

Amigos do Beto - ICPC Library

Contents

1 Data Structures

1.1	BIT 2D Comprimida	1
1.2	Segment Tree with Lazy Propagation	2
1.3	Treap	2
1.4	Persistent Treap	3
1.5	KD-Tree	3
1.6	Link Cut Tree	4
1.7	Sparse Table	5
1.8	Max Queue	5
1.9	Policy Based Structures	5
1.10	Color Updates Structure	5

2 Graph Algorithms

2.1	Simple Disjoint Set	5
2.2	Blossom	6
2.3	Boruvka	7
2.4	Dinic Max Flow	7
2.5	Minimum Vertex Cover	7
2.6	Min Cost Max Flow	7
2.7	Euler Path and Circuit	8
2.8	Articulation Points/Bridges/Biconnected Components	8
2.9	SCC - Strongly Connected Components / 2SAT	9
2.10	LCA - Lowest Common Ancestor	9
2.11	Heavy Light Decomposition	9
2.12	Centroid Decomposition	10
2.13	Sack	10
2.14	Hungarian Algorithm - Maximum Cost Matching	10
2.15	Burunduk	11
2.16	Minimum Arborescence	11

3 Dynamic Programming

3.1	Line Container	12
3.2	Li Chao Tree	12
3.3	Divide and Conquer Optimization	13
3.4	Knuth Optimization	13

4 Math

4.1	Chinese Remainder Theorem	13
4.2	Diophantine Equations	13
4.3	Discrete Logarithm	14
4.4	Discrete Root	14
4.5	Division Trick	14
4.6	Modular Sum	14
4.7	Primitive Root	14
4.8	Extended Euclides	15
4.9	Matrix Fast Exponentiation	15
4.10	FFT - Fast Fourier Transform	15
4.11	NTT - Number Theoretic Transform	16
4.12	Fast Walsh Hadamard Transform	17
4.13	Miller and Rho	18
4.14	Determinant using Mod	18
4.15	Lagrange Interpolation	19
4.16	Count integer points inside triangle	19

5 Geometry

5.1	Geometry	19
5.2	Convex Hull	21
5.3	Cut Polygon	22
5.4	Smallest Enclosing Circle	22
5.5	Minkowski	22
5.6	Half Plane Intersection	23
5.7	Closest Pair	23
5.8	Voronoi	23

6 String Algorithms

6.1	KMP	23
6.2	KMP Automaton	24
6.3	Trie	24
6.4	Aho-Corasick	24
6.5	Algoritmo de Z	24
6.6	Suffix Array	25
6.7	Suffix Tree	25
6.8	Suffix Automaton	25
6.9	Manacher	26
6.10	Polish Notation	26

7 Miscellaneous

7.1	Ternary Search	26
7.2	Count Sort	26
7.3	Random Number Generator	26
7.4	Rectangle Hash	27
7.5	Unordered Map Tricks	27
7.6	Submask Enumeration	27
7.7	Sum over Subsets DP	27
7.8	Dates	27
7.9	Lat Long	27

8 Teoremas e formulas uteis

8.1	Grafos	28
8.2	Math	28
8.3	Geometry	28
8.4	Mersenne's Primes	28

1 Data Structures

1.1 BIT 2D Comprimida

```
// src: tfg50
template<class T = int>
struct Bit2D {
public:
    Bit2D(vector<pair<T, T>> pts) {
        sort(pts.begin(), pts.end());
        for(auto a : pts) {
            if(ord.empty() || a.first != ord.back()) {
                ord.push_back(a.first);
            }
        }
        fw.resize(ord.size() + 1);
        coord.resize(fw.size());
        for(auto &a : pts) {
            swap(a.first, a.second);
        }
        sort(pts.begin(), pts.end());
        for(auto &a : pts) {
            swap(a.first, a.second);
            for(int on = upper_bound(ord.begin(), ord.end(), a.first) - ord.begin(); on < fw.size(); on += on & -on) {
                if(coord[on].empty() || coord[on].back() != a.second) {
                    coord[on].push_back(a.second);
                }
            }
        }
        for(int i = 0; i < fw.size(); i++) {
            fw[i].assign(coord[i].size() + 1, 0);
        }
    }

    void upd(T x, T y, T v) {
        for(int xx = upper_bound(ord.begin(), ord.end(), x) - ord.begin(); xx < fw.size(); xx += xx & -xx) {
            for(int yy = upper_bound(coord[xx].begin(), coord[xx].end(), y) - coord[xx].begin(); yy < fw[xx].size(); yy += yy & -yy) {
                fw[xx][yy] += v;
            }
        }
    }
};
```

```

T qry(T x, T y) {
    T ans = 0;
    for(int xx = upper_bound(ord.begin(), ord.end(), x) - ord.begin(); xx > 0; xx -=
        xx & -xx) {
        for(int yy = upper_bound(coord[xx].begin(), coord[xx].end(), y) - coord[xx].
            begin(); yy > 0; yy -= yy & -yy) {
            ans += fw[xx][yy];
        }
    }
    return ans;
}

T qry(T x1, T y1, T x2, T y2) {
    return qry(x2, y2) - qry(x2, y1 - 1) - qry(x1 - 1, y2) + qry(x1 - 1, y1 - 1);
}

void upd(T x1, T y1, T x2, T y2, T v) { // !insert these points
    upd(x1, y1, v);
    upd(x1, y2 + 1, -v);
    upd(x2 + 1, y1, -v);
    upd(x2 + 1, y2 + 1, v);
}

private:
    vector<T> ord;
    vector<vector<T>> fw, coord;
};

```

1.2 Segment Tree with Lazy Propagation

```

struct LazyContext {
    LazyContext() {}
    void reset() {}
    void operator += (LazyContext &o) {}
    // add attributes
};

struct Node {
    Node() {} // neutral element or empty node
    Node() {} // init
    Node(Node &l, Node &r) {} // merge
    void apply(LazyContext &lazy) {}
    // add attributes
};

template <class i_t, class e_t, class lazy_cont = int> //internal node, external node
class SegmentTree { // Range update, range query segment tree
public:
    void init(vector<e_t> base) {
        n = base.size();
        h = 0;
        while((1 << h) < n) h++;
        tree.resize(2 * n);
        dirty.assign(n, false);
        lazy.resize(n);
        for(int i = 0; i < n; ++i) {
            tree[i + n] = i_t(base[i]); //add arguments if needed to init
        }
        for(int i = n - 1; i > 0; --i) {
            tree[i] = i_t(tree[i << 1], tree[i << 1 | 1]);
            lazy[i].reset();
        }
    }

    i_t qry(int l, int r) { // [l, r)
        if(l >= r) return i_t();
        l += n, r += n;
        push(l);
        push(r - 1);
        i_t lp, rp;
        for(; l < r; l >= 1, r >= 1) {
            if(l & 1) lp = i_t(lp, tree[l++]);
            if(r & 1) rp = i_t(tree[--r], rp);
        }
        return i_t(lp, rp);
    }

    void upd(int l, int r, lazy_cont lc) { // [l, r)
        if(l >= r) return;
        l += n, r += n;
        push(l);
        push(r - 1);
        int l0 = l, r0 = r;
        for(; l < r; l >= 1, r >= 1) {
            if(l & 1) apply(l++, lc);

```

```

            if(r & 1) apply(--r, lc);
        }
        build(l0);
        build(r0 - 1);
    }

private:
    int n, h;
    vector<bool> dirty;
    vector<i_t> tree;
    vector<lazy_cont> lazy;
    void apply(int p, lazy_cont &lc) {
        tree[p].apply(lc);
        if(p < n) {
            dirty[p] = true;
            lazy[p] += lc;
        }
    }

    void push(int p) {
        for(int s = h; s > 0; --s) {
            int i = p >> s;
            if(dirty[i]) {
                apply(i << 1, lazy[i]);
                apply(i << 1 | 1, lazy[i]);
                lazy[i].reset();
                dirty[i] = false;
            }
        }
    }

    void build(int p) {
        for(p >= 1; p > 0; p >= 1) {
            tree[p] = i_t(tree[p << 1], tree[p << 1 | 1]);
            if(dirty[p]) {
                tree[p].apply(lazy[p]);
            }
        }
    }
};

```

1.3 Treap

```

mt19937 rng ((int) chrono::steady_clock::now().time_since_epoch().count());

typedef int Value;
typedef struct item * pitem;

struct item {
    item() {}
    item(Value v) { // add key if not implicit
        value = v;
        prio = uniform_int_distribution<int>() (rng);
        cnt = 1;
        rev = 0;
        l = r = 0;
    }
    pitem l, r;
    Value value;
    int prio, cnt;
    bool rev;
};

int cnt (pitem it) {
    return it ? it->cnt : 0;
}

void fix (pitem it) {
    if (it)
        it->cnt = cnt(it->l) + cnt(it->r) + 1;
}

void pushLazy (pitem it) {
    if (it && it->rev) {
        it->rev = false;
        swap(it->l, it->r);
        if (it->l) it->l->rev ^= true;
        if (it->r) it->r->rev ^= true;
    }
}

void merge (pitem &t, pitem l, pitem r) {
    pushLazy (l); pushLazy (r);

```

```

    if (!l || !r) t = l ? l : r;
    else if (l->prio > r->prio)
        merge(l->r, l->r, r), t = l;
    else
        merge(r->l, l, r->l), t = r;
    fix(t);
}

void split(pitem t, pitem &l, pitem &r, int key) {
    if (!t) return void( l = r = 0 );
    pushLazy(t);
    int cur_key = cnt(t->l); // t->key if not implicit
    if (key <= cur_key)
        split(t->l, l, t->l, key), r = t;
    else
        split(t->r, t->r, r, key - (1 + cnt(t->l))), l = t;
    fix(t);
}

void reverse(pitem t, int l, int r) {
    pitem t1, t2, t3;
    split(t, t1, t2, l);
    split(t2, t2, t3, r-1+1);
    t2->rev ^= true;
    merge(t, t1, t2);
    merge(t, t, t3);
}

void unite(pitem &t, pitem l, pitem r) {
    if (!l || !r) return void( t = l ? l : r );
    if (l->prio < r->prio) swap(l, r);
    pitem lt, rt;
    split(r, lt, rt, l->key);
    unite(l->l, l->l, lt);
    unite(l->r, r, l->r, rt);
    t = l;
}

```

1.4 Persistent Treap

```

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());

typedef int Key;
struct Treap {
    Treap(){}
    Treap(char k) {
        key = 1;
        size = 1;
        l = r = NULL;
        val = k;
    }

    Treap *l, *r;
    Key key;
    char val;
    int size;
};

typedef Treap * PTreap;

bool leftSide(PTreap l, PTreap r) {
    return (int) (rng() % (l->size + r->size)) < l->size;
}

void fix(PTreap t) {
    if (t == NULL) {
        return;
    }
    t->size = 1;
    t->key = 1;
    if (t->l) {
        t->size += t->l->size;
        t->key += t->l->size;
    }
    if (t->r) {
        t->size += t->r->size;
    }
}

void split(PTreap t, Key key, PTreap &l, PTreap &r) {
    if (t == NULL) {

```

```

        l = r = NULL;
    } else if (t->key <= key) {
        l = new Treap();
        *l = *t;
        split(t->r, key - t->key, l->r, r);
        fix(l);
    } else {
        r = new Treap();
        *r = *t;
        split(t->l, key, l, r->l);
        fix(r);
    }
}

void merge(PTreap &t, PTreap l, PTreap r) {
    if (!l || !r) {
        t = l ? l : r;
        return;
    }
    t = new Treap();
    if (leftSide(l, r)) {
        *t = *l;
        merge(t->r, l->r, r);
    } else {
        *t = *r;
        merge(t->l, l, r->l);
    }
    fix(t);
}

vector<PTreap> ver = {NULL};

PTreap build(int l, int r, string& s) {
    if (l >= r) return NULL;
    int mid = (l + r) >> 1;
    auto ans = new Treap(s[mid]);
    ans->l = build(l, mid, s);
    ans->r = build(mid + 1, r, s);
    fix(ans);
    return ans;
}

int last = 0;

void go(PTreap t, int f) {
    if (!t) return;
    go(t->l, f);
    cout << t->val;
    last += (t->val == 'c') * f;
    go(t->r, f);
}

void insert(PTreap t, int pos, string& s) {
    PTreap l, r;
    split(t, pos + 1, l, r);
    PTreap mid = build(0, s.size(), s);
    merge(mid, l, mid);
    merge(mid, mid, r);
    ver.push_back(mid);
}

void erase(PTreap t, int L, int R) {
    PTreap l, mid, r;
    split(t, L, l, mid);
    split(mid, R - L + 1, mid, r);
    merge(l, l, r);
    ver.push_back(l);
}

```

1.5 KD-Tree

```

int d;
long long getValue(const PT &a) {return (d & 1) == 0 ? a.x : a.y; }
bool comp(const PT &a, const PT &b) {
    if ((d & 1) == 0) { return a.x < b.x; }
    else { return a.y < b.y; }
}

long long sqrDist(PT a, PT b) { return (a - b) * (a - b); }

class KD_Tree {
public:
    struct Node {

```

```

    PT point;
    Node *left, *right;
};

void init(std::vector<PT> pts) {
    if(pts.size() == 0) {
        return;
    }
    int n = 0;
    tree.resize(2 * pts.size());
    build(pts.begin(), pts.end(), n);
    //assert(n <= (int) tree.size());
}

long long nearestNeighbor(PT point) {
    // assert(tree.size() > 0);
    long long ans = (long long) 1e18;
    nearestNeighbor(&tree[0], point, 0, ans);
    return ans;
}

private:
std::vector<Node> tree;

Node* build(std::vector<PT>::iterator l, std::vector<PT>::iterator r, int &n, int h
            = 0) {
    int id = n++;
    if(r - l == 1) {
        tree[id].left = tree[id].right = NULL;
        tree[id].point = *l;
    } else if(r - l > 1) {
        std::vector<PT>::iterator mid = l + ((r - l) / 2);
        d = h;
        std::nth_element(l, mid - 1, r, comp);
        tree[id].point = *(mid - 1);
        // BE CAREFUL!
        // DO EVERYTHING BEFORE BUILDING THE LOWER PART!
        tree[id].left = build(l, mid, n, h^1);
        tree[id].right = build(mid, r, n, h^1);
    }
    return &tree[id];
}

void nearestNeighbor(Node* node, PT point, int h, long long &ans) {
    if(!node) {
        return;
    }
    if(point != node->point) {
        // THIS WAS FOR A PROBLEM
        // THAT YOU DON'T CONSIDER THE DISTANCE TO ITSELF!
        ans = std::min(ans, sqrDist(point, node->point));
    }
    d = h;
    long long delta = getValue(point) - getValue(node->point);
    if(delta <= 0) {
        nearestNeighbor(node->left, point, h^1, ans);
        if(ans > delta * delta) {
            nearestNeighbor(node->right, point, h^1, ans);
        }
    } else {
        nearestNeighbor(node->right, point, h^1, ans);
        if(ans > delta * delta) {
            nearestNeighbor(node->left, point, h^1, ans);
        }
    }
}
};

```

1.6 Link Cut Tree

```

/**
 * Author: Simon Lindholm
 * Date: 2016-07-25
 * Source: https://github.com/ngthanhtung23/ACM_Notebook_new/blob/master/
 *         DataStructure/LinkCut.h
 * Description: Represents a forest of unrooted trees. You can add and remove
 * edges (as long as the result is still a forest), and check whether
 * two nodes are in the same tree.
 * Time: All operations take amortized  $O(\log N)$ .
 * Status: Fuzz-tested a bit for  $N \leq 20$ 
 */

```

```

#pragma once

struct Node { // Splay tree. Root's pp contains tree's parent.
    Node *p = 0, *pp = 0, *c[2];
    bool flip = 0;
    Node() { c[0] = c[1] = 0; fix(); }
    void fix() {
        if (c[0]) c[0]->p = this;
        if (c[1]) c[1]->p = this;
        // (+ update sum of subtree elements etc. if wanted)
    }
    void push_flip() {
        if (!flip) return;
        flip = 0; swap(c[0], c[1]);
        if (c[0]) c[0]->flip ^= 1;
        if (c[1]) c[1]->flip ^= 1;
    }
    int up() { return p ? p->c[1] == this : -1; }
    void rot(int i, int b) {
        int h = i ^ b;
        Node *x = c[i], *y = b == 2 ? x : x->c[h], *z = b ? y : x;
        if ((y->p == p)) p->c[up()] = y;
        c[i] = z->c[i ^ 1];
        if (b < 2) {
            x->c[h] = y->c[h ^ 1];
            z->c[h ^ 1] = b ? x : this;
        }
        y->c[i ^ 1] = b ? this : x;
        fix(); x->fix(); y->fix();
        if (p) p->fix();
        swap(pp, y->pp);
    }
    void splay() { /// Splay this up to the root. Always finishes without flip set.
        for (push_flip(); p; ) {
            if (p->p) p->p->push_flip();
            p->push_flip(); push_flip();
            int c1 = up(), c2 = p->up();
            if (c2 == -1) p->rot(c1, 2);
            else p->p->rot(c2, c1 != c2);
        }
    }
    Node* first() { /// Return the min element of the subtree rooted at this, splayed to
        the top.
        push_flip();
        return c[0] ? c[0]->first() : (splay(), this);
    }
};

struct LinkCut {
    vector<Node> node;
    LinkCut(int N) : node(N) {}

    void link(int u, int v) { // add an edge (u, v)
        assert(!connected(u, v));
        make_root(&node[u]);
        node[u].pp = &node[v];
    }

    void cut(int u, int v) { // remove an edge (u, v)
        Node *x = &node[u], *top = &node[v];
        make_root(top); x->splay();
        assert(top == (x->pp ? x->c[0]));
        if (x->pp) x->pp = 0;
        else {
            x->c[0] = top->p = 0;
            x->fix();
        }
    }

    bool connected(int u, int v) { // are u, v in the same tree?
        Node* nu = access(&node[u])->first();
        return nu == access(&node[v])->first();
    }

    void make_root(Node* u) { /// Move u to root of represented tree.
        access(u);
        u->splay();
        if(u->c[0]) {
            u->c[0]->p = 0;
            u->c[0]->flip ^= 1;
            u->c[0]->pp = u;
            u->c[0] = 0;
            u->fix();
        }
    }

    Node* access(Node* u) { /// Move u to root aux tree. Return the root of the root aux

```

```

    tree.
    u->splay();
    while (Node* pp = u->pp) {
        pp->splay(); u->pp = 0;
        if (pp->c[1]) {
            pp->c[1]->p = 0; pp->c[1]->pp = pp; }
        pp->c[1] = u; pp->fix(); u = pp;
    }
    return u;
};

```

1.7 Sparse Table

```

vector<vector<int>> table;
vector<int> lg2;

void build(int n, vector<int> v) {
    lg2.resize(n + 1);
    lg2[1] = 0;
    for (int i = 2; i <= n; i++) {
        lg2[i] = lg2[i >> 1] + 1;
    }
    table.resize(lg2[n] + 1);
    for (int i = 0; i < lg2[n] + 1; i++) {
        table[i].resize(n + 1);
    }
    for (int i = 0; i < n; i++) {
        table[0][i] = v[i];
    }
    for (int i = 0; i < lg2[n]; i++) {
        for (int j = 0; j < n; j++) {
            if (j + (1 << i) >= n) break;
            table[i + 1][j] = min(table[i][j], table[i][j + (1 << i)]);
        }
    }

    int get(int l, int r) {
        int k = lg2[r - l + 1];
        return min(table[k][l], table[k][r - (1 << k) + 1]);
    }
}

```

1.8 Max Queue

```

template <class T, class C = less<T>>
struct MaxQueue {
    MaxQueue() { clear(); }
    void clear() {
        id = 0;
        q.clear();
    }
    void push(T x) {
        pair<int, T> nxt(1, x);
        while (q.size() > id && cmp(q.back().second, x)) {
            nxt.first += q.back().first;
            q.pop_back();
        }
        q.push_back(nxt);
    }
    T qry() { return q[id].second; }
    void pop() {
        q[id].first--;
        if (q[id].first == 0) { id++; }
    }
private:
    vector<std::pair<int, T>> q;
    int id;
    C cmp;
};

```

1.9 Policy Based Structures

```

#include <ext/pb_ds/assoc_container.hpp> // Common file
#include <ext/pb_ds/tree_policy.hpp> // Including tree_order_statistics_node_update

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.find_by_order(0);
X.order_of_key(-5);
end(X), begin(X);

```

1.10 Color Updates Structure

```

struct range {
    int l, r;
    int v;

    range(int l = 0, int r = 0, int v = 0) : l(l), r(r), v(v) {}

    bool operator < (const range &a) const {
        return l < a.l;
    }
};

set<range> ranges;

vector<range> update(int l, int r, int v) { // [l, r)
    vector<range> ans;
    if (l >= r) return ans;
    auto it = ranges.lower_bound(l);
    if (it != ranges.begin()) {
        it--;
        if (it->r > l) {
            auto cur = *it;
            ranges.erase(it);
            ranges.insert(range(cur.l, l, cur.v));
            ranges.insert(range(l, cur.r, cur.v));
        }
    }
    it = ranges.lower_bound(r);
    if (it != ranges.begin()) {
        it--;
        if (it->r > r) {
            auto cur = *it;
            ranges.erase(it);
            ranges.insert(range(cur.l, r, cur.v));
            ranges.insert(range(r, cur.r, cur.v));
        }
    }
    for (it = ranges.lower_bound(l); it != ranges.end() && it->l < r; it++) {
        ans.push_back(*it);
    }
    ranges.erase(ranges.lower_bound(l), ranges.lower_bound(r));
    ranges.insert(range(l, r, v));
    return ans;
}

int query(int v) { // Substituir -1 por flag para quando nao houver resposta
    auto it = ranges.upper_bound(v);
    if (it == ranges.begin()) {
        return -1;
    }
    it--;
    return it->r >= v ? it->v : -1;
}

```

2 Graph Algorithms

2.1 Simple Disjoint Set

```

struct dsu {
    vector<int> hist, par, sz;
    vector<ii> changes;
    int n;
    dsu (int n) : n(n) {
        hist.assign(n, 1e9);
        par.resize(n);
        iota(par.begin(), par.end(), 0);
        sz.assign(n, 1);
    }

    int root (int x, int t) {
        if(hist[x] > t) return x;
        return root(par[x], t);
    }

    void join (int a, int b, int t) {
        a = root(a, t);
        b = root(b, t);
        if (a == b) { changes.emplace_back(-1, -1); return; }
        if (sz[a] > sz[b]) swap(a, b);
        par[a] = b;
        sz[b] += sz[a];
        hist[a] = t;
        changes.emplace_back(a, b);
        n--;
    }

    bool same (int a, int b, int t) {
        return root(a, t) == root(b, t);
    }

    void undo () {
        int a, b;
        tie(a, b) = changes.back();
        changes.pop_back();
        if (a == -1) return;
        sz[b] -= sz[a];
        par[a] = a;
        hist[a] = 1e9;
        n++;
    }

    int when (int a, int b) {
        while (1) {
            if (hist[a] > hist[b]) swap(a, b);
            if (par[a] == b) return hist[a];
            if (hist[a] == 1e9) return 1e9;
            a = par[a];
        }
    }
};

```

2.2 Blossom

```

#define MAXN 110
#define MAXM MAXN*MAXN
int n, m;
int mate[MAXN], first[MAXN], label[MAXN];
int adj[MAXN][MAXN], nadj[MAXN], from[MAXM], to[MAXM];
queue<int> q;
#define OUTER(x) (label[x] >= 0)
void L(int x, int y, int nxy) {
    int join, v, r = first[x], s = first[y];
    if (r == s) { return; }
    nxy += n + 1;
    label[r] = label[s] = -nxy;
    while (1) {
        if (s != 0) { swap(r, s); }
        r = first[label[mate[r]]];
        if (label[r] != -nxy) { label[r] = -nxy; }
        else {
            join = r;
            break;
        }
    }
    v = first[x];
    while (v != join) {
        if (!OUTER(v)) { q.push(v); }
        label[v] = nxy;
    }
}

```

```

    first[v] = join;
    v = first[label[mate[v]]];
}
v = first[y];
while (v != join) {
    if (!OUTER(v)) { q.push(v); }
    label[v] = nxy;
    first[v] = join;
    v = first[label[mate[v]]];
}
for (int i = 0; i <= n; i++) {
    if (OUTER(i) && OUTER(first[i])) { first[i] = join; }
}

void R(int v, int w) {
    int t = mate[v];
    mate[v] = w;
    if (mate[t] != v) { return; }
    if (label[v] >= 1 && label[v] <= n) {
        mate[t] = label[v];
        R(label[v], t);
        return;
    }
    int x = from[label[v] - n - 1], y = to[label[v] - n - 1];
    R(x, y);
    R(y, x);
}

int E() {
    memset(mate, 0, sizeof(mate));
    int r = 0;
    bool e7;
    for (int u = 1; u <= n; u++) {
        memset(label, -1, sizeof(label));
        while (!q.empty()) { q.pop(); }
        if (mate[u]) { continue; }
        label[u] = first[u] = 0;
        q.push(u);
        e7 = false;
        while (!q.empty() && !e7) {
            int x = q.front();
            q.pop();
            for (int i = 0; i < nadj[x]; i++) {
                int y = from[adj[x][i]];
                if (y == x) { y = to[adj[x][i]]; }
                if (!mate[y] && y != u) {
                    mate[y] = x;
                    R(x, y);
                    r++;
                    e7 = true;
                    break;
                } else if (OUTER(y)) { L(x, y, adj[x][i]); }
                else {
                    int v = mate[y];
                    if (!OUTER(v)) {
                        label[v] = x;
                        first[v] = y;
                        q.push(v);
                    }
                }
            }
        }
        label[0] = -1;
    }
    return r;
}

/*Exemplo simples de uso*/
memset(nadj, 0, sizeof nadj);
for (int i = 0; i < m; ++i) { // arestas
    scanf("%d%d", &a, &b);
    a++, b++; // nao utilizar o vertice 0
    adj[a][nadj[a]++] = i;
    adj[b][nadj[b]++] = i;
    from[i] = a;
    to[i] = b;
}

printf("O emparelhamento tem tamanho %d\n", E());
for (int i = 1; i <= n; i++) {
    if (mate[i] > i) { printf("%d com %d\n", i - 1, mate[i] - 1); }
}

```

2.3 Boruvka

```
struct edge {
    int u, v;
    int w;
    int id;
    edge () {}
    edge (int u, int v, int w = 0, int id = 0) : u(u), v(v), w(w), id(id) {};
    bool operator < (edge &other) const { return w < other.w; };
};

vector<edge> boruvka (vector<edge> &edges, int n) {
    vector<edge> mst;
    vector<edge> best(n);
    initDSU(n);
    bool f = 1;
    while (f) {
        f = 0;
        for (int i = 0; i < n; i++) best[i] = edge(i, i, inf);
        for (auto e : edges) {
            int pu = root(e.u), pv = root(e.v);
            if (pu == pv) continue;
            if (e < best[pu]) best[pu] = e;
            if (e < best[pv]) best[pv] = e;
        }
        for (int i = 0; i < n; i++) {
            edge e = best[root(i)];
            if (e.w != inf) {
                join(e.u, e.v);
                mst.push_back(e);
                f = 1;
            }
        }
    }
    return mst;
}
```

2.4 Dinic Max Flow

```
const int ms = 1e3; // Quantidade maxima de vertices
const int me = 1e5; // Quantidade maxima de arestas

int adj[ms], to[me], ant[me], wt[me], z, n;
int copy_adj[ms], fila[ms], level[ms];

void clear() { // Lembrar de chamar no main
    memset(adj, -1, sizeof adj);
    z = 0;
}

void add(int u, int v, int k) {
    to[z] = v;
    ant[z] = adj[u];
    wt[z] = k;
    adj[u] = z++;
    swap(u, v);
    to[z] = v;
    ant[z] = adj[u];
    wt[z] = 0; // Lembrar de colocar = 0
    adj[u] = z++;
}

int bfs(int source, int sink) {
    memset(level, -1, sizeof level);
    level[source] = 0;
    int front = 0, size = 0, v;
    fila[size++] = source;
    while(front < size) {
        v = fila[front++];
        for(int i = adj[v]; i != -1; i = ant[i]) {
            if(wt[i] && level[to[i]] == -1) {
                level[to[i]] = level[v] + 1;
                fila[size++] = to[i];
            }
        }
    }
    return level[sink] != -1;
}

int dfs(int v, int sink, int flow) {
```

```
    if(v == sink) return flow;
    int f;
    for(int &i = copy_adj[v]; i != -1; i = ant[i]) {
        if(wt[i] && level[to[i]] == level[v] + 1 &&
            (f = dfs(to[i], sink, min(flow, wt[i])))) {
            wt[i] -= f;
            wt[i ^ 1] += f;
            return f;
        }
    }
    return 0;
}

int maxflow(int source, int sink) {
    int ret = 0, flow;
    while(bfs(source, sink)) {
        memcpy(copy_adj, adj, sizeof adj);
        while((flow = dfs(source, sink, 1 << 30))) {
            ret += flow;
        }
    }
    return ret;
}
```

2.5 Minimum Vertex Cover

```
// + Dinic
vector<int> coverU, U, coverV, V; // ITA - Particao U LEFT, particao V RIGHT, 0
indexed
bool Zu[mx], Zv[mx];
int pairU[mx], pairV[mx];
void getreach(int u) {
    if (u == -1 || Zu[u]) return;
    Zu[u] = true;
    for (int i = adj[u]; ~i; i = ant[i]) {
        int v = to[i];
        if (v == SOURCE || v == pairU[u]) continue;
        Zv[v] = true;
        getreach(pairV[v]);
    }
}

void minimumcover () {
    memset(pairU, -1, sizeof pairU);
    memset(pairV, -1, sizeof pairV);
    for (auto i : U) {
        for (int j = adj[i]; ~j; j = ant[j]) {
            if (!(j&1) && !wt[j]) {
                pairU[i] = to[j], pairV[to[j]] = i;
            }
        }
    }
    memset(Zu, 0, sizeof Zu);
    memset(Zv, 0, sizeof Zv);
    for (auto u : U) {
        if (pairU[u] == -1) getreach(u);
    }
    coverU.clear(), coverV.clear();
    for (auto u : U) {
        if (!Zu[u]) coverU.push_back(u);
    }
    for (auto v : V) {
        if (Zv[v]) coverV.push_back(v);
    }
}
```

2.6 Min Cost Max Flow

```
template <class T = int>
class MCMF {
public:
    struct Edge {
        Edge(int a, T b, T c) : to(a), cap(b), cost(c) {}
        int to;
        T cap, cost;
    };
};
```

```

MCMF(int size) {
    n = size;
    edges.resize(n);
    pot.assign(n, 0);
    dist.resize(n);
    visit.assign(n, false);
}

pair<T, T> mcmf(int src, int sink) {
    pair<T, T> ans(0, 0);
    if(!SPFA(src, sink)) return ans;
    fixPot();
    // can use dijkstra to speed up depending on the graph
    while(SPFA(src, sink)) {
        auto flow = augment(src, sink);
        ans.first += flow.first;
        ans.second += flow.first * flow.second;
        fixPot();
    }
    return ans;
}

void addEdge(int from, int to, T cap, T cost) {
    edges[from].push_back(list.size());
    list.push_back(Edge(to, cap, cost));
    edges[to].push_back(list.size());
    list.push_back(Edge(from, 0, -cost));
}

private:
int n;
vector<vector<int>> edges;
vector<Edge> list;
vector<int> from;
vector<T> dist, pot;
vector<bool> visit;
pair<T, T> augment(int src, int sink) {
    pair<T, T> flow = {list[from[sink]].cap, 0};
    for(int v = sink; v != src; v = list[from[v]^1].to) {
        flow.first = min(flow.first, list[from[v]].cap);
        flow.second += list[from[v]].cost;
    }
    for(int v = sink; v != src; v = list[from[v]^1].to) {
        list[from[v]].cap -= flow.first;
        list[from[v]^1].cap += flow.first;
    }
    return flow;
}

queue<int> q;
bool SPFA(int src, int sink) {
    T INF = numeric_limits<T>::max();
    dist.assign(n, INF);
    from.assign(n, -1);
    q.push(src);
    dist[src] = 0;
    while(!q.empty()) {
        int on = q.front();
        q.pop();
        visit[on] = false;
        for(auto e : edges[on]) {
            auto ed = list[e];
            if(ed.cap == 0) continue;
            T toDist = dist[on] + ed.cost + pot[on] - pot[ed.to];
            if(toDist < dist[ed.to]) {
                dist[ed.to] = toDist;
                from[ed.to] = e;
                if(!visit[ed.to]) {
                    visit[ed.to] = true;
                    q.push(ed.to);
                }
            }
        }
    }
    return dist[sink] < INF;
}

void fixPot() {
    T INF = numeric_limits<T>::max();
    for(int i = 0; i < n; i++) {
        if(dist[i] < INF) pot[i] += dist[i];
    }
}
};

```

2.7 Euler Path and Circuit

```

int pathV[me], szV, del[me], pathE, szE;
int adj[ms], to[me], ant[me], wt[me], z, n;

// Funcao de add e clear no dinic

void eulerPath(int u) {
    for(int i = adj[u]; ~i; i = ant[u]) if(!del[i]) {
        del[i] = del[i^1] = 1;
        eulerPath(to[i]);
        pathE[szE++] = i;
    }
    pathV[szV++] = u;
}

```

2.8 Articulation Points/Bridges/Biconnected Components

```

int adj[ms], to[me], ant[me], z;
int num[ms], low[ms], timer;
bool art[ms], bridge[me], f[me];
int bc[ms], nbc;
stack<int> st, stk;
vector<vector<int>> comps;

void clear() { // Lembrar de chamar no main
    memset(adj, -1, sizeof adj);
    z = 0;
}

void add(int u, int v) {
    to[z] = v;
    ant[z] = adj[u];
    adj[u] = z++;
}

void generateBc (int v) {
    while (!st.empty()) {
        int u = st.top();
        st.pop();
        bc[u] = nbc;
        if (v == u) break;
    }
    ++nbc;
}

void dfs (int v, int p) {
    st.push(v), stk.push(v);
    low[v] = num[v] = ++timer;
    for (int i = adj[v]; i != -1; i = ant[i]) {
        if (f[i] || f[i^1]) continue;
        f[i] = 1;
        int u = to[i];
        if (num[u] == -1) {
            dfs(u, v);
            low[v] = min(low[v], low[u]);
            if (low[u] > num[v]) bridge[i] = bridge[i^1] = 1;
            if (low[u] >= num[v]) {
                art[v] = (num[v] > 1 || num[u] > 2);
                comps.push_back({v});
                while (comps.back().back() != u)
                    comps.back().push_back(stk.top()), stk.pop();
            }
        } else {
            low[v] = min(low[v], num[u]);
        }
    }
    if (low[v] == num[v]) generateBc(v);
}

void biCon (int n) {
    nbc = 0, timer = 0;
    memset(num, -1, sizeof num);
    memset(bc, -1, sizeof bc);
    memset(bridge, 0, sizeof bridge);
    memset(art, 0, sizeof art);
    memset(f, 0, sizeof f);
    for (int i = 0; i < n; i++) {

```



```

    if (num[i] == -1) {
        timer = 0;
        dfs(i, 0);
    }
}

vector<int> g[ms];
int id[ms];
void buildBlockCut (int n) {
    int z = 0;
    for (int u = 0; u < n; ++u) {
        if (art[u]) id[u] = z++;
    }
    for (auto &comp : comps) {
        int node = z++;
        for (int u : comp) {
            if (!art[u]) id[u] = node;
            else {
                g[node].push_back(id[u]);
                g[id[u]].push_back(node);
            }
        }
    }
}
}

```

2.9 SCC - Strongly Connected Components / 2SAT

```

const int ms = 212345;

vector<int> g[ms];
int idx[ms], low[ms], z, comp[ms], ncomp;
stack<int> st;

int dfs(int u) {
    if(!idx[u]) return idx[u] ? idx[u] : z;
    low[u] = idx[u] = z++;
    st.push(u);
    for(int v : g[u]) {
        low[u] = min(low[u], dfs(v));
    }
    if(low[u] == idx[u]) {
        while(st.top() != u) {
            int v = st.top();
            idx[v] = 0;
            low[v] = low[u];
            comp[v] = ncomp;
            st.pop();
        }
        idx[st.top()] = 0;
        st.pop();
        comp[u] = ncomp++;
    }
    return low[u];
}

bool solveSat(int n) {
    memset(idx, -1, sizeof idx);
    z = 1; ncomp = 0;
    for(int i = 0; i < 2*n; i++) dfs(i);
    for(int i = 0; i < 2*n; i++) if(comp[i] == comp[i^1]) return false;
    return true;
}

int trad(int v) { return v < 0 ? (~v)*2^1 : v * 2; }
void add(int a, int b) { g[trad(a)].push_back(trad(b)); }
void addOr(int a, int b) { add(~a, b); add(~b, a); }
void addImp(int a, int b) { addOr(~a, b); }
void addEqual(int a, int b) { addOr(a, ~b); addOr(~a, b); }
void addDiff(int a, int b) { addEqual(a, ~b); }
// value[i] = comp[trad(i)] < comp[trad(~id)];

```

2.10 LCA - Lowest Common Ancestor

```

int par[ms][mlg+1], lvl[ms];
vector<int> g[ms];

```

```

void dfs(int v, int p, int l = 0) { // chamar como dfs(root, root)
    lvl[v] = l;
    par[v][0] = p;
    for(int k = 1; k <= mlg; k++) {
        par[v][k] = par[par[v][k-1]][k-1];
    }
    for(int u : g[v]) {
        if(u != p) dfs(u, v, l + 1);
    }
}

int lca(int a, int b) {
    if(lvl[b] > lvl[a]) swap(a, b);
    for(int i = mlg; i >= 0; i--) {
        if(lvl[a] - (1 << i) >= lvl[b]) a = par[a][i];
    }
    if(a == b) return a;
    for(int i = mlg; i >= 0; i--) {
        if(par[a][i] != par[b][i]) a = par[a][i], b = par[b][i];
    }
    return par[a][0];
}

```

2.11 Heavy Light Decomposition

```

class HLD {
public:
    void init(int n) { /* resize everything */ }
    void addEdge(int u, int v) {
        edges[u].push_back(v);
        edges[v].push_back(u);
    }
    void setRoot(int r) {
        t = 0;
        p[r] = r;
        h[r] = 0;
        prep(r, r);
        nxt[r] = r;
        hld(r);
    }
    int getLCA(int u, int v) {
        while(!inSubtree(nxt[u], v)) u = p[nxt[u]];
        while(!inSubtree(nxt[v], u)) v = p[nxt[v]];
        return in[u] < in[v] ? u : v;
    }
    // is v in the subtree of u?
    bool inSubtree(int u, int v) {
        return in[u] <= in[v] && in[v] < out[u];
    }
    // returns ranges [l, r) that the path has
    vector<pair<int, int>> getPath(int u, int anc) {
        vector<std::pair<int, int>> ans;
        //assert(inSubtree(anc, u));
        while(nxt[u] != nxt[anc]) {
            ans.emplace_back(in[nxt[u]], in[u] + 1);
            u = p[nxt[u]];
        }
        // this includes the ancestor! care
        ans.emplace_back(in[anc], in[u] + 1);
        return ans;
    }
private:
    vector<int> in, out, p, rin, sz, nxt, h;
    vector<vector<int>> edges;
    int t;
    void prep(int on, int par) {
        sz[on] = 1;
        p[on] = par;
        for(int i = 0; i < (int) edges[on].size(); i++) {
            int &u = edges[on][i];
            if(u == par) {
                swap(u, edges[on].back());
                edges[on].pop_back();
            }
            i--;
        }
        h[on] = 1 + h[on];
        prep(u, on);
        sz[on] += sz[u];
        if(sz[u] > sz[edges[on][0]]) {

```

```

        swap(edges[on][0], u);
    }
}
void hld(int on) {
    in[on] = t++;
    rin[in[on]] = on;
    for(auto u : edges[on]) {
        nxt[u] = (u == edges[on][0] ? nxt[on] : u);
        hld(u);
    }
    out[on] = t;
}
};

```

2.12 Centroid Decomposition

```

template<typename T>
struct CentroidDecomposition {
    vector<int> sz, h, dad;
    vector<vector<pair<int, T>>> adj;
    vector<vector<T>> dis;
    vector<bool> removed;

    CentroidDecomposition (int n) {
        sz.resize(n);
        h.resize(n);
        dis.resize(n, vector<T>(30, 0));
        adj.resize(n);
        removed.resize(n, 0);
        dad.resize(n);
    }

    void add (int a, int b, T w = 1) {
        adj[a].push_back({b, w});
        adj[b].push_back({a, w});
    }

    void dfsSize (int v, int par){
        sz[v] = 1;
        for (auto u : adj[v]){
            if (u.x == par || removed[u.x]) continue;
            dfsSize(u.x, v);
            sz[v] += sz[u.x];
        }
    }

    int getCentroid (int v, int par, int tam){
        for (auto u : adj[v]) {
            if (u.x == par || removed[u.x]) continue;
            if ((sz[u.x]<<1) > tam) return getCentroid(u.x, v, tam);
        }
        return v;
    }

    void setDis (int v, int par, int nv){
        for (auto u : adj[v]) {
            if (u.x == par || removed[u.x]) continue;
            dis[u.x][nv] = dis[v][nv]+u.y;
            setDis(u.x, v, nv);
        }
    }

    void decompose (int v, int par = -1, int nv = 0){
        dfsSize(v, par);
        int c = getCentroid(v, par, sz[v]);
        dad[c] = par;
        removed[c] = 1;
        h[c] = nv;
        setDis(c, par, nv);
        for (auto u : adj[c]){
            if (!removed[u.x]){
                decompose(u.x, c, nv + 1);
            }
        }
    }

    int operator [] (const int idx) const {
        return dad[idx];
    }
};

```

```

    }

    T dist (int u, int v) {
        if (h[u] < h[v]) swap(u, v);
        return dis[u][h[v]];
    }
};

```

2.13 Sack

```

void dfs(int v, int par = -1, bool keep = 0) {
    int big = -1;
    for (int u : adj[v]) {
        if (u == par) continue;
        if (big == -1 || sz[u] > sz[big]) {
            big = u;
        }
    }
    for (int u : adj[v]) {
        if (u == par || u == big) {
            continue;
        }
        dfs(u, v, 0);
    }
    if (big != -1) {
        dfs(big, v, 1);
    }
    for (int u : adj[v]) {
        if (u == par || u == big) {
            continue;
        }
        put(u, v);
    }
    if (!keep) {
    }
}

```

2.14 Hungarian Algorithm - Maximum Cost Matching

```

//input: matrix n x m, n <= m
//return vector p of size n, where p[i] is the match for i
// and minimum cost
// time complexity: O(n^2 * m)

int u[ms], v[ms], p[ms], way[ms], minv[ms];
bool used[ms];

pair<vector<int>, int> solve(const vector<vector<int>> &matrix) {
    int n = matrix.size();
    if(n == 0) return {vector<int>(), 0};
    int m = matrix[0].size();
    assert(n <= m);
    memset(u, 0, (n+1)*sizeof(int));
    memset(v, 0, (m+1)*sizeof(int));
    memset(p, 0, (m+1)*sizeof(int));
    for(int i = 1; i <= n; i++) {
        memset(minv, 0x3f, (m+1)*sizeof(int));
        memset(way, 0, (m+1)*sizeof(int));
        for(int j = 0; j <= m; j++) used[j] = 0;
        p[0] = i;
        int k0 = 0;
        do {
            used[k0] = 1;
            int i0 = p[k0], delta = inf, k1;
            for(int j = 1; j <= m; j++) {
                if(!used[j]) {
                    int cur = matrix[i0-1][j-1] - u[i0] - v[j];
                    if (cur < minv[j]) {
                        minv[j] = cur;
                        way[j] = k0;
                    }
                    if(minv[j] < delta) {
                        delta = minv[j];
                        k1 = j;
                    }
                }
            }
        } while (delta < inf);
        for(int j = 1; j <= m; j++) {
            if(used[j]) {
                u[p[j-1]] += delta;
                v[j] -= delta;
            } else {
                way[j] = k0;
            }
        }
        k0 = k1;
    }
    return {p, 0};
}

```

```

    }
}
for(int j = 0; j <= m; j++) {
    if(used[j]) {
        u[p[j]] += delta;
        v[j] -= delta;
    } else {
        minv[j] -= delta;
    }
}
k0 = k1;
} while(p[k0]);
do {
    int k1 = way[k0];
    p[k0] = p[k1];
    k0 = k1;
} while(k0);
}
vector<int> ans(n, -1);
for(int j = 1; j <= m; j++) {
    if(!p[j]) continue;
    ans[p[j] - 1] = j - 1;
}
return {ans, -v[0]};
}

```

2.15 Burunduk

```

struct edge {
    int a, b, l, r;
};

typedef vector<edge> List;

int cnt[N + 1], ans[N], u[N], color[N], deg[N];
vi g[N];

void add (int a, int b) {
    g[a].pb(b), g[b].pb(a);
}

void dfs (int v, int value) {
    u[v] = 1, color[v] = value;
    forn(i, sz(g[v]))
        if (!u[g[v][i]])
            dfs(g[v][i], value);
}

int compress (List &vl, int vn, int &add_vn) {
    int vn1 = 0;
    forn (i, vn) u[i] = 0;
    forn (i, vn) {
        if (!u[i]) deg[vn1] = 0, dfs(i, vn1++);
    }
    forn (i, sz(vl)) {
        vl[i].a = color[vl[i].a];
        vl[i].b = color[vl[i].b];
        if (vl[i].a != vl[i].b)
            deg[vl[i].a]++, deg[vl[i].b]++;
    }
    vn = vn1, vn1 = 0;
    forn (i, vn) {
        u[i] = vn1, vn1 += (deg[i] > 0), add_vn += !deg[i];
    }
    forn (i, sz(vl)) {
        vl[i].a = u[vl[i].a];
        vl[i].b = u[vl[i].b];
    }
    return vn1;
}

void go (int l, int r, const List &v, int vn, int add_vn) {
    if (cnt[l] == cnt[r]) return;
    if (!sz(v)) {
        while (l < r)
            ans[l++] = vn + add_vn;
        return;
    }
}

```

```

List vl;
forn (i, vn) {
    g[i].clear();
}
forn (i, sz(v)) {
    if (v[i].a != v[i].b) {
        if (v[i].l <= l && v[i].r >= r)
            add(v[i].a, v[i].b);
        else if (l < v[i].r && r > v[i].l)
            vl.pb(v[i]);
    }
}
int vn1 = compress(vl, vn, add_vn);
int m = (l + r) / 2;
go(l, m, vl, vn1, add_vn);
go(m, r, vl, vn1, add_vn);
}

```

2.16 Minimum Arborescence

```

// uncommented O(V^2) arborescence
struct Edges {
    //set<pair<long long, int>> cost; O(Elog^2)
    long long cost[ms];

    // possible optimization, use vector of size n
    // instead of ms
    long long sum = 0;

    Edges() {
        memset(cost, 0x3f, sizeof cost);
    }

    void addEdge(int u, long long c) {
        // cost.insert({c - sum, u}); O(Elog^2)
        cost[u] = min(cost[u], c - sum);
    }

    pair<long long, int> getMin() {
        //return *cost.begin(); O(E*log^2)
        pair<long long, int> ans(cost[0], 0);
        // in this loop can change ms to n to make it faster for many cases
        for(int i = 1; i < ms; i++) {
            if(cost[i] < ans.first) {
                ans = pair<long long, int>(cost[i], i);
            }
        }
        return ans;
    }

    void unite(Edges &e) {
        /*
        O(E*log^2E)
        if(e.cost.size() > cost.size()) {
            cost.swap(e.cost);
            swap(sum, e.sum);
        }
        for(auto i : e.cost) {
            addEdge(i.second, i.first + e.sum);
        }
        e.cost.clear();
        */

        // O(V^2)
        // can change ms to n
        for(int i = 0; i < ms; i++) {
            cost[i] = min(cost[i], e.cost[i] + e.sum - sum);
        }
    }
};

typedef vector<vector<pair<long long, int>>> Graph;

Edges ed[ms];
int par[ms];
long long best[ms];
int col[ms];
int getPar(int x) { return par[x] < 0 ? x : par[x] = getPar(par[x]); }
void makeUnion(int a, int b) {
    a = getPar(a);
}

```

```

b = getPar(b);
if(a == b) return;
ed[a].unite(ed[b]);
par[b] = a;
}

long long arborescence(Graph edges) {
    // root is 0
    // edges has transposed adjacency list (cost, from)
    // edge from i to j cost c is
    // edge[j].emplace_back(c, i)
    int n = (int) edges.size();
    long long ans = 0;
    for(int i = 0; i < n; i++) {
        ed[i] = Edges();
        par[i] = -1;
        for(auto j : edges[i]) {
            ed[i].addEdge(j.second, j.first);
        }
        col[i] = 0;
    }
    // to change the root you can simply change this next line to
    // col[root] = 2;
    col[0] = 2;
    for(int i = 0; i < n; i++) {
        if(col[getPar(i)] == 2) {
            continue;
        }
        int on = getPar(i);
        vector<int> st;
        while(col[on] != 2) {
            assert(getPar(on) == on);
            if(col[on] == 1) {
                // found cycle
                int v = on;
                vector<int> cycle;
                //cout << "found cycle\n";
                while(st.back() != v) {
                    //cout << st.back() << endl;
                    cycle.push_back(st.back());
                    st.pop_back();
                }
                // compress cycle
                for(auto u : cycle) {
                    makeUnion(v, u);
                }
                v = getPar(v);
                col[v] = 0;
                on = v;
            } else {
                // still no cycle
                // while best is in compressed cycle, remove
                /*
                THIS IS TO MAKE O(E*log^2) ALGORITHM!!
                while(!ed[on].cost.empty() && getPar(on) == getPar(ed[on].getMin().second)) {
                    ed[on].cost.erase(ed[on].cost.begin());
                }
                */

                // O(V^2)
                for(int x = 0; x < n; x++) {
                    if(on == getPar(x)) {
                        ed[on].cost[x] = 0x3f3f3f3f3f3f3fLL;
                    }
                }

                // best edge
                auto e = ed[on].getMin();
                // O(E*log^2) assert(!ed[on].cost.empty()) if every vertex appears in the
                // arborescence
                // O(V^2)
                assert(e.first < 0x3f3f3f3f3f3f3fLL);
                int v = getPar(e.second);
                //cout << "found not cycle to " << v << " of cost " << e.first + ed[on].sum <<
                // '\n';
                assert(v != on);
                best[on] = e.first + ed[on].sum;
                ans += best[on];
                // compress edges
                ed[on].sum = -(e.first);
                st.push_back(on);
                col[on] = 1;
                on = v;
            }
        }
    }
}

```

```

    }
    // make everything 2
    for(auto u : st) {
        assert(getPar(u) == u);
        col[u] = 2;
    }
    return ans;
}

```

3 Dynamic Programming

3.1 Line Container

```

typedef long long int ll;

bool Q;
struct Line {
    mutable ll k, m, p;
    bool operator<(const Line& o) const {
        return Q ? p < o.p : k < o.k;
    }
};

struct LineContainer : multiset<Line> {
    // (for doubles, use inf = 1/.0, div(a,b) = a/b)
    const ll inf = LLONG_MAX;
    ll div(ll a, ll b) { // floored division
        return a / b - ((a ^ b) < 0 && a % b);
    }
    bool isect(iterator x, iterator y) {
        if (y == end()) { x->p = inf; return false; }
        if (x->k == y->k) x->p = x->m > y->m ? inf : -inf;
        else x->p = div(y->m - x->m, x->k - y->k);
        return x->p >= y->p;
    }
    void add(ll k, ll m) {
        auto z = insert({k, m, 0}); y = z++, x = y;
        while (isect(y, z)) z = erase(z);
        if (x != begin() && isect(--x, y)) isect(x, y = erase(y));
        while ((y = x) != begin() && (--x)->p >= y->p)
            isect(x, erase(y));
    }
    ll query(ll x) {
        assert(!empty());
        Q = 1; auto l = *lower_bound({0,0,x}); Q = 0;
        return l.k * x + l.m;
    }
};

```

3.2 Li Chao Tree

```

// by luucasv
typedef long long T;
const T INF = 1e18, EPS = 1;
const int BUFFER_SIZE = 1e4;

struct Line {
    T m, b;

    Line(T m = 0, T b = INF) : m(m), b(b) {}
    T apply(T x) { return x * m + b; }
};

struct Node {
    Node *left, *right;
    Line line;
    Node() : left(NULL), right(NULL) {}
};

struct LiChaoTree {
    Node *root, buffer[BUFFER_SIZE];
    T min_value, max_value;
    int buffer_pointer;
};

```

```

LiChaoTree(T min_value, T max_value): min_value(min_value), max_value(max_value + 1)
{
    clear();
}

void clear() { buffer_pointer = 0; root = newNode(); }

void insert_line(T m, T b) { update(root, min_value, max_value, Line(m, b)); }

T eval(T x) { return query(root, min_value, max_value, x); }

void update(Node *cur, T l, T r, Line line) {
    T m = l + (r - l) / 2;
    bool left = line.apply(l) < cur->line.apply(l);
    bool mid = line.apply(m) < cur->line.apply(m);
    bool right = line.apply(r) < cur->line.apply(r);
    if (mid) {
        swap(cur->line, line);
    }
    if (r - l <= EPS) return;
    if (left == right) return;
    if (mid != left) {
        if (cur->left == NULL) cur->left = newNode();
        update(cur->left, l, m, line);
    } else {
        if (cur->right == NULL) cur->right = newNode();
        update(cur->right, m, r, line);
    }
}

T query(Node *cur, T l, T r, T x) {
    if (cur == NULL) return INF;
    if (r - l <= EPS) {
        return cur->line.apply(x);
    }
    T m = l + (r - l) / 2;
    T ans;
    if (x < m) {
        ans = query(cur->left, l, m, x);
    } else {
        ans = query(cur->right, m, r, x);
    }
    return min(ans, cur->line.apply(x));
}

Node* newNode() {
    Node* node = new Node();
    buffer[buffer_pointer] = node;
    return &buffer[buffer_pointer++];
}
};

```

3.3 Divide and Conquer Optimization

```

int n, k;
ll dpold[ms], dp[ms], c[ms][ms]; // c(i, j) pode ser funcao

void compute(int l, int r, int optl, int opttr) {
    if (l > r) return;
    int mid = (l+r)/2;
    pair<ll, int> best = {inf, -1}; // long long inf
    for (int k = optl; k <= min(mid, opttr); k++) {
        best = min(best, {dpold[k-1] + c[k][mid], k});
    }
    dp[mid] = best.first;
    int opt = best.second;
    compute(l, mid-1, optl, opt);
    compute(mid+1, r, opt, opttr);
}

ll solve() {
    dp[0] = 0;
    for (int i = 1; i <= n; i++) dp[i] = inf; // initialize row 0 of the dp
    for (int i = 1; i <= k; i++) {
        swap(dpold, dp);
        compute(0, n, 0, n); // solve row i of the dp
    }
    return dp[n]; // return dp[k][n]
}

```

3.4 Knuth Optimization

```

int n, m, mid[ms][ms];
ll dp[ms][ms];

```

```

void knuth() {
    for (int i = n; i >= 0; i--) { // limites entre 0 e n
        dp[i][i+1] = 0; mid[i][i+1] = i; // caso base
        for (int j = i+2; j <= n; j++) {
            dp[i][j] = inf; // long long inf
            for (int k = mid[i][j-1]; k <= mid[i+1][j]; k++) {
                if (dp[i][j] > dp[i][k] + dp[k][j]) {
                    dp[i][j] = dp[i][k] + dp[k][j];
                    mid[i][j] = k;
                }
            }
            dp[i][j] += c(i, j); // custo associado ao intervalo
        }
    }
}

```

4 Math

4.1 Chinese Remainder Theorem

```

long long modinverse(long long a, long long b, long long s0 = 1, long long s1 = 0) {
    if (!b) return s0;
    else return modinverse(b, a % b, s1, s0 - s1 * (a / b));
}

long long gcd(long long a, long long b) {
    if (!b) return a;
    else return gcd(b, a % b);
}

ll mul(ll a, ll b, ll m) {
    ll q = (long double) a * (long double) b / (long double) m;
    ll r = a * b - q * m;
    return (r + 5 * m) % m;
}

long long safemod(long long a, long long m) {
    return (a % m + m) % m;
}

struct equation {
    equation(long long a, long long m) { mod = m, ans = a, valid = true; }
    equation() { valid = false; }
    equation(equation a, equation b) {
        if (!a.valid || !b.valid) {
            valid = false;
            return;
        }
        long long g = gcd(a.mod, b.mod);
        if ((a.ans - b.ans) % g != 0) {
            valid = false;
            return;
        }
        valid = true;
        mod = a.mod * (b.mod / g);
        ans = a.ans +
            mul(a.mod, modinverse(a.mod, b.mod), mod),
            (b.ans - a.ans) / g,
            mod;
        ans = safemod(ans, mod);
    }
    long long mod, ans;
    bool valid;

    void print() {
        if (!valid)
            std::cout << "equation is not valid\n";
        else
            std::cout << "equation is " << ans << " mod " << mod << '\n';
    }
};

```

4.2 Diophantine Equations

```
int gcd_ext(int a, int b, int& x, int& y) {
    if (b == 0) {
        x = 1, y = 0;
        return a;
    }
    int nx, ny;
    int gc = gcd_ext(b, a % b, nx, ny);
    x = ny;
    y = nx - (a / b) * ny;
    return gc;
}

vector<int> diophantine(int D, vector<int> l) {
    int n = l.size();
    vector<int> gc(n), ans(n);
    gc[n - 1] = l[n - 1];
    for (int i = n - 2; i >= 0; i--) {
        int x, y;
        gc[i] = gcd_ext(l[i], gc[i + 1], x, y);
    }
    if (D % gc[0] != 0) {
        return vector<int>();
    }
    for (int i = 0; i < n; i++) {
        if (i == n - 1) {
            ans[i] = D / l[i];
            D -= l[i] * ans[i];
            continue;
        }
        int x, y;
        gcd_ext(l[i] / gc[i], gc[i + 1] / gc[i], x, y);
        ans[i] = (long long int) D / gc[i] * x % (gc[i + 1] / gc[i]);
        if (D < 0 && ans[i] > 0) {
            ans[i] -= (gc[i + 1] / gc[i]);
        }
        if (D > 0 && ans[i] < 0) {
            ans[i] += (gc[i + 1] / gc[i]);
        }
        D -= l[i] * ans[i];
    }
    return ans;
}
```

4.3 Discrete Logarithm

```
ll discreteLog(ll a, ll b, ll m) {
    a %= m; b %= m;
    ll n = (ll) sqrt(m + .0) + 1, an = 1;
    for (ll i = 0; i < n; i++) {
        an = (an * a) % m;
    }
    map<ll, ll> vals;
    for (ll i = 1, cur = an; i <= n; i++) {
        if (!vals.count(cur)) vals[cur] = i;
        cur = (cur * an) % m;
    }
    ll ans = 1e18; //inf
    for (ll i = 0, cur = b; i <= n; i++) {
        if (vals.count(cur)) {
            ans = min(ans, vals[cur] * n - i);
        }
        cur = (cur * a) % m;
    }
    return ans;
}
```

4.4 Discrete Root

```
//x^k = a % mod
ll discreteRoot(ll k, ll a, ll mod) {
    ll g = primitiveRoot(mod);
    ll y = discreteLog(fexp(g, k, mod), a, mod);
    if (y == -1) {
        return y;
    }
    return fexp(g, y, mod);
}
```

4.5 Division Trick

```
for(int l = 1, r; l <= n; l = r + 1) {
    r = n / (n / l);
    // n / i has the same value for l <= i <= r
}
```

4.6 Modular Sum

```
//calcula (sum(0 <= i <= n) P(i)) % mod,
//onde P(i) eh uma PA modular (com outro modulo)
namespace sum_pa_mod{
    ll calc(ll a, ll b, ll n, ll mod){
        assert(a < b);
        if(a >= b){
            ll ret = ((n*(n+1)/2)%mod)*(a/b);
            if(a%b) ret = (ret + calc(a%b, b, n, mod))%mod;
            else ret = (ret+n+1)%mod;
            return ret;
        }
        return ((n+1)*(((n*a)/b+1)%mod) - calc(b, a, (n*a)/b, mod) + mod + n/b + 1)%mod;
    }
    //P(i) = a*i mod m
    ll solve(ll a, ll n, ll m, ll mod){
        a = (a%m + m)%m;
        if(!a) return 0;
        ll ret = (n*(n+1)/2)%mod;
        ret = (ret*a)%mod;
        ll g = __gcd(a, m);
        ret -= m*(calc(a/g, m/g, n, mod)-n-1);
        return (ret%mod + mod)%mod;
    }
    //P(i) = a + r*i mod m
    ll solve(ll a, ll r, ll n, ll m, ll mod){
        a = (a%m + m)%m;
        r = (r%m + m)%m;
        if(!r) return (a*(n+1))%mod;
        if(!a) return solve(r, n, m, mod);
        ll g, x, y;
        g = gcdExtended(r, m, x, y);
        x = (x%m + m)%m;
        ll d = a - (a/g)*g;
        a -= d;
        x = (x*(a/g))%m;
        return (solve(r, n+x, m, mod) - solve(r, x-1, m, mod) + mod + d*(n+1))%mod;
    }
};
```

4.7 Primitive Root

```
int primitiveRoot(int p) {
    vector<int> fact;
    int phi = p - 1, n = phi;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            fact.push_back(i);
            while (n % i == 0) {
                n /= i;
            }
        }
    }
    if (n > 1) {
        fact.push_back(n);
    }
    for (int res = 2; res <= p; res++) {
        bool ok = true;
        for (auto it : fact) {
            ok &= fexp(res, phi / it, p) != 1;
            if (!ok) {
                break;
            }
        }
        if (ok) {
            return res;
        }
    }
}
```

```

    }
    return -1;
}

```

4.8 Extended Euclides

```

// euclides estendido: acha u e v da equacao:
// u * x + v * y = gcd(x, y);
// u eh inverso modular de x no modulo y
// v eh inverso modular de y no modulo x

pair<ll, ll> euclides(ll a, ll b) {
    ll u = 0, oldu = 1, v = 1, oldv = 0;
    while(b) {
        ll q = a / b;
        oldv = oldv - v * q;
        oldu = oldu - u * q;
        a = a - b * q;
        swap(a, b);
        swap(u, oldu);
        swap(v, oldv);
    }
    return make_pair(oldu, oldv);
}

```

4.9 Matrix Fast Exponentiation

```

const ll mod = 1e9+7;
const int m = 2; // size of matrix

struct Matrix {
    ll mat[m][m];
    Matrix operator * (const Matrix &p) {
        Matrix ans;
        for(int i = 0; i < m; i++)
            for(int j = 0; j < m; j++)
                for(int k = 0; k < m; k++)
                    ans.mat[i][j] = (ans.mat[i][j] + mat[i][k] * p.mat[k][j]) % mod;
        return ans;
    }
};

Matrix fExp(Matrix a, ll b) {
    Matrix ans;
    for(int i = 0; i < m; i++) for(int j = 0; j < m; j++)
        ans.mat[i][j] = i == j;
    while(b) {
        if(b & 1) ans = ans * a;
        a = a * a;
        b >>= 1;
    }
    return ans;
}

```

4.10 FFT - Fast Fourier Transform

```

typedef double ld;

const ld PI = acos(-1);

struct Complex {
    ld real, imag;
    Complex conj() { return Complex(real, -imag); }
    Complex(ld a = 0, ld b = 0) : real(a), imag(b) {}
    Complex operator + (const Complex &o) const { return Complex(real + o.real, imag + o.imag); }
    Complex operator - (const Complex &o) const { return Complex(real - o.real, imag - o.imag); }
    Complex operator * (const Complex &o) const { return Complex(real * o.real - imag * o.imag, real * o.imag + imag * o.real); }
    Complex operator / (ld o) const { return Complex(real / o, imag / o); }
    void operator *= (Complex o) { *this = *this * o; }
    void operator /= (ld o) { real /= o, imag /= o; }
}

```

```

};

typedef std::vector<Complex> CVector;

const int ms = 1 << 22;

int bits[ms];
Complex root[ms];

void initFFT() {
    root[1] = Complex(1);
    for(int len = 2; len < ms; len += len) {
        Complex z(cos(PI / len), sin(PI / len));
        for(int i = len / 2; i < len; i++) {
            root[2 * i] = root[i];
            root[2 * i + 1] = root[i] * z;
        }
    }
}

void pre(int n) {
    int LOG = 0;
    while(1 << (LOG + 1) < n) {
        LOG++;
    }
    for(int i = 1; i < n; i++) {
        bits[i] = (bits[i >> 1] >> 1) | ((i & 1) << LOG);
    }
}

CVector fft(CVector a, bool inv = false) {
    int n = a.size();
    pre(n);
    if(inv) {
        std::reverse(a.begin() + 1, a.end());
    }
    for(int i = 0; i < n; i++) {
        int to = bits[i];
        if(to > i) {
            std::swap(a[to], a[i]);
        }
    }
    for(int len = 1; len < n; len *= 2) {
        for(int i = 0; i < n; i += 2 * len) {
            for(int j = 0; j < len; j++) {
                Complex u = a[i + j], v = a[i + j + len] * root[len + j];
                a[i + j] = u + v;
                a[i + j + len] = u - v;
            }
        }
    }
    if(inv) {
        for(int i = 0; i < n; i++)
            a[i] /= n;
    }
    return a;
}

void fft2in1(CVector &a, CVector &b) {
    int n = (int) a.size();
    for(int i = 0; i < n; i++) {
        a[i] = Complex(a[i].real, b[i].real);
    }
    auto c = fft(a);
    for(int i = 0; i < n; i++) {
        a[i] = (c[i] + c[(n-i) % n].conj()) * Complex(0.5, 0);
        b[i] = (c[i] - c[(n-i) % n].conj()) * Complex(0, -0.5);
    }
}

void ifft2in1(CVector &a, CVector &b) {
    int n = (int) a.size();
    for(int i = 0; i < n; i++) {
        a[i] = a[i] + b[i] * Complex(0, 1);
    }
    a = fft(a, true);
    for(int i = 0; i < n; i++) {
        b[i] = Complex(a[i].imag, 0);
        a[i] = Complex(a[i].real, 0);
    }
}

std::vector<long long> mod_mul(const std::vector<long long> &a, const std::vector<long

```

```

    long> &b, long long cut = 1 << 15) {
// TODO cut memory here by /2
int n = (int) a.size();
CVector C[4];
for(int i = 0; i < 4; i++) {
    C[i].resize(n);
}
for(int i = 0; i < n; i++) {
    C[0][i] = a[i] % cut;
    C[1][i] = a[i] / cut;
    C[2][i] = b[i] % cut;
    C[3][i] = b[i] / cut;
}
fft2in1(C[0], C[1]);
fft2in1(C[2], C[3]);
for(int i = 0; i < n; i++) {
    // 00, 01, 10, 11
    Complex cur[4];
    for(int j = 0; j < 4; j++) cur[j] = C[j/2+2][i] * C[j % 2][i];
    for(int j = 0; j < 4; j++) C[j][i] = cur[j];
}
ifft2in1(C[0], C[1]);
ifft2in1(C[2], C[3]);
std::vector<long long> ans(n, 0);
for(int i = 0; i < n; i++) {
    // if there are negative values, care with rounding
    ans[i] += (long long) (C[0][i].real + 0.5);
    ans[i] += (long long) (C[1][i].real + C[2][i].real + 0.5) * cut;
    ans[i] += (long long) (C[3][i].real + 0.5) * cut * cut;
}
return ans;
}

std::vector<int> mul(const std::vector<int> &a, const std::vector<int> &b) {
    int n = 1;
    while (n - 1 < (int) a.size() + (int) b.size() - 2) n += n;
    CVector poly(n);
    for(int i = 0; i < n; i++) {
        if(i < (int) a.size()) {
            poly[i].real = a[i];
        }
        if(i < (int) b.size()) {
            poly[i].imag = b[i];
        }
    }
    poly = fft(poly);
    for(int i = 0; i < n; i++) {
        poly[i] *= poly[i];
    }
    poly = fft(poly, true);
    std::vector<int> c(n, 0);
    for(int i = 0; i < n; i++) {
        c[i] = (int) (poly[i].imag / 2 + 0.5);
    }
    while (c.size() > 0 && c.back() == 0) c.pop_back();
    return c;
}

```

4.11 NTT - Number Theoretic Transform

```

const int MOD = 998244353;
const int me = 15;
const int ms = 1 << me;
ll fexp(ll x, ll e, ll mod = MOD) {
    ll ans = 1;
    x %= mod;
    for(; e > 0; e /= 2) {
        if(e & 1) {
            ans = ans * x % mod;
        }
        x = x * x % mod;
    }
    return ans;
}
//is n primitive root of p ?
bool test(ll x, ll p) {
    ll m = p - 1;
    for(int i = 2; i * i <= m; ++i) if(!(m % i)) {
        if(fexp(x, i, p) == 1) return false;
        if(fexp(x, m / i, p) == 1) return false;
    }
}

```

```

    }
    return true;
}
//find the largest primitive root for p
int search(int p) {
    for(int i = p - 1; i >= 2; --i) if(test(i, p)) return i;
    return -1;
}
map<int, int> roots;
int get_root(int p) {
    if(roots[p]) {
        return roots[p];
    } else {
        roots[p] = search(p);
        return roots[p];
    }
}

#define add(x, y) x+=y>=MOD?x+y-MOD:x+y

const int gen = search(MOD);
int bits[ms], root[ms];

void initFFT() {
    root[1] = 1;
    for(int len = 2; len < ms; len += len) {
        int z = fexp(gen, (MOD - 1) / len / 2);
        for(int i = len / 2; i < len; i++) {
            root[2 * i] = root[i];
            root[2 * i + 1] = (long long) root[i] * z % MOD;
        }
    }
}

void pre(int n) {
    int LOG = 0;
    while(1 << (LOG + 1) < n) {
        LOG++;
    }
    for(int i = 1; i < n; i++) {
        bits[i] = (bits[i >> 1] >> 1) | ((i & 1) << LOG);
    }
}

vector<int> fft(vector<int> a, int mod, bool inv = false) {
    int n = (int) a.size();
    pre(n);
    if(inv) {
        reverse(a.begin() + 1, a.end());
    }
    for(int i = 0; i < n; i++) {
        int to = bits[i];
        if(i < to)
            swap(a[i], a[to]);
    }
    for(int len = 1; len < n; len *= 2) {
        for(int i = 0; i < n; i += len * 2) {
            for(int j = 0; j < len; j++) {
                int u = a[i + j], v = (ll) a[i + j + len] * root[len + j] % mod;
                a[i + j] = add(u, v);
                a[i + j + len] = add(u, mod - v);
            }
        }
    }
    if(inv) {
        int rev = fexp(n, mod-2, mod);
        for(int i = 0; i < n; i++)
            a[i] = (ll) a[i] * rev % mod;
    }
    return a;
}

std::vector<int> shift(const std::vector<int> &a, int s) {
    int n = std::max(0, s + (int) a.size());
    std::vector<int> b(n, 0);
    for(int i = std::max(-s, 0); i < (int) a.size(); i++) {
        b[i + s] = a[i];
    }
    return b;
}

std::vector<int> cut(const std::vector<int> &a, int n) {
    std::vector<int> b(n, 0);
}

```



```

for(int i = 0; i < (int) a.size() && i < n; i++) {
    b[i] = a[i];
}
return b;
}

std::vector<int> operator +(std::vector<int> a, const std::vector<int> &b) {
    int sz = (int) std::max(a.size(), b.size());
    a.resize(sz, 0);
    for(int i = 0; i < (int) b.size(); i++) {
        a[i] = add(a[i], b[i]);
    }
    return a;
}

std::vector<int> operator -(std::vector<int> a, const std::vector<int> &b) {
    int sz = (int) std::max(a.size(), b.size());
    a.resize(sz, 0);
    for(int i = 0; i < (int) b.size(); i++) {
        a[i] = add(a[i], MOD - b[i]);
    }
    return a;
}

std::vector<int> operator *(std::vector<int> a, std::vector<int> b) {
    while(!a.empty() && a.back() == 0) a.pop_back();
    while(!b.empty() && b.back() == 0) b.pop_back();
    if(a.empty() || b.empty()) return std::vector<int>(0, 0);
    int n = 1;
    while(n-1 < (int) a.size() + (int) b.size() - 2) n += n;
    a.resize(n, 0);
    b.resize(n, 0);
    a = fft(a, false);
    b = fft(b, false);
    for(int i = 0; i < n; i++) {
        a[i] = (int) ((long long) a[i] * b[i] % MOD);
    }
    return fft(a, true);
}

std::vector<int> inverse(const std::vector<int> &a, int k) {
    assert(!a.empty() && a[0] != 0);
    if(k == 0) {
        return std::vector<int>(1, (int) fexp(a[0], MOD - 2));
    } else {
        int n = 1 << k;
        auto c = inverse(a, k-1);
        return cut(c * cut(std::vector<int>(1, 2) - cut(a, n) * c, n), n);
    }
}

std::vector<int> log(const std::vector<int> &a, int k) {
    assert(!a.empty() && a[0] != 0);
    int n = 1 << k;
    std::vector<int> b(n, 0);
    for(int i = 0; i+1 < (int) a.size() && i < n; i++) {
        b[i] = (int) ((i + 1LL) * a[i+1] % MOD);
    }
    b = cut(b * inverse(a, k), n);
    assert((int) b.size() == n);
    for(int i = n - 1; i > 0; i--) {
        b[i] = (int) (b[i-1] * fexp(i, MOD - 2) % MOD);
    }
    b[0] = 0;
    return b;
}

std::vector<int> exp(const std::vector<int> &a, int k) {
    assert(!a.empty() && a[0] == 0);
    if(k == 0) {
        return std::vector<int>(1, 1);
    } else {
        auto b = exp(a, k-1);
        int n = 1 << k;
        return cut(b * cut(std::vector<int>(1, 1) + cut(a, n) - log(b, k), n), n);
    }
}

```

```

vector<ll> FWHT(char oper, vector<ll> a, const bool inv = false) {
    int n = (int) a.size();
    for(int len = 1; len < n; len += len) {
        for(int i = 0; i < n; i += 2 * len) {
            for(int j = 0; j < len; j++) {
                auto u = a[i + j] % mod, v = a[i + j + len] % mod;
                if(oper == '^') {
                    a[i + j] = (u + v) % mod;
                    a[i + j + len] = (u - v + mod) % mod;
                }
                if(oper == '|') {
                    if(!inv) {
                        if(!inv) {
                            a[i + j + len] = (u + v) % mod;
                        } else {
                            a[i + j + len] = (v - u + mod) % mod;
                        }
                    }
                    if(oper == '&') {
                        if(!inv) {
                            a[i + j] = (u + v) % mod;
                        } else {
                            a[i + j] = (u - v + mod) % mod;
                        }
                    }
                }
            }
        }
    }
    if(oper == '^' && inv) {
        ll rev = fexp(n, mod - 2);
        for(int i = 0; i < n; i++) {
            a[i] = a[i] * rev % mod;
        }
    }
    return a;
}

vector<ll> multiply(char oper, vector<ll> a, vector<ll> b) {
    int n = 1;
    while (n < (int) max(a.size(), b.size())) {
        n <<= 1;
    }
    vector<ll> ans(n);
    while (a.size() < ans.size()) a.push_back(0);
    while (b.size() < ans.size()) b.push_back(0);
    a = FWHT(oper, a);
    b = FWHT(oper, b);
    for (int i = 0; i < n; i++) {
        ans[i] = a[i] * b[i] % mod;
    }
    ans = FWHT(oper, ans, true);
    return ans;
}

const int mxlog = 17;

vector<ll> subset_multiply(vector<ll> a, vector<ll> b) {
    int n = 1;
    while (n < (int) max(a.size(), b.size())) {
        n <<= 1;
    }
    vector<ll> ans(n);
    while (a.size() < ans.size()) a.push_back(0);
    while (b.size() < ans.size()) b.push_back(0);
    vector<vector<ll>> A(mxlog + 1, vector<ll>(a.size())), B(mxlog + 1, vector<ll>(b.size()));
    for (int i = 0; i < n; i++) {
        A[__builtin_popcount(i)][i] = a[i];
        B[__builtin_popcount(i)][i] = b[i];
    }
    for (int i = 0; i <= mxlog; i++) {
        A[i] = FWHT('|', A[i]);
        B[i] = FWHT('|', B[i]);
    }
    for (int i = 0; i <= mxlog; i++) {
        vector<ll> C(n);
        for (int x = 0; x <= i; x++) {
            int y = i - x;
            for (int j = 0; j < n; j++) {
                C[j] = (C[j] + A[x][j] * B[y][j] % mod) % mod;
            }
        }
        C = FWHT('|', C, true);
    }
    for (int j = 0; j < n; j++) {

```

4.12 Fast Walsh Hadamard Transform

```

        if (__builtin_popcount(j) == i) {
            ans[j] = (ans[j] + C[j]) % mod;
        }
    }
    return ans;
}

```

4.13 Miller and Rho

```

//miller_rabin
typedef unsigned long long ull;
typedef long double ld;

ull fmul(ull a, ull b, ull m) {
    ull q = (ld) a * (ld) b / (ld) m;
    ull r = a * b - q * m;
    return (r + m) % m;
}

bool miller(ull p, ull a) {
    ull s = p - 1;
    while(s % 2 == 0) s >>= 1;
    while(a >= p) a >>= 1;
    ull mod = fexp(a, s, p);
    while(s != p - 1 && mod != 1 && mod != p - 1) {
        mod = fmul(mod, mod, p);
        s <<= 1;
    }
    if(mod != p - 1 && s % 2 == 0) return false;
    else return true;
}

bool prime(ull p) {
    if(p <= 3)
        return true;
    if(p % 2 == 0)
        return false;
    return miller(p, 2) && miller(p, 3)
        && miller(p, 5) && miller(p, 7)
        && miller(p, 11) && miller(p, 13)
        && miller(p, 17) && miller(p, 19)
        && miller(p, 23) && miller(p, 29)
        && miller(p, 31) && miller(p, 37);
}

//pollard_rho
ull func(ull x, ull c, ull n) {
    return (fmul(x, x, n) + c) % n;
}

ull gcd(ull a, ull b) {
    if(!b) return a;
    else return gcd(b, a % b);
}

ull rho(ull n) {
    if(n % 2 == 0) return 2;
    if(prime(n)) return n;
    while(1) {
        ull c;
        do {
            c = rand() % n;
        } while(c == 0 || (c + 2) % n == 0);
        ull x = 2, y = 2, d = 1;
        ull pot = 1, lam = 1;
        do {
            if(pot == lam) {
                x = y;
                pot <<= 1;
                lam = 0;
            }
            y = func(y, c, n);
            lam++;
            d = gcd(x >= y ? x - y : y - x, n);
        } while(d == 1);
        if(d != n) return d;
    }
}

vector<ull> factors(ull n) {
    vector<ull> ans, rest, times;
    if(n == 1) return ans;
    rest.push_back(n);
    times.push_back(1);
    while(!rest.empty()) {
        ull x = rho(rest.back());
        if(x == rest.back()) {

```

```

            int freq = 0;
            for(int i = 0; i < rest.size(); i++) {
                int cur_freq = 0;
                while(rest[i] % x == 0) {
                    rest[i] /= x;
                    cur_freq++;
                }
                freq += cur_freq * times[i];
                if(rest[i] == 1) {
                    swap(rest[i], rest.back());
                    swap(times[i], times.back());
                    rest.pop_back();
                    times.pop_back();
                    i--;
                }
            }
            while(freq-- > 0) {
                ans.push_back(x);
            }
            continue;
        }
        ull e = 0;
        while(rest.back() % x == 0) {
            rest.back() /= x;
            e++;
        }
        e *= times.back();
        if(rest.back() == 1) {
            rest.pop_back();
            times.pop_back();
        }
        rest.push_back(x);
        times.push_back(e);
    }
    return ans;
}

```

4.14 Determinant using Mod

```

// by zchao1995
// Determinante com coordenadas inteiras usando Mod

ll mat[ms][ms];

ll det (int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            mat[i][j] %= mod;
        }
    }
    ll res = 1;
    for (int i = 0; i < n; i++) {
        if (!mat[i][i]) {
            bool flag = false;
            for (int j = i + 1; j < n; j++) {
                if (mat[j][i]) {
                    flag = true;
                    for (int k = i; k < n; k++) {
                        swap (mat[i][k], mat[j][k]);
                    }
                    res = -res;
                    break;
                }
            }
            if (!flag) {
                return 0;
            }
        }
        for (int j = i + 1; j < n; j++) {
            while (mat[j][i]) {
                ll t = mat[i][i] / mat[j][i];
                for (int k = i; k < n; k++) {
                    mat[i][k] = (mat[i][k] - t * mat[j][k]) % mod;
                    swap (mat[i][k], mat[j][k]);
                }
                res = -res;
            }
        }
        res = (res * mat[i][i]) % mod;
    }
    return (res + mod) % mod;
}

```

4.15 Lagrange Interpolation

```

class LagrangePoly {
public:
    LagrangePoly(std::vector<long long> _a) {
        //f(i) = _a[i]
        //interpola o vetor em um polinomio de grau y.size() - 1
        y = _a;
        den.resize(y.size());
        int n = (int) y.size();
        for(int i = 0; i < n; i++) {
            y[i] = (y[i] % MOD + MOD) % MOD;
            den[i] = ifat[n - i - 1] * ifat[i] % MOD;
            if((n - i - 1) % 2 == 1) {
                den[i] = (MOD - den[i]) % MOD;
            }
        }

        long long getVal(long long x) {
            int n = (int) y.size();
            x %= MOD;
            if(x < n) {
                //return y[(int) x];
            }
            std::vector<long long> l, r;
            l.resize(n);
            l[0] = 1;
            for(int i = 1; i < n; i++) {
                l[i] = l[i - 1] * (x - (i - 1) + MOD) % MOD;
            }
            r.resize(n);
            r[n - 1] = 1;
            for(int i = n - 2; i >= 0; i--) {
                r[i] = r[i + 1] * (x - (i + 1) + MOD) % MOD;
            }
            long long ans = 0;
            for(int i = 0; i < n; i++) {
                long long coef = l[i] * r[i] % MOD;
                ans = (ans + coef * y[i] % MOD * den[i]) % MOD;
            }
            return ans;
        }

private:
    std::vector<long long> y, den;
};

int main() {
    fat[0] = ifat[0] = 1;
    for(int i = 1; i < ms; i++) {
        fat[i] = fat[i - 1] * i % MOD;
        ifat[i] = fexp(fat[i], MOD - 2);
    }
    // Codeforces 622F
    int x, k;
    std::cin >> x >> k;
    std::vector<long long> a;
    a.push_back(0);
    for(long long i = 1; i <= k + 1; i++) {
        a.push_back((a.back() + fexp(i, k)) % MOD);
    }
    LagrangePoly f(a);
    std::cout << f.getVal(x) << '\n';
}

```

4.16 Count integer points inside triangle

```

//gcd(p, q) == 1
ll get(ll p, ll q, ll n, bool floor = true) {
    if (n == 0) {
        return 0;
    }
    if (p % q == 0) {

```

```

        return n * (n + 1) / 2 * (p / q);
    }
    if (p > q) {
        return n * (n + 1) / 2 * (p / q) + get(p % q, q, n, floor);
    }
    ll new_n = p * n / q;
    ll ans = (n + 1) * new_n - get(q, p, new_n, false);
    if (!floor) {
        ans += n - n / q;
    }
    return ans;
}

```

5 Geometry

5.1 Geometry

```

const double inf = 1e100, eps = 1e-9;
const double PI = acos(-1.0L);

int cmp (double a, double b = 0) {
    if (abs(a-b) < eps) return 0;
    return (a < b) ? -1 : +1;
}

struct PT {
    double x, y;
    PT(double x = 0, double y = 0) : x(x), y(y) {}
    PT operator + (const PT &p) const { return PT(x+p.x, y+p.y); }
    PT operator - (const PT &p) const { return PT(x-p.x, y-p.y); }
    PT operator * (double c) const { return PT(x*c, y*c); }
    PT operator / (double c) const { return PT(x/c, y/c); }

    bool operator < (const PT &p) const {
        if(cmp(x, p.x) != 0) return x < p.x;
        return cmp(y, p.y) < 0;
    }

    bool operator == (const PT &p) const {
        return !cmp(x, p.x) && !cmp(y, p.y);
    }

    bool operator != (const PT &p) const {
        return !(p == *this);
    }
};

double dot (PT p, PT q) { return p.x * q.x + p.y*q.y; }
double cross (PT p, PT q) { return p.x * q.y - p.y*q.x; }
double dist2 (PT p, PT q = PT(0, 0)) { return dot(p-q, p-q); }
double dist (PT p, PT q) { return hypot(p.x-q.x, p.y-q.y); }
double norm (PT p) { return hypot(p.x, p.y); }
PT normalize (PT p) { return p/hypot(p.x, p.y); }
double angle (PT p, PT q) { return atan2(cross(p, q), dot(p, q)); }
double angle (PT p) { return atan2(p.y, p.x); }
double polarAngle (PT p) {
    double a = atan2(p.y, p.x);
    return a < 0 ? a + 2*PI : a;
}

// - p.y*sen(+90), p.x*sen(+90)
PT rotateCCW90 (PT p) { return PT(-p.y, p.x); }
// - p.y*sen(-90), p.x*sen(-90)
PT rotateCW90 (PT p) { return PT(p.y, -p.x); }

PT rotateCCW (PT p, double t) {
    return PT(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t));
}

// !!! PT (int, int)
typedef pair<PT, int> Line;
PT getDir (PT a, PT b) {
    if (a.x == b.x) return PT(0, 1);
    if (a.y == b.y) return PT(1, 0);
    int dx = b.x-a.x;
    int dy = b.y-a.y;
    int g = __gcd(abs(dx), abs(dy));
    if (dx < 0) g = -g;
    return PT(dx/g, dy/g);
}

Line getLine (PT a, PT b) {
    PT dir = getDir(a, b);
    return {dir, cross(dir, a)};
}

```

```

}
// Projeta ponto c na linha a - b assumindo a != b
// a.b = |a| cos $\theta$  * |b|
PT projectPointLine (PT a, PT b, PT c) {
    return a + (b-a) * dot(b-a, c-a)/dot(b-a, b-a);
}

PT reflectPointLine (PT a, PT b, PT c) {
    PT p = projectPointLine(a, b, c);
    return p*2 - c;
}

// Projeta ponto c no segmento a - b
PT projectPointSegment (PT a, PT b, PT c) {
    double r = dot(b-a, b-a);
    if (cmp(r) == 0) return a;
    r = dot(b-a, c-a)/r;
    if (cmp(r, 0) < 0) return a;
    if (cmp(r, 1) > 0) return b;
    return a + (b - a) * r;
}

// Calcula distancia entre o ponto c e o segmento a - b
double distancePointSegment (PT a, PT b, PT c) {
    return dist(c, projectPointSegment(a, b, c));
}

// Parallel and opposite directions
// Determina se o ponto c esta em um segmento a - b
bool ptInSegment (PT a, PT b, PT c) {
    if (a == b) return a == c;
    a = a-c, b = b-c;
    return cmp(cross(a, b)) == 0 && cmp(dot(a, b)) <= 0;
}

// Determina se as linhas a - b e c - d sao paralelas ou colineares
bool parallel (PT a, PT b, PT c, PT d) {
    return cmp(cross(b - a, c - d)) == 0;
}

bool collinear (PT a, PT b, PT c, PT d) {
    return parallel(a, b, c, d) && cmp(cross(a - b, a - c)) == 0 && cmp(cross(c - d, c - a)) == 0;
}

// Calcula distancia entre o ponto (x, y, z) e o plano ax + by + cz = d
double distancePointPlane(double x, double y, double z, double a, double b, double c, double d) {
    return abs(a * x + b * y + c * z - d) / sqrt(a * a + b * b + c * c);
}

// Determina se o segmento a - b intersecta com o segmento c - d
bool segmentsIntersect (PT a, PT b, PT c, PT d) {
    if (collinear(a, b, c, d)) {
        if (cmp(dist(a, c)) == 0 || cmp(dist(a, d)) == 0 || cmp(dist(b, c)) == 0 || cmp(dist(b, d)) == 0) return true;
        if (cmp(dot(c - a, c - b)) > 0 && cmp(dot(d - a, d - b)) > 0 && cmp(dot(c - b, d - b)) > 0) return false;
        return true;
    }
    if (cmp(cross(d - a, b - a) * cross(c - a, b - a)) > 0) return false;
    if (cmp(cross(a - c, d - c) * cross(b - c, d - c)) > 0) return false;
    return true;
}

// Calcula a intersecao entre as retas a - b e c - d assumindo que uma unica intersecao existe
// Para intersecao de segmentos, cheque primeiro se os segmentos se intersectam e que nao sao paralelos
// r = a1 + t*d1, (r - a2) x d2 = 0
PT computeLineIntersection (PT a, PT b, PT c, PT d) {
    b = b - a; d = c - d; c = c - a;
    assert(cmp(cross(b, d)) != 0);
    return a + b * cross(c, d) / cross(b, d);
}

// Calcula centro do circulo dado tres pontos
PT computeCircleCenter (PT a, PT b, PT c) {
    b = (a + b) / 2; // bissector
    c = (a + c) / 2; // bissector
    return computeLineIntersection(b, b + rotateCW90(a - b), c, c + rotateCW90(a - c));
}

vector<PT> circle2PtsRad (PT p1, PT p2, double r) {
    vector<PT> ret;

```

```

    double d2 = dist2(p1, p2);
    double det = r * r / d2 - 0.25;
    if (det < 0.0) return ret;
    double h = sqrt(det);
    for (int i = 0; i < 2; i++) {
        double x = (p1.x + p2.x) * 0.5 + (p1.y - p2.y) * h;
        double y = (p1.y + p2.y) * 0.5 + (p2.x - p1.x) * h;
        ret.push_back(PT(x, y));
        swap(p1, p2);
    }
    return ret;
}

// Calcula intersecao da linha a - b com o circulo centrado em c com raio r > 0
bool circleLineIntersection(PT a, PT b, PT c, double r) {
    return cmp(dist(c, projectPointLine(a, b, c)), r) <= 0;
}

vector<PT> circleLine (PT a, PT b, PT c, double r) {
    vector<PT> ret;
    PT p = projectPointLine(a, b, c), pl;
    double h = norm(c-p);
    if (cmp(h, r) == 0) {
        ret.push_back(p);
    } else if (cmp(h, r) < 0) {
        double k = sqrt(r*r - h*h);
        pl = p + (b-a)/(norm(b-a))*k;
        ret.push_back(pl);
        pl = p - (b-a)/(norm(b-a))*k;
        ret.push_back(pl);
    }
    return ret;
}

bool ptInsideTriangle(PT p, PT a, PT b, PT c) {
    if(cross(b-a, c-b) < 0) swap(a, b);
    if(ptInSegment(a,b,p)) return 1;
    if(ptInSegment(b,c,p)) return 1;
    if(ptInSegment(c,a,p)) return 1;
    bool x = cross(b-a, p-b) < 0;
    bool y = cross(c-b, p-c) < 0;
    bool z = cross(a-c, p-a) < 0;
    return x == y && y == z;
}

// Determina se o ponto esta num poligono convexo em O(lgn)
bool pointInConvexPolygon(const vector<PT> &p, PT q) {
    if (p.size() == 1) return p.front() == q;
    int l = 1, r = p.size()-1;
    while (abs(r-l) > 1) {
        int m = (r+l)/2;
        if(cross(p[m]-p[0], q-p[0]) < 0) r = m;
        else l = m;
    }
    return ptInsideTriangle(q, p[0], p[l], p[r]);
}

// Determina se o ponto esta num poligono possivelmente nao-convexo
// Retorna 1 para pontos estritamente dentro, 0 para pontos estritamente fora do poligono
// e 0 ou 1 para os pontos restantes
// Eh possivel converter num teste exato usando inteiros e tomando cuidado com a divisao
// e entao usar testes exatos para checar se esta na borda do poligono
bool pointInPolygon(const vector<PT> &p, PT q) {
    bool c = 0;
    for(int i = 0; i < p.size(); i++){
        int j = (i + 1) % p.size();
        if((p[i].y <= q.y && q.y < p[j].y || p[j].y <= q.y && q.y < p[i].y) && q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) / (p[j].y - p[i].y))
            c = !c;
    }
    return c;
}

// Determina se o ponto esta na borda do poligono
bool pointOnPolygon(const vector<PT> &p, PT q) {
    for(int i = 0; i < p.size(); i++){
        if(cmp(dist2(projectPointSegment(p[i], p[(i + 1) % p.size()], q), q)) < 0)
            return true;
        return false;
    }
}

// area / semiperimeter
double rIncircle (PT a, PT b, PT c) {

```

```

double ab = norm(a-b), bc = norm(b-c), ca = norm(c-a);
return abs(cross(b-a, c-a)/(ab+bc+ca));
}

// Calcula intersecao do circulo centrado em a com raio r e o centrado em b com raio R
vector<PT> circleCircle (PT a, double r, PT b, double R) {
    vector<PT> ret;
    double d = norm(a-b);
    if (d > r + R || d + min(r, R) < max(r, R)) return ret;
    double x = (d*d - R*R + r*r) / (2*d); // x = r*cos(R opposite angle)
    double y = sqrt(r*r - x*x);
    PT v = (b - a)/d;
    ret.push_back(a + v*x + rotateCCW90(v)*y);
    if (cmp(y) > 0)
        ret.push_back(a + v*x - rotateCCW90(v)*y);
    return ret;
}

double circularSegArea (double r, double R, double d) {
    double ang = 2 * acos((d*d - R*R + r*r) / (2*d*r)); // cos(R opposite angle) = x/r
    double tri = sin(ang) * r * r;
    double sector = ang * r * r;
    return (sector - tri) / 2;
}

// Calcula a area ou o centroide de um poligono (possivelmente nao-convexo)
// assumindo que as coordenadas estao listada em ordem horaria ou anti-horaria
// O centroide eh equivalente a o centro de massa ou centro de gravidade
double computeSignedArea (const vector<PT> &p) {
    double area = 0;
    for (int i = 0; i < p.size(); i++) {
        int j = (i+1) % p.size();
        area += p[i].x*p[j].y - p[j].x*p[i].y;
    }
    return area/2.0;
}

double computeArea(const vector<PT> &p) {
    return abs(computeSignedArea(p));
}

PT computeCentroid(const vector<PT> &p) {
    PT c(0,0);
    double scale = 6.0 * ComputeSignedArea(p);
    for(int i = 0; i < p.size(); i++){
        int j = (i + 1) % p.size();
        c = c + (p[i] + p[j]) * (p[i].x * p[j].y - p[j].x * p[i].y);
    }
    return c / scale;
}

// Testa se o poligono listada em ordem CW ou CCW eh simples (nenhuma linha se
// intersecta)
bool isSimple(const vector<PT> &p) {
    for(int i = 0; i < p.size(); i++) {
        for(int k = i + 1; k < p.size(); k++) {
            int j = (i + 1) % p.size();
            int l = (k + 1) % p.size();
            if (i == l || j == k) continue;
            if (segmentsIntersect(p[i], p[j], p[k], p[l]))
                return false;
        }
    }
    return true;
}

vector< pair<PT, PT> > getTangentSegs (PT c1, double r1, PT c2, double r2) {
    if (r1 < r2) swap(c1, c2), swap(r1, r2);
    vector<pair<PT, PT> > ans;
    double d = dist(c1, c2);
    if (cmp(d) <= 0) return ans;
    double dr = abs(r1 - r2), sr = r1 + r2;
    if (cmp(dr, d) >= 0) return ans;
    double u = acos(dr / d);
    PT dc1 = normalize(c2 - c1)*r1;
    PT dc2 = normalize(c2 - c1)*r2;
    ans.push_back(make_pair(c1 + rotateCCW(dc1, +u), c2 + rotateCCW(dc2, +u)));
    ans.push_back(make_pair(c1 + rotateCCW(dc1, -u), c2 + rotateCCW(dc2, -u)));
    if (cmp(sr, d) >= 0) return ans;
    double v = acos(sr / d);
    dc2 = normalize(c1 - c2)*r2;
    ans.push_back({c1 + rotateCCW(dc1, +v), c2 + rotateCCW(dc2, +v)});
    ans.push_back({c1 + rotateCCW(dc1, -v), c2 + rotateCCW(dc2, -v)});
}

```

```

    return ans;
}

```

5.2 Convex Hull

```

vector<PT> convexHull(vector<PT> p, bool needs = 1) {
    if(needs) sort(p.begin(), p.end());
    p.erase(unique(p.begin(), p.end()), p.end());
    int n = p.size(), k = 0;
    if(n <= 1) return p;
    vector<PT> h(n + 2);
    for(int i = 0; i < n; i++) {
        while(k >= 2 && cross(h[k - 1] - h[k - 2], p[i] - h[k - 2]) <= 0) k--;
        h[k++] = p[i];
    }
    for(int i = n - 2, t = k + 1; i >= 0; i--) {
        while(k >= t && cross(h[k - 1] - h[k - 2], p[i] - h[k - 2]) <= 0) k--;
        h[k++] = p[i];
    }
    h.resize(k); // n+1 points where the first is equal to the last
    return h;
}

void sortByAngle (vector<PT>::iterator first, vector<PT>::iterator last, const PT o) {
    first = partition(first, last, [&o] (const PT &a) { return a == o; });
    auto pivot = partition(first, last, [&o] (const PT &a) {
        return !(a < o || a == o); // PT(a.y, a.x) < PT(o.y, o.x)
    });
    auto acmp = [&o] (const PT &a, const PT &b) { // C++11 only
        if (cmp(cross(a-o, b-o)) != 0) return cross(a-o, b-o) > 0;
        else return cmp(norm(a-o), norm(b-o)) < 0;
    };
    sort(first, pivot, acmp);
    sort(pivot, last, acmp);
}

vector<PT> graham (vector<PT> v) {
    sort(v.begin(), v.end());
    sortByAngle(v.begin(), v.end(), v[0]);
    vector<PT> u (v.size());
    int top = 0;
    for (int i = 0; i < v.size(); i++) {
        while (top > 1 && cmp(cross(u[top-1] - u[top-2], v[i]-u[top-2])) <= 0) top--;
        u[top++] = v[i];
    }
    u.resize(top);
    return u;
}

vector<PT> splitHull(const vector<PT> &hull) {
    vector<PT> ans(hull.size());
    for(int i = 0, j = (int) hull.size()-1, k = 0; k < (int) hull.size(); k++) {
        if(hull[i] < hull[j]) {
            ans[k] = hull[i++];
        } else {
            ans[k] = hull[j--];
        }
    }
    return ans;
}

vector<PT> ConvexHull(const vector<PT> &a, const vector<PT> &b) {
    auto A = splitHull(a);
    auto B = splitHull(b);
    vector<PT> C(A.size() + B.size());
    merge(A.begin(), A.end(), B.begin(), B.end(), C.begin());
    return ConvexHull(C, false);
}

int maximizeScalarProduct(const vector<PT> &hull, PT vec) {
    // this code assumes that there are no 3 colinear points
    int ans = 0;
    int n = hull.size();
    if(n < 20) {
        for(int i = 0; i < n; i++) {
            if(dot(hull[i], vec) > dot(hull[ans], vec)) {
                ans = i;
            }
        }
    }
}

```

```

} else {
    if(dot(hull[l], vec) > dot(hull[ans], vec)) {
        ans = l;
    }
    for(int rep = 0; rep < 2; rep++) {
        int l = 2, r = n - 1;
        while(l != r) {
            int mid = (l + r + 1) / 2;
            bool flag = dot(hull[mid], vec) >= dot(hull[mid-1], vec);
            if(rep == 0) { flag = flag && dot(hull[mid], vec) >= dot(hull[0], vec); }
            else { flag = flag || dot(hull[mid-1], vec) < dot(hull[0], vec); }
            if(flag) {
                l = mid;
            } else {
                r = mid - 1;
            }
        }
        if(dot(hull[ans], vec) < dot(hull[l], vec)) {
            ans = l;
        }
    }
    return ans;
}

```

5.3 Cut Polygon

```

struct Segment {
    typedef long double T;
    PT p1, p2;
    T a, b, c;

    Segment() {}

    Segment(PT st, PT en) {
        p1 = st, p2 = en;
        a = -(st.y - en.y);
        b = st.x - en.x;
        c = a * en.x + b * en.y;
    }

    T plug(T x, T y) {
        // plug >= 0 is to the right
        return a * x + b * y - c;
    }

    T plug(PT p) {
        return plug(p.x, p.y);
    }

    bool inLine(PT p) { return cross((p - p1), (p2 - p1)) == 0; }
    bool inSegment(PT p) {
        return inLine(p) && dot((p1 - p2), (p - p2)) >= 0 && dot((p2 - p1), (p - p1)) >= 0;
    }

    PT lineIntersection(Segment s) {
        long double A = a, B = b, C = c;
        long double D = s.a, E = s.b, F = s.c;
        long double x = (long double) C * E - (long double) B * F;
        long double y = (long double) A * F - (long double) C * D;
        long double tmp = (long double) A * E - (long double) B * D;
        x /= tmp;
        y /= tmp;
        return PT(x, y);
    }

    bool polygonIntersection(const vector<PT> &poly) {
        long double l = -1e18, r = 1e18;
        for(auto p : poly) {
            long double z = plug(p);
            l = max(l, z);
            r = min(r, z);
        }
        return l - r > eps;
    }
};

vector<PT> cutPolygon(vector<PT> poly, Segment seg) {
    int n = (int) poly.size();
    vector<PT> ans;
    for(int i = 0; i < n; i++) {
        double z = seg.plug(poly[i]);

```

```

        if(z > -eps) {
            ans.push_back(poly[i]);
        }
        double z2 = seg.plug(poly[(i + 1) % n]);
        if((z > eps && z2 < -eps) || (z < -eps && z2 > eps)) {
            ans.push_back(seg.lineIntersection(Segment(poly[i], poly[(i + 1) % n])));
        }
    }
    return ans;
}

```

5.4 Smallest Enclosing Circle

```

typedef pair<PT, double> circle;
bool inCircle(circle c, PT p) {
    return cmp(dist(c.first, p), c.second) <= 0;
}

PT circumcenter(PT p, PT q, PT r) {
    PT a = p - r, b = q - r;
    PT c = PT(dot(a, p + r) / 2, dot(b, q + r) / 2);
    return PT(cross(c, PT(a.y, b.y)), cross(PT(a.x, b.x), c)) / cross(a, b);
}

mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
circle spanningCircle(vector<PT> &v) {
    int n = v.size();
    shuffle(v.begin(), v.end(), rng);
    circle C(PT(), -1);
    for(int i = 0; i < n; i++) if(!inCircle(C, v[i])) {
        C = circle(v[i], 0);
        for(int j = 0; j < i; j++) if(!inCircle(C, v[j])) {
            C = circle((v[i] + v[j]) / 2, dist(v[i], v[j]) / 2);
            for(int k = 0; k < j; k++) if(!inCircle(C, v[k])) {
                PT o = circumcenter(v[i], v[j], v[k]);
                C = circle(o, dist(o, v[k]));
            }
        }
    }
    return C;
}

```

5.5 Minkowski

```

bool comp(PT a, PT b) {
    int hp1 = (a.x < 0 || (a.x == 0 && a.y < 0));
    int hp2 = (b.x < 0 || (b.x == 0 && b.y < 0));
    if(hp1 != hp2) return hp1 < hp2;
    long long R = cross(a, b);
    if(R) return R > 0;
    return dot(a, a) < dot(b, b);
}

// This code assumes points are ordered in ccw and the first points in both vectors
// is the min lexicographically
vector<PT> minkowskiSum(const vector<PT> &a, const vector<PT> &b) {
    if(a.empty() || b.empty()) return vector<PT>();
    vector<PT> ret;
    int n1 = a.size(), n2 = b.size();
    if(min(n1, n2) < 2) {
        for(int i = 0; i < n1; i++) {
            for(int j = 0; j < n2; j++) {
                ret.push_back(a[i] + b[j]);
            }
        }
        return ret;
    }
    PT v1, v2, p = a[0] + b[0];
    ret.push_back(p);
    for(int i = 0, j = 0; i + j + 1 < n1 + n2; ) {
        v1 = a[(i + 1) % n1] - a[i];
        v2 = b[(j + 1) % n2] - b[j];
        if(j == n2 || (i < n1 && comp(v1, v2))) p = p + v1, i++;
        else p = p + v2, j++;
        while(ret.size() >= 2 && cmp(cross(p - ret.back(), p - ret[(int)ret.size() - 2])) == 0)
            ret.pop_back();
    }
}

```

```

    // removing colinear points
    // needs the scalar product stuff if the result is a line
    ret.pop_back();
}
ret.push_back(p);
}
return ret;
}

```

5.6 Half Plane Intersection

```

struct L {
    PT a, b;
    L() {}
    L(PT a, PT b) : a(a), b(b) {}
};

double angle (L la) { return atan2(-(la.a.y - la.b.y), la.b.x - la.a.x); }

bool comp (L la, L lb) {
    if (cmp(angle(la), angle(lb)) == 0) return cross((lb.b - lb.a), (la.b - lb.a)) >
        eps;
    return cmp(angle(la), angle(lb)) < 0;
}

PT computeLineIntersection (L la, L lb) {
    return computeLineIntersection(la.a, la.b, lb.a, lb.b);
}

bool check (L la, L lb, L lc) {
    PT p = computeLineIntersection(lb, lc);
    double det = cross((la.b - la.a), (p - la.a));
    return cmp(det) < 0;
}

vector<PT> hpi (vector<L> line) { // salvar (i, j) CCW, (j, i) CW
    sort(line.begin(), line.end(), comp);
    vector<L> pl(1, line[0]);
    for (int i = 0; i < (int)line.size(); ++i) if (cmp(angle(line[i]), angle(pl.back()))
        != 0) pl.push_back(line[i]);
    deque<int> dq;
    dq.push_back(0);
    dq.push_back(1);
    for (int i = 2; i < (int)pl.size(); ++i) {
        while ((int)dq.size() > 1 && check(pl[i], pl[dq.back()], pl[dq[dq.size() -
            2]])) dq.pop_back();
        while ((int)dq.size() > 1 && check(pl[i], pl[dq[0]], pl[dq[1]])) dq.pop_front();
        dq.push_back(i);
    }
    while ((int)dq.size() > 1 && check(pl[dq[0]], pl[dq.back()], pl[dq[dq.size() -
        2]])) dq.pop_back();
    while ((int)dq.size() > 1 && check(pl[dq.back()], pl[dq[0]], pl[dq[1]])) dq.
        pop_front();
    vector<PT> res;
    for (int i = 0; i < (int)dq.size(); ++i) {
        res.emplace_back(computeLineIntersection(pl[dq[i]], pl[dq[(i + 1) % dq.size()
            ]]));
    }
    return res;
}

```

5.7 Closest Pair

```

double closestPair(vector<PT> p) {
    int n = p.size(), k = 0;
    sort(p.begin(), p.end());
    double d = inf;
    set<PT> ptsInv;
    for (int i = 0; i < n; i++) {
        while(k < i && p[k].x < p[i].x - d) {
            ptsInv.erase(swapCoord(p[k++]));
        }
        for(auto it = ptsInv.lower_bound(PT(p[i].y - d, p[i].x - d));
            it != ptsInv.end() && it->x <= p[i].y + d; it++) {
            d = min(d, dist(p[i] - swapCoord(*it), PT(0, 0)));
        }
        ptsInv.insert(swapCoord(p[i]));
    }
}

```

```

    }
    return d;
}

```

5.8 Voronoi

```

Segment getBisector(PT a, PT b) {
    Segment ans(a, b);
    swap(ans.a, ans.b);
    ans.b *= -1;
    ans.c = ans.a * (a.x + b.x) * 0.5 + ans.b * (a.y + b.y) * 0.5;
    return ans;
}

// BE CAREFUL!
// the first point may be any point
// O(N^3)
vector<PT> getCell(vector<PT> pts, int i) {
    vector<PT> ans;
    ans.emplace_back(0, 0);
    ans.emplace_back(1e6, 0);
    ans.emplace_back(1e6, 1e6);
    ans.emplace_back(0, 1e6);
    for(int j = 0; j < (int) pts.size(); j++) {
        if(j != i) {
            ans = cutPolygon(ans, getBisector(pts[i], pts[j]));
        }
    }
    return ans;
}

// O(N^2) expected time
vector<vector<PT>> getVoronoi(vector<PT> pts) {
    // assert(pts.size() > 0);
    int n = (int) pts.size();
    vector<int> p(n, 0);
    for(int i = 0; i < n; i++) {
        p[i] = i;
    }
    shuffle(p.begin(), p.end(), rng);
    vector<vector<PT>> ans(n);
    ans[0].emplace_back(0, 0);
    ans[0].emplace_back(w, 0);
    ans[0].emplace_back(w, h);
    ans[0].emplace_back(0, h);
    for(int i = 1; i < n; i++) {
        ans[i] = ans[0];
    }
    for(auto i : p) {
        for(auto j : p) {
            if(j == i) break;
            auto bi = getBisector(pts[j], pts[i]);
            if(!bi.polygonIntersection(ans[j])) continue;
            ans[j] = cutPolygon(ans[j], getBisector(pts[j], pts[i]));
            ans[i] = cutPolygon(ans[i], getBisector(pts[i], pts[j]));
        }
    }
    return ans;
}

```

6 String Algorithms

6.1 KMP

```

vector<int> getBorder(string str) {
    int n = str.size();
    vector<int> border(n, -1);
    for(int i = 1, j = -1; i < n; i++) {
        while(j >= 0 && str[i] != str[j + 1]) {
            j = border[j];
        }
        if(str[i] == str[j + 1]) {
            j++;
        }
        border[i] = j;
    }
}

```

```

    }
    return border;
}

int matchPattern(const string &txt, const string &pat, const vector<int> &border) {
    int freq = 0;
    for(int i = 0, j = -1; i < txt.size(); i++) {
        while(j >= 0 && txt[i] != pat[j + 1]) {
            j = border[j];
        }
        if(pat[j + 1] == txt[i]) {
            j++;
        }
        if(j + 1 == (int) pat.size()) {
            //found occurrence
            freq++;
            j = border[j];
        }
    }
    return freq;
}

```

6.2 KMP Automaton

```

// trad converts a char to its index
int trad(char ch) { return (int) ch; }
// sigma should be greater then the greatest value returned by trad
vector<vector<int>> buildAutomaton(string p, int sigma=300) {
    vector<vector<int>> A(p.size() + 1, vector<int>(sigma));
    int brd = 0;
    for (int i = 0; i < sigma; i++) A[0][i] = 0;
    A[0][trad(p[0])] = 1;
    for (int i = 1; i <= p.size(); i++) {
        for (int ch = 0; ch < sigma; ch++) {
            A[i][ch] = A[brd][ch];
        }
        if (i < p.size()) {
            A[i][trad(p[i])] = i + 1;
            brd = A[brd][trad(p[i])];
        }
    }
    return A;
}

```

6.3 Trie

```

int trie[ms][sigma], terminal[ms], z;

void init() {
    memset(trie[0], -1, sizeof trie[0]);
    z = 1;
}

int get_id(char c) {
    return c - 'a';
}

void insert(string &p) {
    int cur = 0;
    for(int i = 0; i < p.size(); i++) {
        int id = get_id(p[i]);
        if(trie[cur][id] == -1) {
            memset(trie[z], -1, sizeof trie[z]);
            trie[cur][id] = z++;
        }
        cur = trie[cur][id];
    }
    terminal[cur]++;
}

int count(string &p) {
    int cur = 0;
    for(int i = 0; i < p.size(); i++) {
        int id = get_id(p[i]);
        if(trie[cur][id] == -1) {
            return false;
        }
    }
}

```

```

    }
    cur = trie[cur][id];
    return terminal[cur];
}

const int ms = 1e6; // quantidade de caracteres
const int sigma = 26; // tamanho do alfabeto
int trie[ms][sigma], fail[ms], terminal[ms], qtd;

void init() {
    qtd = 1;
    memset(trie[0], -1, sizeof trie[0]);
}

void add(string &s) {
    int node = 0;
    for (char ch : s) {
        int pos = val(ch); // no caso de alfabeto a-z: val(ch) = ch - 'a'
        if (trie[node][pos] == -1) {
            memset(trie[qtd], -1, sizeof trie[qtd]);
            terminal[qtd] = 0;
            trie[node][pos] = qtd++;
        }
        node = trie[node][pos];
    }
    ++terminal[node]; // trocar pela info que quiser
}

void buildFailure() {
    memset(fail, 0, sizeof(int) * qtd);
    queue<int> Q;
    Q.push(0);
    while (Q.size()) {
        int node = Q.front();
        Q.pop();
        for (int pos = 0; pos < sigma; ++pos) {
            int &v = trie[node][pos];
            int f = node == 0 ? 0 : trie[fail[node]][pos];
            if (v == -1) {
                v = f;
            } else {
                fail[v] = f;
                Q.push(v);
            }
            // dar merge nas infos (por ex: terminal[v] += terminal[f])
        }
    }
}

void search(string &s) {
    int node = 0;
    for (char ch : s) {
        int pos = val(ch);
        node = trie[node][pos];
        // processar infos no no atual (por ex: ocorrencias += terminal[node])
    }
}

```

6.5 Algoritmo de Z

```

template <class T>
vector<int> ZFunc(const vector<T> &v) {
    vector<int> z(v.size(), 0);
    int n = (int) v.size(), a = 0, b = 0;
    if (!z.empty()) z[0] = n;
    for (int i = 1; i < n; i++) {
        int end = i; if (i < b) end = min(i + z[i - a], b);
        while(end < n && v[end] == v[end - i]) ++end;
        z[i] = end - i; if(end > b) a = i, b = end;
    }
    return z;
}

```


6.6 Suffix Array

```
vector<int> buildSa(const string& in) {
    int n = in.size(), c = 0;
    vector<int> temp(n), posBucket(n), bucket(n), bpos(n), out(n);
    for (int i = 0; i < n; i++) out[i] = i;
    sort(out.begin(), out.end(), [&](int a, int b) { return in[a] < in[b]; });
    for (int i = 0; i < n; i++) {
        bucket[i] = c;
        if (i + 1 == n || in[out[i]] != in[out[i + 1]]) c++;
    }
    for (int h = 1; h < n && c < n; h <= 1) {
        for (int i = 0; i < n; i++) posBucket[out[i]] = bucket[i];
        for (int i = n - 1; i >= 0; i--) bpos[bucket[i]] = i;
        for (int i = 0; i < n; i++) {
            if (out[i] >= n - h) temp[bpos[bucket[i]]++] = out[i];
        }
        for (int i = 0; i < n; i++) {
            if (out[i] >= h) temp[bpos[posBucket[out[i] - h]]++] = out[i] - h;
        }
        c = 0;
        for (int i = 0; i + 1 < n; i++) {
            int a = (bucket[i] != bucket[i + 1]) || (temp[i] >= n - h)
                || (posBucket[temp[i + 1] + h] != posBucket[temp[i] + h]);
            bucket[i] = c;
            c += a;
        }
        bucket[n - 1] = c++;
        temp.swap(out);
    }
    return out;
}

vector<int> buildLcp(string s, vector<int> sa) {
    int n = (int) s.size();
    vector<int> pos(n), lcp(n, 0);
    for (int i = 0; i < n; i++) {
        pos[sa[i]] = i;
    }
    int k = 0;
    for (int i = 0; i < n; i++) {
        if (pos[i] + 1 == n) {
            k = 0;
            continue;
        }
        int j = sa[pos[i] + 1];
        while (i + k < n && j + k < n && s[i + k] == s[j + k]) k++;
        lcp[pos[i]] = k;
        k = max(k - 1, 0);
    }
    return lcp;
}
```

6.7 Suffix Tree

```
//by adamant

#define fpos adla
const int inf = 1e9;
const int maxn = 1e4;
char s[maxn];
map<int, int> to[maxn];
int len[maxn], fpos[maxn], link[maxn];
int node, pos;
int sz = 1, n = 0;

int make_node(int _pos, int _len)
{
    fpos[sz] = _pos;
    len[sz] = _len;
    return sz++;
}

void go_edge()
{
    while (pos > len[to[node][s[n - pos]]])
    {
        node = to[node][s[n - pos]];
        pos -= len[node];
    }
}
```

```

    }
}

void add_letter(int c)
{
    s[n++] = c;
    pos++;
    int last = 0;
    while (pos > 0)
    {
        go_edge();
        int edge = s[n - pos];
        int &v = to[node][edge];
        int t = s[fpos[v] + pos - 1];
        if (v == 0)
        {
            v = make_node(n - pos, inf);
            link[last] = node;
            last = 0;
        }
        else if (t == c)
        {
            link[last] = node;
            return;
        }
        else
        {
            int u = make_node(fpos[v], pos - 1);
            to[u][c] = make_node(n - 1, inf);
            to[u][t] = v;
            fpos[v] += pos - 1;
            len[v] -= pos - 1;
            v = u;
            link[last] = u;
            last = u;
        }
        if (node == 0)
            pos--;
        else
            node = link[node];
    }
}

//len[0] = inf
```

6.8 Suffix Automaton

```
int len[ms*2], link[ms*2], nxt[ms*2][sigma];
int sz, last;

void build(string &s) {
    len[0] = 0; link[0] = -1;
    sz = 1; last = 0;
    memset(nxt[0], -1, sizeof nxt[0]);
    for (char ch : s) {
        int c = ch - 'a', cur = sz++;
        len[cur] = len[last] + 1;
        memset(nxt[cur], -1, sizeof nxt[cur]);
        int p = last;
        while (p != -1 && nxt[p][c] == -1) {
            nxt[p][c] = cur; p = link[p];
        }
        if (p == -1) {
            link[cur] = 0;
        }
        else {
            int q = nxt[p][c];
            if (len[p] + 1 == len[q]) {
                link[cur] = q;
            }
            else {
                len[sz] = len[p] + 1; link[sz] = link[q];
                memcpy(nxt[sz], nxt[q], sizeof nxt[q]);
                while (p != -1 && nxt[p][c] == q) {
                    nxt[p][c] = sz; p = link[p];
                }
                link[q] = link[cur] = sz++;
            }
        }
        last = cur;
    }
}
```

6.9 Manacher

```

/*
 * Manacher's Algorithm O(n)
 */

void manacher(char str[], int L[]) {
    int n = strlen(str), c = 0, r = 0;
    for(int i = 0; i < n; i++) {
        if(i < r && 2*c >= i) L[i] = min(L[2*c-i], r-i);
        else L[i] = 0;
        while(i-L[i]-1 >= 0 && i+L[i]+1 < n &&
            str[i-L[i]-1] == str[i+L[i]+1]) L[i]++;
        if(i+L[i]>r) { c=i; r=i+L[i]; }
    }
}

/*
 * Longest Palindromic Substring O(n)
 */

int LPS(char T[]) {
    int n = 2*strlen(T) + 1;
    char tmp[n+1];
    for(int i = 0, k = 0; T[i]; i++) {
        tmp[k++] = '|'; tmp[k++] = T[i];
    }
    tmp[n-1] = '|'; tmp[n] = '\0';
    int L[n], ans = 1;
    manacher(tmp, L);
    for(int i=0; i<n; i++)
        ans = max(ans, L[i]);
    return ans;
}

```

6.10 Polish Notation

```

/*
 * Parenthetic to polish expression conversion
 */

inline bool isOp(char c) {
    return c=='+' || c=='-' || c=='*' || c=='/' || c=='^';
}

inline bool isCarac(char c) {
    return (c>='a' && c<='z') || (c>='A' && c<='Z') || (c>='0' && c<='9');
}

int paren2polish(char* paren, char* polish) {
    map<char, int> prec;
    prec['('] = 0;
    prec['+'] = prec['-'] = 1;
    prec['*'] = prec['/'] = 2;
    prec['^'] = 3;
    int len = 0;
    stack<char> op;
    for (int i = 0; paren[i]; i++) {
        if (isOp(paren[i])) {
            while (!op.empty() && prec[op.top()] >= prec[paren[i]]) {
                polish[len++] = op.top(); op.pop();
            }
            op.push(paren[i]);
        }
        else if (paren[i]=='(') op.push('(');
        else if (paren[i]==')') {
            for (; op.top()!='('; op.pop())
                polish[len++] = op.top();
            op.pop();
        }
        else if (isCarac(paren[i]))
            polish[len++] = paren[i];
    }
    for(; !op.empty(); op.pop())
        polish[len++] = op.top();
    polish[len] = 0;
    return len;
}

```

```

}

```

7 Miscellaneous

7.1 Ternary Search

```

// R
for(int i = 0; i < LOG; i++){
    long double m1 = (A * 2 + B) / 3.0;
    long double m2 = (A + 2 * B) / 3.0;

    if(f(m1) > f(m2))
        A = m1;
    else
        B = m2;
}
ans = f(A);

// Z
while(B - A > 4){
    int m1 = (A + B) / 2;
    int m2 = (A + B) / 2 + 1;
    if(f(m1) > f(m2))
        A = m1;
    else
        B = m2;
}
ans = inf;
for(int i = A; i <= B; i++) ans = min(ans, f(i));

```

7.2 Count Sort

```

int H[(1<<15)+1], to[mx], b[mx];
void sort(int m, int a[]) {
    memset(H, 0, sizeof H);
    for (int i = 1; i <= m; i++) {
        H[a[i] % (1<<15)]++;
    }
    for (int i = 1; i < 1<<15; i++) {
        H[i] += H[i-1];
    }
    for (int i = m; i; i--) {
        to[i] = H[a[i] % (1 << 15)]--;
    }
    for (int i = 1; i <= m; i++) {
        b[to[i]] = a[i];
    }
    memset(H, 0, sizeof H);
    for (int i = 1; i <= m; i++) {
        H[b[i]>>15]++;
    }
    for (int i = 1; i < 1<<15; i++) {
        H[i] += H[i-1];
    }
    for (int i = m; i; i--) {
        to[i] = H[b[i]>>15]--;
    }
    for (int i = 1; i <= m; i++) {
        a[to[i]] = b[i];
    }
}

```

7.3 Random Number Generator

```

// mt19937_64 se LL
mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
// Random_Shuffle
shuffle(v.begin(), v.end(), rng);
// Random number in interval
int randomInt = uniform_int_distribution(0, i)(rng);

```

```
double randomDouble = uniform_real_distribution(0, 1)(rng);
// bernoulli_distribution, binomial_distribution, geometric_distribution
// normal_distribution, poisson_distribution, exponential_distribution
```

7.4 Rectangle Hash

```
namespace {
    struct safe_hash {
        static uint64_t splitmix64(uint64_t x) {
            // http://xorshift.di.unimi.it/splitmix64.c
            x += 0x9e3779b97f4a7c15;
            x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
            x = (x ^ (x >> 27)) * 0x94d049b133111eb;
            return x ^ (x >> 31);
        }

        size_t operator()(uint64_t x) const {
            static const uint64_t FIXED_RANDOM = std::chrono::steady_clock::now().
                time_since_epoch().count();
            return splitmix64(x + FIXED_RANDOM);
        }
    };

    struct rect {
        int x1, y1, x2, y2; // x1 < x2, y1 < y2
        rect() {}
        rect(int x1, int y1, int x2, int y2) : x1(x1), x2(x2), y1(y1), y2(y2) {}

        rect inter(const rect& other) const {
            int x3 = max(x1, other.x1);
            int y3 = max(y1, other.y1);
            int x4 = min(x2, other.x2);
            int y4 = min(y2, other.y2);
            return rect(x3, y3, x4, y4);
        }

        uint64_t get_hash() const {
            safe_hash sh;
            uint64_t ret = sh(x1);
            ret ^= sh(ret ^ y1);
            ret ^= sh(ret ^ x2);
            ret ^= sh(ret ^ y2);
            return ret;
        }
    };
};
```

7.5 Unordered Map Tricks

```
// pair<int, int> hash function
struct HASH{
    size_t operator()(const pair<int,int>&x) const{
        return (size_t) x.first * 37U + (size_t) x.second;
    }
};

unordered_map<int,int> mp;
mp.reserve(1024);
mp.max_load_factor(0.25);
```

7.6 Submask Enumeration

```
for (int s=m; ; s=(s-1)&m) {
    ... you can use s ...
    if (s==0) break;
}
```

7.7 Sum over Subsets DP

```
// F[i] = Sum of all A[j] where j is a submask of i
for(int i = 0; i < (1<<N); ++i)
    F[i] = A[i];
for(int i = 0; i < N; ++i) for(int mask = 0; mask < (1<<N); ++mask){
    if(mask & (1<<i))
        F[mask] += F[mask ^ (1<<i)];
}
```

7.8 Dates

```
string dayOfWeek[] = {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"};

// converts Gregorian date to integer (Julian day number)
int dateToInt(int m, int d, int y){
    return
        1461 * (y + 4800 + (m - 14) / 12) / 4 +
        367 * (m - 2 - (m - 14) / 12 * 12) / 12 -
        3 * ((y + 4900 + (m - 14) / 12) / 100) / 4 +
        d - 32075;
}

// converts integer (Julian day number) to Gregorian date: month/day/year
void intToDate(int jd, int &m, int &d, int &y){
    int x, n, i, j;

    x = jd + 68569;
    n = 4 * x / 146097;
    x -= (146097 * n + 3) / 4;
    i = (4000 * (x + 1)) / 1461001;
    x -= 1461 * i / 4 - 31;
    j = 80 * x / 2447;
    d = x - 2447 * j / 80;
    x = j / 11;
    m = j + 2 - 12 * x;
    y = 100 * (n - 49) + i + x;
}

// converts integer (Julian day number) to day of week
string intToDay(int jd){
    return dayOfWeek[jd % 7];
}
```

7.9 Lat Long

```
/*
    Converts from rectangular coordinates to latitude/longitude and vice
    versa. Uses degrees (not radians).
*/
struct ll
{
    double r, lat, lon;
};

struct rect
{
    double x, y, z;
};

ll convert(const rect& P)
{
    ll Q;
    Q.r = sqrt(P.x*P.x + P.y*P.y + P.z*P.z);
    Q.lat = 180/M_PI * asin(P.z/Q.r);
    Q.lon = 180/M_PI * acos(P.x/sqrt(P.x*P.x + P.y*P.y));

    return Q;
}

rect convert(const ll& Q)
{
    rect P;
    P.x = Q.r * cos(Q.lon*M_PI/180) * cos(Q.lat*M_PI/180);
    P.y = Q.r * sin(Q.lon*M_PI/180) * cos(Q.lat*M_PI/180);
    P.z = Q.r * sin(Q.lat*M_PI/180);

    return P;
}
```

8 Teoremas e formulas uteis

8.1 Grafos

Formula de Euler: $V - E + F = 2$ (para grafo planar)
 Handshaking: Numero par de vertices tem grau impar
 Kirchhoff's Theorem: Monta matriz onde $M_{i,i} = \text{Grau}[i]$ e $M_{i,j} = -1$ se houver aresta $i-j$
 ou 0 caso contrario, remove uma linha e uma coluna qualquer e o numero de
 spanning trees nesse grafo eh o det da matriz

Grafo contem caminho hamiltoniano se:
 Dirac's theorem: Se o grau de cada vertice for pelo menos $n/2$
 Ore's theorem: Se a soma dos graus que cada par nao-adjacente de vertices for pelo
 menos n

Trees:
 Tem Catalan(N) Binary trees de N vertices
 Tem Catalan(N-1) Arvores enraizadas com N vertices
 Cayley Formula: $n^{(n-2)}$ arvores em N vertices com label
 Prufer code: Cada etapa voce remove a folha com menor label e o label do vizinho eh
 adicionado ao codigo ate ter 2 vertices

Flow:
 Max Edge-disjoint paths: Max flow com arestas com peso 1
 Max Node-disjoint paths: Faz a mesma coisa mas separa cada vertice em um com as
 arestas de chegadas e um com as arestas de saida e uma aresta de peso 1
 conectando o vertice com aresta de chegada com ele mesmo com arestas de saida
 Konig's Theorem: minimum node cover = maximum matching se o grafo for bipartido,
 complemento eh o maximum independent set
 Min Node disjoint path cover: formar grafo bipartido de vertices duplicados, onde
 aresta sai do vertice tipo A e chega em tipo B, entao o path cover eh $N -$
 matching
 Min General path cover: Mesma coisa mas colocando arestas de A pra B sempre que houver
 caminho de A pra B
 Dilworth's Theorem: Min General Path cover = Max Antichain (set de vertices tal que
 nao existe caminho no grafo entre vertices desse set)
 Hall's marriage: um grafo tem um matching completo do lado X se para cada subconjunto
 W de X,
 $|W| \leq |\text{vizinhosW}|$ onde $|W|$ eh quantos vertices tem em W

8.2 Math

Goldbach's: todo numero par $n > 2$ pode ser representado com $n = a + b$ onde a e b sao
 primos
 Twin prime: existem infinitos pares $p, p + 2$ onde ambos sao primos
 Legendre's: sempre tem um primo entre n^2 e $(n+1)^2$
 Lagrange's: todo numero inteiro pode ser inscrito como a soma de 4 quadrados
 Zeckendorf's: todo numero pode ser representado pela soma de dois numeros de
 fibonnacis diferentes e nao consecutivos
 Euclid's: toda tripla de pitagoras primitiva pode ser gerada com
 $(n^2 - m^2, 2nm, n^2 + m^2)$ onde n, m sao coprimos e um deles eh par
 Wilson's: n eh primo quando $(n-1)! \bmod n = n - 1$
 Mcnugget: Para dois coprimos x, y o maior inteiro que nao pode ser escrito como $ax +$
 by eh $(x-1)(y-1)/2$

Fermat: Se p eh primo entao $a^{(p-1)} \bmod p = 1$
 Se x e m tambem forem coprimos entao $x^k \bmod m = x^{(k \bmod (m-1))} \bmod m$
 Euler's theorem: $x^{\phi(m)} \bmod m = 1$ onde $\phi(m)$ eh o totiente de euler

Chinese remainder theorem:
 Para equacoes no formato $x = a \bmod m_1, \dots, x = a \bmod m_n$ onde todos os pares $m_1,$
 \dots, m_n sao coprimos
 Deixe $X_k = m_1 m_2 \dots m_n / m_k$ e $X_k^{-1} \bmod m_k = \text{inverso de } X_k \bmod m_k$, entao
 $x = \text{somatorio com k de 1 ate n de } a_k * X_k * (X_k^{-1} \bmod m_k)$
 Para achar outra solucao so somar $m_1 m_2 \dots m_n$ a solucao existente

Catalan number: exemplo expressoes de parenteses bem formadas
 $C_0 = 1, C_n = \text{somatorio de } i=0 \rightarrow n-1 \text{ de } C_i * C_{(n-1-i)}$

outra forma: $C_n = (2n \text{ escolhe } n) / (n+1)$
 Bertrand's ballot theorem: p votos tipo A e q votos tipo B com $p > q$, prob de em todo
 ponto ter mais As do que Bs antes dele = $(p-q)/(p+q)$
 Se puder empates entao prob = $(p+1-q)/(p+1)$, para achar quantidade de possibilidades
 nos dois casos basta multiplicar por $(p + q \text{ escolhe } q)$

Propriedades de Coeficientes Binomiais:
 Somatorio de $k = 0 \rightarrow m$ de $(-1)^k * (n \text{ escolhe } k) = (-1)^m * (n-1 \text{ escolhe } m)$
 $(N \text{ escolhe } K) = (N \text{ escolhe } N-K)$
 $(N \text{ escolhe } K) = N/K * (n-1 \text{ escolhe } k-1)$
 Somatorio de $k = 0 \rightarrow n$ de $(n \text{ escolhe } k) = 2^n$
 Somatorio de $m = 0 \rightarrow n$ de $(m \text{ escolhe } k) = (n+1 \text{ escolhe } k+1)$
 Somatorio de $k = 0 \rightarrow m$ de $(n+k \text{ escolhe } k) = (n+m+1 \text{ escolhe } m)$
 Somatorio de $k = 0 \rightarrow n$ de $(n \text{ escolhe } k)^2 = (2n \text{ escolhe } n)$
 Somatorio de $k = 0$ ou $1 \rightarrow n$ de $k * (n \text{ escolhe } k) = n * 2^{(n-1)}$
 Somatorio de $k = 0 \rightarrow n$ de $(n-k \text{ escolhe } k) = \text{Fib}(n+1)$

Hockey-stick: Somatorio de $i = r \rightarrow n$ de $(i \text{ escolhe } r) = (n+1 \text{ escolhe } r+1)$
 Vandermonde: $(m+n \text{ escolhe } r) = \text{somatorio de } k = 0 \rightarrow r \text{ de } (m \text{ escolhe } k) * (n \text{ escolhe } r - k)$

Burnside lemma: colares diferentes nao contando rotacoes quando $m = \text{cores}$ e $n =$
 comprimento
 $(m^n + \text{somatorio } i=1 \rightarrow n-1 \text{ de } m^{\text{gcd}(i, n)})/n$

Distribuicao uniforme $a, a+1, \dots, b$ Expected[X] = $(a+b)/2$
 Distribuicao binomial com n tentativas de probabilidade p , $X = \text{sucessos}$:
 $P(X = x) = p^x * (1-p)^{(n-x)} * (n \text{ escolhe } x)$ e $E[X] = p*n$
 Distribuicao geometrica onde continuamos ate ter sucesso, $X = \text{tentativas}$:
 $P(X = x) = (1-p)^{(x-1)} * p$ e $E[X] = 1/p$
 Linearity of expectation: Tendo duas variaveis X e Y e constantes a e b , o valor
 esperado de $aX + bY = aE[X] + bE[Y]$
 $V(X) = E((X-u)^2)$
 $V(X) = E(X^2) - E(X)^2$

PG: $a_1 * (q^n - 1)/(q - 1)$

8.3 Geometry

Formula de Euler: $V - E + F = 2$
 Pick Theorem: Para achar pontos em coords inteiras num poligono Area = $i + b/2 - 1$
 onde i eh o o numero de pontos dentro do poligono e b de pontos no perimetro do
 poligono
 Two ears theorem: Todo poligono simples com mais de 3 vertices tem pelo menos 2
 orelhas, vertices que podem ser removidos sem criar um crossing, remover orelhas
 repetidamente triangula o poligono
 Incentro triangulo: $(a(X_a, Y_a) + b(X_b, Y_b) + c(X_c, Y_c))/(a+b+c)$ onde $a = \text{lado oposto}$
 ao vertice a , incentro eh onde cruzam as bissetrizes, eh o centro da
 circunferencia inscrita e eh equidistante aos lados

Delaunay Triangulation: Triangulacao onde nenhum ponto esta dentro de nenhum circulo
 circunscrito nos triangulos
 Eh uma triangulacao que maximiza o menor angulo e a MST euclidiana de um conjunto de
 pontos eh um subconjunto da triangulacao

Brahmagupta's formula: Area cyclic quadrilateral
 $s = (a+b+c+d)/2$
 area = $\sqrt{(s-a)*(s-b)*(s-c)*(s-d)}$
 $d = 0 \Rightarrow \text{area} = \sqrt{(s-a)*(s-b)*(s-c)*s}$

8.4 Mersenne's Primes

Primos de Mersenne $2^n - 1$
 Lista de Ns que resultam nos primeiros 41 primos de Mersenne:
 2; 3; 5; 7; 13; 17; 19; 31; 61; 89; 107; 127; 521; 607; 1.279; 2.203; 2.281; 3.217;
 4.253; 4.423; 9.689; 9.941; 11.213; 19.937; 21.701; 23.209; 44.497; 86.243;
 110.503; 132.049; 216.091; 756.839; 859.433; 1.257.787; 1.398.269; 2.976.221;
 3.021.377; 6.972.593; 13.466.917; 20.996.011; 24.036.583;