

## The AC is a lie - ICPC Library

## Contents

## 1 String Algorithms

1.1	String Alignment	1
1.2	KMP	1
1.3	Trie	2
1.4	Aho-Corasick	2
1.5	Algoritmo de Z	2
1.6	Suffix Array	2

## 2 Data Structures

2.1	BIT - Binary Indexed Tree	3
2.2	BIT 2D	3
2.3	BIT 2D Comprimida	3
2.4	Iterative Segment Tree	4
2.5	Iterative Segment Tree with Interval Updates	4
2.6	Iterative Segment Tree with Lazy Propagation	4
2.7	Recursive Segment Tree	5
2.8	Segment Tree with Lazy Propagation	6
2.9	Persistent Segment Tree	6
2.10	Treap	7
2.11	Sparse Table	7
2.12	Policy Based Structures	7
2.13	Color Updates Structure	8
2.14	Centroid Decomposition	8
2.15	Li Chao Tree	9

## 3 Graph Algorithms

3.1	Dinic Max Flow	9
3.2	Euler Path and Circuit	10
3.3	Articulation Points/Bridges/Biconnected Components	10
3.4	SCC - Strongly Connected Components / 2SAT	10
3.5	LCA - Lowest Common Ancestor	11
3.6	Heavy Light Decomposition	11
3.7	Sack	12
3.8	Min Cost Max Flow	12
3.9	Hungarian Algorithm - Maximum Cost Matching	13

## 4 Math

4.1	Discrete Logarithm	13
4.2	GCD - Greatest Common Divisor	14
4.3	Extended Euclides	14
4.4	Fast Exponentiation	14
4.5	Matrix Fast Exponentiation	14
4.6	FFT - Fast Fourier Transform	14
4.7	NTT - Number Theoretic Transform	15
4.8	Miller and Rho	16
4.9	Determinant using Mod	16

## 5 Geometry

5.1	Geometry	17
5.2	Convex Hull	17
5.3	Closest Pair	19
5.4	Intersection Points	19
5.5	Delaunay Triangulation	19
5.6	Java Geometry Library	21

## 6 Dynamic Programming

6.1	Convex Hull Trick	22
6.2	Divide and Conquer Optimization	23
6.3	Knuth Optimization	23

## 7 Miscellaneous

7.1	LIS - Longest Increasing Subsequence	23
7.2	Ternary Search	24

7.3	Random Number Generator	24
7.4	Submask Enumeration	24
7.5	Java Fast I/O	24

## 8 Teoremas e formulas uteis

8.1	Grafos	25
8.2	Math	25
8.3	Geometry	26
8.4	Mersenne's Primes	26

## 1 String Algorithms

## 1.1 String Alignment

```
int pd[ms][ms];

int edit_distance(string &a, string &b) {
    int n = a.size(), m = b.size();
    for(int i = 0; i <= n; i++) pd[i][0] = i;
    for(int j = 0; j <= m; j++) pd[0][j] = j;
    for(int i = 1; i <= n; i++) {
        for(int j = 1; j <= m; j++) {
            int del = pd[i][j-1] + 1;
            int ins = pd[i-1][j] + 1;
            int mod = pd[i-1][j-1] + (a[i-1] != b[j-1]);
            pd[i][j] = min(del, min(ins, mod));
        }
    }
    return pd[n][m];
}
```

## 1.2 KMP

```
string p, t;
int b[ms], n, m;

void kmpPreprocess() {
    int i = 0, j = -1;
    b[0] = -1;
    while(i < m) {
        while(j >= 0 && p[i] != p[j]) j = b[j];
        b[++i] = ++j;
    }
}

void kmpSearch() {
    int i = 0, j = 0, ans = 0;
    while(i < n) {
        while(j >= 0 && t[i] != p[j]) j = b[j];
        i++; j++;
        if(j == m) {
            //ocorrencia aqui comecando em i - j
            ans++;
            j = b[j];
        }
    }
    return ans;
}
```

## 1.3 Trie

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

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

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

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

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

## 1.4 Aho-Corasick

```
// Construa a Trie do seu dicionario com o codigo acima

int fail[ms];
queue<int> q;

void buildFailure() {
    q.push(0);
    while(!q.empty()) {
        int node = q.front();
        q.pop();
        for(int pos = 0; pos < sigma; pos++) {
            int &v = trie[node][pos];
            int f = node == 0 ? 0 : trie[fail[node]][pos];
            if(v == -1) {
                v = f;
            } else {
                fail[v] = f;
                q.push(v);
            }
        }
    }
}
```

```
// juntar as informacoes da borda para o V ja q um match em V
// implica um match na borda
terminal[v] += terminal[f];
}
}
}

int search(string &txt) {
    int node = 0;
    int ans = 0;
    for(int i = 0; i < txt.length(); i++) {
        int pos = get_id(txt[i]);
        node = trie[node][pos];
        // processar informacoes no no atual
        ans += terminal[node];
    }
    return ans;
}
```

## 1.5 Algoritmo de Z

```
string s;
int fz[ms], n;

void zfunc() {
    fz[0] = n;
    for(int i = 1, l = 0, r = 0; i < n; i++) {
        if(l && i + fz[i-1] < r)
            fz[i] = fz[i-1];
        else {
            int j = min(l ? fz[i-1] : 0, i > r ? 0 : r - i);
            while(s[i+j] == s[j] && ++j);
            fz[i] = j; l = i; r = i + j;
        }
    }
}
```

## 1.6 Suffix Array

```
namespace SA {
    typedef pair<int, int> ii;

    vector<int> buildSA(string s) {
        int n = (int) s.size();
        vector<int> ids(n), pos(n);
        vector<ii> pairs(n);
        for(int i = 0; i < n; i++) {
            ids[i] = i;
            pairs[i] = ii(s[i], -1);
        }
        sort(ids.begin(), ids.end(), [&](int a, int b) -> bool {
            return pairs[a] < pairs[b];
        });
        int on = 0;
        for(int i = 0; i < n; i++) {
            if (i && pairs[ids[i-1]] != pairs[ids[i]]) on++;
            pos[ids[i]] = on;
        }
    }
}
```

```

}
for(int offset = 1; offset < n; offset <= 1) {
    //ja tao ordenados pelos primeiros offset caracteres
    for(int i = 0; i < n; i++) {
        pairs[i].first = pos[i];
        if (i + offset < n) {
            pairs[i].second = pos[i + offset];
        } else {
            pairs[i].second = -1;
        }
    }
    sort(ids.begin(), ids.end(), [&](int a, int b) -> bool {
        return pairs[a] < pairs[b];
    });
    int on = 0;
    for(int i = 0; i < n; i++) {
        if (i && pairs[ids[i - 1]] != pairs[ids[i]]) on++;
        pos[ids[i]] = on;
    }
    return ids;
}

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;
}
};

```

## 2 Data Structures

### 2.1 BIT - Binary Indexed Tree

```

int bit[ms], n;

void update(int v, int idx) {
    while(idx <= n) {
        bit[idx] += v;
        idx += idx & -idx;
    }
}

int query(int idx) {

```

```

    int r = 0;
    while(idx > 0) {
        r += bit[idx];
        idx -= idx & -idx;
    }
    return r;
}

```

### 2.2 BIT 2D

```

int bit[ms][ms], n, m;

void update(int v, int x, int y) {
    while(x <= n) {
        while(y <= m) {
            bit[x][y] += v;
            y += y & -y;
        }
        x += x & -x;
    }
}

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

```

### 2.3 BIT 2D Comprimida

```

// by TFG
#include <vector>
#include <utility>
#include <algorithm>

typedef std::pair<int, int> ii;

struct Bit2D {
public:
    Bit2D(std::vector<ii> pts) {
        std::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)
            std::swap(a.first, a.second);
        std::sort(pts.begin(), pts.end());
        for(auto &a : pts) {
            std::swap(a.first, a.second);

```

```

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

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

private:
    std::vector<int> ord;
    std::vector<std::vector<int>> fw, coord;
};

```

## 2.4 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);
}

```

## 2.5 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;
    }
}

```

## 2.6 Iterative Segment Tree with Lazy Propagation

```

struct LazyContext {
    int v;

    LazyContext(int v = 0) : v(v) {}

    void reset() {
        v = 0;
    }

    void operator += (LazyContext o) {
        v += o.v;
    }
};

struct Node {
    int sz, v;
}

```

```

Node() { // neutral element
    v = 0; sz = 0;
}

Node(int i) { // init
    v = i; sz = 1;
}

Node(Node &l, Node &r) { // merge
    sz = l.sz + r.sz;
    v = l.v + r.v;
}

void apply(LazyContext lazy) {
    v += lazy.v * sz;
}

Node tree[2*ms];
LazyContext lazy[ms];
bool dirty[ms];
int n, h, a[ms];

void init() {
    h = 0;
    while((1 << h) < n) h++;
    for(int i = 0; i < n; i++) {
        tree[i + n] = Node(a[i]);
    }
    for(int i = n - 1; i > 0; i--) {
        tree[i] = Node(tree[i + 1], tree[i + 1 + 1]);
        lazy[i].reset();
        dirty[i] = 0;
    }
}

void apply(int p, LazyContext &lc) {
    tree[p].apply(lc);
    if(p < n) {
        dirty[p] = true;
        lazy[p] += lc;
    }
}

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

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

```

```

}

Node query(int l, int r) {
    if(l > r) return Node();
    l += n, r += n+1;
    push(l);
    push(r - 1);
    Node lp, rp;
    for(; l < r; l /= 2, r /= 2) {
        if(l & 1) lp = Node(lp, tree[l++]);
        if(r & 1) rp = Node(tree[--r], rp);
    }
    return Node(lp, rp);
}

void update(int l, int r, LazyContext lc) {
    if(l > r) return;
    l += n, r += n+1;
    push(l);
    push(r - 1);
    int lo = l, ro = r;
    for(; l < r; l /= 2, r /= 2) {
        if(l & 1) apply(l++, lc);
        if(r & 1) apply(--r, lc);
    }
    build(lo);
    build(ro - 1);
}

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

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

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

void update(int V, int I, int idx = 0, int l = 0, int r = n - 1) {
    int mid = (l+r)/2, left = 2 * idx + 1, right = 2 * idx + 2;
    if(l > I || r < I) return;
    if(l == r) {
        arr[I] = V;
        seg[idx] = V; // Aplicar Update
        return;
    }
    build(left, l, mid); build(right, mid + 1, r);
    seg[idx] = seg[left] + seg[right]; // Merge
}

```

## 2.7 Recursive Segment Tree

```

    }
    update(V, I, left, l, mid); update(V, I, right, mid + 1, r);
    seg[idx] = seg[left] + seg[right]; // Merge
}

```

## 2.8 Segment Tree with Lazy Propagation

```

int arr[4 * 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, left = 2 * idx + 1, right = 2 * idx + 2;
    lazy[idx] = 0;
    if(l == r) {
        seg[idx] = arr[l];
        return;
    }
    build(left, l, mid); build(right, mid + 1, r);
    seg[idx] = seg[left] + seg[right]; // Merge
}

void propagate(int idx, int l, int r) {
    int mid = (l+r)/2, left = 2 * idx + 1, right = 2 * idx + 2;
    if(lazy[idx]) {
        seg[idx] += lazy[idx] * (r - l + 1); // Aplicar lazy no seg
        if(l < r) {
            lazy[2*idx+1] += lazy[idx]; // Merge de lazy
            lazy[2*idx+2] += lazy[idx]; // Merge de lazy
        }
        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, left = 2 * idx + 1, right = 2 * idx + 2;
    propagate(idx, l, r);
    if(R < l || L > r) return 0; // Valor que nao atrapalhe
    if(L <= l && r <= R) return seg[idx];
    return query(L, R, left, l, mid) + query(L, R, right, mid + 1, r);
    // Merge
}

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

```

## 2.9 Persistent Segment Tree

```

struct PSEGTREE{

```

```

private:
    int z, t, sz, *tree, *L, *R, head[112345];

    void _build(int l, int r, int on, vector<int> &v) {
        if(l == r) {
            tree[on] = v[l];
            return;
        }
        L[on] = ++z;
        int mid = (l+r)>>1;
        _build(l, mid, L[on], v);
        R[on] = ++z;
        _build(mid+1, r, R[on], v);
        tree[on] = tree[L[on]] + tree[R[on]];
    }

    int _upd(int ql, int qr, int val, int l, int r, int on) {
        if(l > qr || r < ql) return on;
        int curr = ++z;
        if(l >= ql && r <= qr) {
            tree[curr] = tree[on] + val;
            return curr;
        }
        int mid = (l+r)>>1;
        L[curr] = _upd(ql, qr, val, l, mid, L[on]);
        R[curr] = _upd(ql, qr, val, mid+1, r, R[on]);
        tree[curr] = tree[L[curr]] + tree[R[curr]];
        return curr;
    }

    int _query(int ql, int qr, int l, int r, int on) {
        if(l > qr || r < ql) return 0;
        if(l >= ql && r <= qr) {
            return tree[on];
        }
        int mid = (l+r)>>1;
        return _query(ql, qr, l, mid, L[on]) + _query(ql, qr, mid+1, r, R[on]);
    }

public:
    PSEGTREE(vector<int> &v) {
        tree = new int[1123456];
        L = new int[1123456];
        R = new int[1123456];
        build(v);
    }

    void build(vector<int> &v) {
        t = 0, z = 0;
        sz = v.size();
        head[0] = 0;
        _build(0, sz-1, 0, v);
    }

    void upd(int pos, int val, int idx) {
        head[++t] = _upd(pos, pos, val, 0, sz-1, head[idx]);
    }

    int query(int l, int r, int idx) {
        return _query(l, r, 0, sz-1, head[idx]);
    }
}

```

```
};
```

## 2.10 Treap

```
typedef struct item * pitem;
struct item {
    int prior, value, cnt;
    bool rev;
    pitem l, r;
};

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

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

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

void merge (pitem & t, pitem l, pitem r) {
    push (l), push (r);
    if (!l || !r) t = l ? l : r;
    else if (l->prior > r->prior)
        merge (l->r, l->r, r), t = l;
    else
        merge (r->l, l, r->l), t = r;
    upd_cnt (t);
}

void split (pitem t, pitem & l, pitem & r, int key) {
    if (!t) return void( l = r = 0 );
    push (t);
    int cur_key = cnt(t->l);
    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;
    upd_cnt (t);
}

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

pitem unite (pitem l, pitem r) {
    if (!l || !r) return l ? l : r;
    if (l->prior < r->prior) swap (l, r);
    pitem lt, rt;
```

```
split (r, l->key, lt, rt);
l->l = unite (l->l, lt);
l->r = unite (l->r, rt);
return l;
```

```
}
```

## 2.11 Sparse Table

```
template<class Info_t>
class SparseTable {
private:
    vector<int> log2;
    vector<vector<Info_t>> table;

    Info_t merge(Info_t &a, Info_t &b) {

    }

public:
    SparseTable(int n, vector<Info_t> v) {
        log2.resize(n + 1);
        log2[1] = 0;
        for (int i = 2; i <= n; i++) {
            log2[i] = log2[i >> 1] + 1;
        }
        table.resize(n + 1);
        for (int i = 0; i < n; i++) {
            table[i].resize(log2[n] + 1);
        }
        for (int i = 0; i < n; i++) {
            table[i][0] = v[i];
        }
        for (int i = 0; i < log2[n]; i++) {
            for (int j = 0; j < n; j++) {
                if (j + (1 << i) >= n) break;
                table[j][i + 1] = merge(table[j][i], table[j + (1 << i)][i]);
            }
        }

        int get(int l, int r) {
            int k = log2[r - l + 1];
            return merge(table[l][k], table[r - (1 << k) + 1][k]);
        }
    };
};
```

## 2.12 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);

```

## 2.13 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;
}

```

## 2.14 Centroid Decomposition

```

//Centroid decomposition1

void dfsSize(int v, int pa) {
    sz[v] = 1;
    for(int u : adj[v]) {
        if (u == pa || rem[u]) continue;
        dfsSize(u, v);
        sz[v] += sz[u];
    }
}

int getCentroid(int v, int pa, int tam) {
    for(int u : adj[v]) {
        if (u == pa || rem[u]) continue;
        if (2 * sz[u] > tam) return getCentroid(u, v, tam);
    }
    return v;
}

void decompose(int v, int pa = -1) {
    //cout << v << ' ' << pa << '\n';
    dfsSize(v, pa);
    int c = getCentroid(v, pa, sz[v]);
    //cout << c << '\n';
    par[c] = pa;
    rem[c] = 1;
    for(int u : adj[c]) {
        if (!rem[u] && u != pa) decompose(u, c);
    }
    adj[c].clear();
}

//Centroid decomposition2

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

int getCentroid(int v, int par, int tam) {
    for(int u : adj[v]) {
        if (u == par || removed[u]) continue;
        if (2 * sz[u] > tam) return getCentroid(u, v, tam);
    }
    return v;
}

void setDis(int v, int par, int nv, int d) {
    dis[v][nv] = d;
    for(int u : adj[v]) {
        if (u == par || removed[u]) continue;
        setDis(u, v, nv, d + 1);
    }
}

```



```

    }
}

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

```

## 2.15 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 Graph Algorithms

### 3.1 Dinic Max Flow

```

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

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

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

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

int bfs(int source, int sink) {
    memset(level, -1, sizeof level);
    level[source] = 0;
    int front = 0, size = 0, v;
    fila[size++] = source;
    while(front < size) {

```

```

v = fila[front++];
for(int i = adj[v]; i != -1; i = ant[i]) {
    if(wt[i] && level[to[i]] == -1) {
        level[to[i]] = level[v] + 1;
        fila[size++] = to[i];
    }
}
return level[sink] != -1;
}

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

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

```

## 3.2 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;
}

```

## 3.3 Articulation Points/Bridges/Biconnected Components

```

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

int adj[ms], to[me], ant[me], z, n;

```

```

int idx[ms], bc[me], ind, nbc, child, st[me], top;

// Funcao de add e clear no dinic

void generateBc(int edge) {
    while(st[--top] != edge) {
        bc[st[top]] = nbc;
    }
    bc[edge] = nbc++;
}

int dfs(int v, int par = -1) {
    int low = idx[v] = ind++;
    for(int i = adj[v]; i > -1; i = ant[i]) {
        if(idx[to[i]] == -1) {
            if(par == -1) child++;
            st[top++] = i;
            int temp = dfs(to[i], v);
            if(par == -1 && child > 1 || ~par && temp >= idx[v]) generateBc(i);
            if(temp >= idx[v]) art[v] = true;
            if(temp > idx[v]) bridge[i] = true;
            low = min(low, temp);
        } else if(to[i] != par && idx[to[i]] < low) {
            low = idx[to[i]];
            st[top++] = i;
        }
    }
    return low;
}

void biconnected() {
    ind = 0;
    nbc = 0;
    top = -1;
    memset(idx, -1, sizeof idx);
    memset(art, 0, sizeof art);
    memset(bridge, 0, sizeof bridge);
    for(int i = 0; i < n; i++) if(idx[i] == -1) {
        child = 0;
        dfs(i);
    }
}

```

## 3.4 SCC - Strongly Connected Components / 2SAT

```

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

// Operacoes comuns de 2-sat
// ~v = "nao v"
#define trad(v) (v<0?((~v)*2)^1:v*2)
void addImp(int a, int b) { g[trad(a)].push(trad(b)); }
void addOr(int a, int b) { addImp(~a, b); addImp(~b, a); }
void addEqual(int a, int b) { addOr(a, ~b); addOr(~a, b); }
void addDiff(int a, int b) { addEqual(a, ~b); }
// valoracao: value[v] = comp[trad(v)] < comp[trad(~v)]

```

### 3.5 LCA - Lowest Common Ancestor

```

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

void dfs(int v, int p, int l = 0) {
    lvl[v] = l;
    par[v][0] = p;
    for(int u : g[v]) {
        if(u != p) dfs(u, v, l + 1);
    }
}

void processAncestors(int root = 0) {
    dfs(root, root);
    for(int k = 1; k <= mlg; k++) {
        for(int i = 0; i < n; i++) {
            par[i][k] = par[par[i][k-1]][k-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];
}

```

### 3.6 Heavy Light Decomposition

```

// HLD + Euler Tour by adamant

int sz[ms], par[ms], h[ms];
int t, in[ms], out[ms], rin[ms], nxt[ms];

void dfs_sz(int v = 0, int p = -1) {
    sz[v] = 1;
    for(int i = 0; i < g[v].size(); i++) {
        int &u = g[v][i];
        if(u == p) continue;
        h[u] = h[v]+1, par[u] = v;
        dfs_sz(u, v);
        sz[v] += sz[u];
        if(g[v][0] == p || sz[u] > sz[g[v][0]]) {
            swap(u, g[v][0]);
        }
    }
}

void dfs_hld(int v = 0, int p = -1) {
    in[v] = t++;
    rin[in[v]] = v;
    for(int i = 0; i < g[v].size(); i++) {
        int &u = g[v][i];
        if(u == p) continue;
        nxt[u] = u == g[v][0] ? nxt[v] : u;
        dfs_hld(u, v);
    }
    out[v] = t;
}

int up(int v) {
    return (nxt[v] != v) ? nxt[v] : (~par[v] ? par[v] : v);
}

int getLCA(int a, int b) {
    while(nxt[a] != nxt[b]) {
        if(h[a] == 0 || h[up(a)] < h[up(b)]) swap(a, b);
        a = up(a);
    }
    return h[a] < h[b] ? a : b;
}

int queryUp(int a, int p = 0) {
    int ans = 0;
    while(nxt[a] != nxt[p]) {
        ans += query(in[nxt[a]], in[a]);
        a = par[nxt[a]];
    }
    ans += query(in[p], in[a]);
    return ans;
}

int queryPath(int u, int v) {
    int lca = getLCA(u, v);
}

```

```

return queryUp(u, lca) + queryUp(v, lca) - queryUp(lca, lca);
}

```

### 3.7 Sack

```

void solve(int a, int p, bool f){
    int big = -1;
    for(auto &b : adj[a]){
        if(b != p && (big == -1 || en[b]-st[b] > en[big]-st[big])){
            big = b;
        }
    }
    for(auto &b : adj[a]){
        if(b == p || b == big) continue;
        solve(b, a, 0);
    }
    if(~big) solve(big, a, 1);
    add(cnt[v[a]], -1);
    cnt[v[a]]++;
    add(cnt[v[a]], +1);
    for(auto &b : adj[a]){
        if(b == p || b == big) continue;
        for(int i = st[b]; i < en[b]; i++){
            add(cnt[ett[i]], -1);
            cnt[ett[i]]++;
            add(cnt[ett[i]], +1);
        }
    }
    for(auto &q : Q[a]){
        ans[q.first] = query(mx-1)-query(q.second-1);
    }
    if(!f){
        for(int i = st[a]; i < en[a]; i++){
            add(cnt[ett[i]], -1);
            cnt[ett[i]]--;
            add(cnt[ett[i]], +1);
        }
    }
}

```

### 3.8 Min Cost Max Flow

```

template <class flow_t, class cost_t>
class MinCostMaxFlow {
private:
    typedef pair<cost_t, int> ii;

    struct Edge {
        int to;
        flow_t cap;
        cost_t cost;
        Edge(int to, flow_t cap, cost_t cost) : to(to), cap(cap), cost(
            cost) {}
    };

    int n;
    vector<vector<int>>> adj;
    vector<Edge> edges;

```

```

vector<cost_t> dis;
vector<int> prev, id_prev;
vector<int> q;
vector<bool> inq;

pair<flow_t, cost_t> spfa(int src, int sink) {
    fill(dis.begin(), dis.end(), int(1e9)); //cost_t inf
    fill(prev.begin(), prev.end(), -1);
    fill(inq.begin(), inq.end(), false);
    q.clear();
    q.push_back(src);
    inq[src] = true;
    dis[src] = 0;
    for(int on = 0; on < (int) q.size(); on++) {
        int cur = q[on];
        inq[cur] = false;
        for(auto id : adj[cur]) {
            if (edges[id].cap == 0) continue;
            int to = edges[id].to;
            if (dis[to] > dis[cur] + edges[id].cost) {
                prev[to] = cur;
                id_prev[to] = id;
                dis[to] = dis[cur] + edges[id].cost;
                if (!inq[to]) {
                    q.push_back(to);
                    inq[to] = true;
                }
            }
        }
    }
    flow_t mn = flow_t(1e9);
    for(int cur = sink; prev[cur] != -1; cur = prev[cur]) {
        int id = id_prev[cur];
        mn = min(mn, edges[id].cap);
    }
    if (mn == flow_t(1e9) || mn == 0) return make_pair(0, 0);
    pair<flow_t, cost_t> ans(mn, 0);
    for(int cur = sink; prev[cur] != -1; cur = prev[cur]) {
        int id = id_prev[cur];
        edges[id].cap -= mn;
        edges[id ^ 1].cap += mn;
        ans.second += mn * edges[id].cost;
    }
    return ans;
}

public:
MinCostMaxFlow(int a = 0) {
    n = a;
    adj.resize(n + 2);
    edges.clear();
    dis.resize(n + 2);
    prev.resize(n + 2);
    id_prev.resize(n + 2);
    inq.resize(n + 2);
}

void init(int a) {
    n = a;
    adj.resize(n + 2);
    edges.clear();
    dis.resize(n + 2);
    prev.resize(n + 2);

```

```

    id_prev.resize(n + 2);
    inq.resize(n + 2);
}
void add(int from, int to, flow_t cap, cost_t cost) {
    adj[from].push_back(int(edges.size()));
    edges.push_back(Edge(to, cap, cost));
    adj[to].push_back(int(edges.size()));
    edges.push_back(Edge(from, 0, -cost));
}
pair<flow_t, cost_t> maxflow(int src, int sink) {
    pair<flow_t, cost_t> ans(0, 0), got;
    while((got = spfa(src, sink)).first > 0) {
        ans.first += got.first;
        ans.second += got.second;
    }
    return ans;
}
};

```

### 3.9 Hungarian Algorithm - Maximum Cost Matching

```

const int inf = 0x3f3f3f3f;

int n, w[ms][ms], maxm;
int lx[ms], ly[ms], xy[ms], yx[ms];
int slack[ms], slackx[ms], prev[ms];
bool S[ms], T[ms];

void init_labels() {
    memset(lx, 0, sizeof lx); memset(ly, 0, sizeof ly);
    for(int x = 0; x < n; x++) for(int y = 0; y < n; y++) {
        lx[x] = max(lx[x], cos[x][y]);
    }
}

void updateLabels() {
    int delta = inf;
    for(int y = 0; y < n; y++) if(!T[y]) delta = min(delta, slack[y]);
    for(int x = 0; x < n; x++) if(S[x]) lx[x] -= delta;
    for(int y = 0; y < n; y++) if(T[y]) ly[y] += delta;
    for(int y = 0; y < n; y++) if(!T[y]) slack[y] -= delta;
}

void addTree(int x, int prevx) {
    S[x] = 1; prev[x] = prevx;
    for(int y = 0; y < n; y++) if(lx[x] + ly[y] - w[x][y] < slack[y]) {
        slack[y] = lx[x] + ly[y] - cost[x][y];
        slackx[y] = x;
    }
}

void augment() {
    if(maxm == n) return;
    int x, y, root;
    int q[ms], wr = 0, rd = 0;
    memset(S, 0, sizeof S); memset(T, 0, sizeof T);
    memset(prev, -1, sizeof prev);
    for(int x = 0; x < n; x++) if(xy[x] == -1) {
        q[wr++] = root = x;
        prev[x] = -2;

```

```

        S[x] = 1;
        break;
    }
    for(int y = 0; y < n; y++) {
        slack[y] = lx[root] + ly[y] - w[root][y];
        slackx[y] = root;
    }
    while(true) {
        while(rd < wr) {
            x = q[rd++];
            for(y = 0; y < n; y++) if(w[x][y] == lx[x] + ly[y] && !T[y]) {
                if(yx[y] == -1) break;
                T[y] = 1;
                q[wr++] = yx[y];
                addTree(yx[y], x);
            }
            if(y < n) break;
        }
        if(y < n) break;
        updateLabels();
        wr = rd = 0;
        for(y = 0; y < n; y++) if(!T[y] && !slack[y]) {
            if(yx[y] == -1) {
                x = slackx[y];
                break;
            } else {
                T[y] = true;
                if(!S[yx[y]]) {
                    q[wr++] = yx[y];
                    addTree(yx[y], slackx[y]);
                }
            }
        }
        if(y < n) break;
    }
    if(y < n) {
        maxm++;
        for(int cx = x, cy = y, ty; cx != -2; cx = prev[cx], cy = ty) {
            ty = xy[cx];
            yx[cy] = cx;
            xy[cx] = cy;
        }
        augment();
    }
}

int hungarian() {
    int ans = 0; maxm = 0;
    memset(xy, -1, sizeof xy); memset(yx, -1, sizeof yx);
    init_labels(); augment();
    for(int x = 0; x < n; x++) ans += w[x][xy[x]];
    return ans;
}

```

## 4 Math

### 4.1 Discrete Logarithm

```

ll discreteLog(ll a, ll b, ll m) {

```

```

// a^ans == b mod m
// ou -1 se nao existir
ll cur = a, on = 1;
for(int i = 0; i < 100; i++) {
    cur = cur * a % m;
}
while(on * on <= m) {
    cur = cur * a % m;
    on++;
}
map<ll, ll> position;
for(ll i = 0, x = 1; i * i <= m; i++) {
    position[x] = i * on;
    x = x * cur % m;
}
for(ll i = 0; i <= on + 20; i++) {
    if(position.count(b)) {
        return position[b] - i;
    }
    b = b * a % m;
}
return -1;
}

```

## 4.2 GCD - Greatest Common Divisor

```

ll gcd(ll a, ll b) {
    while(b) a %= b, swap(a, b);
    return a;
}

```

## 4.3 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);
}

```

## 4.4 Fast Exponentiation

```

const ll mod = 1e9+7;

```

```

ll fExp(ll a, ll b) {
    ll ans = 1;
    while(b) {
        if(b & 1) ans = ans * a % mod;
        a = a * a % mod;
        b >>= 1;
    }
    return ans;
}

```

## 4.5 Matrix Fast Exponentiation

```

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

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

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

```

## 4.6 FFT - Fast Fourier Transform

```

typedef complex<double> Complex;
typedef long double ld;
typedef long long ll;

const int ms = maiortamanhoderesposta * 2;
const ld pi = acosl(-1.0);

int rbit[1 << 23];
Complex a[ms], b[ms];

void calcReversedBits(int n) {
    int lg = 0;
    while(1 << (lg + 1) < n) {
        lg++;
    }
}

```

```

    for(int i = 1; i < n; i++) {
        rbit[i] = (rbit[i] >> 1) >> 1 | ((i & 1) << lg);
    }
}

void fft(Complex a[], int n, bool inv = false) {
    for(int i = 0; i < n; i++) {
        if(rbit[i] > i) swap(a[i], a[rbit[i]]);
    }
    double ang = inv ? -pi : pi;
    for(int m = 1; m < n; m += m) {
        Complex d(cosl(ang/m), sinl(ang/m));
        for(int i = 0; i < n; i += m+m) {
            Complex cur = 1;
            for(int j = 0; j < m; j++) {
                Complex u = a[i + j], v = a[i + j + m] * cur;
                a[i + j] = u + v;
                a[i + j + m] = u - v;
                cur *= d;
            }
        }
    }
    if(inv) {
        for(int i = 0; i < n; i++) a[i] /= n;
    }
}

void multiply(ll x[], ll y[], ll ans[], int nx, int ny, int &n) {
    n = 1;
    while(n < nx+ny) n <= 1;
    calcReversedBits(n);
    for(int i = 0; i < n; i++) {
        a[i] = Complex(x[i]);
        b[i] = Complex(y[i]);
    }
    fft(a, n); fft(b, n);
    for(int i = 0; i < n; i++) {
        a[i] = a[i] * b[i];
    }
    fft(a, n, true);
    for(int i = 0; i < n; i++) {
        ans[i] = ll(a[i].real() + 0.5);
    }
    n = nx + ny;
}

```

## 4.7 NTT - Number Theoretic Transform

```

long long int mod = (11911 << 23) + 1, c_root = 3;

namespace NTT {
    typedef long long int ll;

    ll fexp(ll base, ll e) {
        ll ans = 1;
        while(e > 0) {
            if (e & 1) ans = ans * base % mod;
            base = base * base % mod;
            e >>= 1;
        }
    }
}

```

```

    return ans;
}

ll inv_mod(ll base) {
    return fexp(base, mod - 2);
}

void ntt(vector<ll>& a, bool inv) {
    int n = (int) a.size();
    if (n == 1) return;

    for(int i = 0, j = 0; i < n; i++) {
        if (i > j) {
            swap(a[i], a[j]);
        }
        for(int l = n / 2; (j ^= 1) < 1; l >>= 1);
    }

    for(int sz = 1; sz < n; sz <= 1) {
        ll delta = fexp(c_root, (mod - 1) / (2 * sz)); //delta = w_2sz
        if (inv) {
            delta = inv_mod(delta);
        }
        for(int i = 0; i < n; i += 2 * sz) {
            ll w = 1;
            for(int j = 0; j < sz; j++) {
                ll u = a[i + j], v = w * a[i + j + sz] % mod;
                a[i + j] = (u + v + mod) % mod;
                a[i + j] = (a[i + j] + mod) % mod;
                a[i + j + sz] = (u - v + mod) % mod;
                a[i + j + sz] = (a[i + j + sz] + mod) % mod;
                w = w * delta % mod;
            }
        }
    }
    if (inv) {
        ll inv_n = inv_mod(n);
        for(int i = 0; i < n; i++) {
            a[i] = a[i] * inv_n % mod;
        }
    }
    for(int i = 0; i < n; i++) {
        a[i] %= mod;
        a[i] = (a[i] + mod) % mod;
    }
}

void multiply(vector<ll> &a, vector<ll> &b, vector<ll> &ans) {
    int lim = (int) max(a.size(), b.size());
    int n = 1;
    while(n < lim) n <= 1;
    n <= 1;
    a.resize(n);
    b.resize(n);
    ans.resize(n);
    ntt(a, false);
    ntt(b, false);
    for(int i = 0; i < n; i++) {
        ans[i] = a[i] * b[i] % mod;
    }
    ntt(ans, true);
}

```

```

    }
};

```

## 4.8 Miller and Rho

```

typedef long long int ll;

bool overflow(ll a, ll b) {
    return b && (a >= (1ll << 62) / b);
}

ll add(ll a, ll b, ll md) {
    return (a + b) % md;
}

ll mul(ll a, ll b, ll md) {
    if (!overflow(a, b)) return (a * b) % md;
    ll ans = 0;
    while(b) {
        if (b & 1) ans = add(ans, a, md);
        a = add(a, a, md);
        b >>= 1;
    }
    return ans;
}

ll fexp(ll a, ll e, ll md) {
    ll ans = 1;
    while(e) {
        if (e & 1) ans = mul(ans, a, md);
        a = mul(a, a, md);
        e >>= 1;
    }
    return ans;
}

ll my_rand() {
    ll ans = rand();
    ans = (ans << 31) | rand();
    return ans;
}

ll gcd(ll a, ll b) {
    while(b) {
        ll t = a % b;
        a = b;
        b = t;
    }
    return a;
}

bool miller(ll p, int iteracao) {
    if(p < 2) return 0;
    if(p % 2 == 0) return (p == 2);
    ll s = p - 1;
    while(s % 2 == 0) s >>= 1;
    for(int i = 0; i < iteracao; i++) {
        ll a = rand() % (p - 1) + 1, temp = s;
        ll mod = fexp(a, temp, p);
        while(temp != p - 1 && mod != 1 && mod != p - 1) {

```

```

            mod = mul(mod, mod, p);
            temp <=<= 1;
        }
        if(mod != p - 1 && temp % 2 == 0) return 0;
    }
    return 1;
}

ll rho(ll n) {
    if (n == 1 || miller(n, 10)) return n;
    if (n % 2 == 0) return 2;
    while(1) {
        ll x = my_rand() % (n - 2) + 2, y = x;
        ll c = 0, cur = 1;
        while(c == 0) {
            c = my_rand() % (n - 2) + 1;
        }
        while(cur == 1) {
            x = add(mul(x, x, n), c, n);
            y = add(mul(y, y, n), c, n);
            y = add(mul(y, y, n), c, n);
            cur = gcd((x >= y ? x - y : y - x), n);
        }
        if (cur != n) return cur;
    }
}

```

## 4.9 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;
}

```

## 5 Geometry

### 5.1 Geometry

```

const double inf = 1e100, eps = 1e-9;

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

double dot(PT p, PT q) { return p.x * q.x + p.y * q.y; }
double dist2(PT p, PT q) { return dot(p - q, p - q); }
double dist(PT p, PT q) { return hypot(p.x - q.x, p.y - q.y); }
double cross(PT p, PT q) { return p.x * q.y - p.y * q.x; }

// Rotaciona o ponto CCW ou CW ao redor da origem
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));
}

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

// Projeta ponto c no segmento a - b
PT projectPointSegment(PT a, PT b, PT c) {
    double r = dot(b - a, b - a);
    if(abs(r) < eps) return a;
    r = dot(c - a, b - a) / r;

```

```

    if(r < 0) return a;
    if(r > 1) return b;
    return a + (b - a) * r;
}

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

// Determina se o ponto c esta em um segmento a - b
bool ptInSegment(PT a, PT b, PT c) {
    bool x = min(a.x, b.x) <= c.x && c.x <= max(a.x, b.x);
    bool y = min(a.y, b.y) <= c.y && c.y <= max(a.y, b.y);
    return x && y && (cross((b-a), (c-a)) == 0); // testar com eps se for
    double

}

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

// Determina se as linhas a - b e c - d sao paralelas ou colineares
bool linesParallel(PT a, PT b, PT c, PT d) {
    return abs(cross(b - a, c - d)) < eps;
}

bool linesCollinear(PT a, PT b, PT c, PT d) {
    return linesParallel(a, b, c, d) && abs(cross(a - b, a - c)) < eps
        && abs(cross(c - d, c - a)) < eps;
}

// Determina se o segmento a - b intersecta com o segmento c - d
bool segmentsIntersect(PT a, PT b, PT c, PT d) {
    if(linesCollinear(a, b, c, d)) {
        if(dist2(a, c) < eps || dist2(a, d) < eps || dist2(b, c) < eps
            || dist2(b, d) < eps) return true;
        if(dot(c - a, c - b) > 0 && dot(d - a, d - b) > 0 && dot(c - b,
            , d - b) > 0) return false;
        return true;
    }
    if(cross(d - a, b - a) * cross(c - a, b - a) > 0) return false;
    if(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 paralelos
PT computeLineIntersection(PT a, PT b, PT c, PT d) {
    b = b - a; d = c - d; c = c - a;
    assert(cross(b, d) != 0); // garante que as retas nao sao
        paralelas, remover pra evitar tle
    return a + b * cross(c, d) / cross(b, d);
}

// Calcula centro do circulo dado tres pontos

```

```

PT computeCircleCenter(PT a, PT b, PT c) {
    b = (a + b) / 2;
    c = (a + c) / 2;
    return computeLineIntersection(b, b + rotateCW90(a - b), c, c +
        rotateCW90(a - c));
}

// Determina se o ponto p esta dentro do triangulo (a, b, c)
bool ptInsideTriangle(PT p, PT a, PT b, PT c) {
    if(cross(b-a, c-b) < 0) swap(a, b);
    ll x = cross(b-a, p-b);
    ll y = cross(c-b, p-c);
    ll z = cross(a-c, p-a);
    if(x > 0 && y > 0 && z > 0) return true;
    if(!x) return ptInSegment(a,b,p);
    if(!y) return ptInSegment(b,c,p);
    if(!z) return ptInSegment(c,a,p);
    return false;
}

// Determina se o ponto esta num poligono convexo em O(lgn)
bool pointInConvexPolygon(const vector<PT> &p, PT q) {
    PT pivot = p[0];
    int x = 1, y = p.size();
    while(y-x != 1) {
        int z = (x+y)/2;
        PT diagonal = pivot - p[z];
        if(cross(p[x] - pivot, q - pivot) * cross(q-pivot, p[z] - pivot)
            >= 0) y = z;
        else x = z;
    }
    return ptInsideTriangle(q, p[x], p[y], pivot);
}

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

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

```

```

}

// Calcula intersecao da linha a - b com o circulo centrado em c com
// raio r > 0
vector<PT> circleLineIntersection(PT a, PT b, PT c, double r) {
    vector<PT> ans;
    b = b - a;
    a = a - c;
    double x = dot(b, b);
    double y = dot(a, b);
    double z = dot(a, a) - r * r;
    double w = y * y - x * z;
    if (w < -eps) return ans;
    ans.push_back(c + a + b * (-y + sqrt(w + eps)) / x);
    if (w > eps)
        ans.push_back(c + a + b * (-y - sqrt(w)) / x);
    return ans;
}

// Calcula intersecao do circulo centrado em a com raio r e o centrado
// em b com raio R
vector<PT> circleCircleIntersection(PT a, PT b, double r, double R) {
    vector<PT> ans;
    double d = sqrt(dist2(a, b));
    if (d > r + R || d + min(r, R) < max(r, R)) return ans;
    double x = (d * d - R * R + r * r) / (2 * d);
    double y = sqrt(r * r - x * x);
    PT v = (b - a) / d;
    ans.push_back(a + v * x + rotateCCW90(v) * y);
    if (y > 0)
        ans.push_back(a + v * x - rotateCCW90(v) * y);
    return ans;
}

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

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

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

```

```

}

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

```

## 5.2 Convex Hull

```

vector<PT> convexHull(vector<PT> p) {
    int n = p.size(), k = 0;
    vector<PT> h(2 * n);
    sort(p.begin(), p.end());
    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);
    return h;
}

```

## 5.3 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, !p[i] - swapCoord(*it));
        }
        ptsInv.insert(swapCoord(p[i]));
    }
    return d;
}

```

## 5.4 Intersection Points

```

// Utiliza uma seg tree

int intersectionPoints(vector<pair<PT, PT>> v) {
    int n = v.size();
    vector<pair<int, int>> events, vertInt;
    for(int i = 0; i < n; i++) {
        if(v.first.x == v.second.x) { // Segmento Vertical
            int y0 = min(v.first.y, v.second.y), y1 = max(v.first.y, v.second.y);
            events.push_back({v.first.x, vertInt.size()}); // Tipo = Indice
            // no array
            vertInt.push_back({y0, y1});
        } else { // Segmento Horizontal
            int x0 = min(v.first.x, v.second.x), x1 = max(v.first.x, v.second.x);
            events.push_back({x0, -1}); // Inicio de Segmento
            events.push_back({x1, inf}); // Final de Segmento
        }
    }
    sort(events.begin(), events.end());
    int ans = 0;
    for(int i = 0; i < events.size(); i++) {
        int t = events[i].second;
        if(t == -1) {
            segUpdate(events[i].first, 1);
        } else if(t == inf) {
            segUpdate(events[i].first, 0);
        } else {
            ans += segQuery(vertInt[t].first, vertInt[t].second);
        }
    }
    return ans;
}

```

## 5.5 Delaunay Triangulation

```

typedef long long ll;

bool ge(const ll& a, const ll& b) { return a >= b; }
bool le(const ll& a, const ll& b) { return a <= b; }
bool eq(const ll& a, const ll& b) { return a == b; }
bool gt(const ll& a, const ll& b) { return a > b; }
bool lt(const ll& a, const ll& b) { return a < b; }
int sgn(const ll& a) { return a >= 0 ? a ? 1 : 0 : -1; }

struct pt {
    ll x, y;
    pt() {}
    pt(ll _x, ll _y) : x(_x), y(_y) {}
    pt operator-(const pt& p) const {
        return pt(x - p.x, y - p.y);
    }
    ll cross(const pt& p) const {
        return x * p.y - y * p.x;
    }
    ll cross(const pt& a, const pt& b) const {

```

```

        return (a - *this).cross(b - *this);
    }
    ll dot(const pt& p) const {
        return x * p.x + y * p.y;
    }
    ll dot(const pt& a, const pt& b) const {
        return (a - *this).dot(b - *this);
    }
    ll sqrLength() const {
        return this->dot(*this);
    }
    bool operator==(const pt& p) const {
        return eq(x, p.x) && eq(y, p.y);
    }
};

const pt inf_pt = pt(1e18, 1e18);

struct QuadEdge {
    pt origin;
    QuadEdge* rot = nullptr;
    QuadEdge* onext = nullptr;
    bool used = false;
    QuadEdge* rev() const {
        return rot->rot;
    }
    QuadEdge* lnext() const {
        return rot->rev()->onext->rot;
    }
    QuadEdge* oprev() const {
        return rot->onext->rot;
    }
    pt dest() const {
        return rev()->origin;
    }
};

QuadEdge* make_edge(pt from, pt to) {
    QuadEdge* e1 = new QuadEdge;
    QuadEdge* e2 = new QuadEdge;
    QuadEdge* e3 = new QuadEdge;
    QuadEdge* e4 = new QuadEdge;
    e1->origin = from;
    e2->origin = to;
    e3->origin = e4->origin = inf_pt;
    e1->rot = e3;
    e2->rot = e4;
    e3->rot = e2;
    e4->rot = e1;
    e1->onext = e1;
    e2->onext = e2;
    e3->onext = e4;
    e4->onext = e3;
    return e1;
}

void splice(QuadEdge* a, QuadEdge* b) {
    swap(a->onext->rot->onext, b->onext->rot->onext);
    swap(a->onext, b->onext);
}

```

```

void delete_edge(QuadEdge* e) {
    splice(e, e->oprev());
    splice(e->rev(), e->rev()->oprev());
    delete e->rot;
    delete e->rev()->rot;
    delete e;
    delete e->rev();
}

QuadEdge* connect(QuadEdge* a, QuadEdge* b) {
    QuadEdge* e = make_edge(a->dest(), b->origin);
    splice(e, a->lnext());
    splice(e->rev(), b);
    return e;
}

bool left_of(pt p, QuadEdge* e) {
    return gt(p.cross(e->origin, e->dest()), 0);
}

bool right_of(pt p, QuadEdge* e) {
    return lt(p.cross(e->origin, e->dest()), 0);
}

template <class T>
T det3(T a1, T a2, T a3, T b1, T b2, T b3, T c1, T c2, T c3) {
    return a1 * (b2 * c3 - c2 * b3) - a2 * (b1 * c3 - c1 * b3) +
        a3 * (b1 * c2 - c1 * b2);
}

bool in_circle(pt a, pt b, pt c, pt d) {
    // If there is __int128, calculate directly.
    // Otherwise, calculate angles.
    #if defined(__LP64__) || defined(_WIN64)
        __int128 det = -det3<__int128>(b.x, b.y, b.sqrLength(), c.x, c.y,
            c.sqrLength(), d.x, d.y, d.sqrLength());
        det += det3<__int128>(a.x, a.y, a.sqrLength(), c.x, c.y, c.sqrLength(), d.x, d.y, d.sqrLength());
        det -= det3<__int128>(a.x, a.y, a.sqrLength(), b.x, b.y, b.sqrLength(), d.x, d.y, d.sqrLength());
        det += det3<__int128>(a.x, a.y, a.sqrLength(), b.x, b.y, b.sqrLength(), c.x, c.y, c.sqrLength());
        return det > 0;
    #else
        auto ang = [] (pt l, pt mid, pt r) {
            ll x = mid.dot(l, r);
            ll y = mid.cross(l, r);
            long double res = atan2((long double)x, (long double)y);
            return res;
        };
        long double kek = ang(a, b, c) + ang(c, d, a) - ang(b, c, d) - ang(d, a, b);
        if (kek > 1e-8)
            return true;
        else
            return false;
    #endif
}

```

```

}

pair<QuadEdge*, QuadEdge*> build_tr(int l, int r, vector<pt>& p) {
    if (r - l + 1 == 2) {
        QuadEdge* res = make_edge(p[l], p[r]);
        return make_pair(res, res->rev());
    }
    if (r - l + 1 == 3) {
        QuadEdge* a = make_edge(p[l], p[l + 1]), *b = make_edge(p[l + 1], p[r]);
        splice(a->rev(), b);
        int sg = sgn(p[l].cross(p[l + 1], p[r]));
        if (sg == 0)
            return make_pair(a, b->rev());
        QuadEdge* c = connect(b, a);
        if (sg == 1)
            return make_pair(a, b->rev());
        else
            return make_pair(c->rev(), c);
    }
    int mid = (l + r) / 2;
    QuadEdge* ldo, *ldi, *rdo, *rdi;
    tie(ldo, ldi) = build_tr(l, mid, p);
    tie(rdi, rdo) = build_tr(mid + 1, r, p);
    while (true) {
        if (left_of(rdi->origin, ldi)) {
            ldi = ldi->lnext();
            continue;
        }
        if (right_of(ldi->origin, rdi)) {
            rdi = rdi->rev()->onext;
            continue;
        }
        break;
    }
    QuadEdge* basel = connect(rdi->rev(), ldi);
    auto valid = [&basel](QuadEdge* e) { return right_of(e->dest(), basel); };
    if (ldi->origin == ldo->origin)
        ldo = basel->rev();
    if (rdi->origin == rdo->origin)
        rdo = basel;
    while (true) {
        QuadEdge* lcand = basel->rev()->onext;
        if (valid(lcand)) {
            while (in_circle(basel->dest(), basel->origin, lcand->dest(), lcand->onext->dest())) {
                QuadEdge* t = lcand->onext;
                delete_edge(lcand);
                lcand = t;
            }
        }
        QuadEdge* rcand = basel->oprev();
        if (valid(rcand)) {
            while (in_circle(basel->dest(), basel->origin, rcand->dest(), rcand->oprev()->dest())) {
                QuadEdge* t = rcand->oprev();
                delete_edge(rcand);
                rcand = t;
            }
        }
    }
}

```

```

}
    if (!valid(lcand) && !valid(rcand))
        break;
    if (!valid(lcand) || (valid(rcand) && in_circle(lcand->dest(), lcand->origin, rcand->origin, rcand->dest())))
        basel = connect(rcand, basel->rev());
    else
        basel = connect(basel->rev(), lcand->rev());
}
return make_pair(ldo, rdo);
}

vector<tuple<pt, pt, pt>> delaunay(vector<pt> p) {
    sort(p.begin(), p.end(), [](const pt& a, const pt& b) {
        return lt(a.x, b.x) || (eq(a.x, b.x) && lt(a.y, b.y));
    });
    auto res = build_tr(0, (int)p.size() - 1, p);
    QuadEdge* e = res.first;
    vector<QuadEdge*> edges = {e};
    while (lt(e->onext->dest().cross(e->dest(), e->origin), 0))
        e = e->onext;
    auto add = [&p, &e, &edges]() {
        QuadEdge* curr = e;
        do {
            curr->used = true;
            p.push_back(curr->origin);
            edges.push_back(curr->rev());
            curr = curr->lnext();
        } while (curr != e);
    };
    add();
    p.clear();
    int kek = 0;
    while (kek < (int)edges.size()) {
        if (!(e = edges[kek++])->used)
            add();
    }
    vector<tuple<pt, pt, pt>> ans;
    for (int i = 0; i < (int)p.size(); i += 3) {
        ans.push_back(make_tuple(p[i], p[i + 1], p[i + 2]));
    }
    return ans;
}

```

## 5.6 Java Geometry Library

```

import java.util.*;
import java.io.*;
import java.awt.geom.*;
import java.lang.*;
//Lazy Geometry
class AWT{
    static Area makeArea(double[] pts){
        Path2D.Double p = new Path2D.Double();
        p.moveTo(pts[0], pts[1]);
        for(int i = 2; i < pts.length; i+=2){
            p.lineTo(pts[i], pts[i+1]);
        }
    }
}

```

```

    p.closePath();
    return new Area(p);
}
static double computePolygonArea(ArrayList<Point2D.Double> points) {
    Point2D.Double[] pts = points.toArray(new Point2D.Double[points.
        size()]);
    double area = 0;
    for (int i = 0; i < pts.length; i++){
        int j = (i+1) % pts.length;
        area += pts[i].x * pts[j].y - pts[j].x * pts[i].y;
    }
    return Math.abs(area)/2;
}
static double computeArea(Area area) {
    double totArea = 0;
    PathIterator iter = area.getPathIterator(null);
    ArrayList<Point2D.Double> points = new ArrayList<Point2D.Double>()
        ;
    while (!iter.isDone()) {
        double[] buffer = new double[6];
        switch (iter.currentSegment(buffer)) {
            case PathIterator.SEG_MOVETO:
            case PathIterator.SEG_LINETO:
                points.add(new Point2D.Double(buffer[0], buffer[1]));
                break;
            case PathIterator.SEG_CLOSE:
                totArea += computePolygonArea(points);
                points.clear();
                break;
        }
        iter.next();
    }
    return totArea;
}
}

```

## 6 Dynamic Programming

### 6.1 Convex Hull Trick

```

typedef long double double_t;
typedef long long int ll;

class HullDynamic {
public:
    const double_t inf = 1e9;

    struct Line {
        ll m, b;
        double_t start;
        bool is_query;

        Line() {}

        Line(ll _m, ll _b, double_t _start, bool _is_query) : m(_m), b(_b)
            , start(_start), is_query(_is_query) {}

        ll eval(ll x) {

```

```

            return m * x + b;
        }
    };

    double_t intersect(const Line& l) const {
        return (double_t) (l.b - b) / (m - l.m);
    }

    bool operator< (const Line& l) const {
        if (is_query == 0) return m > l.m;
        return (start < l.start);
    }
};

typedef set<Line>::iterator iterator_t;

bool has_prev(iterator_t it) {
    return (it != hull.begin());
}

bool has_next(iterator_t it) {
    return (++it != hull.end());
}

bool irrelevant(iterator_t it) {
    if (!has_prev(it) || !has_next(it)) return 0;
    iterator_t prev = it, next = it;
    prev--;
    next++;
    return next->intersect(*prev) <= it->intersect(*prev);
}

void update_left(iterator_t it) {
    if (it == hull.begin()) return;
    iterator_t pos = it;
    --it;
    vector<Line> rem;
    while (has_prev(it)) {
        iterator_t prev = it;
        --prev;
        if (prev->intersect(*pos) <= prev->intersect(*it)) {
            rem.push_back(*it);
        } else {
            break;
        }
        --it;
    }
    double_t start = pos->intersect(*it);
    Line f = *pos;
    for (Line r : rem) hull.erase(r);
    hull.erase(f);
    f.start = start;
    hull.insert(f);
}

void update_right(iterator_t it) {
    if (!has_next(it)) return;
    iterator_t pos = it;
    ++it;
    vector<Line> rem;
    while (has_next(it)) {
        iterator_t next = it;

```

```

++next;
if (next->intersect(*pos) <= pos->intersect(*it)) {
    rem.push_back(*it);
} else {
    break;
}
++it;
}
double_t start = pos->intersect(*it);
Line f = *it;
for (Line r : rem) hull.erase(r);
hull.erase(f);
f.start = start;
hull.insert(f);
}

void insert_line(ll m, ll b) {
    Line f(m, b, -inf, 0);
    iterator_t it = hull.lower_bound(f);
    if (it != hull.end() && it->m == f.m) {
        if (it->b <= f.b) {
            return;
        } else if (it->b > f.b) {
            hull.erase(it);
        }
    }
    hull.insert(f);
    it = hull.lower_bound(f);
    if (irrelevant(it)) {
        hull.erase(it);
        return;
    }
    update_left(it);
    it = hull.lower_bound(f);
    update_right(it);
}

ll get(ll x) {
    Line f(0, 0, x, 1);
    iterator_t it = hull.upper_bound(f);
    assert(it != hull.begin());
    --it;
    return it->m * x + it->b;
}

private:
    set<Line> hull;
};

```

## 6.2 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(l, 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]
}

```

## 6.3 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; // intervalo de tamanho 1 nao
        diminuem
        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
        }
    }
}

```

## 7 Miscellaneous

### 7.1 LIS - Longest Increasing Subsequence

```

int arr[ms], lisArr[ms], n;
// int bef[ms], pos[ms];

int lis() {
    int len = 1;
    lisArr[0] = arr[0];
    // bef[0] = -1;
    for(int i = 1; i < n; i++) {
        // upper_bound se non-decreasing
        int x = lower_bound(lisArr, lisArr + len, arr[i]) - lisArr;
        len = max(len, x + 1);
        lisArr[x] = arr[i];
        // pos[x] = i;
    }
}

```

```

    // bef[i] = x ? pos[x-1] : -1;
}
return len;
}

vi getLis() {
    int len = lis();
    vi ans;
    for(int i = pos[lisArr[len - 1]]; i >= 0; i = bef[i]) {
        ans.push_back(arr[i]);
    }
    reverse(ans.begin(), ans.end());
    return ans;
}

```

## 7.2 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));

```

## 7.3 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 Submask Enumeration

```

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

```

## 7.5 Java Fast I/O

```

import java.util.*;
import java.io.*;
// https://www.spoj.com/problems/INTEST/
class Main{
    public static void main(String[] args) throws Exception{
        Reader s = new Reader();
        PrintWriter out = new PrintWriter(new BufferedOutputStream(System.
            out));
        int n = s.nextInt();
        int k = s.nextInt();
        int count=0;
        while (s.hasNext()) {
            int x = s.nextInt();
            if (x%k == 0)
                count++;
        }
        out.printf("%d\n", count);
        out.close();
        s.close();
    }
    // fast io
    static class Reader {
        final private int BUFFER_SIZE = 1 << 16;
        private DataInputStream din;
        private byte[] buffer;
        private int bufferPointer, bytesRead;

        public Reader() {
            din = new DataInputStream(System.in);
            buffer = new byte[BUFFER_SIZE];
            bufferPointer = bytesRead = 0;
        }

        public Reader(String file_name) throws IOException {
            din = new DataInputStream(new FileInputStream(file_name));
            buffer = new byte[BUFFER_SIZE];
            bufferPointer = bytesRead = 0;
        }

        public String readLine() throws IOException {
            byte[] buf = new byte[64]; // line length
            int cnt = 0, c;
            while ((c = read()) != -1){
                if (c == '\n') break;
                buf[cnt++] = (byte) c;
            }
            return new String(buf, 0, cnt);
        }

        public int nextInt() throws IOException {
            int ret = 0;
            byte c = read();
            while (c <= ' ' ) c = read();

```



```

boolean neg = (c == '-');
if (neg) c = read();

do{
    ret = ret * 10 + c - '0';
} while ((c = read()) >= '0' && c <= '9');

if (neg) return -ret;
return ret;
}

public long nextLong() throws IOException {
    long ret = 0;
    byte c = read();
    while (c <= ' ') c = read();
    boolean neg = (c == '-');
    if (neg) c = read();

    do {
        ret = ret * 10 + c - '0';
    } while ((c = read()) >= '0' && c <= '9');

    if (neg) return -ret;
    return ret;
}

public double nextDouble() throws IOException {
    double ret = 0, div = 1;
    byte c = read();
    while (c <= ' ')
        c = read();
    boolean neg = (c == '-');
    if (neg) c = read();
    do {
        ret = ret * 10 + c - '0';
    } while ((c = read()) >= '0' && c <= '9');

    if (c == '.') {
        while ((c = read()) >= '0' && c <= '9') {
            ret += (c - '0') / (div *= 10);
        }
    }
    if (neg) return -ret;
    return ret;
}

private void fillBuffer() throws IOException {
    bytesRead = din.read(buffer, bufferPointer = 0, BUFFER_SIZE);
    if (bytesRead == -1) buffer[0] = -1;
}

public boolean hasNext() throws IOException {
    if (bufferPointer < bytesRead) return true;
    fillBuffer();
    if (buffer[0] == -1) return false;
    return true;
}

private byte read() throws IOException {
    if (bufferPointer == bytesRead) fillBuffer();
    return buffer[bufferPointer++];
}

```

```

}

public void close() throws IOException {
    if (din == null) return;
    din.close();
}
}

```

## 8 Teoremas e formulas uteis

### 8.1 Grafos

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

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

Trees:  
 Tem Catalan(N) Binary trees de N vertices  
 Tem Catalan(N-1) Arvores enraizadas com N vertices  
 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:  
 Max Edge-disjoint paths: Max flow com arestas com peso 1  
 Max Node-disjoint paths: Faz a mesma coisa mas separa cada vertice em um com as arestas de chegadas e um com as arestas de saida e uma aresta de peso 1 conectando o vertice com aresta de chegada com ele mesmo com arestas de saida  
 Konig's Theorem: minimum node cover = maximum matching se o grafo for bipartido, complemento eh o maximum independent set  
 Min Node disjoint path cover: formar grafo bipartido de vertices duplicados, onde aresta sai do vertice tipo A e chega em tipo B, entao o path cover eh  $N - \text{matching}$   
 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

### 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  $x = \text{somatorio com } k \text{ de } 1 \text{ ate } n \text{ de } a_k * X_k * (X_k, m_k^{-1} \bmod m_k)$   
 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  $(i \text{ escolhe } r) = (n + 1 \text{ escolhe } r)$

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

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

Distribuicao uniforme  $a, a+1, \dots, b$  Expected[X] =  $(a+b)/2$

Distribuicao binomial com  $n$  tentativas de probabilidade  $p$ ,  $X = \text{sucessos}$ :

$P(X = x) = p^x * (1-p)^{(n-x)} * (n \text{ escolhe } x)$  e  $E[X] = p*n$

Distribuicao geometrica onde continuamos ate ter sucesso,  $X = \text{tentativas}$ :

$P(X = x) = (1-p)^{(x-1)} * p$  e  $E[X] = 1/p$

Linearity of expectation: Tendo duas variaveis  $X$  e  $Y$  e constantes  $a$  e  $b$ , o valor esperado de  $aX + bY = aE[X] + bE[Y]$

## 8.3 Geometry

Formula de Euler:  $V - E + F = 2$

Pick Theorem: Para achar pontos em coords inteiras num poligono Area =  $i + b/2 - 1$  onde  $i$  eh o o numero de pontos dentro do poligono e  $b$  de pontos no perimetro do poligono

Two ears theorem: Todo poligono simples com mais de 3 vertices tem pelo menos 2 orelhas, vertices que podem ser removidos sem criar um crossing, remover orelhas repetidamente triangula o poligono

Incentro triangulo:  $(a(X_a, Y_a) + b(X_b, Y_b) + c(X_c, Y_c)) / (a+b+c)$  onde  $a = \text{lado oposto ao vertice } a$ , incentro eh onde cruzam as bissetrizes, eh o centro da circunferencia inscrita e eh equidistante aos lados

## 8.4 Mersenne's Primes

Primos de Mersenne  $2^n - 1$

Lista de Ns que resultam nos primeiros 41 primos de Mersenne:

2; 3; 5; 7; 13; 17; 19; 31; 61; 89; 107; 127; 521; 607; 1.279; 2.203; 2.281; 3.217; 4.253; 4.423; 9.689; 9.941; 11.213; 19.937; 21.701; 23.209; 44.497; 86.243; 110.503; 132.049; 216.091; 756.839; 859.433; 1.257.787; 1.398.269; 2.976.221; 3.021.377; 6.972.593; 13.466.917; 20.996.011; 24.036.583;