

## Amigos do Beto - ICPC Library

## Contents

## 1 Data Structures

1.1	BIT 2D Comprimida	1
1.2	Iterative Segment Tree	2
1.3	Iterative Segment Tree with Interval Updates	2
1.4	Segment Tree with Lazy Propagation	3
1.5	Segment Tree with Lazy Propagation	3
1.6	Persistent Segment Tree	4
1.7	Treap	4
1.8	Persistent Treap	5
1.9	KD-Tree	6
1.10	Link Cut Tree	6
1.11	Sparse Table	7
1.12	Max Queue	8
1.13	Policy Based Structures	8
1.14	Color Updates Structure	8

## 2 Graph Algorithms

2.1	Simple Disjoint Set	8
2.2	Blossom	9
2.3	Boruvka	10
2.4	Dinic Max Flow	10
2.5	Min Cost Max Flow	11
2.6	Euler Path and Circuit	12
2.7	Articulation Points/Bridges/Biconnected Components	12
2.8	SCC - Strongly Connected Components / 2SAT	12
2.9	LCA - Lowest Common Ancestor	13
2.10	LCA $O(1)$	13
2.11	Heavy Light Decomposition	13
2.12	Centroid Decomposition	14
2.13	Sack	14
2.14	Hungarian Algorithm - Maximum Cost Matching	15
2.15	Burunduk	15
2.16	Minimum Arborescence	16
2.17	Dominator Tree	17
2.18	Kuhn	17
2.19	Bipartite Check	18
2.20	Link Cut Tree Padrao	18
2.21	Link Cut Tree Aresta	19
2.22	Link Cut Tree Vertice	20

## 3 Dynamic Programming

3.1	Line Container	21
3.2	Li Chao Tree	21
3.3	Divide and Conquer Optimization	21
3.4	Knuth Optimization	22

## 4 Math

4.1	Chinese Remainder Theorem	22
4.2	Diophantine Equations	22
4.3	Discrete Logarithm	23
4.4	Discrete Root	23
4.5	Division Trick	23
4.6	Modular Sum	23
4.7	Primitive Root	23
4.8	Linear Sieve	24
4.9	Extended Euclides	24
4.10	Fast Exponentiation	24
4.11	Matrix	24
4.12	FFT - Fast Fourier Transform	24
4.13	NTT - Number Theoretic Transform	26
4.14	Fast Walsh Hadamard Transform	27
4.15	Miller and Rho	28
4.16	Determinant using Mod	29

4.17	Gauss	29
4.18	Lagrange Interpolation	30
4.19	Lagrange extracting polynomial	30
4.20	Count integer points inside triangle	30
4.21	Prime Counting	31
4.22	Berlekamp Massey	31

## 5 Geometry

5.1	Geometry	32
5.2	Convex Hull	34
5.3	Cut Polygon	34
5.4	Smallest Enclosing Circle	35
5.5	Minkowski	35
5.6	Half Plane Intersection	36
5.7	Closest Pair	36
5.8	Voronoi	36

## 6 String Algorithms

6.1	KMP	37
6.2	KMP Automaton	37
6.3	Aho-Corasick	37
6.4	Algoritmo de Z	38
6.5	Suffix Array	38
6.6	Suffix Tree	38
6.7	Suffix Automaton	39
6.8	Manacher	39
6.9	Polish Notation	39
6.10	String Hash	40

## 7 Miscellaneous

7.1	Ternary Search	40
7.2	Count Sort	40
7.3	Random Number Generator	40
7.4	Rectangle Hash	41
7.5	Safe Hash	41
7.6	Unordered Map Tricks	41
7.7	Iterate masks in bitcount order	41
7.8	Submask Enumeration	41
7.9	Sum over Subsets DP	41
7.10	Dates	42
7.11	Regular Expressions	42
7.12	Lat Long	42
7.13	Stable Marriage	42
7.14	Mo	43

## 8 Teoremas e formulas uteis

8.1	Grafos	43
8.2	Math	44
8.3	Geometry	44
8.4	Dynamic Programming	45
8.5	Mersenne's Primes	45

## 1 Data Structures

## 1.1 BIT 2D Comprimida

```

template<class T = int>
struct Bit2D {
public:
    // send updated points
    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) {
    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 Iterative Segment Tree

```
int n, t[2 * ms];
```

```

void build() {
    for(int i = n - 1; i > 0; --i) t[i] = t[i<<1] + t[i<<1|1]; // Merge
}

void update(int p, int value) { // set value at position p
    for(t[p += n] = value; p > 1; p >>= 1) t[p>>1] = t[p] + t[p^1]; //
        Merge
}

int query(int l, int r) {
    int res = 0;
    for(l += n, r += n+1; l < r; l >>= 1, r >>= 1) {
        if(l&1) res += t[l++]; // Merge
        if(r&1) res += t[--r]; // Merge
    }
    return res;
}

// If is non-commutative
S query(int l, int r) {
    S resl, resr;
    for (l += n, r += n+1; l < r; l >>= 1, r >>= 1) {
        if (l&1) resl = combine(resl, t[l++]);
        if (r&1) resr = combine(t[--r], resr);
    }
    return combine(resl, resr);
}

```

## 1.3 Iterative Segment Tree with Interval Updates

```

int n, t[2 * ms];

void build() {
    for(int i = n - 1; i > 0; --i) t[i] = t[i<<1] + t[i<<1|1]; // Merge
}

void update(int v, int l, int r) {
    for(l += n, r += n+1; l < r; l >>= 1, r >>= 1) {
        if(l&1) t[l++] += v; // Merge
        if(r&1) t[--r] += v; // Merge
    }
}

int query(int p) {
    int res = 0;
    for(p += n; p > 0; p >>= 1) res += t[p]; // Merge
    return res;
}

void push() { // push modifications to leafs
    for(int i = 1; i < n; i++) {
        t[i<<1] += t[i]; // Merge
        t[i<<1|1] += t[i]; // Merge
        t[i] = 0;
    }
}

```

## 1.4 Segment Tree with Lazy Propagation

```

struct LazyContext {
    LazyContext() { }
    void reset() { }
    void operator += (LazyContext o) { }
};

struct Node {
    Node() { }
    Node() { }
    Node(Node l, Node r) { }
    bool canBreak(LazyContext lazy) { } // false if non beats
    bool canApply(LazyContext lazy) { } // true if non beats
    void apply(LazyContext &lazy) { }
};

template <class i_t, class e_t, class lazy_cont>
class SegmentTree {
public:
    void init(std::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]);
        }
        for(int i = n - 1; i > 0; i--) {
            tree[i] = i_t(tree[i + 1], tree[i + i + 1]);
            lazy[i].reset();
        }
    }

    i_t qry(int l, int r) {
        if(l >= r) return i_t();
        l += n, r += n;
        push(l);
        push(r - 1);
        i_t lp, rp;
        for(; l < r; l /= 2, r /= 2) {
            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) {
        if(l >= r) return;
        l += n, r += n;
        push(l);
        push(r - 1);
        int l0 = l, r0 = r;
        for(; l < r; l /= 2, r /= 2) {
            if(l & 1) downUpd(l++, lc);
            if(r & 1) downUpd(--r, lc);
        }
        build(l0);
        build(r0 - 1);
    }

    void upd(int pos, e_t v) {
        pos += n;
        push(pos);

```

```

        tree[pos] = i_t(v);
        build(pos);
    }

private:
    int n, h;
    std::vector<bool> dirty;
    std::vector<i_t> tree;
    std::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 pushSingle(int p) {
        if(dirty[p]) {
            downUpd(p + p, lazy[p]);
            downUpd(p + p + 1, lazy[p]);
            lazy[p].reset();
            dirty[p] = false;
        }
    }

    void push(int p) {
        for(int s = h; s > 0; s--) {
            pushSingle(p >> s);
        }
    }

    void downUpd(int p, lazy_cont lc) {
        if(tree[p].canBreak(lc)) {
            return;
        } else if(tree[p].canApply(lc)) {
            apply(p, lc);
        } else {
            pushSingle(p);
            downUpd(p + p, lc);
            downUpd(p + p + 1, lc);
            tree[p] = i_t(tree[p + p], tree[p + p + 1]);
        }
    }

    void build(int p) {
        for(p /= 2; p > 0; p /= 2) {
            tree[p] = i_t(tree[p + p], tree[p + p + 1]);
            if(dirty[p]) {
                tree[p].apply(lazy[p]);
            }
        }
    }
};

```

## 1.5 Segment Tree with Lazy Propagation

```

int arr[ms], seg[4 * ms], lazy[4 * ms], n;

void build(int idx = 0, int l = 0, int r = n-1) {
    int mid = (l+r)/2;
    lazy[idx] = 0;
    if(l == r) {
        seg[idx] = arr[l];
        return;
    }

```

```

    }
    build(2*idx+1, l, mid); build(2*idx+2, mid+1, r);
    seg[idx] = seg[2*idx+1] + seg[2*idx+2]; // Merge
}

void apply(int idx, int l, int r) {
    if(lazy[idx] && !canBreak) { // if not beats canBreak = false
        if(l < r) {
            lazy[2*idx+1] += lazy[idx]; // Merge de lazy
            lazy[2*idx+2] += lazy[idx]; // Merge de lazy
        }
        if(canApply) { // if not beats canApply = true
            seg[idx] += lazy[idx] * (r - l + 1); // Aplicar lazy no seg
        } else {
            apply(2*idx+1, l, mid); apply(2*idx+2, mid+1, r);
            seg[idx] = seg[2*idx+1] + seg[2*idx+2]; // Merge
        }
    }
    lazy[idx] = 0; // Limpar a lazy
}

int query(int L, int R, int idx = 0, int l = 0, int r = n-1) {
    int mid = (l+r)/2;
    apply(idx, l, r);
    if(l > R || r < L) return 0; // Valor que nao atrapalhe
    if(L <= l && r <= R) return seg[idx];
    return query(L, R, 2*idx+1, l, mid) + query(L, R, 2*idx+2, mid+1, r)
        ; // Merge
}

void update(int L, int R, int V, int idx = 0, int l = 0, int r = n-1)
{
    int mid = (l+r)/2;
    apply(idx, l, r);
    if(l > R || r < L) return;
    if(L <= l && r <= R) {
        lazy[idx] = V;
        apply(idx, l, r);
        return;
    }
    update(L, R, V, 2*idx+1, l, mid); update(L, R, V, 2*idx+2, mid+1, r)
        ;
    seg[idx] = seg[2*idx+1] + seg[2*idx+2]; // Merge
}

```

## 1.6 Persistent Segment Tree

```

struct Node{
    int v = 0;
    Node *l = this, *r = this;
};
int CNT = 1;
Node buffer[ms * 20];
Node* update(Node *root, int l, int r, int idx, int val){
    Node *node = &buffer[CNT++];
    *node = *root;
    int mid = (l + r) / 2;
    node->v += val;
    if(l+1 != r){

```

```

        if(idx < mid) node->l = update(root->l, l, mid, idx,
            val);
        else node->r = update(root->r, mid, r, idx, val);
    }
    return node;
}

int query(Node *node, int tl, int tr, int l, int r){
    if(l <= tl && tr <= r) return node->v;
    if(tr <= l || tl >= r) return 0;
    int mid = (tl+tr) / 2;
    return query(node->l, tl, mid, l, r) + query(node->r, mid, tr,
        l, r);
}

```

## 1.7 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 insert (pitem & t, pitem it) {
    if (!t)
        t = it;
    else if (it->prio > t->prio)
        split (t, it->key, it->l, it->r), t = it;
    else
        insert (t->key <= it->key ? t->r : t->l, it);
}

void merge (pitem & t, pitem l, pitem r) {
    pushLazy (l); pushLazy (r);
    if (!l || !r) t = l ? l : r;

```

## 1.8 Persistent Treap

```

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 erase (pitem & t, int key) {
    if (t->key == key) {
        pitem th = t;
        merge (t, t->l, t->r);
        delete th;
    }
    else
        erase (key < t->key ? t->l : t->r, key);
}
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-l+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, l->r, rt);
    t = l;
}
pitem kth_element(pitem t, int k) {
    if (!t) return NULL;
    if (t->l) {
        if (t->l->size >= k) return kth_element(t->l, k);
        else k -= t->l->size;
    }
    return (k == 1) ? t : kth_element(t->r, k - 1);
}
int countLeft(pitem t, int key) {
    if (!t) return 0;
    else if (t->key < key)
        return 1 + (t->l ? t->l->size : 0) + countLeft(t->r, key);
    else
        return countLeft(t->l, key);
}
}

```

```

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->key;
    }
    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);
}

```

## 1.9 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);
}
long long nearestNeighbor(PT point) {
    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);
        }
    }
}
};

```

## 1.10 Link Cut Tree

```

#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;
    }
};

```

## 1.11 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]);
}

```

## 1.12 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;
};
```

## 1.13 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);
```

## 1.14 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; // vertices
const int me = 1e5; // 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 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;

    /*bool dij(int src, int sink) {
        T INF = std::numeric_limits<T>::max();
        dist.assign(n, INF);
        from.assign(n, -1);
        visit.assign(n, false);
        dist[src] = 0;
        for(int i = 0; i < n; i++) {
            int best = -1;
            for(int j = 0; j < n; j++) {
                if(visit[j]) continue;
                if(best == -1 || dist[best] > dist[j]) best = j;
            }
            if(dist[best] >= INF) break;
            visit[best] = true;
            for(auto e : edges[best]) {
                auto ed = list[e];
```

```
                if(ed.cap == 0) continue;
                T toDist = dist[best] + ed.cost + pot[best] - pot[ed.to];
                assert(toDist >= dist[best]);
                if(toDist < dist[ed.to]) {
                    dist[ed.to] = toDist;
                    from[ed.to] = e;
                }
            }
        }
        return dist[sink] < INF;
    }*/

    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];
        }
    }
};
```

## 2.6 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;
}
```

## 2.7 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);
            }
        }
    }
}
}
```

## 2.8 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)];

```

## 2.9 LCA - Lowest Common Ancestor

```

int par[ms][mlg+1], lvl[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];
}

```

## 2.10 LCA O(1)

```

template<class T>
struct RMQ {
    vector<vector<T>> jmp;

    RMQ(const vector<T>& V) : jmp(1, V) {
        for (int pw = 1, k = 1; pw * 2 <= (int)size(V); pw *= 2, ++k) {

```

```

            jmp.emplace_back(size(V) - pw * 2 + 1);
            for (int j = 0; j < (int)size(jmp[k]); ++j)
                jmp[k][j] = min(jmp[k-1][j], jmp[k-1][j + pw]);
        }
    }
    T query(int a, int b) {
        assert(a < b); // or return inf if a == b
        int dep = 31 - __builtin_clz(b - a);
        return min(jmp[dep][a], jmp[dep][b - (1 << dep)]);
    }
};

struct LCA {
    int T = 0;
    vector<int> time, path, ret;
    RMQ<int> rmq;

    LCA(vector<vector<int>>& C) : time(size(C)), rmq((dfs(C,0,-1), ret)) {}
    void dfs(vector<vector<int>>& C, int v, int par) {
        time[v] = T++;
        for (int y : C[v]) if (y != par) {
            path.push_back(v), ret.push_back(time[v]);
            dfs(C, y, v);
        }
    }
    int lca(int a, int b) {
        if (a == b) return a;
        tie(a, b) = minmax(time[a], time[b]);
        return path[rmq.query(a, b)];
    }
};

```

## 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--;
        } else {
            h[u] = 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

```

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;
                }
            }
        }
        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 &v1, 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(v1)) {
        v1[i].a = color[v1[i].a];
        v1[i].b = color[v1[i].b];
        if (v1[i].a != v1[i].b)
            deg[v1[i].a]++, deg[v1[i].b]++;
    }
    vn = vn1, vn1 = 0;
    forn (i, vn) {
        u[i] = vn1, vn1 += (deg[i] > 0), add_vn += !deg[i];
    }
    forn (i, sz(v1)) {
        v1[i].a = u[v1[i].a];
        v1[i].b = u[v1[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;
for (i, vn) {
    g[i].clear();
}
for (i, sz(v)) {
    if (v[i].a != v[i].b) {
        if (v[i].l <= 1 && v[i].r >= r)
            add(v[i].a, v[i].b);
        else if (1 < 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(E*log^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) {
                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();
                }
                for (auto u : cycle) { // compress 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;
}

```

## 2.17 Dominator Tree

```

struct dominator_tree {
    vector<basic_string<int>> g, rg, bucket;
    vector<int> arr, par, rev, sdom, dom, dsu, label;
    int n, t;
    dominator_tree(int n) : g(n), rg(n), bucket(n), arr(n, -1),
        par(n), rev(n), sdom(n), dom(n), dsu(n), label(n), n(n), t(0) {}
    void add_edge(int u, int v) { g[u] += v; }
    void dfs(int u) {
        arr[u] = t;
        rev[t] = u;
        label[t] = sdom[t] = dsu[t] = t;
        t++;
        for (int w : g[u]) {
            if (arr[w] == -1) {
                dfs(w);
            }
        }
    }
}

```

```

        par[arr[w]] = arr[u];
    }
    rg[arr[w]] += arr[u];
}
}
int find(int u, int x=0) {
    if (u == dsu[u])
        return x ? -1 : u;
    int v = find(dsu[u], x+1);
    if (v < 0)
        return u;
    if (sdom[label[dsu[u]]] < sdom[label[u]])
        label[u] = label[dsu[u]];
    dsu[u] = v;
    return x ? v : label[u];
}
vector<int> run(int root) {
    dfs(root);
    iota(dom.begin(), dom.end(), 0);
    for (int i=t-1; i>=0; i--) {
        for (int w : rg[i])
            sdom[i] = min(sdom[i], sdom[find(w)]);
        if (i)
            bucket[sdom[i]] += i;
        for (int w : bucket[i]) {
            int v = find(w);
            if (sdom[v] == sdom[w])
                dom[w] = sdom[w];
            else
                dom[w] = v;
        }
        if (i > 1)
            dsu[i] = par[i];
    }
    for (int i=1; i<t; i++) {
        if (dom[i] != sdom[i])
            dom[i] = dom[dom[i]];
    }
    vector<int> outside_dom(n);
    iota(begin(outside_dom), end(outside_dom), 0);
    for (int i=0; i<n; i++)
        outside_dom[rev[i]] = rev[dom[i]];
    return outside_dom;
}
};

```

## 2.18 Kuhn

```

int n, m;
vector<vector<int>> g;
vector<int> mt;
vector<bool> used;

bool try_kuhn(int v) {
    if (used[v]) return false;
    used[v] = true;
    for (int to : g[v]) {
        if (mt[to] == -1 || try_kuhn(mt[to])) {
            mt[to] = v;
            return true;
        }
    }
}

```

```

    }
    return false;
}

int main () {
    mt.assign(m, -1);
    vector<bool> used1(n, false);
    for (int i = 0; i < n; i++) {
        for (int to : g[i]) {
            if (mt[to] == -1) {
                mt[to] = i;
                used1[i] = true;
                break;
            }
        }
    }

    for (int i = 0; i < n; i++) {
        if (used1[i]) continue;
        used.assign(n, false);
        try_kuhn(i);
    }
}

```

## 2.19 Bipartite Check

```

void make_set(int v) {
    parent[v] = make_pair(v, 0);
    h[v] = 0;
}

pair<int, int> find_set(int v) {
    if (v != parent[v].first) {
        int parity = parent[v].second;
        parent[v] = find_set(parent[v].first);
        parent[v].second ^= parity;
    }
    return parent[v];
}

bool add_edge(int a, int b) {
    int x, y;
    tie(a, x) = find_set(a);
    tie(b, y) = find_set(b);
    if (a == b) {
        if (x == y)
            return false;
    } else {
        if (h[a] < h[b])
            swap(a, b);
        else if (h[a] == h[b])
            ++h[a];
        parent[b] = make_pair(a, x^y^1);
    }
    return true;
}

```

## 2.20 Link Cut Tree Padrao

```

// by brunomaletta
// Link-cut tree padrao
//
// Todas as operacoes sao O(log(n)) amortizado
// e4e663

namespace lct {
    struct node {
        int p, ch[2];
        node() { p = ch[0] = ch[1] = -1; }
    };

    node t[MAX];

    bool is_root(int x) {
        return t[x].p == -1 or (t[t[x].p].ch[0] != x and t[t[x].p].ch[1] != x);
    }

    void rotate(int x) {
        int p = t[x].p, pp = t[p].p;
        if (!is_root(p)) t[pp].ch[t[pp].ch[1] == p] = x;
        bool d = t[p].ch[0] == x;
        t[p].ch[!d] = t[x].ch[d], t[x].ch[d] = p;
        if (t[p].ch[!d+1] t[t[p].ch[!d]].p = p;
        t[x].p = pp, t[p].p = x;
    }

    void splay(int x) {
        while (!is_root(x)) {
            int p = t[x].p, pp = t[p].p;
            if (!is_root(p)) rotate((t[pp].ch[0] == p) ^ (t[p].ch[0] == x) ? x : p);
            rotate(x);
        }
    }

    int access(int v) {
        int last = -1;
        for (int w = v; w+1; last = w, splay(v), w = t[v].p)
            splay(w), t[w].ch[1] = (last == -1 ? -1 : v);
        return last;
    }

    int find_root(int v) {
        access(v);
        while (t[v].ch[0]+1) v = t[v].ch[0];
        return splay(v), v;
    }

    void link(int v, int w) { // v deve ser raiz
        access(v);
        t[v].p = w;
    }

    void cut(int v) { // remove aresta de v pro pai
        access(v);
        t[v].ch[0] = t[t[v].ch[0]].p = -1;
    }

    int lca(int v, int w) {
        return access(v), access(w);
    }
}

```

## 2.21 Link Cut Tree Aresta

```
// by brunomaletta
// Valores nas arestas
// rootify(v) torna v a raiz de sua arvore
// query(v, w) retorna a soma do caminho v--w
// update(v, w, x) soma x nas arestas do caminho v--w
//
// Todas as operacoes sao O(log(n)) amortizado
// 9ce48f

namespace lct {
    struct node {
        int p, ch[2];
        ll val, sub;
        bool rev;
        int sz, ar;
        ll lazy;
        node() {}
        node(int v, int ar_) :
            p(-1), val(v), sub(v), rev(0), sz(ar_), ar(ar_), lazy
            (0) {
                ch[0] = ch[1] = -1;
            }
    };

    node t[2*MAX]; // MAXN + MAXQ
    map<pair<int, int>, int> aresta;
    int sz;

    void prop(int x) {
        if (t[x].lazy) {
            if (t[x].ar) t[x].val += t[x].lazy;
            t[x].sub += t[x].lazy*t[x].sz;
            if (t[x].ch[0]+1) t[t[x].ch[0]].lazy += t[x].
                lazy;
            if (t[x].ch[1]+1) t[t[x].ch[1]].lazy += t[x].
                lazy;
        }
        if (t[x].rev) {
            swap(t[x].ch[0], t[x].ch[1]);
            if (t[x].ch[0]+1) t[t[x].ch[0]].rev ^= 1;
            if (t[x].ch[1]+1) t[t[x].ch[1]].rev ^= 1;
        }
        t[x].lazy = 0, t[x].rev = 0;
    }

    void update(int x) {
        t[x].sz = t[x].ar, t[x].sub = t[x].val;
        for (int i = 0; i < 2; i++) if (t[x].ch[i]+1) {
            prop(t[x].ch[i]);
            t[x].sz += t[t[x].ch[i]].sz;
            t[x].sub += t[t[x].ch[i]].sub;
        }
    }

    bool is_root(int x) {
        return t[x].p == -1 or (t[t[x].p].ch[0] != x and t[t[x]
            ].p.ch[1] != x);
    }

    void rotate(int x) {
        int p = t[x].p, pp = t[p].p;
        if (!is_root(p)) t[pp].ch[t[pp].ch[1] == p] = x;
    }
}
```

```
bool d = t[p].ch[0] == x;
t[p].ch[!d] = t[x].ch[d], t[x].ch[d] = p;
if (t[p].ch[!d]+1) t[t[p].ch[!d]].p = p;
t[x].p = pp, t[p].p = x;
update(p), update(x);
}

int splay(int x) {
    while (!is_root(x)) {
        int p = t[x].p, pp = t[p].p;
        if (!is_root(p)) prop(pp);
        prop(p), prop(x);
        if (!is_root(p)) rotate((t[pp].ch[0] == p)^(t[
            p].ch[0] == x) ? x : p);
        rotate(x);
    }
    return prop(x), x;
}

int access(int v) {
    int last = -1;
    for (int w = v; w+1; update(last = w), splay(v), w = t
        [v].p)
        splay(w), t[w].ch[1] = (last == -1 ? -1 : v);
    return last;
}

void make_tree(int v, int w=0, int ar=0) { t[v] = node(w, ar);
}

int find_root(int v) {
    access(v), prop(v);
    while (t[v].ch[0]+1) v = t[v].ch[0], prop(v);
    return splay(v);
}

bool conn(int v, int w) {
    access(v), access(w);
    return v == w ? true : t[v].p != -1;
}

void rootify(int v) {
    access(v);
    t[v].rev ^= 1;
}

ll query(int v, int w) {
    rootify(w), access(v);
    return t[v].sub;
}

void update(int v, int w, int x) {
    rootify(w), access(v);
    t[v].lazy += x;
}

void link_(int v, int w) {
    rootify(w);
    t[w].p = v;
}

void link(int v, int w, int x) { // v--w com peso x
    int id = MAX + sz++;
    aresta[make_pair(v, w)] = id;
    make_tree(id, x, 1);
    link_(v, id), link_(id, w);
}

void cut_(int v, int w) {
    rootify(w), access(v);
    t[v].ch[0] = t[t[v].ch[0]].p = -1;
}
}
```

```

void cut(int v, int w) {
    int id = aresta[make_pair(v, w)];
    cut_(v, id), cut_(id, w);
}

int lca(int v, int w) {
    access(v);
    return access(w);
}
}

```

## 2.22 Link Cut Tree Vertice

```

// by brunomaletta
// Valores nos vertices
// make_tree(v, w) cria uma nova arvore com um
// vertice soh com valor 'w'
// rootify(v) torna v a raiz de sua arvore
// query(v, w) retorna a soma do caminho v--w
// update(v, w, x) soma x nos vertices do caminho v--w
//
// Todas as operacoes sao O(log(n)) amortizado
// f9f489

namespace lct {
    struct node {
        int p, ch[2];
        ll val, sub;
        bool rev;
        int sz;
        ll lazy;
        node() {}
        node(int v) : p(-1), val(v), sub(v), rev(0), sz(1),
            lazy(0) {
            ch[0] = ch[1] = -1;
        }
    };

    node t[MAX];

    void prop(int x) {
        if (t[x].lazy) {
            t[x].val += t[x].lazy, t[x].sub += t[x].lazy *
                t[x].sz;
            if (t[x].ch[0] != -1) t[t[x].ch[0]].lazy += t[x].
                lazy;
            if (t[x].ch[1] != -1) t[t[x].ch[1]].lazy += t[x].
                lazy;
        }
        if (t[x].rev) {
            swap(t[x].ch[0], t[x].ch[1]);
            if (t[x].ch[0] != -1) t[t[x].ch[0]].rev ^= 1;
            if (t[x].ch[1] != -1) t[t[x].ch[1]].rev ^= 1;
        }
        t[x].lazy = 0, t[x].rev = 0;
    }

    void update(int x) {
        t[x].sz = 1, t[x].sub = t[x].val;
        for (int i = 0; i < 2; i++) if (t[x].ch[i] != -1) {
            prop(t[x].ch[i]);
            t[x].sz += t[t[x].ch[i]].sz;
        }
    }
}

```

```

        t[x].sub += t[t[x].ch[i]].sub;
    }

    bool is_root(int x) {
        return t[x].p == -1 or (t[t[x].p].ch[0] != x and t[t[x].
            p].ch[1] != x);
    }

    void rotate(int x) {
        int p = t[x].p, pp = t[p].p;
        if (!is_root(p)) t[pp].ch[t[pp].ch[1] == p] = x;
        bool d = t[p].ch[0] == x;
        t[p].ch[!d] = t[x].ch[d], t[x].ch[d] = p;
        if (t[p].ch[!d] != -1) t[t[p].ch[!d]].p = p;
        t[x].p = pp, t[p].p = x;
        update(p), update(x);
    }

    int splay(int x) {
        while (!is_root(x)) {
            int p = t[x].p, pp = t[p].p;
            if (!is_root(p)) prop(pp);
            prop(p), prop(x);
            if (!is_root(p)) rotate((t[pp].ch[0] == p) ^ (t[
                p].ch[0] == x) ? x : p);
            rotate(x);
        }
        return prop(x), x;
    }

    int access(int v) {
        int last = -1;
        for (int w = v; w != -1; w = t[w].p) {
            splay(w), t[w].ch[1] = (last == -1 ? -1 : v);
            return last;
        }
    }

    void make_tree(int v, int w) { t[v] = node(w); }

    int find_root(int v) {
        access(v), prop(v);
        while (t[v].ch[0] != -1) v = t[v].ch[0], prop(v);
        return splay(v);
    }

    bool connected(int v, int w) {
        access(v), access(w);
        return v == w ? true : t[v].p != -1;
    }

    void rootify(int v) {
        access(v);
        t[v].rev ^= 1;
    }

    ll query(int v, int w) {
        rootify(w), access(v);
        return t[v].sub;
    }

    void update(int v, int w, int x) {
        rootify(w), access(v);
        t[v].lazy += x;
    }

    void link(int v, int w) {
        rootify(w);
        t[w].p = v;
    }

    void cut(int v, int w) {

```

```

        rootify(w), access(v);
        t[v].ch[0] = t[t[v].ch[0]].p = -1;
    }
    int lca(int v, int w) {
        access(v);
        return access(w);
    }
}

```

## 3 Dynamic Programming

### 3.1 Line Container

```

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() {
        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 optr) {
    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, optr); k++) {
    best = min(best, {dpold[k-1] + c[k][mid], k});
}
dp[mid] = best.first;
int opt = best.second;
compute(1, mid-1, optl, opt);
compute(mid+1, r, opt, optr);
}
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(
                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

```
//is n primitive root of p ?
bool test(long long x, long long p) {
    long long 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 smallest primitive root for p
int search(int p) {
    for(int i = 2; i < p; i++) if(test(i, p)) return i;
    return -1;
}
```

## 4.8 Linear Sieve

```
//check long long
vector<int> prime;
bool is_composite[MAXN];
int cnt[MAXN];
long long primePow[MAXN];
long long func[MAXN];

long long getFunction(int i, int p) {
    return cnt[i] + 1;
}

void sieve (int n) {
    fill(is_composite, is_composite + n, false);
    func[1] = 1;
    for (int i = 2; i < n; ++i) {
        if (!is_composite[i]) {
            prime.push_back (i);
            func[i] = 1; // base case
            cnt[i] = 1; primePow[i] = i;
        }
        for (int j = 0; j < prime.size () && i * prime[j] < n; ++j) {
            is_composite[i * prime[j]] = true;
            if (i % prime[j] == 0) {
                func[i * prime[j]] = func[i / primePow[i]] * getFunction(i,
                    prime[j]); // f(ip) = f(i / primePow[i]) * f(primePow[i] *
                    prime[j])
                cnt[i * prime[j]] = cnt[i] + 1;
                primePow[i * prime[j]] = primePow[i] * prime[j];
                break;
            } else {
                func[i * prime[j]] = func[i] * func[prime[j]]; // f(ip) = f(i)
                    * f(p)
                cnt[i * prime[j]] = 1;
                primePow[i * prime[j]] = prime[j];
            }
        }
    }
}
```

## 4.9 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);
}
```

## 4.10 Fast Exponentiation

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

int fexp(int a, int b) {
    int ans = 1;
    while(b) {
        if(b & 1) ans = ans * a % mod;
        a = a * a % mod;
        b >>= 1;
    }
    return ans;
}
```

## 4.11 Matrix

```
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;
    }
};
```

## 4.12 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) {
    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;
}

```

## 4.13 NTT - Number Theoretic Transform

```

const int MOD = 998244353;
const int me = 15;
const int ms = 1 << me;

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

const int gen = 3; // use search() from PrimitiveRoot.cpp if MOD isn't
998244353

int bits[ms], root[ms];
void initFFT() {
    root[1] = 1;
    for(int len = 2; len < ms; len *= 2) {
        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, 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);
    }
}

```

## 4.14 Fast Walsh Hadamard Transform

```

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

```

## 4.15 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;
}

```

## 4.16 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;
}

```

## 4.17 Gauss

```

const double eps = 1e-7;

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

```

```

    vector<int> where (m, -1);
    for (int col=0, row=0; col<m && row<n; ++col) {
        int sel = row;
        for (int i=row; i<n; ++i){
            if (abs (a[i][col]) > abs (a[sel][col]))
                sel = i;
        }
        if (abs (a[sel][col]) < eps) continue;
        for (int i=col; i<=m; ++i)
            swap (a[sel][i], a[row][i]);
        where[col] = row;
        for (int i=0; i<n; ++i){
            if (i != row) {
                double c = a[i][col] / a[row][col];
                for (int j=col; j<=m; ++j)
                    a[i][j] -= a[row][j] * c;
            }
        }
        ++row;
    }
    ans.assign (m, 0);
    for (int i=0; i<m; ++i){
        if (where[i] != -1)
            ans[i] = a[where[i]][m] / a[where[i]][i];
    }
    for (int i=0; i<n; ++i) {
        double sum = 0;
        for (int j=0; j<m; ++j)
            sum += ans[j] * a[i][j];
        if (abs (sum - a[i][m]) > eps)
            return 0;
    }
    for (int i=0; i<m; ++i){
        if (where[i] == -1)
            return INF;
    }
    return 1;
}

// mod 2 (xor);
int gauss (vector<bitset<ms>> a, int m) {
    int n = (int) a.size();
    vector<int> where (m, -1);
    for (int col=0, row=0; col<m && row<n; ++col) {
        for (int i=row; i<n; ++i){
            if (a[i][col]) {
                swap (a[i], a[row]);
                break;
            }
        }
        if (!a[row][col]) continue;
        where[col] = row;
        for (int i=0; i<n; ++i){
            if (i != row && a[i][col])
                a[i] ^= a[row];
        }
        ++row;
    }
    //same above
}

```

## 4.18 Lagrange Interpolation

```
class LagrangePoly {
public:
    LagrangePoly(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];
        }
        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:
    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;
    cin >> x >> k;
    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);
    cout << f.getVal(x) << '\n';
}
```

## 4.19 Lagrange extracting polynomial

```
// O(n^2), recebe v {x, y} e retorna o polinomio em fracao
vector<pair<int, int>> interpolate(vector<ii> v) {
    int n = v.size();
    vector<int> prod(n+1);
    prod[0] = 1;
    for(auto p : v) {
        for(int i = n; i > 0; i--) {
            prod[i] = prod[i-1] - p.first * prod[i];
        }
        prod[0] = -p.first * prod[0];
    }
    vector<pair<int, int>> ans(n+1);
    for(int i = 0; i <= n; i++) ans[i].second = 1;
    for(int i = 0; i < n; i++) {
        vector<int> pol(n+1); // (x - v[i].first)
        for(int j = n; j > 0; j--) {
            pol[j-1] = prod[j] + pol[j] * v[i].first;
        }
        for(int j = 0; j < n; j++) {
            pol[j] *= v[i].second;
        }
        int k = 1;
        for(int j = 0; j < n; j++) {
            if(i==j) continue;
            k *= v[i].first - v[j].first;
        }
        if(k < 0) {
            k = -k;
            for(auto &p : pol) p = -p;
        }
        for(int i = 0; i < n; i++) {
            ans[i] = {ans[i].first*k + pol[i]*ans[i].second, k*ans[i].second};
        }
        if(ans[i].first == 0) ans[i].second = 1;
        else {
            int gc = __gcd(abs(ans[i].first), ans[i].second);
            ans[i].first /= gc;
            ans[i].second /= gc;
        }
    }
    return ans;
}
```

## 4.20 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;
}
}

```

## 4.21 Prime Counting

```

const int ms = 5001000, lim_n = 3e5, lim_p = 1e2;
std::vector<int> primes;
int id[ms];
int memo[lim_n][lim_p];
void pre() {
    std::vector<bool> isPrime(ms, true);
    for(int i = 2; i < ms; i++) {
        id[i] = (int) primes.size();
        if(!isPrime[i]) continue;
        id[i]++;
        primes.push_back(i);
        for(int j = i+1; j < ms; j += i) isPrime[j] = false;
    }
    for(int i = 1; i < lim_n; i++) {
        memo[i][0] = i;
        for(int j = 1; j < lim_p; j++) memo[i][j] = memo[i][j-1] - memo[i/
            primes[j-1]][j-1];
    }
}
int cbc(long long n) {
    int ans = std::max(0, (int) pow((double) n, 1.0 / 3) - 2);
    while((ll) ans * ans * ans < n) ans++;
    return ans;
}
long long dp(long long n, int i) {
    if(n == 0) return 0; if(i == 0) return n;
    if(primes[i-1] >= n) return 1;
    if((ll) primes[i-1] * primes[i-1] > n && n < ms) return id[n] - (i
        - 1);
    else if(n < lim_n && i < lim_p) return memo[n][i];
    else return dp(n, i-1) - dp(n / primes[i-1], i-1);
}
long long primeFunction(long long n) {
    if(n < ms) return id[(int)n];
    int i = id[cbc(n)];
    long long ans = dp(n, i) + i - 1;
    while((long long) primes[i] * primes[i] <= n) {
        ans -= primeFunction(n / primes[i]) - i;
        i++;
    }
    return ans;
}
}

```

## 4.22 Berlekamp Massey

```

vector<int> berlekampMassey(const vector<int> &s) {
    int n = (int) s.size(), l = 0, m = 1;
    vector<int> b(n), c(n);
    int ld = b[0] = c[0] = 1;
    for (int i=0; i<n; i++, m++) {
        int d = s[i];
        for (int j=1; j<=l; j++)
            d = (d + c[j] * s[i-j]) % mod;
        if (d == 0) continue;
        vector<int> temp = c;
        int coef = d * fexp(ld, mod-2) % mod;
        for (int j=m; j<n; j++)
            (c[j] = (c[j] - coef * b[j-m]) % mod + mod) % mod;
        if (2 * l <= i) {
            l = i + 1 - l;
            b = temp;
            ld = d;
            m = 0;
        }
    }
    c.resize(l + 1);
    c.erase(c.begin());
    for (int &x : c)
        x = mod-x;
    return c;
}

// p = p*q % h
void mull(vector<int> &p, vector<int> &q, vector<int> &h, int m) {
    vector<int> t_(m+m);
    for(int i=0; i<m; ++i) if(p[i])
        for(int j=0; j<m; ++j)
            t_[i+j] = (t_[i+j] + p[i] * q[j]) % mod;
    for(int i=m+m-1; i>=m; --i) if(t_[i])
        //miuns t_[i]x^{i-m} (x^{m-\sum_{j=0}^{m-1} x^{m-j-1}}h_j
        )
        for(int j=m-1; ~j; --j)
            t_[i-j-1] = (t_[i-j-1] + t_[i] * h[j]) % mod;
    for(int i=0; i<m; ++i) p[i] = t_[i];
}

// a = caso base, h = recorrência, m = tamanho da recorrência
inline int calc(vector<int> &a, vector<int> &h, int K, int m) {
    vector<int> s(m), t(m);
    //init
    s[0]=1; if(m!=1) t[1]=1; else t[0]=h[0];
    //binary-exponentiation
    while(K) {
        if(K&1) mull(s,t,h,m);
        mull(t,t,h,m); K>>=1;
    }
    int su=0;
    for(int i=0; i<m; ++i) su = (su+s[i]*a[i]) % mod;
    return (su%mod+mod) % mod;
}

```

## 5 Geometry

### 5.1 Geometry

```
const double inf = 1e100, eps = 1e-12;
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); }
};
ostream &operator<<(ostream &os, const PT &p) {
    os << "(" << p.x << ", " << p.y << ")";
}
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;
}
PT rotateCCW90 (PT p) { return PT(-p.y, p.x); }
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));
}
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)};
}
PT projPtLine (PT a, PT b, PT c) { // ponto c na linha a - b, a.b = |a
```

```
    | cost * |b|
    return a + (b-a) * dot(b-a, c-a)/dot(b-a, b-a);
}
PT reflectPointLine (PT a, PT b, PT c) {
    PT p = projPtLine(a, b, c);
    return p*2 - c;
}
PT projPtSeg (PT a, PT b, PT c) { // c no segmento a - b
    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;
}
double distancePointSegment (PT a, PT b, PT c) { // ponto c e o
    segmento a - b
    return dist(c, projPtSeg(a, b, c));
}
bool ptInSegment (PT a, PT b, PT c) { // ponto c esta em um segmento a
    - b
    if (a == b) return a == c;
    a = a-c, b = b-c;
    return cmp(cross(a, b)) == 0 && cmp(dot(a, b)) <= 0;
}
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 distPtPlane(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
    );
}
bool segInter (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
PT lineLine (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);
}
```



```

}
PT circleCenter (PT a, PT b, PT c) {
    b = (a + b) / 2; // bissector
    c = (a + c) / 2; // bissector
    return lineLine(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;
}
bool circleLineIntersection(PT a, PT b, PT c, double r) {
    return cmp(dist(c, projPtLine(a, b, c)), r) <= 0;
}
vector<PT> circleLine (PT a, PT b, PT c, double r) {
    vector<PT> ret;
    PT p = projPtLine(a, b, c), p1;
    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);
        p1 = p + (b-a)/(norm(b-a))*k;
        ret.push_back(p1);
        p1 = p - (b-a)/(norm(b-a))*k;
        ret.push_back(p1);
    }
    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;
}
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 poligno

```

```

// e 0 ou 1 para os pontos restantes
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 poligno
bool pointOnPolygon(const vector<PT> &p, PT q) {
    for(int i = 0; i < p.size(); i++)
        if(cmp(dist(projPtSeg(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));
}
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;
}
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 == 1 || j == k) continue;
            if (segInter(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); // se der wa bota 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;
}

```

```

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;
}
// uniao de convex hulls
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[1], vec) > dot(hull[ans], vec)) {
            ans = 1;
        }
        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>(0);
    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) {
        // 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.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.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 Aho-Corasick

```
const int ms = 1e6; // quantidade de caracteres
const int sigma = 26; // tamanho do alfabeto
int trie[ms][sigma], fail[ms], superfail[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), memset(superfail, 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];
        // int sf = present[f] ? f : superfail[f];
        // present marks if that vertex is a terminal node or not
        // if summing up on terminal, doesn't work
        if (v == -1) {
            v = f;
        } else {
            fail[v] = f;
            // superfail[v] = sf;
            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])
    }
}

// se tiver usando super fail, cuidado com o estado que voce ta, antes
// de subir pro sf, porque pode ser que o estado que ta nao seja no
// terminal

```

## 6.4 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;
}

```

## 6.5 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;
}

```

## 6.6 Suffix Tree

```

#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

```

## 6.7 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;
}

```

## 6.8 Manacher

```

std::array<std::vector<int>, 2> manacher(const std::string& s) {
    int n = (int) s.size();
    std::array<std::vector<int>, 2> p = {std::vector<int>(n+1),
        std::vector<int>(n)};
    for(int z = 0; z < 2; z++) for (int i = 0, l = 0, r = 0; i < n
        ; i++) {
        int t = r-i+!z;
        if (i<r) p[z][i] = std::min(t, p[z][l+t]);
        int L = i-p[z][i], R = i+p[z][i]-!z;
        while (L>=1 && R+1<n && s[L-1] == s[R+1])
            p[z][i]++, L--, R++;
        if (R>r) l=L, r=R;
    }
    return p;
} // pra cada centro o tamanho max do palindromo centrado ali,
    qualquer coisa printa a saida pra abacabaab

```

## 6.9 Polish Notation

```

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;
}

```

## 6.10 String Hash

```

struct StringHashing {
    const uint64_t MOD = (1LL << 61) - 1;
    const int base = 31;
    vector<uint64_t> h, p;

    uint64_t modMul(uint64_t a, uint64_t b) {
        uint64_t l1 = (uint32_t)a, h1 = a >> 32, l2 = (uint32_t)b, h2 = b
            >> 32;
        uint64_t l = l1 * l2, m = l1 * h2 + l2 * h1, h = h1 * h2;
        uint64_t ret = (l & MOD) + (l >> 61) + (h << 3) + (m >> 29) + ((m
            << 35) >> 3) + 1;
        ret = (ret & MOD) + (ret >> 61);
        ret = (ret & MOD) + (ret >> 61);
        return ret - 1;
    }

    uint64_t getKey(int l, int r) { // [l, r]
        uint64_t res = h[r];
        if (l > 0) res = (res + MOD - modMul(p[r - l + 1], h[l - 1])) % MOD;
        return res;
    }

    uint64_t getInt(char c) {
        return c - 'a' + 1;
    }

    StringHashing(string &s) {
        int n = s.size();
        h.resize(n);
        p.resize(n);
        p[0] = 1;
    }
}

```

```

h[0] = getInt(s[0]);
for (int i = 1; i < n; ++i) {
    p[i] = modMul(p[i - 1], base);
    h[i] = (modMul(h[i - 1], base) + getInt(s[i])) % MOD;
}
};

```

## 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)) * 0x94d049bb133111eb;
            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(rect other) {
        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() {
    safe_hash sh;
    uint64_t ret = sh(x1);
    ret ^= sh(ret ^ y1);
    ret ^= sh(ret ^ x2);
    ret ^= sh(ret ^ y2);
    return ret;
}
};
```

## 7.5 Safe 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)) * 0x94d049bb133111eb;
        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);
    }
};
```

## 7.6 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);
```

## 7.7 Iterate masks in bitcount order

```
for(int k = n-1; k >= 0; k--) {
    unsigned int i = (1 << k) - 1;
    while(i < (1 << n)) {
        // do what you want
        unsigned int t = (i | (i - 1)) + 1;
        if(i == 0) break;
        i = t | (((t & -t) / (i & -i)) >> 1) - 1;
    }
}
```

## 7.8 Submask Enumeration

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

## 7.9 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)];
    }
}

```

## 7.10 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];
}

```

## 7.11 Regular Expressions

```

import java.util.*;
import java.util.regex.*;

public class Main {
    public static String BuildRegex () {
        return "^" + sentence + "$";
    }
    public static void main (String args[]){
        String regex = BuildRegex();
        // check pattern documentation
        Pattern pattern = Pattern.compile (regex);
        Scanner s = new Scanner(System.in);
        String sentence = s.nextLine().trim();
        boolean found = pattern.matcher(sentence).find()
    }
}

```

```

    }
}

```

## 7.12 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(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(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;
}

```

## 7.13 Stable Marriage

```

std::vector<std::vector<int>> stableMarriage(std::vector<std::vector<
int>> first, std::vector<std::vector<int>> second, std::vector<int>
> cap) {
    assert(cap.size() == second.size());
    int n = (int) first.size(), m = (int) second.size();
    // if O(N * M) first in memory, use table
    std::map<std::pair<int, int>, int> prio;
    std::vector<std::set<std::pair<int, int>>> current(m);
    for(int i = 0; i < n; i++) {
        std::reverse(first[i].begin(), first[i].end());
    }
    for(int i = 0; i < m; i++) {
        for(int j = 0; j < (int) second[i].size(); j++) {
            prio[{second[i][j], i}] = j;
        }
    }
    for(int i = 0; i < n; i++) {

```

```

int on = i;
while(!first[on].empty()) {
    int to = first[on].back();
    first[on].pop_back();
    if(cap[to]) {
        cap[to]--;
        assert(prio.count({on, to}));
        current[to].insert({prio[{on, to}], on});
        break;
    }
    assert(!current[to].empty());
    auto it = current[to].end();
    it--;
    if(it->first > prio[{on, to}]) {
        int nxt = it->second;
        current[to].erase(it);
        current[to].insert({prio[{on, to}], on});
        on = nxt;
    }
}
std::vector<std::vector<int>> ans(m);
for(int i = 0; i < m; i++) {
    for(auto it : current[i]) {
        ans[i].push_back(it.second);
    }
}
return ans;
}

```

```

qnt[x]--;
if (qnt[x] == 0) {
    diff--;
}

void mos() {
    int curr_l = 0, curr_r = -1;
    sort(queries.begin(), queries.end());
    for (Query q : queries) {
        while (curr_l > q.l) {
            curr_l--;
            add(curr_l);
        }
        while (curr_r < q.r) {
            curr_r++;
            add(curr_r);
        }
        while (curr_l < q.l) {
            remove(curr_l);
            curr_l++;
        }
        while (curr_r > q.r) {
            remove(curr_r);
            curr_r--;
        }
        ans[q.idx] = diff;
    }
}

```

## 7.14 Mo

```

const int blk_sz = 170;

struct Query {
    int l, r, idx;
    bool operator < (Query a) {
        if (l / blk_sz == a.l / blk_sz) {
            return r < a.r;
        }
        return (l / blk_sz) < (a.l / blk_sz);
    }
};

vector<Query> queries;
int a[MAXN], ans[MAXN], qnt[1000010];
int diff = 0;

void add(int x) {
    x = a[x];
    if (qnt[x] == 0) {
        diff++;
    }
    qnt[x]++;
}

void remove(int x) {
    x = a[x];

```

## 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  $n$   
 Boruvka's: enquanto grafo nao conexo, para cada componente conexa use a aresta que sai de menor custo.

Trees:  
 Tem Catalan(N) Binary trees de N vertices  
 Tem Catalan(N-1) Arvores enraizadas com N vertices  
 Caley 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:  
 Recuperar min cut eh ver se  $\text{level}[u] \neq -1$  ai eh do lado do source  
 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 vertex cover sao os vertices da particao do source que nao tao do lado do source do cut e os do sink que tao do lado do source, independent set o contrario

Min edge cover eh  $N - \text{match}$ , pega as arestas do match mais qualquer aresta dos outros vertices

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  $N - \text{matching}$

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

feasible flow in a network with both upper and lower capacity constraints, no source or sink: capacities are changed to upper bound - lower bound. Add a new source and a sink. let  $M[v] = (\text{sum of lower bounds of ingoing edges to } v) - (\text{sum of lower bounds of outgoing edges from } v)$ . For all v, if  $M[v] > 0$  then add edge (S,v) with capacity M, otherwise add (v,T) with capacity -M. If all outgoing edges from S are full, then a feasible flow exists, it is the flow plus the original lower\_bounds

## 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  $a^{(p-1)} \% p = 1$

Se x e m tambem forem coprimos entao  $x^k \% m = x^{(k \bmod (m-1))} \% m$

Euler's theorem:  $x^{(\phi(m))} \bmod m = 1$  onde  $\phi(m)$  eh o totiente de euler

Chinese remainder theorem:

Para equacoes no formato  $x = a_1 \bmod m_1, \dots, x = a_n \bmod m_n$  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 = 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, ..., b Expected[X] =  $(a+b)/2$

Distribuicao binomial com n tentativas de probabilidade p, X = 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 = 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  $aX + bY = aE[X] + bE[X]$

$V(X) = E((X-u)^2)$

$V(X) = E(X^2) - E(X^2)$

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

Mobius Inverse:  $\text{Sum}(d|n): \text{mobius}(d) = [n = 1]$  (expressao booleana)

Soma dos cubos de 1 a N =  $a^2$  onde a = somatorio de 1 a N

Lindstrom-Gessel-Viennot: quantidade de caminhos disjuntos nas linhas do grid eh o determinante da matriz de qnts caminhos

## 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  
 $a$  = 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  
 $s = (a+b+c+d)/2$   
 $area = \sqrt{(s-a)*(s-b)*(s-c)*(s-d)}$   
 $d = 0 \Rightarrow area = \sqrt{(s-a)*(s-b)*(s-c)*s}$

---

## 8.4 Dynamic Programming

Divide and conquer -  $dp[i][j] = \min_k \{ dp[i-1][k] + C[k][j] \}$   
 dividir o subsegmento ate  $j$  em  $i$  segmentos com custo, valido se  $A[i][j]$   
 $j \leq A[i][j+1]$

Knuth -  $p[i][j] = \min_k \{ dp[i][k] + dp[k][j] \} + C[i][j]$ , valido  
 se  $A[i, j-1] \leq A[i][j] \leq A[i+1, j]$   
 onde  $A[i][j]$  eh o menor  $k$  que da a resposta otima  
 slope trick - funcao piecewise linear convexa, descrita pelos pontos  
 de mudanca de slope (multiset/heap)  
 lembre que existe fft, cht, aliens trick e bitset

---

## 8.5 Mersenne's Primes

Primos de Mersenne  $2^n - 1$   
 Lista de  $N$ s 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;

---