & 0 & ... & 0 \\ 0 & 0 & 0 & 1 & ... & 0 \\ ... & ... & ... & ... & ... & ... \\ c[k] & c[k-1] & c[k-2] & ... & c[1] & 0 \end{pmatrix}^n \times \begin{pmatrix} dp[0] \\ dp[1] \\ dp[2] \\ ... \\ dp[k-1] \end{pmatrix}$$

- –

Exemplo de DP:

$$dp[i] = dp[i-1] + 2 \cdot i^2 + 3 \cdot i + 5$$

Nesses casos é preciso fazer uma linha para manter cada constante e potência do índice.

Mapeamento:

$$\begin{pmatrix} 1 & 5 & 3 & 2 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 2 & 1 \end{pmatrix}^n \times \begin{pmatrix} dp[0] \\ 1 \\ 1 \\ 1 \end{pmatrix} \begin{array}{c} \text{mantém } dp[i] \\ \text{mantém } 1 \\ \text{mantém } i \\ \text{mantém } i^2 \end{array}$$

Exemplo de DP:

$$dp[n] = c \times \prod_{i=1}^{k} dp[n-i]$$

Nesses casos é preciso trabalhar com o logaritmo e temos o caso padrão:

$$\log(dp[n]) = log(c) + \sum_{i=1}^{k} log(dp[n-i])$$

Se a resposta precisar ser inteira, deve-se fatorar a constante e os valores inicias e então fazer uma exponenciação para cada fator primo. Depois é só juntar a resposta no final.

```
ll dp[100];
mat T;

#define MOD 1000000007

mat mult(mat a, mat b) {
    mat res(a.size(), vi(b[0].size()));
    for (int i = 0; i < a.size(); i++) {
        for (int j = 0; j < b[0].size(); j++) {
            for (int k = 0; k < b.size(); k++) {
                res[i][j] += a[i][k] * b[k][j] % MOD;
                res[i][j] %= MOD;
            }
        }
    }
    return res;
}

mat exp_mod(mat b, ll exp) {
    mat res(b.size(), vi(b.size()));
    for (int i = 0; i < b.size(); i++) {
        res[i][i] = 1;
    }

    while (exp) {
        if (exp & 1) {
            res = mult(res, b);
        }
        b = mult(b, b);
        exp /= 2;
    }
    return res;
}

// MUDA MUITO DE ACORDO COM O PROBLEMA
// LEIA COMO FAZER O MAPEAMENTO NO README
ll solve(ll exp, ll dim) {
    if (exp < dim) {
        return dp[exp];
    }

    T.assign(dim, vi(dim));
    // TO DO: Preencher a Matriz que vai ser
    // exponenciada T[0][1] = 1; T[1][0] = 1;
    // T[1][1] = 1;
```

```
    mat prod = exp_mod(T, exp);

    mat vec;
    vec.assign(dim, vi(1));
    for (int i = 0; i < dim; i++) {
        vec[i][0] = dp[i]; // Valores iniciais
    }

    mat ans = mult(prod, vec);
    return ans[0][0];
}
```

4.8 Mo

Resolve Queries Complicadas Offline de forma rápida.

É preciso manter uma estrutura que adicione e remova elementos nas extremeidades de um range (tipo janela).

- Complexidade de tempo (Query offline): O(N * sqrt(N))

Mo com Update

Resolve Queries Complicadas Offline de forma rápida.

Permite que existam **UPDATES PONTUAIS!**

É preciso manter uma estrutura que adicione e remova elementos nas extremidades de um range (tipo janela).

- Complexidade de tempo: $O(Q * N^(2/3))$

```
typedef pair<int, int> ii;
int block_sz; // Better if 'const';

namespace mo {
    struct query {
        int l, r, idx;
        bool operator<(query q) const {
            int _l = l / block_sz;
            int _ql = q.l / block_sz;
            return ii(_l, (_l & 1 ? -r : r)) < ii(_ql, (_ql & 1 ?
```

```
            -q.r : q.r));
    }
};
vector<query> queries;

void build(int n) {
    block_sz = (int)sqrt(n);
    // TODO: initialize data structure
}
inline void add_query(int l, int r) {
    queries.push_back({l, r, (int)queries.size()});
}
inline void remove(int idx) {
    // TODO: remove value at idx from data
    // structure
}
inline void add(int idx) {
    // TODO: add value at idx from data
    // structure
}
inline int get_answer() {
    // TODO: extract the current answer of the
    // data structure
    return 0;
}

vector<int> run() {
    vector<int> answers(queries.size());
    sort(queries.begin(), queries.end());
    int L = 0;
    int R = -1;
    for (query q : queries) {
        while (L > q.l) {
            add(--L);
        }
        while (R < q.r) {
            add(++R);
        }
        while (L < q.l) {
            remove(L++);
        }
        while (R > q.r) {
            remove(R--);
        }
        answers[q.idx] = get_answer();
    }
    return answers;
}

};
```

```
typedef pair<int, int> ii;
typedef tuple<int, int, int> iii;
int block_sz; // Better if 'const';
vector<int> vec;
namespace mo {
    struct query {
        int l, r, t, idx;
        bool operator<(query q) const {
            int _l = l / block_sz;
            int _r = r / block_sz;
            int _ql = q.l / block_sz;
            int _qr = q.r / block_sz;
            return iii(_l, (_l & 1 ? -_r : _r), (_r & 1 ? t : -t)) <
                   iii(_ql, (_ql & 1 ? -_qr : _qr), (_qr & 1 ? q.t :
                       -q.t));
        }
    };
    vector<query> queries;
    vector<ii> updates;
```

```cpp
    void build(int n) {
        block_sz = pow(1.4142 * n, 2.0 / 3);
        // TODO: initialize data structure
    }
    inline void add_query(int l, int r) {
        queries.push_back({l, r, (int)updates.size(),
            (int)queries.size()});
    }
    inline void add_update(int x, int v) {
        updates.push_back({x, v});
    }
    inline void remove(int idx) {
        // TODO: remove value at idx from data
        // structure
    }
    inline void add(int idx) {
        // TODO: add value at idx from data
        // structure
    }
    inline void update(int l, int r, int t) {
        auto &[x, v] = updates[t];
        if (l <= x && x <= r) {
            remove(x);
        }
        swap(vec[x], v);
        if (l <= x && x <= r) {
            add(x);
        }
    }
    inline int get_answer() {
        // TODO: extract the current answer from
        // the data structure
        return 0;
    }

    vector<int> run() {
        vector<int> answers(queries.size());
        sort(queries.begin(), queries.end());
        int L = 0;
        int R = -1;
        int T = 0;
        for (query q : queries) {
            while (T < q.t) {
                update(L, R, T++);
            }
            while (T > q.t) {
                update(L, R, --T);
            }
            while (L > q.l) {
                add(--L);
            }
            while (R < q.r) {
                add(++R);
            }
            while (L < q.l) {
                remove(L++);
            }
            while (R > q.r) {
                remove(R--);
            }
            answers[q.idx] = get_answer();
        }
        return answers;
    }
};
```

Capítulo 5

Matemática

5.1 Eliminação Gaussiana

5.1.1 Gauss

Método de eliminação gaussiana para resolução de sistemas lineares com coeficientes reais.

- Complexidade de tempo: $O(n^3)$

```
const double EPS = 1e-9;
const int INF = 2; // it doesn't actually have to
                   // be infinity or a big number

int gauss(vector<vector<double>> a, vector<double> &ans) {
    int n = (int)a.size();
    int m = (int)a[0].size() - 1;

    vector<int> where(m, -1);
    for (int col = 0, row = 0; col < m && row < n; ++col) {
        int sel = row;
        for (int i = row; i < n; ++i) {
            if (abs(a[i][col]) > abs(a[sel][col])) {
                sel = i;
            }
        }
        if (abs(a[sel][col]) < EPS) {
            continue;
```

```
        }
        for (int i = col; i <= m; ++i) {
            swap(a[sel][i], a[row][i]);
        }
        where[col] = row;

        for (int i = 0; i < n; ++i) {
            if (i != row) {
                double c = a[i][col] / a[row][col];
                for (int j = col; j <= m; ++j) {
                    a[i][j] -= a[row][j] * c;
                }
            }
        }
        ++row;
    }

    ans.assign(m, 0);
    for (int i = 0; i < m; ++i) {
        if (where[i] != -1) {
            ans[i] = a[where[i]][m] / a[where[i]][i];
        }
    }
    for (int i = 0; i < n; ++i) {
        double sum = 0;
        for (int j = 0; j < m; ++j) {
            sum += ans[j] * a[i][j];
        }
        if (abs(sum - a[i][m]) > EPS) {
            return 0;
        }
    }

    for (int i = 0; i < m; ++i) {
        if (where[i] == -1) {
            return INF;
        }
    }
    return 1;
}
```

5.1.2 Gauss Mod 2

Método de eliminação gaussiana para resolução de sistemas lineares com coeficientes em $\mathbb{Z}_2$ (inteiros módulo 2).

- Complexidade de tempo: $O(n^3/32)$

```
const int N = 105;
const int INF = 2; // tanto faz

// n -> numero de equacoes, m -> numero de
// variaveis a[i][j] para j em [0, m - 1] ->
// coeficiente da variavel j na iesima equacao
// a[i][j] para j == m -> resultado da equacao da
// iesima linha ans -> bitset vazio, que retornara
```

```
// a solucao do sistema (caso exista)

int gauss(vector<bitset<N>> a, int n, int m, bitset<N> &ans) {
    vector<int> where(m, -1);

    for (int col = 0, row = 0; col < m && row < n; col++) {
        for (int i = row; i < n; i++) {
            if (a[i][col]) {
                swap(a[i], a[row]);
                break;
            }
        }
        if (!a[row][col]) {
            continue;
        }
        where[col] = row;

        for (int i = 0; i < n; i++) {
            if (i != row && a[i][col]) {
                a[i] ^= a[row];
            }
        }
        row++;
    }

    for (int i = 0; i < m; i++) {
        if (where[i] != -1) {
            ans[i] = a[where[i]][m] / a[where[i]][i];
        }
    }
    for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = 0; j < m; j++) {
            sum += ans[j] * a[i][j];
        }
        if (abs(sum - a[i][m]) > 0) {
            return 0; // Sem solucao
        }
    }

    for (int i = 0; i < m; i++) {
        if (where[i] == -1) {
            return INF; // Infinitas solucoes
        }
    }
    return 1; // Unica solucao (retornada no
              // bitset ans)
}
```

5.2 Exponenciação Modular Rápida

Computa $(base^{exp})\%mod$.

- Complexidade de tempo: $O(log(exp))$.
- Complexidade de espaço: $O(1)$

```
ll exp_mod(ll base, ll exp) {
    ll b = base, res = 1;
    while (exp) {
        if (exp & 1) {
            res = (res * b) % MOD;
        }
        b = (b * b) % MOD;
        exp /= 2;
    }
    return res;
}
```

5.3 FFT

Algoritmo que computa a transformada rápida de fourier para convolução de polinômios.

Computa convolução (multiplicação) de polinômios.

- Complexidade de tempo (caso médio): O(N * log(N))
- Complexidade de tempo (considerando alto overhead): $O(n * log^2(n) * log(log(n)))$

Garante que não haja erro de precisão para polinômios com grau até $3 * 10^5$ e constantes até 10^6.

```
typedef complex<double> cd;
typedef vector<cd> poly;
const double PI = acos(-1);

void fft(poly &a, bool invert = 0) {
    int n = a.size(), log_n = 0;
    while ((1 << log_n) < n) {
        log_n++;
    }

    for (int i = 1, j = 0; i < n; ++i) {
        int bit = n >> 1;
        for (; j >= bit; bit >>= 1) {
            j -= bit;
        }
        j += bit;
        if (i < j) {
            swap(a[i], a[j]);
        }
    }

    double angle = 2 * PI / n * (invert ? -1 : 1);
    poly root(n / 2);
    for (int i = 0; i < n / 2; ++i) {
        root[i] = cd(cos(angle * i), sin(angle * i));
    }

    for (long long len = 2; len <= n; len <<= 1) {
```

```
        long long step = n / len;
        long long aux = len / 2;
        for (long long i = 0; i < n; i += len) {
            for (int j = 0; j < aux; ++j) {
                cd u = a[i + j], v = a[i + j + aux] * root[step * j];
                a[i + j] = u + v;
                a[i + j + aux] = u - v;
            }
        }
    }
    if (invert) {
        for (int i = 0; i < n; ++i) {
            a[i] /= n;
        }
    }
}

vector<long long> convolution(vector<long long> &a, vector<long long>
    &b) {
    int n = 1, len = a.size() + b.size();
    while (n < len) {
        n <<= 1;
    }
    a.resize(n);
    b.resize(n);
    poly fft_a(a.begin(), a.end());
    fft(fft_a);
    poly fft_b(b.begin(), b.end());
    fft(fft_b);

    poly c(n);
    for (int i = 0; i < n; ++i) {
        c[i] = fft_a[i] * fft_b[i];
    }
    fft(c, 1);

    vector<long long> res(n);
    for (int i = 0; i < n; ++i) {
        res[i] = round(c[i].real()); // res = c[i].real();
                                     // se for vector de
                                     // double
    }
    // while(size(res) > 1 && res.back() == 0)
    // res.pop_back(); // apenas para quando os
    // zeros direita nao importarem
    return res;
}
```

5.4 Fatoração

Algortimos para fatorar um número.

Fatoração Simples

Fatora um número N.

- Complexidade de tempo: $O(\sqrt{n})$

Crivo Linear

Pré-computa todos os fatores primos até MAX.

Utilizado para fatorar um número N menor que MAX.

- Complexidade de tempo: Pré-processamento O(MAX)
- Complexidade de tempo: Fatoraração O(quantidade de fatores de N)
- Complexidade de espaço: O(MAX)

Fatoração Rápida

Utiliza Pollar-Rho e Miller-Rabin (ver em Primos) para fatorar um número N.

- Complexidade de tempo: $O(N^{1/4} \cdot log(N))$

Pollard-Rho

Descobre um divisor de um número N.

- Complexidade de tempo: $O(N^{1/4} \cdot log(N))$
- Complexidade de espaço: $O(N^{1/2})$

```
vector<int> factorize(int n) {
    vector<int> factors;
    for (long long d = 2; d * d <= n; d++) {
        while (n % d == 0) {
            factors.push_back(d);
            n /= d;
        }
    }
    if (n != 1) {
        factors.push_back(n);
    }
    return factors;
}
```

```
namespace sieve {
    const int MAX = 1e4;
    int lp[MAX + 1], factor[MAX + 1];
    vector<int> pr;
    void build() {
        for (int i = 2; i <= MAX; ++i) {
            if (lp[i] == 0) {
                lp[i] = i;
                pr.push_back(i);
            }
            for (int j = 0; i * pr[j] <= MAX; ++j) {
                lp[i * pr[j]] = pr[j];
                factor[i * pr[j]] = i;
                if (pr[j] == lp[i]) {
                    break;
                }
            }
        }
    }
    vector<int> factorize(int x) {
        if (x < 2) {
            return {};
        }
        vector<int> v;
        for (int lpx = lp[x]; x >= lpx; x = factor[x]) {
            v.emplace_back(lp[x]);
        }
        return v;
    }
}
```

```
long long mod_mul(long long a, long long b, long long m) {
    return (__int128)a * b % m;
}

long long pollard_rho(long long n) {
    auto f = [n](long long x) {
        return mod_mul(x, x, n) + 1;
    };
    long long x = 0, y = 0, t = 30, prd = 2, i = 1, q;
    while (t++ % 40 || __gcd(prd, n) == 1) {
        if (x == y) {
            x = ++i, y = f(x);
        }
        if ((q = mod_mul(prd, max(x, y) - min(x, y), n))) {
            prd = q;
        }
        x = f(x), y = f(f(y));
    }
    return __gcd(prd, n);
}
```

```
// usa miller_rabin.cpp!! olhar em
// matematica/primos usa pollar_rho.cpp!! olhar em
// matematica/fatoracao

vector<long long> factorize(long long n) {
    if (n == 1) {
```

```
        return {};
    }
    if (miller_rabin(n)) {
        return {n};
    }
    long long x = pollard_rho(n);
    auto l = factorize(x), r = factorize(n / x);
    l.insert(l.end(), all(r));
    return l;
}
```

5.5 GCD

Algoritmo Euclides para computar o Máximo Divisor Comum (MDC em português; GCD em inglês), e variações.

Read in [English](README.en.md)

Algoritmo de Euclides

Computa o Máximo Divisor Comum (MDC em português; GCD em inglês).

- Complexidade de tempo: O(log(n))

Mais demorado que usar a função do compilador C++ __gcd(a,b).

Algoritmo de Euclides Estendido

Algoritmo extendido de euclides que computa o Máximo Divisor Comum e os valores x e y tal que a * x + b * y = gcd(a, b).

- Complexidade de tempo: O(log(n))

```
long long gcd(long long a, long long b) {
    return (b == 0) ? a : gcd(b, a % b);
}
```

```cpp
int extended_gcd(int a, int b, int &x, int &y) {
    x = 1, y = 0;
    int x1 = 0, y1 = 1;
    while (b) {
        int q = a / b;
        tie(x, x1) = make_tuple(x1, x - q * x1);
        tie(y, y1) = make_tuple(y1, y - q * y1);
        tie(a, b) = make_tuple(b, a - q * b);
    }
    return a;
}
```

```cpp
ll extended_gcd(ll a, ll b, ll &x, ll &y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    } else {
        ll g = extended_gcd(b, a % b, y, x);
        y -= a / b * x;
        return g;
    }
}
```

5.6 Inverso Modular

Algoritmos para calcular o inverso modular de um número. O inverso modular de um inteiro a é outro inteiro x tal que $a \cdot x \equiv 1 \pmod{MOD}$

The modular inverse of an integer a is another integer x such that a * x is congruent to 1 (mod MOD).

Modular Inverse

Calculates the modular inverse of a.

Uses the [exp_mod](/Matemática/Exponenciação%20Modular%20Rápida/exp_mod.cpp) algorithm, thus expects MOD to be prime.

* Time Complexity: O(log(MOD)).

* Space Complexity: O(1).

Modular Inverse by Extended GDC

Calculates the modular inverse of a.

Uses the [extended_gcd](/Matemática/GCD/extended_gcd.cpp) algorithm, thus expects MOD to be coprime with a.

Returns -1 if this assumption is broken.

* Time Complexity: O(log(MOD)).

* Space Complexity: O(1).

Modular Inverse for 1 to MAX

Calculates the modular inverse for all numbers between 1 and MAX.

expects MOD to be prime.

* Time Complexity: O(MAX).

* Space Complexity: O(MAX).

Modular Inverse for all powers

Let b be any integer.

Calculates the modular inverse for all powers of b between b^0 and b^MAX.

Needs you calculate beforehand the modular inverse of b, for 2 it is always (MOD+1)/2.

expects MOD to be coprime with b.

* Time Complexity: O(MAX).

* Space Complexity: O(MAX).

```
ll inv[MAX];

void compute_inv(const ll m = MOD) {
    inv[1] = 1;
    for (int i = 2; i < MAX; i++) {
        inv[i] = m - (m / i) * inv[m % i] % m;
    }
}
```

```
const ll INVB = (MOD + 1) / 2; // Modular inverse of the base,
                               // for 2 it is (MOD+1)/2

ll inv[MAX]; // Modular inverse of b^i

void compute_inv() {
    inv[0] = 1;
    for (int i = 1; i < MAX; i++) {
        inv[i] = inv[i - 1] * INVB % MOD;
    }
}
```

```
ll inv(ll a) {
    return exp_mod(a, MOD - 2);
}
```

```
int inv(int a) {
    int x, y;
    int g = extended_gcd(a, MOD, x, y);
    if (g == 1) {
        return (x % m + m) % m;
    }
    return -1;
}
```

5.7 NTT

Computa a multiplicação de polinômios com coeficientes inteiros módulo um número primo.

Computa multiplicação de polinômino; **Somente para inteiros**.

- Complexidade de tempo: O(N * log(N))

Constantes finais devem ser menor do que 10^9.

Para constantes entre 10^9 e 10^{18} é necessário codar também [big_convolution](big_convolution.cpp).

```
typedef long long ll;
typedef vector<ll> poly;

ll mod[3] = {998244353LL, 1004535809LL, 1092616193LL};
ll root[3] = {102292LL, 12289LL, 23747LL};
ll root_1[3] = {116744195LL, 313564925LL, 642907570LL};
ll root_pw[3] = {1LL << 23, 1LL << 21, 1LL << 21};

ll modInv(ll b, ll m) {
    ll e = m - 2;
    ll res = 1;
    while (e) {
        if (e & 1) {
            res = (res * b) % m;
        }
        e /= 2;
        b = (b * b) % m;
    }
    return res;
}

void ntt(poly &a, bool invert, int id) {
    ll n = (ll)a.size(), m = mod[id];
    for (ll i = 1, j = 0; i < n; ++i) {
        ll bit = n >> 1;
        for (; j >= bit; bit >>= 1) {
            j -= bit;
        }
        j += bit;
        if (i < j) {
            swap(a[i], a[j]);
        }
    }
    for (ll len = 2, wlen; len <= n; len <<= 1) {
        wlen = invert ? root_1[id] : root[id];
        for (ll i = len; i < root_pw[id]; i <<= 1) {
            wlen = (wlen * wlen) % m;
        }
        for (ll i = 0; i < n; i += len) {
            ll w = 1;
            for (ll j = 0; j < len / 2; j++) {
                ll u = a[i + j], v = (a[i + j + len / 2] * w) % m;
                a[i + j] = (u + v) % m;
                a[i + j + len / 2] = (u - v + m) % m;
                w = (w * wlen) % m;
            }
        }
    }
    if (invert) {
        ll inv = modInv(n, m);
        for (ll i = 0; i < n; i++) {
            a[i] = (a[i] * inv) % m;
        }
    }
}

poly convolution(poly a, poly b, int id = 0) {
    ll n = 1LL, len = (1LL + a.size() + b.size());
    while (n < len) {
        n *= 2;
    }
    a.resize(n);
    b.resize(n);
    ntt(a, 0, id);
    ntt(b, 0, id);
    poly answer(n);
    for (ll i = 0; i < n; i++) {
        answer[i] = (a[i] * b[i]);
    }
    ntt(answer, 1, id);
    return answer;
}
```

```
ll mod_mul(ll a, ll b, ll m) {
    return (__int128)a * b % m;
}
ll ext_gcd(ll a, ll b, ll &x, ll &y) {
    if (!b) {
        x = 1;
        y = 0;
        return a;
    } else {
        ll g = ext_gcd(b, a % b, y, x);
        y -= a / b * x;
        return g;
    }
}

// convolution mod 1,097,572,091,361,755,137
poly big_convolution(poly a, poly b) {
    poly r0, r1, answer;
    r0 = convolution(a, b, 1);
    r1 = convolution(a, b, 2);

    ll s, r, p = mod[1] * mod[2];
    ext_gcd(mod[1], mod[2], r, s);

    answer.resize(r0.size());
    for (int i = 0; i < (int)answer.size(); i++) {
        answer[i] = (mod_mul((s * mod[2] + p) % p, r0[i], p) +
                     mod_mul((r * mod[1] + p) % p, r1[i], p) + p) %
                    p;
    }
    return answer;
}
```

5.8 Primos

Algortimos relacionados a números primos.

Crivo de Eratóstenes

Computa a primalidade de todos os números até N, quase tão rápido quanto o crivo linear.

- Complexidade de tempo: O(N * log(log(N)))

Demora 1 segundo para LIM igual a $3 * 10^7$.

Miller-Rabin

Teste de primalidade garantido para números menores do que 2^64.

- Complexidade de tempo: O(log(N))

Teste Ingênuo

Computa a primalidade de um número N.

- Complexidade de tempo: $O(N^(1/2))$

```
vector<bool> sieve(int n) {
    vector<bool> is_prime(n + 5, true);
    is_prime[0] = false;
    is_prime[1] = false;
    long long sq = sqrt(n + 5);
    for (long long i = 2; i <= sq; i++) {
        if (is_prime[i]) {
            for (long long j = i * i; j < n; j += i) {
                is_prime[j] = false;
            }
        }
    }
    return is_prime;
}
```

```
bool is_prime(int n) {
    for (long long d = 2; d * d <= n; d++) {
        if (n % d == 0) {
            return false;
        }
    }
    return true;
}
```

```
long long power(long long base, long long e, long long mod) {
    long long result = 1;
    base %= mod;
    while (e) {
        if (e & 1) {
            result = (__int128)result * base % mod;
        }
        base = (__int128)base * base % mod;
        e >>= 1;
    }
    return result;
}
```

```
bool is_composite(long long n, long long a, long long d, int s) {
    long long x = power(a, d, n);
    if (x == 1 || x == n - 1) {
        return false;
    }
    for (int r = 1; r < s; r++) {
        x = (__int128)x * x % n;
        if (x == n - 1) {
            return false;
        }
    }
    return true;
}

bool miller_rabin(long long n) {
    if (n < 2) {
        return false;
    }
    int r = 0;
    long long d = n - 1;
    while ((d & 1) == 0) {
        d >>= 1, ++r;
    }
    for (int a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {
        if (n == a) {
            return true;
        }
        if (is_composite(n, a, d, r)) {
            return false;
        }
    }
    return true;
}
```

5.9 Sum of floor (n div i)

Computa

$$\sum_{i=1}^{n} \lfloor \frac{n}{i} \rfloor$$

- Complexidade de tempo: $O(\sqrt{n})$

```
const int MOD = 1e9 + 7;

long long sumoffloor(long long n) {
    long long answer = 0, i;
    for (i = 1; i * i <= n; i++) {
        answer += n / i;
        answer %= MOD;
    }
    i--;
    for (int j = 1; n / (j + 1) >= i; j++) {
        answer += (((n / j - n / (j + 1)) % MOD) * j) % MOD;
        answer %= MOD;
    }
    return answer;
}
```

5.10 Teorema do Resto Chinês

Algoritmo que resolve o sistema $x \equiv a_i \pmod{m_i}$, onde m_i são primos entre si.

Retorna -1 se a resposta não existir.

```
ll extended_gcd(ll a, ll b, ll &x, ll &y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    } else {
        ll g = extended_gcd(b, a % b, y, x);
        y -= a / b * x;
        return g;
    }
}

ll crt(vector<ll> rem, vector<ll> mod) {
    int n = rem.size();
    if (n == 0) {
        return 0;
    }
    __int128 ans = rem[0], m = mod[0];
    for (int i = 1; i < n; i++) {
        ll x, y;
        ll g = extended_gcd(mod[i], m, x, y);
        if ((ans - rem[i]) % g != 0) {
            return -1;
        }
        ans = ans + (__int128)1 * (rem[i] - ans) * (m / g) * y;
        m = (__int128)(mod[i] / g) * (m / g) * g;
        ans = (ans % m + m) % m;
    }
    return ans;
}
```

5.11 Totiente de Euler

Código para computar o totiente de Euler.

Totiente de Euler (Phi) para um número

Computa o totiente para um único número N.

- Complexidade de tempo: $O(N^{(}1/2))$

Totiente de Euler (Phi) entre 1 e N

Computa o totiente entre 1 e N.

- Complexidade de tempo: O(N * log(log(N)))

```
vector<int> phi_1_to_n(int n) {
    vector<int> phi(n + 1);
    for (int i = 0; i <= n; i++) {
        phi[i] = i;
    }
    for (int i = 2; i <= n; i++) {
        if (phi[i] == i) {
            for (int j = i; j <= n; j += i) {
                phi[j] -= phi[j] / i;
            }
        }
    }
    return phi;
}
```

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

```
    }
    return result;
}
```

Capítulo 6

Theoretical

6.1 Some Prime Numbers

6.1.1 Left-Truncatable Prime

Prime number such that any suffix of it is a prime number
357,686,312,646,216,567,629,137

6.1.2 Mersenne Primes

Prime numbers of the form $2^m - 1$

Exponent (m)	**Decimal representation**
2	3
3	7
5	31
7	127
13	8,191
17	131,071
19	524,287
31	2,147,483,647
61	$2,3 * 10^{18}$
89	$6,1 * 10^{26}$
107	$1,6 * 10^{32}$
127	$1,7 * 10^{38}$

6.2 C++ constants

Constant	C++ Name	Value
π	`M_PI`	3.141592...
$\pi/2$	`M_PI_2`	1.570796...
$\pi/4$	`M_PI_4`	0.785398...
$1/\pi$	`M_1_PI`	0.318309...
$2/\pi$	`M_2_PI`	0.636619...
$2/\sqrt{\pi}$	`M_2_SQRTPI`	1.128379...
$\sqrt{2}$	`M_SQRT2`	1.414213...
$1/\sqrt{2}$	`M_SQRT1_2`	0.707106...
e	`M_E`	2.718281...
$\log_2 e$	`M_LOG2E`	1.442695...
$\log_{10} e$	`M_LOG10E`	0.434294...
$\ln 2$	`M_LN2`	0.693147...
$\ln 10$	`M_LN10`	2.302585...

6.3 Linear Operators

6.3.1 Rotate counter-clockwise by $\theta°$

$$\begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix}$$

6.3.2 Reflect about the line $y = mx$

$$\frac{1}{m^2+1}\begin{bmatrix} 1-m^2 & 2m \\ 2m & m^2-1 \end{bmatrix}$$

6.3.3 Inverse of a 2x2 matrix A

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{det(A)} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

6.3.4 Horizontal shear by K

$$\begin{bmatrix} 1 & K \\ 0 & 1 \end{bmatrix}$$

6.3.5 Vertical shear by K

$$\begin{bmatrix} 1 & 0 \\ K & 1 \end{bmatrix}$$

6.3.6 Change of basis

$\vec{a}_\beta$ are the coordinates of vector $\vec{a}$ in basis β.
$\vec{a}$ are the coordinates of vector $\vec{a}$ in the canonical basis.
$\vec{b1}$ and $\vec{b2}$ are the basis vectors for β.
C is a matrix that changes from basis β to the canonical basis.

$$C\vec{a}_\beta = \vec{a}$$

$$C^{-1}\vec{a} = \vec{a}_\beta$$

$$C = \begin{bmatrix} b1_x & b2_x \\ b1_y & b2_y \end{bmatrix}$$

6.3.7 Properties of matrix operations

$$(AB)^{-1} = A^{-1}B^{-1}$$
$$(AB)^T = B^T A^T$$
$$(A^{-1})^T = (A^T)^{-1}$$
$$(A+B)^T = A^T + B^T$$
$$det(A) = det(A^T)$$
$$det(AB) = det(A)det(B)$$

Let A be an NxN matrix:

$$det(kA) = K^N det(A)$$