ETH zürich

Team Code Reference

Team RaclETH

ICPC 2019 World Finals

Testsession: Check if GNU builtins are available, as well as `__int128`.

0.1 Vim

Turn off auto-tabbing by adding to the `.vimrc`: `filetype indent off`.

0.2 Memory

10^5 time 32 bits is 0.4 MegaBytes. 10^6 times 64 bits is 8 MB. $10^6 \cdot \log 10^6$ times 64 bits is 48 MB. In general, one megabyte is $8 \cdot 10^6$ bits. A `char`, `short`, `int`, `long long` is $8, 16, 32, 64$ bits. A `float`, `double`, `long double` is $32, 64, 80$ bits. Finally note $\log_2 10^{3k} \approx 10k$.

0.3 C++

Fast IO in C++: initialize with `ios::sync_with_stdio(false)` and `cin.tie(nullptr)`. For debugging locally, it might help to add `#define _GLIBCXX_DEBUG`.

1 Datastructures

1.1 Union Find

```cpp
struct UnionFind {
    std::vector<int> par, rank, size;
    int c;
    UnionFind(int n) : par(n), rank(n, 0), size(n, 1), c(n) {
        for(int i = 0; i < n; ++i) par[i] = i;
    }
    int find(int i) { return (par[i] == i ? i : (par[i] = find(par[i]))); }
    bool same(int i, int j) { return find(i) == find(j); }
    int get_size(int i) { return size[find(i)]; }
    int count() { return c; }
    int merge(int i, int j) {
        if((i = find(i)) == (j = find(j))) return -1;
        --c;
        if(rank[i] > rank[j]) swap(i, j);
        par[i] = j;
        size[j] += size[i];
        if(rank[i] == rank[j]) rank[j]++;
        return j;
    }
};
```

1.2 Fenwick Tree

Can be generalized to arbitrary dimensions by duplicating loops.

```cpp
template <class T>
struct FenwickTree {        // use 1 based indices!!!
    int n; vector<T> tree;
    FenwickTree(int n) : n(n) { tree.assign(n + 1, 0); }
    T query(int l, int r) { return query(r) - query(l - 1); }
    T query(int r) {
        T s = 0;
        for(; r > 0; r -= (r & (-r))) s += tree[r];
        return s;
    }
    void update(int i, T v) {
        for(; i <= n; i += (i & (-i))) tree[i] += v;
    }
};
```

1.3 Skew Heap

Meldable heap, all operations in $O(\log n)$ time.

```cpp
template <class T>
struct SkewHeap {
    struct node {
        ll key, lazy = 0LL; T val;
        node *lc = nullptr, *rc = nullptr;
        node(ll k, T v) : key(k), val(v) {}
        node *prop() {
            if (lc) lc->lazy += lazy;
            if (rc) rc->lazy += lazy;
            key += lazy, lazy = 0LL;
            return this;
        }
    };
    node *r = nullptr;
    node* merge(node* x, node* y){
        if (!x) return y; else if (!y) return x;
        if (x->prop()->key > y->prop()->key) swap(x, y);
        x->rc = merge(x->rc, y);
        swap(x->lc, x->rc);
        return x;
    }
    bool empty() { return r == nullptr; }
    void insert(ll x, T val) { r = merge(r, new node(x, val)); }
    void adjust(ll c) { if (r) r->lazy += c; }
    pair<ll, T> pop_min() {
        ll w = r->prop()->key;
        T ret = r->val;
        node *nr = merge(r->lc, r->rc);
        swap(r, nr), delete nr;
        return {w, ret};
    }
    void absorb(SkewHeap<T> &o) { r = merge(r, o.r); o.r = nullptr; }
};
```

1.4 Segment Tree

Works bottom up, so not suitable for lazy propagation.

```cpp
template <class T, const T&(*op)(const T&, const T&)>
struct SegmentTree {
    int n; vector<T> tree; T id;
    SegmentTree(int _n, T _id) : n(_n), tree(2 * n, _id), id(_id) { }
    void update(int i, T val) {
        for (tree[i+n] = val, i = (i+n)/2; i > 0; i /= 2)
            tree[i] = op(tree[2*i], tree[2*i+1]);
    }
    T query(int l, int r) {
        T lhs = T(id), rhs = T(id);
        for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
            if (  l&1 ) lhs = op(lhs, tree[l++]);
            if (!(r&1)) rhs = op(tree[r--], rhs);
        }
        return op(l == r ? op(lhs, tree[l]) : lhs, rhs);
    }
};
```

1.5 Treap

```cpp
struct Node {
    ll v;
    int sz, pr;
    Node *l = nullptr, *r = nullptr;
    Node(ll val) : v(val), sz(1) { pr = rand(); }
};
int size(Node *p) { return p ? p->sz : 0; }
void update(Node* p) {
    if (!p) return;
    p->sz = 1 + size(p->l) + size(p->r);
    // Pull data from children here
}
void propagate(Node *p) {
    if (!p) return;
    // Push data to children here
}
void merge(Node *&t, Node *l, Node *r) {
    propagate(l), propagate(r);
    if (!l)         t = r;
    else if (!r)    t = l;
    else if (l->pr > r->pr)
            merge(l->r, l->r, r), t = l;
    else    merge(r->l, l, r->l), t = r;
    update(t);
}
void spliti(Node *t, Node *&l, Node *&r, int index) {
    propagate(t);
    if (!t) { l = r = nullptr; return; }
    int id = size(t->l);
    if (index <= id) // id \in [index, \infty), so move it right
        spliti(t->l, l, t->l, index), r = t;
    else
        spliti(t->r, t->r, r, index - id), l = t;
    update(t);
}
void splitv(Node *t, Node *&l, Node *&r, ll val) {
    propagate(t);
    if (!t) { l = r = nullptr; return; }
    if (val <= t->v) // t->v \in [val, \infty), so move it right
        splitv(t->l, l, t->l, val), r = t;
    else
        splitv(t->r, t->r, r, val), l = t;
    update(t);
}
void clean(Node *p) {
    if (p) { cleanup(p->l), cleanup(p->r); delete p; }
}
```

1.6 Heavy-Light decomposition

```cpp
struct HLD {
    int V; vvi &graph; // graph can be graph or childs only
    vi p, r, d, h; // parents, path-root; heavy child, depth
    HLD(vvi &graph, int root = 0) : V(graph.size()), graph(graph),
    p(V,-1), r(V,-1), d(V,0), h(V,-1) { dfs(root);
        for(int i=0; i<V; ++i) if (p[i]==-1 || h[p[i]]!=i)
            for (int j=i; j!=-1; j=h[j]) r[j] = i;
    }
    int dfs(int u){
        ii best={-1,-1}; int s=1, ss;   // best, size (of subtree)
        for(auto &v : graph[u]) if(v!=p[u])
            d[v]=d[u]+1, p[v]=u, s += ss=dfs(v), best = max(best,{ss,v});
        h[u] = best.second; return s;
    }
    int lca(int u, int v){
        for(; r[u]!=r[v]; v=p[r[v]]) if(d[r[u]] > d[r[v]]) swap(u,v);
        return d[u] < d[v] ? u : v;
    }
};
```

1.7 Sequence

This is essentially a treap, but it needs special functionality for the Euler Tour Tree

```
template <class T, void M(const T *, T *, const T *) = nullptr>
struct seq {
    T val;
    int size_, priority;
    seq<T, M> *l = nullptr, *r = nullptr, *p = nullptr;
    seq(T _v) : val(_v), size_(1) { priority = rand(); }
    static int size(seq<T, M> *c) { return c ? c->size_ : 0; }
    seq<T, M> *update() {
        size_ = 1;
        if (l) l->p = this, size_ += l->size_;
        if (r) r->p = this, size_ += r->size_;
        if (M) M(l ? &l->val : nullptr, &this->val, r ? &r->val : nullptr);
        return this;
    }
    int index() {
        int ind = size(this->l);
        seq<T, M> *c = this;
        while (c->p) {
            if (c->p->l != c) ind += 1 + size(c->p->l);
            c = c->p;
        }
        return ind;
    }
    seq<T, M> *root() { return this->p ? p->root() : this; }
    seq<T, M> *min() { return this->l ? l->min() : this; }
    seq<T, M> *max() { return this->r ? r->max() : this; }
    seq<T, M> *next() { return this->r ? this->r->min() : this->p; }
    seq<T, M> *prev() { return this->l ? this->l->max() : this->p; }
};
// Note: Assumes both nodes are the roots of their sequences.
template <class T, void M(const T *, T *, const T *)>
seq<T, M> *merge(seq<T, M> *A, seq<T, M> *B) {
    if (!A) return B;
    if (!B) return A;
    if (A->priority > B->priority) {
        A->r = merge(A->r, B);
        return A->update();
    } else {
        B->l = merge(A, B->l);
        return B->update();
    }
}
// Note: Assumes all nodes are the roots of their sequences.
template <class T, void M(const T *, T *, const T *), typename... Seqs>
seq<T, M> *merge(seq<T, M> *l, Seqs... seqs) {
    return merge(l, merge(seqs...));
}
// Split into [0, index) and [index, ..)
template <class T, void M(const T *, T *, const T *)>
pair<seq<T, M> *, seq<T, M> *> split(seq<T, M> *A, int index) {
    if (!A) return {nullptr, nullptr};
    A->p = nullptr;
    if (index <= seq<T, M>::size(A->l)) {
        auto pr = split(A->l, index);
        A->l = pr.second;
        return {pr.first, A->update()};
    } else {
        auto pr = split(A->r, index - (1 + seq<T, M>::size(A->l)));
        A->r = pr.first;
        return {A->update(), pr.second};
    }
}
// return [0, A), [A, ..)
template <class T, void M(const T *, T *, const T *)>
pair<seq<T, M> *, seq<T, M> *> split(seq<T, M> *A) {
    if (!A) return {nullptr, nullptr};
    seq<T, M> *B = A, *lr = A;
    A = A->l;
    if (!A) {
        while (lr->p && lr->p->l == B) lr = B = lr->p;
        if (lr->p)
            lr = A = lr->p, lr->r = B->p = nullptr;
    } else
        A->p = lr->l = nullptr;
    while (lr->update()->p) {
        if (lr->p->l == lr) {
            if (lr == A) swap(A->p, B->p), B->p->l = B;
            lr = B = B->p;
        } else {
            if (lr == B) swap(A->p, B->p), A->p->r = A;
            lr = A = A->p;
        }
    }
    return {A, B};
}
```

1.8 Euler Tour Tree

Maintain information about the trees using the underlying `seq` datastructure.

```
#include "sequence.cpp"
struct EulerTourTree {
    struct edge { int u, v; };
    vector<seq<edge>> vertices;
    vector<map<int, seq<edge>>> edges;
    EulerTourTree(int n) {
        vertices.reserve(n); edges.reserve(n);
        for (int i = 0; i < n; ++i) add_vertex();
    }
    // Create a new vertex.
    int add_vertex() {
        int id = (int)vertices.size();
        vertices.push_back(edge{id, id});
        edges.emplace_back();
        return id;
    }
    // Find root of the subtree containg this vertex.
    int root(int u) { return vertices[u].root()->min()->val.u; }
    bool connected(int u, int v) {
        return vertices[u].root() == vertices[v].root();
    }
    int size(int u) { return (vertices[u].root()->size_ + 2) / 3; }
    // Make v the parent of u. Assumes u has no parent!
    void attach(int u, int v) {
        seq<edge> *i1, *i2;
        tie(i1, i2) = split(&vertices[v]);
        ::merge(i1,
                &(edges[v].emplace(u, seq<edge>{edge{v, u}}).first)->second,
                vertices[u].root(),
                &(edges[u].emplace(v, seq<edge>{edge{u, v}}).first)->second,
                i2);
    }
    // Reroot the tree containing u at u.
    void reroot(int u) {
        seq<edge> *i1, *i2;
        tie(i1, i2) = split(&vertices[u]);
        merge(i2, i1);
    }
    // Links u and v.
    void link(int u, int v) { reroot(u); attach(u, v); }
    // Cut {u, v}. Assumes it exists!!
    void cut(int u, int v) {
        auto uv = edges[u].find(v), vu = edges[v].find(u);
        if (uv->second.index() > vu->second.index())
            swap(u, v), swap(uv, vu);
        seq<edge> *i1, *i2;
        tie(i1, i2) = split(&uv->second); split(i2, 1);
        merge(i1, split(split(&vu->second).second, 1).second);
        edges[u].erase(uv); edges[v].erase(vu);
    }
};
```

1.9 Suffix Array

An $O(n \log n)$ implementation. Note that the resulting array maps values to their position in the suffix array, so invert if necessary.

```
struct SuffixArray {
    string s;
    int n;
    vvi P;
    SuffixArray(string &_s) : s(_s), n(_s.length()) { construct(); }
    void construct() {
        P.push_back(vi(n, 0));
        compress();
        vi occ(n + 1, 0), s1(n, 0), s2(n, 0);
        for (int k = 1, cnt = 1; cnt / 2 < n; ++k, cnt *= 2) {
            P.push_back(vi(n, 0));
            fill(occ.begin(), occ.end(), 0);
            for (int i = 0; i < n; ++i)
                occ[i+cnt<n ? P[k-1][i+cnt]+1 : 0]++;
            partial_sum(occ.begin(), occ.end(), occ.begin());
            for (int i = n - 1; i >= 0; --i)
                s1[--occ[i+cnt<n ? P[k-1][i+cnt]+1 : 0]] = i;
            fill(occ.begin(), occ.end(), 0);
            for (int i = 0; i < n; ++i)
                occ[P[k-1][s1[i]]]++;
            partial_sum(occ.begin(), occ.end(), occ.begin());
            for (int i = n - 1; i >= 0; --i)
                s2[--occ[P[k-1][s1[i]]]] = s1[i];
            for (int i = 1; i < n; ++i) {
                P[k][s2[i]] = same(s2[i], s2[i - 1], k, cnt)
                    ? P[k][s2[i - 1]] : i;
            }
        }
    }
    bool same(int i, int j, int k, int l) {
        return P[k - 1][i] == P[k - 1][j]
            && (i + l < n ? P[k - 1][i + l] : -1)
            == (j + l < n ? P[k - 1][j + l] : -1);
    }
    void compress() {
        vi cnt(256, 0);
        for (int i = 0; i < n; ++i) cnt[s[i]]++;
        for (int i = 0, mp = 0; i < 256; ++i)
            if (cnt[i] > 0) cnt[i] = mp++;
        for (int i = 0; i < n; ++i) P[0][i] = cnt[s[i]];
    }
    const vi &get_array() { return P.back(); }
    int lcp(int x, int y) {
        int ret = 0;
        if (x == y) return n - x;
        for (int k = P.size() - 1; k >= 0 && x < n && y < n; --k)
            if (P[k][x] == P[k][y]) {
                x += 1 << k;
                y += 1 << k;
                ret += 1 << k;
            }
        return ret;
    }
};
```

1.10 Convex Hull Set

```
template <class T>
struct ConvexHullSet {
    struct Line {
        T a, b;
        mutable ld x;
```

```cpp
        bool type;
        bool operator<(const Line &rhs) const {
            return type || rhs.type ? x < rhs.x : a < rhs.a;
        }
        ld intersect(const Line &rhs) const {
            return ld(b - rhs.b) / ld(rhs.a - a);
        }
    };
    static constexpr ld MAX = std::numeric_limits<T>::max() / 10;
    static constexpr ld MIN = std::numeric_limits<T>::min() / 10;
    set<Line> lines;
    void adjust(typename set<Line>::iterator it) {
        if(it != lines.begin()) {
            auto pit = prev(it);
            pit->x = it != lines.end() ? pit->intersect(*it) : MAX;
        }
    }
    bool empty() { return lines.empty(); }
    T query(T x) { // Returns max. Insert -ax-b / use -query(x) for min.
        auto it = lines.lower_bound(Line{T(0), T(0), ld(x), true});
        return it != lines.end() ? it->a * x + it->b : MIN;
    }
    void insert(T a, T b) {
        Line ln = Line{a, b, 0.0, false};
        auto it = lines.lower_bound(ln);
        if(it != lines.end() && it->a == a) {
            if(it->b >= b) return;
            adjust(it = lines.erase(it));
        }
        ln.x = it != lines.end() ? it->intersect(ln) : MAX;
        while(it != lines.end() && ln.x >= it->x) {
            adjust(it = lines.erase(it));
            ln.x = it != lines.end() ? it->intersect(ln) : MAX;
        }
        while(it != lines.begin()) {
            --it;
            ld nx = it->intersect(ln);
            if(nx >= it->x) return;
            if(it != lines.begin() && prev(it)->x >= nx)
                adjust(it = lines.erase(it));
            else
                break;
        }
        it = lines.insert(ln).first;
        adjust(it), adjust(next(it));
    }
};
```

1.11 Pareto Front

```cpp
struct pareto_front {
    map<ll, ll> m;
    void insert(ll a, ll b) {
        auto it = m.lower_bound(a);
        if (it != m.end() && it->second >= b)
            return;
        while (!m.empty() && (it = m.upper_bound(a)) != m.begin())
            if ((--it)->first <= a && it->second <= b)
                m.erase(it); else break;
        m[a] = b;
    }
    // max { b | (a, b) \in m, a >= u }, or -LLINF otherwise.
    ll max_tail(ll u) {
        auto it = m.lower_bound(u);
        return (it != m.end() ? it->second : -LLINF);
    }
};
```

1.12 GNU Built-in datastructures

These require gnu so check in test session.

```cpp
// Minimum Heap
#include <queue>
template<class T>
using min_queue = priority_queue<T, vector<T>, greater<T>>;

// Order Statistics Tree
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template<class TIn, class TOut>
using order_tree = tree<
    TIn, TOut, less<TIn>, // key, value types. TOut can be null_type
    rb_tree_tag, tree_order_statistics_node_update>;
// find_by_order(int r) (0-based)
// order_of_key(TIn v)
// use key pair<Tin,int> {value, counter} for multiset/multimap

// Need a much better hash function for gp_hash_table
struct custom_hash {
    static uint64_t splitmix64(uint64_t x) {
        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
            = chrono::steady_clock::now().time_since_epoch().count();
        return splitmix64(x + FIXED_RANDOM); } };

template<class TIn, class TOut>
using table = gp_hash_table<TIn, TOut>;
// Or gp_hash_table<uint64_t, TOut, custom_hash>
```

2 Combinatorics

2.1 Formulae

Permutations

The number of derangements D_n, n-permutations without a fixed point, satisfies $D_{n+1} = n(D_n + D_{n-1}) = (n+1)D_n + (-1)^{n+1}$. Stirling numbers of the first kind $S_1(n, k)$ count permutations on n items with k cycles. $S_1(n,k) = S_1(n-1,k-1) + (n-1)S_1(n-1,k)$ with $S_1(0,0) = 1$. Note $\sum_{k=0}^{n} S_1(n,k)x^k = x(x+1)\dots(x+n-1)$.

Eulerian numbers $E(n,k)$ count the number of permutations on n elments, with exactly k elements that are greater than their predecessor, i.e. k 'ascents'. We have $E(n,k) = (n-k)E(n-1,k-1) + (k+1)E(n-1,k)$ with $E(n,0) = E(n,n-1) = 1$.

Binomials and other partitionings

We have $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} = \prod_{i=1}^{k} \frac{n-i+1}{i}$. This last product may be computed incrementally since any product of k' consecutive values is divisibleby $k'!$.

Basic identities: The hockeystick identity: $\sum_{k=r}^{n}\binom{k}{r} = \binom{n+1}{r+1}$ or $\sum_{k\le n}\binom{r+k}{k} = \binom{r+n+1}{n}$. Also $\sum_{k=0}^{n}\binom{k}{m} = \binom{n+1}{m+1}$.

For $n,m \ge 0$ and p prime. Write n,m in base p, i.e. $n = n_k p^k + \dots + n_1 p + n_0$ and $m = m_k p^k + \dots m_1 p + m_0$. Then by Lucas theorem we have $\binom{n}{m} \equiv \prod_{i=0}^{k}\binom{n_i}{m_i} \mod p$, with the convention that $n_i < m_i \implies \binom{n_i}{m_i} = 0$.

```
ll binom(ll n, ll k){
    ll ans = 1;
    for(ll i = 1; i <= min(k,n-k); ++i) ans = ans*(n+1-i)/i;
    return ans;
}
ll lucas(ll n, ll k, ll p){ // calculate (n \choose k) % p
    ll ans = 1;
    while(n){
        ll np = n%p, kp = k%p;
        if(kp > np) return 0;
        ans *= binom(np,kp);
        n /= p; k /= p;
    }
    return ans;
}
```

We can put n indistinguishable items into k distinguishable bins in exactly $\binom{n+k-1}{k-1}$ ways, or if we require the bins be non-empty, in $\binom{n-1}{k-1}$ ways.

Stirling numbers of the second kind $S_2(n,k)$ count partitions of n distinct elements into exactly k non-empty groups. $S_2(n,k) = S_2(n-1,k-1) + kS_2(n-1,k)$ with $S_2(n,1) = S_2(n,n) = 1$ and

$$S_2(n,k) = \frac{1}{k!}\sum_{i=0}^{k}(-1)^{k-i}\binom{k}{i}i^n$$

Bell numbers B_n count arbitrary partitions of n distinct elements, i.e. $B_n = \sum_k S_2(n,k)$.

Catalan numbers

Catalan numbers C_n satisfy $C_n = \frac{1}{2n+1}\binom{2n}{n}$ as well as $C_0 = 1, C_{n+1} = \sum_i C_i C_{n-i}$. These count, among other things: Valid parenthesis sequences of length $2n$. Sub-diagonal monotone paths from $(0,0)$ to (n,n) in the $n \times n$ grid. Complete binary trees (i.e. no vertices with exactly 1 child) with $n+1$ leaves. Triangulations of a convex polygon with $n+2$ sides. Stack-sortable permutations of size n.

Trees

There are n^{n-2} labeled unrooted (cq. n^{n-1} rooted) trees on n vertices. A bijection between a tree and it's Prüfer sequence $(a_1, a_2, \dots, a_{n-2})$is given by:

- $\leftarrow$ Given the Prüfer sequence $(a_j)_{j=1}^{n-2}$, let d_i be 1 plus the number of occurences of i in a. Now for each a_j for $j = 1\dots$, find the lowest numbered node k with degree 1 and add $\{a_j, k\}$, and decrement the degrees d_{a_j}, d_k. At the end two degree 1 nodes remain, connect them.
- $\rightarrow$ Iterate, at step $i = 1, \dots, n-2$ remove the leaf with the smallest label and set the i^{th} element of a to be its neighbour.

A useful generalization: the number of labelled k-forests on n vertices with roots $1, 2, \dots, k$ is kn^{n-k-1} (careful with $k = n$).

2.2 Burnside's lemma

Given a group G acting on a set X, the number of elements in X up to symmetry is

$$\frac{1}{|G|}\sum_{g\in G}|X^g|$$

with X^g the elements of X invariant under g. For example, if $f(n)$ counts "configurations" of some sort of length n, and we want to count them up to rotational symmetry using $G = \mathbb{Z}/n\mathbb{Z}$, then

$$g(n) = \frac{1}{n}\sum_{k=0}^{n-1} f(\gcd(n,k)) = \frac{1}{n}\sum_{k\|n} f(k)\phi(n/k)$$

I.e. for coloring with c colors we have $f(k) = k^c$.

Relatedly, in Pólya's enumeration theorem we imagine X as a set of n beads with G permuting the beads (e.g. a necklace, with G all rotations and reflections of the n-cycle, i.e. the dihedral group D_n). Suppose further that we had Y colors, then the number of G-invariant colorings Y^X/G is counted by

$$\frac{1}{|G|}\sum_{g\in G}|Y|^{c(g)}$$

with $c(g)$ counting the number of cycles of g when viewed as a permutation of X. We can generalize this to a weighted version: if the color i can occur exactly r_i times, then this is counted by the coefficient of $t_1^{r_1}\dots t_n^{r_n}$ in the polynomial

$$Z(t_1,\dots,t_n) = \frac{1}{|G|}\sum_{g\in G}\prod_{m\ge 1}(t_1^m + \dots + t_n^m)^{c_m(g)}$$

where $c_m(g)$ counts the number of length m cycles in g acting as a permutation on X. Note we get the original formula by setting all $t_i = 1$. Here Z is the cycle index. Note: you can cleverly deal with even/odd sizes by setting some t_i to -1.

2.3 Inclusion-Exclusion

For appropriate f compute $\sum_{S\subseteq T}(-1)^{|T\setminus S|}f(S)$, or if only the size of S matters, $\sum_{s=0}^{n}(-1)^{n-s}\binom{n}{s}f(s)$. In some contexts we might use Stirling numbers, not binomial coefficients!

Some useful applications:

Graph coloring Let $I(S)$ count the number of independent sets contained in $S \subseteq V$ ($I(\emptyset) = 1$, $I(S) = I(S \setminus v) + I(S \setminus N(v))$). Let $c_k = \sum_{S \subseteq V} (-1)^{|V \setminus S|} I(S)$. Then V is k-colorable iff $v > 0$. Thus we can compute the chromatic number of a graph in $O^*(2^n)$ time.

2.4 Subset enumeration

Note: for sum over subsets use FSC.

```cpp
template<typename F>    // All subsets of size k of {0..N-1}
void iterate_k_subset(ll N, ll k, F f){
    ll mask = (1ll << k) - 1;
    while (!(mask & 1ll<<N)) { f(mask);
        ll t = mask | (mask-1);
        mask = (t+1) | (((~t & -~t) - 1) >> (__builtin_ctzll(mask)+1));
    }
}
template<typename F>    // All subsets of set
void iterate_mask_subset(ll set, F f){ ll mask = set;
    do  f(mask), mask = (mask-1) & set;
    while (mask != set);
}
int logfloor(unsigned long long a) {
    return __builtin_clzll(1) - __builtin_clzll(a);
}
```

Some related gnu built-in functions that might be useful. Note that all of these take *unsigned* values!!

`__builtin_popcount[ll](x)` Counts the number of 1s in `x`.

`__builtin_ctz[ll](x)` Counts the number of trailing 0s in `x`. In other words, the exponent of the largest power of 2 dividing `x`.

`__builtin_clz[ll](x)` Counts the number of leading 0s in `x`.

2.5 Fast Subset Convolution

Sets $a'(S) = \sum_{T \subseteq S} a(T)$ in $O(n2^n)$ time (rather than $O(3^n)$ or even $O(4^n)$). Note that if you flip the `+=` to a `-=` you get inclusion-exclusion for all subsets!

```cpp
void fsc(vi &a, bool inverse = false) {
    for (int s = 1, k = (int)a.size(); s < k; s <<= 1)
        for (int i = 0; i < k; ++i)
            if (!(i&s))
                a[i|s] += a[i] * (!inverse ? 1 : -1);
}
```

2.6 Stable Marriage Algorithm

Given n men and n women, with each man ranking each woman by preference, and vice versa, (i.e., permutations), we can find a stable marriage by applying this procedure:

- Initialize all men m and women w to 'free'.
- While there is a free man m for whom there is some woman w he has not proposed to, let w be the first/most prefered such woman. Now m proposes to w.
  - If w is free, then (m, w) become engaged.
  - If w is engaged to m' and w prefers m', nothing happens.
  - If w is engaged to m' and w prefers m, then (m, w) become engaged and m' becomes free.

Implement in $O(n^2)$ with the right bookkeeping. The resulting configuration marries everyone, with no unstable marriages (where two unmarried people prefer each other to their partners). When the number of men/women don't match add dummy people noone prefers.

2.7 2-SAT

```cpp
#include "../graphs/tarjan.cpp"
struct TwoSAT {
    int n;
    vvi imp; // implication graph
    Tarjan tj;

    TwoSAT(int _n) : n(_n), imp(2 * _n, vi()), tj(imp) { }

    // Only copy the needed functions:
    void add_implies(int c1, bool v1, int c2, bool v2) {
        int u = 2 * c1 + (v1 ? 1 : 0),
            v = 2 * c2 + (v2 ? 1 : 0);
        imp[u].push_back(v);        //  u =>  v
        imp[v^1].push_back(u^1);    // -v => -u
    }
    void add_equivalence(int c1, bool v1, int c2, bool v2) {
        add_implies(c1, v1, c2, v2);
        add_implies(c2, v2, c1, v1);
    }
    void add_or(int c1, bool v1, int c2, bool v2) {
        add_implies(c1, !v1, c2, v2);
    }
    void add_and(int c1, bool v1, int c2, bool v2) {
        add_true(c1, v1); add_true(c2, v2);
    }
    void add_xor(int c1, bool v1, int c2, bool v2) {
        add_or(c1, v1, c2, v2);
        add_or(c1, !v1, c2, !v2);
    }
    void add_true(int c1, bool v1) {
        add_implies(c1, !v1, c1, v1);
    }

    // on true: a contains an assignment.
    // on false: no assignment exists.
    bool solve(vb &a) {
        vi com;
        tj.find_sccs(com);
        for (int i = 0; i < n; ++i)
            if (com[2 * i] == com[2 * i + 1])
                return false;

        vvi bycom(com.size());
        for (int i = 0; i < 2 * n; ++i)
            bycom[com[i]].push_back(i);

        a.assign(n, false);
        vb vis(n, false);
        for(auto &&component : bycom){
            for (int u : component) {
                if (vis[u / 2]) continue;
                vis[u / 2] = true;
                a[u / 2] = (u % 2 == 1);
            }
        }
        return true;
    }
};
```

2.8 Simplex Algorithm

Maximize $c^t x$ subject to $Ax \le b$ and $x \ge 0$. With $A[m \times n], b[m], c[n], x[n]$. Solution in x. The dual problem is to maximize $b^t x$ subject to $A^t x \ge c$. Strong duality says the optima coincide.

```
#define REP(i, n) for(auto i = decltype(n)(0); i < (n); i++)
using T     = long double;
using vd    = vector<T>;
using vvd   = vector<vd>;
const T EPS = 1e-9;
struct LPSolver {
    int m, n;
    vi B, N;
    vvd D;
    LPSolver(const vvd &A, const vd &b, const vd &c)
        : m(b.size()), n(c.size()), B(m), N(n + 1), D(m + 2, vd(n + 2)) {
        REP(i, m) REP(j, n) D[i][j] = A[i][j];
        REP(i, m) B[i] = n + i, D[i][n] = -1, D[i][n + 1] = b[i];
        REP(j, n) N[j] = j, D[m][j] = -c[j];
        N[n]         = -1;
        D[m + 1][n] = 1;
    }
    void Pivot(int r, int s) {
        D[r][s] = 1.0 / D[r][s];
        REP(i, m + 2) if(i != r) REP(j, n + 2) if(j != s) D[i][j] -= D[r][j]
             * D[i][s] * D[r][s];
        REP(j, n + 2) if(j != s) D[r][j] *= D[r][s];
        REP(i, m + 2) if(i != r) D[i][s] *= -D[r][s];
        swap(B[r], N[s]);
    }
    bool Simplex(int phase) {
        int x = phase == 1 ? m + 1 : m;
        while(true) {
            int s = -1;
            REP(j, n + 1) {
                if(phase == 2 && N[j] == -1) continue;
                if(s == -1 || D[x][j] < D[x][s] || (D[x][j] == D[x][s] && N[
                    j] < N[s])) s = j;
            }
            if(D[x][s] >= -EPS) return true;
            int r = -1;
            REP(i, m) {
                if(D[i][s] <= EPS) continue;
                if(r == -1 || D[i][n + 1] / D[i][s] < D[r][n + 1] / D[r][s]
                    ||
                  (D[i][n + 1] / D[i][s] == D[r][n + 1] / D[r][s] && B[i] <
                       B[r]))
                    r = i;
            }
            if(r == -1) return false;
            Pivot(r, s);
        }
    }
    T Solve(vd &x) {
        int r = 0;
        for(int i = 1; i < m; i++)
            if(D[i][n + 1] < D[r][n + 1]) r = i;
        if(D[r][n + 1] <= -EPS) {
            Pivot(r, n);
            if(!Simplex(1) || D[m + 1][n + 1] < -EPS) return -INF;
            REP(i, m) if(B[i] == -1) {
                int s = -1;
                REP(j, n + 1)
                if(s == -1 || D[i][j] < D[i][s] || (D[i][j] == D[i][s] && N[
                    j] < N[s])) s = j;
                Pivot(i, s);
            }
        }
        if(!Simplex(2)) return INF;
        x = vd(n);
        REP(i, m) if(B[i] < n) x[B[i]] = D[i][n + 1];
        return D[m][n + 1];
    }
};
```

3 Mathematics

3.1 Elementary Number Theory

```
ll gcd(ll a, ll b) { while (b) { a %= b; swap(a, b); } return a; }
ll lcm(ll a, ll b) { return (a / gcd(a, b)) * b; }
ll mod(ll a, ll b) { return ((a % b) + b) % b; }
// Finds x, y s.t. ax + by = d = gcd(a, b).
void extended_euclid(ll a, ll b, ll &x, ll &y, ll &d) {
    ll xx = y = 0;
    ll yy = x = 1;
    while (b) {
        ll q = a / b;
        ll t = b; b = a % b; a = t;
        t = xx; xx = x - q * xx; x = t;
        t = yy; yy = y - q * yy; y = t;
    }
    d = a;
}
// solves ab = 1 (mod n), -1 on failure
```

```
ll mod_inverse(ll a, ll n) {
    ll x, y, d;
    extended_euclid(a, n, x, y, d);
    return (d > 1 ? -1 : mod(x, n));
}
// All modular inverses of [1..n] mod P in O(n) time.
vi inverses(ll n, ll P) {
    vi I(n+1, 1LL);
    for (ll i = 2; i <= n; ++i)
        I[i] = mod(-(P/i) * I[P%i], P);
    return I;
}
// (a*b)%m
ll mulmod(ll a, ll b, ll m){
    ll x = 0, y=a%m;
    while(b>0){
        if(b&1) x = (x+y)%m;
        y = (2*y)%m, b /= 2;
    }
    return x % m;
}
// Finds b^e % m in O(lg n) time, ensure that b < m to avoid overflow!
ll powmod(ll b, ll e, ll m) {
    ll p = e<2 ? 1 : powmod((b*b)%m,e/2,m);
    return e&1 ? p*b%m : p;
}
// Solve ax + by = c, returns false on failure.
bool linear_diophantine(ll a, ll b, ll c, ll &x, ll &y) {
    ll d = gcd(a, b);
    if (c % d) {
        return false;
    } else {
        x = c / d * mod_inverse(a / d, b / d);
        y = (c - a * x) / b;
        return true;
    }
}
```

3.2 Chinese Remainder Theorem

```
#include "./elementary-nt.cpp"
// Solves x = a1 mod m1, x = a2 mod m2, x is unique modulo lcm(m1, m2).
// Returns {0, -1} on failure, {x, lcm(m1, m2)} otherwise.
pair<ll, ll> crt(ll a1, ll m1, ll a2, ll m2) {
    ll s, t, d;
    extended_euclid(m1, m2, s, t, d);
    if (a1 % d != a2 % d) return {0, -1};
    return {mod(s*a2 %m2 * m1 + t*a1 %m1 * m2, m1 * m2) / d, m1 / d * m2};
}

// Solves x = ai mod mi. x is unique modulo lcm mi.
// Returns {0, -1} on failure, {x, lcm mi} otherwise.
pair<ll, ll> crt(vector<ll> &a, vector<ll> &m) {
    pair<ll, ll> res = {a[0], m[0]};
    for (ull i = 1; i < a.size(); ++i) {
        res = crt(res.first, res.second, mod(a[i], m[i]), m[i]);
        if (res.second == -1) break;
    }
    return res;
}
```

3.3 Tonelli-Shanks

```
#include "elementary-nt.cpp"
ll legendre(ll a, ll p) {
    return (powmod(a, (p-1)/2, p)+1) % p - 1;
}
// Returns a root of `a` modulo p, or -1 on failure. Note that if x>0
// is one root, n-x is the other. Exactly half of all values in
// [1..p-1] have two roots, the other half have no root.
// Runtime: O(log^2 p) if the generalized Riemann hypothesis is true.
ll sqrtp(ll a, ll p) {
    a = mod(a, p);
    if (a == 0) return 0;
    if (p == 2) return a;
    if (legendre(a, p) != 1) return -1;
    if ((p&3) == 3)             // Special cases
        return powmod(a, (p+1)/4, p);
    if ((p&7) == 5) {
        ll c = powmod(a, p/8+1, p);
        return (c*c%p == a ? c : c*powmod(2, p/4, p)%p);
    }
    ll q = p-1, s = 0, z = 1;
    while (!(q&1)) ++s, q >>= 1;
    while (legendre(z, p) != -1) ++z;
    ll x = powmod(a, (q+1)/2, p);
    for (ll c = powmod(z, q, p), t = powmod(a, q, p); t != 1;) {
        ll i = 1, t2i = t*t%p;
        while (t2i != 1) t2i = t2i*t2i%p, ++i;
        ll b = powmod(c, 1LL<<(s-i-1), p);
        x = (x*b)%p, c = (b*b)%p, t = (t*c)%p, s = i;
    }
    return x;
}
// Root modulo a prime power. Must have q = p^k for some k>1
ll sqrtp(ll a, ll p, ll q) {
    ll x = sqrtp(a, p);
    if (x < 0) return -1;
    ll r = q / p, e = (q - 2*r + 1)/2;
    return mulmod(powmod(x, r, q), powmod(a, e, q), q);
}
```

3.4 Euler-Phi function

For $n > 0$, $\phi(n)$ counts all numbers in $\{1, \ldots, n-1\}$ coprime to n. Note for $n = {p_1}^{k_1} \ldots p_m^{k_m}$ we have

$$\phi(n) = n \prod_i \frac{p_i - 1}{p_i} = n \prod_i (1 - \frac{1}{p_i})$$

Also $\sum_{d \| n} \phi(d) = n$. Finally for a and n relatively prime, $a^{\phi(n)} \equiv 1 \mod n$.

```
vi calculate_phi(int n) {
    vi phi(n + 1, 0LL);
    iota(phi.begin(), phi.end(), 0LL);
    for (ll i = 2LL; i <= n; ++i)
        if (phi[i] == i)
            for (ll j = i; j <= n; j += i)
                phi[j] -= phi[j] / i;
    return phi;
}
```

3.5 Floor sums

All in $O(\log n)$ time.

```
// Computes \sum_{k=1}^N \lfloor kp/q \rfloor
ll floor_sum(ll N, ll p, ll q) {
    ll t = __gcd(p, q), s = 0, z = 1;
    p /= t, q /= t;
    while (q > 0 && N > 0) {
        t = p/q;
        s += N*(N+1)/2*z*t;
        p -= q*t, t = N/q;
        s += z*p*t*(N+1) - z*t*(p*q*t+p+q-1)/2;
        N -= q*t, t = N*p/q;
        s += z*t*N;
        N = t, swap(p, q), z *= -1;
    }
    return s;
}
// Computes #{x,y \in ZxZ | x,y > 0, ax + by \leq c}
ll count_triangle(ll a, ll b, ll c) {
    assert(a > 0 && b > 0 && c > 0);
    if (b > a) swap(a, b);
    ll m = c/a;
    if (a == b) return m*(m-1)/2;
    ll k = (a-1)/b, h = (c-a*m)/b;
    return m*(m-1)/2*k + m*h + count_triangle(b, a-b*k, c-b*(k*m+h));
}
```

3.6 Continued Fractions

Continued fractions are typically written as:

$$[a_0; a_1, a_2, \dots] \leftrightarrow a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \frac{1}{\ddots}}}$$

Particularly well-known is $e \leftrightarrow [2; 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, \dots]$ and $\phi \leftrightarrow [1; 1, 1, \dots]$. The continued fraction representation of a number $r \in \mathbb{R}$ is finite if and only if $r \in \mathbb{Q}$. Real numbers with a periodic continued fraction are exactly the quadratic irrationals (roots of irreducible quadratics in $\mathbb{Q}[X]$, i.e. $\frac{a+b\sqrt{c}}{d}$ with c square free).

The convergents of a continued fraction $r \leftrightarrow [a_0; a_1, a_2, \dots]$ are given as $\frac{h_0}{k_0} = \frac{a_0}{1}$, $\frac{h_1}{k_1} = \frac{a_1 a_0 + 1}{a_1}$ and $\frac{h_n}{k_n} = \frac{a_n h_{n-1} + h_{n-2}}{a_n k_{n-1} + k_{n-2}}$. This sequences has the following useful properties:

$$[a_0; a_1, \dots, a_{n-1}, z] = \frac{z h_{n-1} + h_{n-2}}{z k_{n-1} + k_{n-2}}$$
$$\gcd(h_n, k_n) = 1$$

Also, the even convergents continually increase, whereas the odd convergents always decrease. Most importantly, these convergents give, in some sense, an ‘optimal’ approximation to r, that is, for any $\frac{h_n}{k_n}$ there exists no closer rational to r with smaller numerator or denominator.

Finally (Pell’s equation), for positive integers p, q and non-square n, it holds that $p^2 - nq^2 = \pm 1$ if and only if $\frac{p}{q}$ is a convergent of $\sqrt{n}$. Here, the following relation is useful for computing convergents of $\sqrt{D}$ where $D = \frac{p}{q}$: take $a_0 = \lfloor \sqrt{D} \rfloor, b_1 = a_0, c_1 = D - a_0^2$, and follow

$$a_{n-1} = \lfloor \tfrac{a_0 + b_{n-1}}{c_{n-1}} \rfloor, b_n = a_{n-1} c_{n-1} - b_{n-1}, c_n = \tfrac{D - b_n^2}{c_{n-1}}$$

Here b_n and c_n remain bounded by $2\sqrt{D}$. Maintain a cache to find the eventual period. The length of the period is bounded by pq, though usually it is much smaller. In fact, for $D \in \mathbb{Z}$ the period is $O(\sqrt{D} \log D)$. Additionally, the period will repeat from the second term, simplifying caching.

```
from fractions import Fraction
F, fl = Fraction, floor
cf = lambda f: [fl(f)] + ([] if not f-fl(f) else cf(1 / (f - fl(f))))
icf = lambda l: F(l[0]) if len(l) == 1 else l[0] + 1 / icf(l[1:])
approx = lambda f, c: f.limit_denominator(c)

# Takes two continued fractions in list form, and returns a continued
# fraction in (l, r) that has smallest numerator and denominator possible.
def lwithin(l, r):
    if len(r) == 0 or r[0] > l[0] + int(len(r) == 1): return [l[0]+1]
    if l[0]+1 == r[0]: r = [r[0]-1, 1]
    return [l[0]] + lwithin(r[1:], l[1:])

def cmp(l, r):
    if not r or not l or l[0] < r[0]: return bool(r)
    return cmp(r[1:], l[1:])
```

3.7 Enumerating $\mathbb{Q}$

For the **Stern-Brocot tree**, initially take the two ‘fractions’ $\frac{0}{1}$ and $\frac{1}{0}$. For any two fractions $\frac{a}{b}$ and $\frac{c}{d}$ define their mediant as $\frac{a+c}{b+d}$. We can iterate this process on every new layer of the tree, so the first value (the root) will be $\frac{0+1}{1+0} = \frac{1}{1}$, then $\frac{0+1}{1+1} = \frac{1}{2}$ and $\frac{1+1}{1+0} = \frac{2}{1}$, etc. This tree satisfies:

1. All rationals are enumerated in their lowest terms, exactly once.
2. If $\frac{a}{b}$ is a direct ancestor of $\frac{c}{d}$ then $ad - bc = \pm 1$.

The **Farey sequence** of order n is the sorted sequence of rationals in $[0, 1]$ in lowest terms whose denominators do not exceed n, e.g. $F_4 = \{0/1, 1/4, 1/3, 1/2, 2/3, 3/4, 1/1\}$. Note that $|F_n| = |F_{n-1}| + \phi(n) = 1 + \sum_{k=1}^{n} \phi(k)$. We can generate the Farey sequence in a manner similar to the Stern-Brocot tree, inserting $\frac{a+c}{b+d}$ between $\frac{a}{b}$ and $\frac{c}{d}$ when $c + d \leq n$.

3.8 Primes

Note that sieving into a bitset might be more space efficient.

$\leq N$	10^3	10^6	10^9	10^{12}	10^{18}
m	840	720720	735134400	963761198400	897612484786617600
$d(m)$	32	240	1344	6270	103680

Table 1: Maximizers of $d(n)$ upto some bound N.

```
#include "numbertheory.cpp"
ll SIZE;
vector<ll> primes;
vector<int> spf;
void sieve(ll size=1e6) {   // Initialize once in main.
    spf.assign((SIZE = size) + 1, 0);
    spf[0] = spf[1] = 1;
    for (ll i = 2; i <= SIZE; i++) {
        if (!spf[i]) primes.push_back(spf[i] = i);
        for (ll p : primes) {
            if (i*p > SIZE) break;
            spf[i*p] = p;
            if (spf[i] == p) break;
        }
    }
}

bool is_prime(ll n) {
    assert(n <= SIZE*SIZE);
    if (n <= SIZE) return spf[n] == n;
    for (ll p : primes) if (n % p == 0LL) return false;
    return true;
}
struct Factor{ ll p; ll exp; };
using FS = vector<Factor>;
FS factor(ll n){ // Up to SIZE^2
    FS fs;
    for (size_t i = 0; i < primes.size() && n > SIZE; ++i)
        if (n % primes[i] == 0) {
            fs.push_back({primes[i], 0});
            while (n % primes[i] == 0)
                fs.back().exp++, n /= primes[i];
        }
    if (n <= SIZE) {
        for(; n>1; n /= spf[n]) {
            if (!fs.empty() && fs.back().p == spf[n])
                ++fs.back().exp;
            else fs.push_back({spf[n], 1});
        }
    } else fs.push_back({n, 1});
    return fs;
}
```

3.9 Lucy's algorithm

Modified Meissel-Lehmer algorithm for summing complete multiplicative functions over the primes up to n in $O(n^{3/4})$ time. Example `f` and `F` compute the sum of squares.

```
ll f(ll n) { return n; }
ll F(ll n) { return n*(n+1)/2LL; } // = \sum_{i \leq n} f(i)
size_t id(ll i, ll N, const vi &V) {
    return i <= V.size()/2 ? V.size()-i : N/i-1;
}   // When adding 0 to V, add a -1 ^ here.
ll lucy(ll N) {
    vi V, S;
    for (ll i = 1; i <= N; i = N/(N/i)+1)
        V.push_back(N/i), S.push_back(F(N/i)-F(1));
    for (ll p = 2LL, sp; p*p <= N; ++p)
        if (S[id(p, N, V)] > (sp = S[id(p-1, N, V)]))
            for (size_t i = 0; i < V.size() && V[i] >= p*p; ++i)
                S[i] -= f(p)*(S[id(V[i]/p, N, V)] - sp); // <= UPDATE
    return S[0]; // Or return all of S for summation upto all N/k.
}
```

3.10 Miller-Rabin

```
#include "numbertheory.cpp"
vector<ll> test_primes  = {2,7,61};                  // <= 2^32
vector<ll> test_primes2 = {2,13,23,1662803};         // <= 1.1e12
vector<ll> test_primes3 = {2,3,5,7,11,13,17,19,23}; // <= 3.8e18, v <= 2^64
vector<ll> test_primes4 = {2,325,9375,28178,450775,9780504,1795265022};
//using ll = __int128   // uncomment for n>=1e9, or use mulmod instead
bool miller_rabin(const ll n){  // true when prime
    if(n<2) return false;
    if(n%2==0) return n==2;
    ll s = 0, d = n-1; // n-1 = 2^s * d
    while(~d&1) s++, d/=2;
    for(auto a : test_primes){
        if(a > n-2) break;
        ll x = powmod(a,d,n);
        if(x == 1 || x == n-1) continue;
        for (int i = 0; i < s - 1; ++i) {
            x = x*x%n;
            if(x==1) return false;
            if(x==n-1) goto next_it;
        }
        return false;
next_it:;
    }
    return true;
}
```

3.11 Pollard-ρ

```
#include "../elementary-nt.h"
using ull = unsigned long long;
ull diff(ull x, ull y) { return x > y ? x-y : y-x; }
ull g(ull x, ull a, ull n) { return (mulmod(x, x, n) + a) % n; }

ull pollard_rho(ull n, int reps = 10) {
    if (1&~n) return 2;
    for (ull a; reps-- > 0;) {
        do { a = rand() % 2321 + 47; } while (a == n - 2);
        ull x = 2+rand()%n, y = x, d = 1;
```

```cpp
        while (d == 1)
            x = g(x, a, n), y = g(g(y, a, n), a, n), d = gcd(diff(x, y), n);
        if (d < n) return d;
    }
    return n;
}
```

3.12 Finite Field

For usage with FFT.

```cpp
#include "./numbertheory.cpp"
template<ll p,ll w> // prime, primitive root
struct Field { using T = Field; ll x; Field(ll x=0) : x{x} {}
    T operator+(T r) const { return {(x+r.x)%p}; }
    T operator-(T r) const { return {(x-r.x+p)%p}; }
    T operator*(T r) const { return {(x*r.x)%p}; }
    T operator/(T r) const { return (*this)*r.inv(); }
    T inv() const { return {mod_inverse(x,p)}; }
    static T root(ll k) { assert( (p-1)%k==0 );      // (p-1)%k == 0?
        auto r = powmod(w,(p-1)/abs(k),p);           // k-th root of unity
        return k>=0 ? T{r} : T{r}.inv();
    }
    bool zero() const { return x == 0LL; }
};
using F1 = Field<1004535809,3 >;
using F2 = Field<1107296257,10>; // 1<<30 + 1<<25 + 1
using F3 = Field<2281701377,3 >; // 1<<31 + 1<<27 + 1
```

3.13 Complex Field

For usage with FFT.

```cpp
constexpr double pi = 3.141592653589793238462643 3; // or std::acos(-1)
struct Complex { using T = Complex; double u,v;
    Complex(double u=0, double v=0) : u{u}, v{v} {}
    T operator+(T r) const { return {u+r.u, v+r.v}; }
    T operator-(T r) const { return {u-r.u, v-r.v}; }
    T operator*(T r) const { return {u*r.u - v*r.v, u*r.v + v*r.u}; }
    T operator/(T r) const {
        auto norm = r.u*r.u+r.v*r.v;
        return {(u*r.u + v*r.v)/norm, (v*r.u - u*r.v)/norm};
    }
    T operator*(double r) const { return T{u*r, v*r}; }
    T operator/(double r) const { return T{u/r, v/r}; }
    T inv() const { return T{1,0}/ *this; }
    T conj() const { return T{u, -v}; }
    static T root(ll k){ return {cos(2*pi/k), sin(2*pi/k)}; }
    bool zero() const { return max(abs(u), abs(v)) < 1e-6; }
};
```

3.14 Fast Fourier Transform

Use Finite Field for NTT. For the complex FFT, `double` is noticably faster (> 2x) than `long double`. Use high precision FFT when worried about precision.

```cpp
#include "./complex.cpp"
#include "./field.cpp"
void brinc(int &x, int k) {
    int i = k - 1, s = 1 << i;
    x ^= s;
    if ((x & s) != s) {
        --i; s >>= 1;
        while (i >= 0 && ((x & s) == s))
            x = x &~ s, --i, s >>= 1;
        if (i >= 0) x |= s;
    }
}
using T = Complex;  // using T=F1,F2,F3
vector<T> roots;
void root_cache(int N) {
    if (N == (int)roots.size()) return;
    roots.assign(N, T{0});
    for (int i = 0; i < N; ++i)
        roots[i] = ((i&-i) == i)
            ? T{cos(2.0*pi*i/N), sin(2.0*pi*i/N)}
            : roots[i&-i] * roots[i-(i&-i)];
}
void fft(vector<T> &A, int p, bool inv = false) {
    int N = 1<<p;
    for(int i = 0, r = 0; i < N; ++i, brinc(r, p))
        if (i < r) swap(A[i], A[r]);
//  Uncomment to precompute roots (for T=Complex). Slower but more precise.
//  root_cache(N);
//               , sh=p-1        , --sh
    for (int m = 2; m <= N; m <<= 1) {
        T w, w_m = T::root(inv ? -m : m);
        for (int k = 0; k < N; k += m) {
            w = T{1};
            for (int j = 0; j < m/2; ++j) {
//              T w = (!inv ? roots[j<<sh] : roots[j<<sh].conj());
                T t = w * A[k + j + m/2];
                A[k + j + m/2] = A[k + j] - t;
                A[k + j] = A[k + j] + t;
                w = w * w_m;
            }
        }
    }
    if(inv){ T inverse = T(N).inv(); for(auto &x : A) x = x*inverse; }
}
// convolution leaves A and B in frequency domain state
// C may be equal to A or B for in-place convolution
void convolution(vector<T> &A, vector<T> &B, vector<T> &C){
    int s = A.size() + B.size() - 1;
    int q = 32 - __builtin_clz(s-1), N=1<<q;    // fails if s=1
    A.resize(N,{}); B.resize(N,{}); C.resize(N,{});
    fft(A, q, false); fft(B, q, false);
    for (int i = 0; i < N; ++i) C[i] = A[i] * B[i];
    fft(C, q, true); C.resize(s);
}
void square_inplace(vector<T> &A) {
    int s = 2*A.size()-1, q = 32 - __builtin_clz(s-1), N=1<<q;
    A.resize(N,{}); fft(A, q, false);
    for(auto &x : A) x = x*x;
    fft(A, q, true); A.resize(s);
}
```

3.15 Fast Walsh-Hadamard Tr.

To compute the xor convolution of $(a_i)_{i=0}^{2^k-1}$ and $(b_j)_{j=0}^{2^k-1}$, given as $c_k = \sum_{i \oplus j = k} a_i b_j$, use the Hadamard matrix $[[1,1],[1,-1]]$ inside of the inner loop of the FFT. In practice, this just means using $\omega = 1$. In particular, there is no need to compute roots and this transform works in any finite field $\mathbb{Z}/p\mathbb{Z}$.

3.16 High-Precision Convolution

A `double` is precise for all integers upto $2^{53} \approx 9e15$. This version breaks up the FFT in lower and upper 15 bits.

```
#include "./fft.cpp"
void convolution_mod(const vi &A, const vi &B, ll MOD, vi &C) {
    int s = A.size() + B.size() - 1; ll m15 = (1LL<<15)-1LL;
    int q = 32 - __builtin_clz(s-1), N=1<<q;    // fails if s=1
    vector<T> Ac(N), Bc(N), R1(N), R2(N);
    for (size_t i = 0; i < A.size(); ++i) Ac[i] = T{A[i]&m15, A[i]>>15};
    for (size_t i = 0; i < B.size(); ++i) Bc[i] = T{B[i]&m15, B[i]>>15};
    fft(Ac, q, false); fft(Bc, q, false);
    for (int i = 0, j = 0; i < N; ++i, j = (N-1)&(N-i)) {
        T as = (Ac[i] + Ac[j].conj()) / 2;
        T al = (Ac[i] - Ac[j].conj()) / T{0, 2};
        T bs = (Bc[i] + Bc[j].conj()) / 2;
        T bl = (Bc[i] - Bc[j].conj()) / T{0, 2};
        R1[i] = as*bs + al*bl*T{0,1}, R2[i] = as*bl + al*bs;
    }
    fft(R1, q, true); fft(R2, q, true);
    ll p15 = (1LL<<15)%MOD, p30 = (1LL<<30)%MOD; C.resize(s);
    for (int i = 0; i < s; ++i) {
        ll l = llround(R1[i].u), m = llround(R2[i].u), h = llround(R1[i].v);
        C[i] = (l + m*p15 + h*p30) % MOD;
    }
}
```

3.17 Polynomial Arithmetic

Power series inversion and polynomial division each in $O(n \log n)$ time.

```
#include "./fft.h"
vector<T> &rev(vector<T> &A) { reverse(A.begin(), A.end()); return A; }
void copy_into(const vector<T> &A, vector<T> &B, size_t n) {
    std::copy(A.begin(), A.begin()+min({n, A.size(), B.size()}), B.begin());
}

// Multiplicative inverse of A modulo x^n. Requires A[0] != 0!!
vector<T> inverse(const vector<T> &A, int n) {
    vector<T> Ai{A[0].inv()};
    for (int k = 0; (1<<k) < n; ++k) {
        vector<T> As(4<<k, T(0)), Ais(4<<k, T(0));
        copy_into(A, As, 2<<k); copy_into(Ai, Ais, Ai.size());
        fft(As, k+2, false); fft(Ais, k+2, false);
        for (int i = 0; i < (4<<k); ++i) As[i] = As[i]*Ais[i]*Ais[i];
        fft(As, k+2, true); Ai.resize(2<<k, {});
        for (int i = 0; i < (2<<k); ++i) Ai[i] = T(2) * Ai[i] - As[i];
    }
    Ai.resize(n);
    return Ai;
}
// Polynomial division. Returns {Q, R} such that A = QB+R, deg R < deg B.
// Requires that the leading term of B is nonzero.
pair<vector<T>, vector<T>> divmod(const vector<T> &A, const vector<T> &B) {
    size_t n = A.size()-1, m = B.size()-1;
    if (n < m) return {vector<T>(1, T(0)), A};

    vector<T> X(A), Y(B), Q, R;
    convolution(rev(X), Y = inverse(rev(Y), n-m+1), Q);
    Q.resize(n-m+1); rev(Q);

    X.resize(Q.size()), copy_into(Q, X, Q.size());
    Y.resize(B.size()), copy_into(B, Y, B.size());
    convolution(X, Y, X);

    R.resize(m), copy_into(A, R, m);
    for (size_t i = 0; i < m; ++i) R[i] = R[i] - X[i];
    while (R.size() > 1 && R.back().zero()) R.pop_back();
    return {Q, R};
}
vector<T> mod(const vector<T> &A, const vector<T> &B) {
    return divmod(A, B).second;
}
```

Some other useful operations:

$\sqrt{P(x)}$ Like Hensel lifting for the inverse, start with a solution $\mod x$. Then if $S_n(x)^2 = P(x) \mod x^n$, we have

$$S_{2n}(x) = \frac{1}{2}(S_n(x) + F(x)S_n(x)^{-1}) \mod x^{2n}$$

3.18 Linear Recurrence Solver

Given a linear recurrence of the form

$$a_n = \sum_{i=0}^{k-1} c_i a_{n-i-1}$$

this code computes a_n in $O(k \log k \log n)$ time. However, in practice the constant factor is quite large. On codeforces it runs in about 1200ms for $k = 3000$ and $n = 10^{18}$. Caching inverses for each input `n` gives about $800ms$.

```
#include "./poly.h"
// x^k mod f
vector<T> xmod(const vector<T> f, ll k) {
    vector<T> r{T(1)};
    for (int b = 62; b >= 0; --b) {
        if (r.size() > 1)
            square_inplace(r), r = mod(r, f);
        if ((k>>b)&1) {
            r.insert(r.begin(), T(0));
            if (r.size() == f.size()) {
                T c = r.back() / f.back();
                for (size_t i = 0; i < f.size(); ++i)
```

```
                    r[i] = r[i] - c * f[i];
                r.pop_back();
            }
        }
    }
    return r;
}
// Given A[0,k) and C[0, k), computes the n-th term of:
// A[n] = \sum_i C[i] * A[n-i-1]
T nth_term(const vector<T> &A, const vector<T> &C, ll n) {
    int k = (int)A.size();
    if (n < k) return A[n];

    vector<T> f(k+1, T{1});
    for (int i = 0; i < k; ++i)
        f[i] = T{-1} * C[k-i-1];
    f = xmod(f, n);

    T r = T{0};
    for (int i = 0; i < k; ++i)
        r = r + f[i] * A[i];
    return r;
}
```

3.19 Matrix Solver

```
#define REP(i, n) for(auto i = decltype(n)(0); i < (n); i++)
using T = double;
constexpr T EPS = 1e-8;
template<int R, int C>
using M = array<array<T,C>,R>;  // matrix
template<int R, int C>
T ReducedRowEchelonForm(M<R,C> &m, int rows) {  // return the determinant
    int r = 0; T det = 1;                           // MODIFIES the input
    for(int c = 0; c < rows && r < rows; c++) {
        int p = r;
        for(int i=r+1; i<rows; i++) if(abs(m[i][c]) > abs(m[p][c])) p=i;
        if(abs(m[p][c]) < EPS){ det = 0; continue; }
        swap(m[p], m[r]);       det = -det;
        T s = 1.0 / m[r][c], t; det *= m[r][c];
        REP(j,C) m[r][j] *= s;              // make leading term in row 1
        REP(i,rows) if (i!=r){ t = m[i][c]; REP(j,C) m[i][j] -= t*m[r][j]; }
        ++r;
    }
    return det;
}
bool error, inconst;    // error => multiple or inconsistent
template<int R,int C>   // Mx = a; M:R*R, v:R*C => x:R*C
M<R,C> solve(const M<R,R> &m, const M<R,C> &a, int rows){
    M<R,R+C> q;
    REP(r,rows){
        REP(c,rows) q[r][c] = m[r][c];
        REP(c,C) q[r][R+c] = a[r][c];
    }
    ReducedRowEchelonForm<R,R+C>(q,rows);
    M<R,C> sol; error = false, inconst = false;
    REP(c,C) for(auto j = rows-1; j >= 0; --j){
        T t=0; bool allzero=true;
        for(auto k = j+1; k < rows; ++k)
            t += q[j][k]*sol[k][c], allzero &= abs(q[j][k]) < EPS;
        if(abs(q[j][j]) < EPS)
            error = true, inconst |= allzero && abs(q[j][R+c]) > EPS;
        else sol[j][c] = (q[j][R+c] - t) / q[j][j]; // usually q[j][j]=1
    }
    return sol;
}
```

3.20 Matrix Exponentiation

Note: swapping the last two loops of the multiplication helps with speed (due to cache).

```
#define ITERATE_MATRIX(w) for (int r = 0; r < (w); ++r) \
                          for (int c = 0; c < (w); ++c)
template <class T, int N>
struct M {
    array<array<T,N>,N> m;
    M() { ITERATE_MATRIX(N) m[r][c] = 0; }
    static M id() {
        M I; for (int i = 0; i < N; ++i) I.m[i][i] = 1; return I;
    }
    M operator*(const M &rhs) const {
        M out;
        ITERATE_MATRIX(N) for (int i = 0; i < N; ++i)
                out.m[r][c] += m[r][i] * rhs.m[i][c];
        return out;
    }
    M raise(ll n) const {
        if(n == 0) return id();
        if(n == 1) return *this;
        auto r = (*this**this).raise(n / 2);
        return (n%2 ? *this*r : r);
    }
};
```

3.21 Game theory

A game can be reduced to Nim if it is a finite impartial game, then for any state x, $g(x) = \inf(\mathbb{N}_0 - \{g(y) : y \in F(x)\})$. Nim and its variants include:

Nim Let $X = \bigoplus_{i=1}^{n} x_i$, then $(x_i)_{i=1}^{n}$ is a winning position iff $X \neq 0$. Find a move by picking k such that $x_k > x_k \oplus X$.

Misère Nim Regular Nim, except that the last player to move *loses*. Play regular Nim until there is only one pile of size larger than 1, reduce it to 0 or 1 such that there is an odd number of piles.

Moore's Nim$_k$ The player may remove from at most k piles (Nim = Nim$_1$). Expand the piles in base 2, do a carry-less addition in base $k+1$ (i.e. the number of ones in each column should be divisible by $k+1$).

Lasker's Nim Players may alternatively split a pile into two new non-empty piles. $g(4k+1) = 4k+1$, $g(4k+2) = 4k+2$, $g(4k+3) = 4k+4$, $g(4k+4) = 4k+3$ ($k \geq 0$).

Hackenbush on trees A tree with stalks $(x_i)_{i=1}^{n}$ may be replaced with a single stalk with length $\bigoplus_{i=1}^{n} x_i$.

For others, bruteforce g for small x and try to spot a pattern. And a useful identity: $\bigoplus_{x=0}^{a-1} x = \{0, a-1, 1, a\}[a\%4]$.

4 Graph Algorithms

4.1 Tarjan's Algorithm

```cpp
struct Tarjan {
    vvi &edges;
    int V, counter = 0, C = 0;
    vi n, l;
    vb vs;
    stack<int> st;
    Tarjan(vvi &e) : edges(e), V(e.size()),
        n(V, -1), l(V, -1), vs(V, false) { }
    void visit(int u, vi &com) {
        l[u] = n[u] = counter++;
        st.push(u); vs[u] = true;
        for (auto &&v : edges[u]) {
            if (n[v] == -1) visit(v, com);
            if (vs[v]) l[u] = min(l[u], l[v]);
        }
        if (l[u] == n[u]) {
            while (true) {
                int v = st.top(); st.pop(); vs[v] = false;
                com[v] = C;     //<== ACT HERE
                if (u == v) break;
            }
            C++;
        }
    }
    int find_sccs(vi &com) { // component indices will be stored in 'com'
        com.assign(V, -1);
        C = 0;
        for (int u = 0; u < V; ++u)
            if (n[u] == -1) visit(u, com);
        return C;
    }
    // scc is a map of the original vertices of the graph to the vertices
    // of the SCC graph, scc_graph is its adjacency list.
    // SCC indices and edges are stored in 'scc' and 'scc_graph'.
    void scc_collapse(vi &scc, vvi &scc_graph) {
        find_sccs(scc);
        scc_graph.assign(C,vi());
        set<ii> rec; // recorded edges
        for (int u = 0; u < V; ++u) {
            assert(scc[u] != -1);
            for (int v : edges[u]) {
                if (scc[v] == scc[u] ||
                    rec.find({scc[u], scc[v]}) != rec.end()) continue;
                scc_graph[scc[u]].push_back(scc[v]);
                rec.insert({scc[u], scc[v]});
            }
        }
    }
};
```

4.2 Biconnected Components

```cpp
struct BCC{      // find AVs and bridges in an undirected graph
    vvi &edges;
    int V, counter = 0, root, rcs;      // root and # children of root
    vi n,l;                             // nodes,low
    stack<int> s;
    BCC(vvi &e) : edges(e), V(e.size()), n(V,-1), l(V,-1) {}
    void visit(int u, int p) {          // also pass the parent
        l[u] = n[u] = counter++; s.push(u);
        for(auto &v : edges[u]){
            if (n[v] == -1) {
                if (u == root) rcs++; visit(v,u);
                if (l[v]>=n[u] && u!=root) {} // u is an articulation point
                if (l[v] > n[u]) {      // u<->v is a bridge
                    while(true){        // biconnected component
                        int w = s.top(); s.pop();   // <= ACT HERE
                        if(w==v) break;
                    }
                }
                l[u] = min(l[u], l[v]);
            } else if (v != p) l[u] = min(l[u], n[v]);
        }
    }
    void run() {
        for (int u = 0; u < V; ++u) if (n[u] == -1) {
            root = u; rcs = 0; visit(u,-1);
            if(rcs > 1) {}              // u is articulation point
            while(!s.empty()){          // biconnected component
                int w = s.top(); s.pop();   // <= ACT HERE
            }
        }
    }
};
```

4.3 MDST

Minimum directed spanning tree rooted at some r. Runs in $O(E \log V)$. About 3s for a complete graph on $n = 2500$ vertices on Kattis.

```cpp
#include "../datastructures/unionfind.cpp"
#include "../datastructures/skew-heap.cpp"
struct arc { int u, v; ll w; };
// Returns the weight of a minimum rooted arborescence, -1 if none exists.
ll solve(int root, int n, const vector<arc> &A) {
    UnionFind uf(n);
    vector<SkewHeap<size_t>> sk(n);
    for (size_t a = 0; a < A.size(); ++a)
        if (A[a].v != root && A[a].u != A[a].v)
            sk[A[a].v].insert(A[a].w, a);
    ll ans = 0LL; vi best(n, -LLINF);
    for (int i = 0; i < n; ++i) {
        if (uf.find(i) == root) continue;
        stack<int> st; st.push(i);
        while (true) {
            int u = st.top(), v; ll w; size_t a;
            if (sk[u].empty()) return -1LL; // If you only want a partial
            tie(w, a) = sk[u].pop_min();    // tree just discard the stack
            v = uf.find(A[a].u);            // and carry on. DONT UNION
            if (v == u) continue;           // ‘root‘ AFTER THE WHILE!!!
            ans += (best[u] = w);   // To reconstruct the solution: before
            if (v == root) break;   // <-this check store the arc A[a], and
            if (best[v] == -LLINF) {// at the end take any DFS tree over
                st.push(v);         // these arcs.
            } else {
                while (true) {
                    sk[st.top()].adjust(-best[st.top()]);
                    if (st.top() != u) sk[u].absorb(sk[st.top()]);
                    if (uf.find(st.top()) == v) break;
                    else uf.merge(st.top(), v), v = uf.find(v), st.pop();
                }
                swap(sk[u], sk[v]);
                st.pop(), st.push(v);
            }
        }
        while (!st.empty()) uf.merge(root, st.top()), st.pop();
        root = uf.find(root);
    }
    return ans;
}
```

4.4 Eulerian paths and tours

In an *undirected graph*, an *Eulerian Circuit* exists iff all vertices have even degree, and all vertices of nonzero degree belong to a single connected component. In an *undirected graph*, an *Eulerian Trail* exists iff at most two vertices have odd degree, and all vertices of nonzero degree belong to a single connected component. In a *directed graph*, an *Eulerian Circuit* exists iff every vertex has equal indegree and outdegree, and all vertices of nonzero degree belong to a single strongly connected component. In a *directed graph*, an *Eulerian Trail* exists iff at most one vertex has $d_{out} - d_{in} = 1$, at most one vertex has $d_{in} - d_{out} = 1$, every other vertex has equal indegree and outdegree, and all vertices of nonzero degree belong to a single connected component *in the underlying undirected graph*.

```
struct edge {
    int v;
    list<edge>::iterator rev;
    edge(int _v) : v(_v) {};
};
void add_edge(vector< list<edge> > &adj, int u, int v) {
    adj[u].push_front(edge(v));
    adj[v].push_front(edge(u));
    adj[u].begin()->rev = adj[v].begin();
    adj[v].begin()->rev = adj[u].begin();
}
void remove_edge(vector< list<edge> > &adj, int s, list<edge>::iterator e) {
    adj[e->v].erase(e->rev);
    adj[s].erase(e);
}
eulerian_circuit(vector< list<edge> > &adj, vi &c, int start = 0) {
    stack<int> st;
    st.push(start);
    while(!st.empty()) {
        int u = st.top().first;
        if (adj[u].empty()) {
            c.push_back(u);
            st.pop();
        } else {
            st.push(adj[u].front().v);
            remove_edge(adj, u, adj[u].begin());
        }
    }
}
```

4.5 Bellmann-Ford

$O(VE)$

```
void bellmann_ford_extended(vvii &e, int source, vi &dist, vb &cyc) {
    dist.assign(e.size(), INF);
    cyc.assign(e.size(), false); // true when u is in a <0 cycle
    dist[source] = 0;
    for (int iter = 0; iter < e.size() - 1; ++iter){
        bool relax = false;
        for (int u = 0; u < e.size(); ++u)
            if (dist[u] == INF) continue;
            else for (auto &e : e[u])
                if(dist[u]+e.second < dist[e.first])
                    dist[e.first] = dist[u]+e.second, relax = true;
        if(!relax) break;
    }
    bool ch = true;
    while (ch) {                // keep going untill no more changes
        ch = false;             // set dist to -INF when in cycle
        for (int u = 0; u < e.size(); ++u)
            if (dist[u] == INF) continue;
            else for (auto &e : e[u])
                if (dist[e.first] > dist[u] + e.second
                    && !cyc[e.first]) {
                    dist[e.first] = -INF;
                    ch = true;      //return true for cycle detection only
                    cyc[e.first] = true;
                }
    }
}
```

4.6 Johnson’s reweighting

For computing all pairs shortest paths in sparse graphs with negative weights. Apply Bellman-Ford to the graph with `d[u] = 0` (as if an extra vertex with zero weight edges were added), then reweight edges to $w_{uv} + h_u - h_v$, then use Dijkstra from every vertex. The result is $O(VE + (V + E)(\log E))$.

4.7 Tree isomorphisms

For a rooted tree isomorphism, from the lowest level to the highest, compute a hash for each subtree based on the ordered or unordered set of hashes of the children. This can be implemented deterministically and fast using radix sorts, but this takes a lot of code - use strong hash functions instead. An example of a good hash function is to pick $s_d \in \mathbb{F}_p$ uniformly at random for each depth d and hash a vertex v at depth d to $h_v = s_d + \prod_{u \text{ child of } v} h_u$.

For an unrooted tree isomorphism, try finding a rooted tree isomorphism between all pairs of centroids.

4.8 Relevant Theorems

Kirchhoff's Theorem Given an undirected graph G on n vertices, let D be the $n \times n$ matrix with all degrees on the diagonal, and A the $n \times n$ incidence matrix of G. Let M be any minor (remove one row and one column) of $D - A$. Then the number of spanning trees of G is $|\det M|$.

Acyclicity A directed graph is acyclic if and only a depth-first search yields no back edges.

5 Flows and cuts

5.1 Flow Graph

Datastructure used by all flow algorithms

```
using F = ll; using W = ll; // types for flow and weight/cost
struct S{
    const int v;             // neighbour
    const int r;    // index of the reverse edge
    F f;            // current flow
    const F cap;    // capacity
    const W cost;   // unit cost
    S(int v, int ri, F c, W cost = 0) :
        v(v), r(ri), f(0), cap(c), cost(cost) {}
    inline F res() const { return cap - f; }
};
struct FlowGraph : vector<vector<S>> {
    FlowGraph(size_t n) : vector<vector<S>>(n) {}
    void add_edge(int u, int v, F c, W cost = 0){ auto &t = *this;
        t[u].emplace_back(v, t[v].size(), c, cost);
        t[v].emplace_back(u, t[u].size()-1, c, -cost);
    }
    void add_arc(int u, int v, F c, W cost = 0){ auto &t = *this;
        t[u].emplace_back(v, t[v].size(), c, cost);
        t[v].emplace_back(u, t[u].size()-1, 0, -cost);
    }
    void clear() { for (auto &E : *this) for (auto &e : E) e.f = 0LL; }
};
```

5.2 Dinic Algorithm

Runs in $O(V^2E)$, or $O(VE\log U)$ after introducing scaling. On unit capacity networks it runs in $O(\min\{V^{2/3}, E^{1/2}\}E)$ time. On networks where every vertex has either a single incoming edge that also has unit capacity, or a single outgoing edge that also has unit capacity, it runs in $O(E\sqrt{V})$.

```
#include "flowgraph.cpp"
struct Dinic{
    FlowGraph &edges; int V,s,t;
    vi l; vector<vector<S>::iterator> its; // levels and iterators
    Dinic(FlowGraph &edges, int s, int t) :
        edges(edges), V(edges.size()), s(s), t(t), l(V,-1), its(V) {}
    ll augment(int u, F c) { // we reuse the same iterators
        if (u == t) return c; ll r = 0LL;
        for(auto &i = its[u]; i != edges[u].end(); i++){
            auto &e = *i;
            if (e.res() && l[u] < l[e.v]) {
                auto d = augment(e.v, min(c, e.res()));
                if (d > 0) { e.f += d; edges[e.v][e.r].f -= d; c -= d;
                    r += d; if (!c) break; }
        }   }
        return r;
    }
    ll run() {
        ll flow = 0, f;
        while(true) {
            fill(l.begin(), l.end(),-1); l[s]=0; // recalculate the layers
            queue<int> q; q.push(s);
            while(!q.empty()){
                auto u = q.front(); q.pop(); its[u] = edges[u].begin();
                for(auto &&e : edges[u]) if(e.res() && l[e.v]<0)
                    l[e.v] = l[u]+1, q.push(e.v);
            }
            if (l[t] < 0) return flow;
            while ((f = augment(s, INF)) > 0) flow += f;
    }   }
};
```

5.3 Minimum Cost Flow

Cannot handle negative cycles

```
#include "flowgraph.cpp"
using F = ll; using W = ll; W WINF = LLINF; F FINF = LLINF;
struct Q{ int u; F c; W w;}; // target, maxflow and total cost/weight
bool operator>(const Q &l, const Q &r){return l.w > r.w;}
struct Edmonds_Karp_Dijkstra{
    FlowGraph &g; int V,s,t; vector<W> pot;
    Edmonds_Karp_Dijkstra(FlowGraph &g, int s, int t) :
        g(g), V(g.size()), s(s), t(t), pot(V) {}
    pair<F,W> run() { // return pair<f, cost>
        F maxflow = 0; W cost = 0;           // Bellman-Ford for potentials
        fill(pot.begin(),pot.end(),WINF); pot[s]=0;
        for (int i = 0; i < V - 1; ++i) {
            bool relax = false;
            for (int u = 0; u < V; ++u) if(pot[u] != WINF)
                for(auto &e : g[u])
                    if(e.cap>e.f)
                        if(pot[u] + e.cost < pot[e.v])
                            pot[e.v] = pot[u] + e.cost, relax=true;
            if(!relax) break;
        }
        for (int u = 0; u < V; ++u) if(pot[u] == WINF) pot[u] = 0;
        while(true){
            priority_queue<Q,vector<Q>,greater<Q>> q;
            vector<vector<S>::iterator> p(V,g.front().end());
            vector<W> dist(V, WINF); F f, tf = -1;
            q.push({s, FINF, 0}); dist[s]=0;
            while(!q.empty()){
                int u = q.top().u; W w = q.top().w;
                f = q.top().c; q.pop();
                if(w!=dist[u]) continue; if(u==t && tf < 0) tf = f;
                for(auto it = g[u].begin(); it!=g[u].end(); it++){
                    auto &e = *it;
                    W d =  w + e.cost + pot[u] - pot[e.v];
                    if(e.cap>e.f && d < dist[e.v]){
                        q.push({e.v, min(f, e.cap-e.f),dist[e.v] = d});
                        p[e.v]=it;
                    }   }   }
            auto it = p[t];
            if(it == g.front().end()) return {maxflow,cost};
            maxflow += f = tf;
            while(it != g.front().end()){
                auto &r = g[it->v][it->r];
                cost += f * it->cost; it->f+=f;
                r.f -= f; it = p[r.v];
            }
            for (int u = 0; u < V; ++u) if(dist[u]!=WINF) pot[u] += dist[u];
        }
    }
};
```

5.4 Minimum Cost Circulation

Fast in practice, can easily do min cost matching upto $n = 1000$.

```
#include "flowgraph.h"
#include "dinic.cpp"
int logfloor(unsigned long long a) {
    return __builtin_clzll(1) - __builtin_clzll(a);
}
struct MinCostCirculation {
    FlowGraph &fg; int V, log_eps = 0, dt = 2;
    ll flow = 0LL, cost = 0LL;
    vi x, p, a; vector<vector<S>::iterator> its;
    MinCostCirculation(FlowGraph &fg)
        : fg(fg), V(fg.size()), x(V), p(V), a(V), its(V) { }
    inline ll eps() { return 1LL<<log_eps; }
    pair<ll, ll> run(int s = -1, int t = -1) {
        if (s>=0 && t>=0) Dinic(fg, s, t).run(); else assert(s<0 && t<0);
        for (const auto &E : fg) for (const auto &s : E)
            log_eps = max(log_eps, 1+logfloor(abs(V*s.cost)));
        stack<int> q; // or queue<int> q;
        for (log_eps = ((log_eps+dt-1)/dt)*dt; log_eps>=0; log_eps-=dt) {
            for (int u = 0; u < V; ++u) {
                its[u] = fg[u].begin();
                for (auto &e : fg[u])
                    if (e.res() && e.cost*V + p[u] < p[e.v])
                        push(u, e, e.res());
            }
            for (int u = 0; u < V; ++u) if (x[u] > 0) q.push(u), a[u] = 1;
            while (!q.empty()) {
                int u = q.top(); q.pop(); a[u] = 0;
                while (x[u] > 0) {
                    if (its[u] == fg[u].end()) relabel(u);
                    for (auto &i = its[u]; i != fg[u].end() && x[u]; ++i) {
                        auto &e = *i;
                        if (e.res() && e.cost*V + p[u] < p[e.v]) {
                            push(u, e, min(e.res(), x[u]));
                            if (x[e.v]>0 && !a[e.v]) q.push(e.v), a[e.v]=1;
        }   }   }   }   }
        for (const auto &e : fg[s]) flow += e.f;
        for (const auto &E : fg) for (const auto &e : E) cost += e.f*e.cost;
        return {flow, cost /= 2LL};
    }
    void push(int u, S &e, ll d) {
        e.f += d, x[u] -= d, fg[e.v][e.r].f -= d, x[e.v] += d;
    }
    void relabel(int u) {
        ll &pu = (p[u] = -LLINF);
        for (auto it = fg[u].begin(); it != fg[u].end(); ++it) {
            auto &e = *it;
            ll pi = p[e.v] - e.cost*V;
            if (e.res() && pu < pi) pu = pi, its[u] = it;
        }
        pu -= eps();
    }
};
```

5.5 Relevant Constructions

Circulation Add a supersource S and supersink T. For an arc (x, y) with lowerbound l and upperbound u, set its new capacity to $u - l$, and add arcs (S, y) and (x, T) with capacity l. Then find a flow from S to T. This corresponds to a circulation if the flow equals $\sum_{(x,y)} l$. Note: This approach can often be sped up significantly by merging double edges.

Tutte matrix For a graph $G = (V, E)$, its Tutte matrix T is defined as T_{ij} equal to 0 if $\{i, j\} \notin E$, x_{ij} if $i < j$ or $-x_{ji}$ if $i > j$. The rank of T is twice the size of the maximum matching in G. To compute: assign x_{ij} randomly in $\mathbb{F}_p$ and compute the rank for some number of iterations k. Take the maximum over all runs, for a total runtime of $O(kn^3)$. Each iteration has a failure probability of at most n/p. Also relevant for reconstruction is the Rabin-Vazirani lemma: if G has a perfect matching then $G - \{v_i, v_j\}$ has a perfect matching if $(T^{-1})_{ij} \neq 0$ where T is the Tutte matrix of G.

Project selection A very non-trivial application of flow is the project selection problem, where we have a set of projects P, and each project $i \in P$ has an associated revenue p_i (which can be positive or negative). There are dependencies between the projects, given by some acylic graph $G = (P, A)$ (cyclic dependencies can just be collapsed into a single vertex). The goal is of course to select some set of projects $S \subseteq P$ so as to maximize profit while satisfying the dependencies.

To solve, build a flow graph G' with (s, i) with capacity p_i if $p_i > 0$, or (i, t) with capacity $-p_i$ if $p_i < 0$. Furthermore add infinite capacity edges to G' for the edges of G. Let $C = \sum_{p_i > 0} p_i$, then we claim that for any cut (A', B'), the set $A = A \setminus \{s\}$

is a set of projects that satisfies $c(A', B') = C - \sum_{i \in A} p_i$. Therefore the minimum (s,t)-cut in G' solves the problem.

5.6 Relevant Theorems

Min-Cut Max-Flow The minimum cut separating s and t equals the maximum flow between s and t.

Königs Theorem There is a bijection between the maximum matchings in a bipartite graph and the minimal vertex covers. To find the cover: find a maximum matching and run minimum cut inference. Then pick all vertices that are on the 'wrong' side. Also note the complement of a minimum vertex cover is a maximum independent set.

Hall's Marriage Theorem For a bipartite graph $(L \cup R, E)$, a matching saturating L exists iff $\forall L' \subseteq L$ we have $|L'| \leq |N(L')|$ where $N(L')$ is the neighbour set of L'.

Dilworth's Theorem The minimum number of disjoint chains into which a partial ordering S can be decomposed equals the length of the longest antichain of S. Compute by defining a bipartite graph with l_x and r_x for each $x \in S$, and add (l_x, r_y) if $x < y$. For a maximum matching of size m the number of disjoint chains is then $|S| - m$. To find the actual antichain, find the minimal vertex cover and take all x with neither l_x nor r_x in the cover.

Mirsky's Theorem Dual theorem to the above, for the smallest decomposition into antichains and the longest chain. Easily computable using dynamic programming.

5.7 Karger's Algorithm

To find a minimum cut in a graph, repeatedly contract a random edge until there are only two components left. One easy way to implement this is to randomly give each edge a weight, and then run Kruskal's algorithm until only two components are left. To find the cut with high probability, repeat the algorithm $\binom{n}{2} \log n$ times for a failure probability of $\frac{1}{n}$. The total running time is then $O(n^2 m \log n)$.

5.8 Gomory-Hu Tree

Does $V - 1$ minimum cut computations. Useful: taking the $k - 1$ lightest edges in the tree gives a $2 - 2/k$ approximation to the minimum k-cut problem

```cpp
#include "./dinic.cpp"
struct edge { int u, v; ll w; };
struct GomoryHuTree {       // After construction, for all n-1 i such that
    int V; vi p, w, c;      // p[i] \geq 0, the edge (i, p[i]) is in the GH
    vector<edge> tree;      // tree with weight w[i].
    void dfs(int u, const FlowGraph &fg) {
        c[u] = 1;
        for (const auto &e : fg[u]) if (!c[e.v] && e.res()) dfs(e.v, fg);
    }
    GomoryHuTree(int n, const vector<edge> &E) : V(n), p(V), w(V), c(V) {
        FlowGraph fg(V);
        for (const edge &e : E) fg.add_edge(e.u, e.v, e.w);
        p[0] = -1, std::fill(p.begin() + 1, p.end(), 0);
        for (int i = 1; i < V; ++i, fg.clear()) {
            w[i] = Dinic(fg, i, p[i]).run();
            std::fill(c.begin(), c.end(), 0);
            dfs(i, fg);
            for (int j = i+1; j < V; ++j)
                if (c[j] && p[j] == p[i]) p[j] = i;
            if (p[p[i]] >= 0 && c[p[p[i]]]) {
                int pi = p[i];
                swap(w[i], w[pi]);
                p[i] = p[pi], p[pi] = i;
            }
        }
    }
};
```

6 Dynamic Programming

6.1 LIS

For the actual sequence, store parent pointers. Note that longest common subsequence can be reduced to LIS with unique elements.

```cpp
// Length only
template<class T>
int longest_increasing_subsequence(vector<T> &a) {
    set<T> st;
    typename set<T>::iterator it;
    for (int i = 0; i < a.size(); ++i) {
        it = st.lower_bound(a[i]);
        if (it != st.end()) st.erase(it);
        st.insert(a[i]);
    }
    return st.size();
}
```

6.2 All Nearest Smaller Values

```cpp
void all_nearest_smaller_values(vi &a, vi &b) {
    b.assign(a.size(), -1);
    for (int i = 1; i < b.size(); ++i) {
        b[i] = i - 1;
        while (b[i] >= 0 && a[i] < a[b[i]])
            b[i] = b[b[i]];
    }
}
```

6.3 DP Optimizations

CH Recurrences are generally of the form $dp_i = dp_j \cdot a_i + b_j$ or equivalent. Insert all lines as functions of a_i into the convex hull set (see datastructures) to query in logarithmic time.

DC Recurrences are generally of the form $dp_{k,i} = \max_{j<i} f(dp_{k-1,j}, i, j, k)$. If the arg max is monotonic in i, do divide and conquer by first finding the answer for $i = \frac{n}{2}$, and then recursing for larger and smaller i.

Knuth Recurrences are generally of form similar to

$$dp_{i,j} = min_{i \le k < j}\big(f(i,j) + dp_{i,k} + dp_{k+1,j}\big)$$

Let $a_{i,j}$ be the argmin for $dp_{i,j}$, then if $a_{i,j-1} \le a_{i,j} \le a_{i+1,j}$, for a fixed $len > 0$ the ranges we have to scan for all $dp_{i,i+len}$ are (almost) disjoint, and we can solve the recurrence in $O(n^2)$ time.

A sufficient condition for applicability on f is, for $i \le j \le k \le l$, monotonicity: $f(j,k) \le f(i,l)$, and the quadrangle inequality: $f(i,k) + f(j,l) \le f(i,l) + f(j,k)$.

7 Geometry

Use 64-bit integer arithmetic whenever possible.

7.1 Formulae I

To **intersect two lines**, assume by equations of the form $ax + by = e$. For a line going through points (x_1, y_1) and (x_2, y_2), note that $\vec{n} = (y_1 - y_2, x_2 - x_1)$ is a normal vector, so the line equation is given by $\vec{n} \cdot \vec{p} = \vec{n} \cdot (x_1, y_1)$ for $\vec{p} \in \mathbb{R}^2$. Note: normalizing $\vec{n}$ might be necessary due to input sizes, but this does result in a loss of precision. Then:

$$\begin{matrix} ax + by = e \\ cx + dy = f \end{matrix} \rightarrow \begin{matrix} x = \dfrac{ed - bf}{ad - bc} \\ y = \dfrac{af - ec}{ad - bc} \end{matrix}$$

This equation is undefined precisely when $ad - bc = 0$, i.e. the given lines coincide $(e = f)$ or are parellel $(e \ne f)$. For segment-segment intersections, just check that (x, y) is in the bounding box of the two segments. For a general equation of the form $Ax = b$, we have $x_i = \det A_i / \det A$ with A_i equal to A with the i^{th} column replaced by b.

When only **deciding whether two segments intersect**, there is no need to compute the intersection point, use determinants instead (or rather, the derived `ccw` function). Specifically, the sequence $\vec{a} \to \vec{b} \to \vec{c}$ goes counterclockwise iff $(\vec{b} - \vec{a}) \times (\vec{c} - \vec{a})$ is positive, and clockwise if it is negative. Colinearity if and only if the determinant is 0. For example:

```cpp
bool intersect(P a1, P a2, P b1, P b2) {
    if (max(a1.x, a2.x) < min(b1.x, b2.x)) return false;
    if (max(b1.x, b2.x) < min(a1.x, a2.x)) return false;
    if (max(a1.y, a2.y) < min(b1.y, b2.y)) return false;
    if (max(b1.y, b2.y) < min(a1.y, a2.y)) return false;
    bool l1 = ccw(a2, b1, a1) * ccw(a2, b2, a1) <= 0;
    bool l2 = ccw(b2, a1, b1) * ccw(b2, a2, b1) <= 0;
    return l1 && l2;
}
```

In C++, the angle between (x_1, y_1) and (x_2, y_2) is `atan2(y1-y2, x1-x2)`. Note that it returns values in $(-\pi, \pi]$. In general though, when sorting by angle, try to use determinants to avoid precision errors.

The **projection** of $\vec{u}$ onto $\vec{v}$ is $\lambda\vec{v}$ for $\lambda = \frac{\vec{u}\cdot\vec{v}}{|\vec{v}|^2}$. When projecting on a segment, truncate λ to $[0, 1]$.

The **reflection** of $\vec{q}$ through the line $\vec{p_1} \to \vec{p_2}$ is $2\vec{q'} - \vec{q}$ where $\vec{q'}$ is the projection of $\vec{q}$ onto the line (see above).

To do a **line-circle intersection** project the center of the circle onto the line, and move left/right over the line using Pythagoras theorem.

The line intersecting two three-dimensional planes can be found by taking the cross product of their normal vectors. The three dimensional cross product is given as:

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} \times \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = \begin{bmatrix} y_1 z_2 - y_2 z_1 \\ z_1 x_2 - z_2 x_1 \\ x_1 y_2 - x_2 y_1 \end{bmatrix}$$

Given $\vec{p}, \vec{q}, \vec{r} \in \mathbb{R}^3$, the associated plane is defined by $\vec{x} \cdot \vec{n} = \vec{r} \cdot \vec{n}$ for $\vec{n} = (\vec{p} - \vec{r}) \times (\vec{q} - \vec{r})$.

7.2 2D Geometry Utilities

```cpp
using C = ld;   // could be long long or long double
constexpr C EPS = 1e-10;    // change to 0 for C=ll
struct P {      // may also be used as a 2D vector
    C x, y;
    P(C x = 0, C y = 0) : x(x), y(y) {}
    P operator+ (const P &p) const { return {x + p.x, y + p.y}; }
    P operator- (const P &p) const { return {x - p.x, y - p.y}; }
    P operator* (C c) const { return {x * c, y * c}; }
    P operator/ (C c) const { return {x / c, y / c}; }
    C operator* (const P &p) const { return x*p.x + y*p.y; }
    C operator^ (const P &p) const { return x*p.y - p.x*y; }
    P perp() const { return P{y, -x}; }
    C lensq() const { return x*x + y*y; }
    ld len() const { return sqrt((ld)lensq()); }
    static ld dist(const P &p1, const P &p2) {
        return (p1-p2).len(); }
    bool operator==(const P &r) const {
        return ((*this)-r).lensq() <= EPS*EPS; }
};
C det(P p1, P p2) { return p1^p2; }
C det(P p1, P p2, P o) { return det(p1-o, p2-o); }
C det(const vector<P> &ps) {
    C sum = 0; P prev = ps.back();
    for(auto &p : ps) sum += det(p, prev), prev = p;
    return sum;
}
// Careful with division by two and C=ll
C area(P p1, P p2, P p3) { return abs(det(p1, p2, p3))/C(2); }
C area(const vector<P> &poly) { return abs(det(poly))/C(2); }
int sign(C c){ return (c > C(0)) - (c < C(0)); }
int ccw(P p1, P p2, P o) { return sign(det(p1, p2, o)); }

// Only well defined for C = ld.
P unit(const P &p) { return p / p.len(); }
P rotate(P p, ld a) { return P{p.x*cos(a)-p.y*sin(a), p.x*sin(a)+p.y*cos(a)
    }; }
```

7.3 Formulae II

$$[ABC] = rs = \frac{1}{2}ab\sin\gamma = \frac{abc}{4R} = \sqrt{s(s-a)(s-b)(s-c)} = \frac{1}{2}\left|(B - A, C - A)^T\right|$$

$$s = \frac{a+b+c}{2} \qquad 2R = \frac{a}{\sin\alpha}$$

cosine rule: $$c^2 = a^2 + b^2 - 2ab\cos\gamma$$

Euler: $$1 + CC = V - E + F$$

Pick: $$\text{Area} = \text{interior points} + \frac{\text{boundary points}}{2} - 1$$

$$p \cdot q = |p||q|\cos(\theta) \qquad |p \times q| = |p||q|\sin(\theta)$$

Given a non-self-intersecting closed polygon on n vertices, given as (x_i, y_i), its centroid (C_x, C_y) is given as:

$$C_x = \frac{1}{6A}\sum_{i=0}^{n-1}(x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i), \quad C_y = \frac{1}{6A}\sum_{i=0}^{n-1}(y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$

$$A = \frac{1}{2}\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1} y_i) = \text{polygon area}$$

7.4 Planar Rotation

```cpp
#include "../geometry/essentials.cpp"
int quad(P p) {
    if (p.x >= 0 && p.y >= 0) return 0;
    if (p.x <= 0 && p.y >= 0) return 1;
    if (p.x <= 0 && p.y <= 0) return 2;
    if (p.x >= 0 && p.y <= 0) return 3;
    __builtin_unreachable();
}

struct pr { size_t i, j; };
void rotate(const vector<P> &pts) {
    size_t n = pts.size();
    vector<pr> prs;
    for (size_t i = 0; i < n; ++i)
        for (size_t j = 0; j < n; ++j)
            if (i != j) prs.push_back(pr{i, j});

    sort(prs.begin(), prs.end(), [&pts](const pr &l, const pr &r) {
        if (quad(pts[l.j]-pts[l.i]) != quad(pts[r.j]-pts[r.i]))
            return quad(pts[l.j]-pts[l.i]) < quad(pts[r.j]-pts[r.i]);
        else
            return ccw(pts[l.j]-pts[l.i], pts[r.j]-pts[r.i], P{0, 0}) > 0;
    });

    vector<size_t> vs(n), vsi(n);
    for (size_t i = 0; i < vs.size(); ++i) vs[i] = i;
    sort(vs.begin(), vs.end(), [&pts](const size_t &i, const size_t &j) {
        return pts[i].y != pts[j].y ? pts[i].y<pts[j].y : pts[i].x<pts[j].x;
    });
    for (size_t i = 0; i < vs.size(); ++i) vsi[vs[i]] = i;
    for (pr p : prs) {
        // Consider the line through pts[p.i] and pts[p.j]. All points
        // are currently sorted along the projection onto the normal.
        // Furthermore, these points are located next to each other at
        // positions vsi[p.i] and vsi[p.j].
        swap(vs[vsi[p.i]], vs[vsi[p.j]]);
        swap(vsi[p.i], vsi[p.j]);
    }
}
```

7.5 Convex Hull

```cpp
#include "essentials.cpp"
struct ConvexHull {         // O(n lg n) monotone chain.
    size_t n;
    vector<size_t> h, c;    // Indices of the hull are in ‘h‘, ccw.
    const vector<P> &p;
    ConvexHull(const vector<P> &_p) : n(_p.size()), c(n), p(_p) {
        std::iota(c.begin(), c.end(), 0);
        std::sort(c.begin(), c.end(), [this](size_t l, size_t r) -> bool {
            return p[l].x != p[r].x ? p[l].x < p[r].x : p[l].y < p[r].y; });
        c.erase(std::unique(c.begin(), c.end(), [this](size_t l, size_t r) {
            return p[l] == p[r]; }), c.end());
        for (size_t s = 1, r = 0; r < 2; ++r, s = h.size()) {
            for (size_t i : c) {
                while (h.size() > s
                        && ccw(p[h.end()[-2]], p[h.end()[-1]], p[i]) <= 0)
                    h.pop_back();
                h.push_back(i);
            }
            reverse(c.begin(), c.end());
        }
        if (h.size() > 1) h.pop_back();
    }
    size_t size() const { return h.size(); }
    template <class T, void U(const P&, const P&, const P&, T&)>
    void rotating_calipers(T &ans) {
        if (size() <= 2) U(p[h[0]], p[h.back()], p[h.back()], ans); else
        for (size_t i = 0, j = 1, s = size(); i < 2*s; ++i) {
            while (det(p[h[(i+1)%s]]-p[h[i%s]], p[h[(j+1)%s]]-p[h[j]]) >= 0)
                j = (j+1)%s;
            U(p[h[i%s]], p[h[(i+1)%s]], p[h[j]], ans);
        }
    }
};
// Example: furthest pair of points. Now set ans = 0LL and call
// ConvexHull(pts).rotating_calipers<ll, update>(ans);
void update(const P &p1, const P &p2, const P &o, ll &ans) {
    ans = max(ans, max((p1-o).lensq(), (p2-o).lensq()));
}
```

7.6 Convex Polygon

```cpp
bool lexo_yx(const P &l, const P &r) {
    return l.y != r.y ? l.y < r.y : l.x < r.x;
}
int quad(P p) {
    if (p.x > 0 && p.y >= 0) return 0;  // This ‘quad‘ is different from
    if (p.y > 0 && p.x <= 0) return 1;  // the one in planar-rotation.cpp,
    if (p.x < 0 && p.y <= 0) return 2;  // make sure to not mix the two.
    if (p.y < 0 && p.x >= 0) return 3;
    return 0; // p.x = 0 \land p.y = 0, go with 0 arbitrarily.
}
struct ConvexPolygon {
    // 21 Polygon is cut up into quadrants, where quadrant
    // 30 i is given by the segment [s[i], s[i+1])
    size_t s[5], N;
    vector<P> Ps;

    // Input a non-degenerate convex polygon. All vertices must have
    // an internal angle that is strictly less than pi.
    ConvexPolygon(const vector<P> &_Ps) : N(_Ps.size()), Ps(_Ps) {
        assert(N >= 3); size_t l = 0;
        for (size_t j = 1; j < N; ++j) l = (lexo_yx(Ps[l], Ps[j]) ? l : j);
        if (l != 0) std::rotate(Ps.begin(), Ps.begin()+l, Ps.end());
        Ps.resize(Ps.size()+3);
        std::copy(Ps.begin(), Ps.begin()+3, Ps.begin()+N);
        for (size_t i = 0; i < N; ++i) Ps.push_back(Ps[i]);
        s[0] = 0;
        for (int i = 1; i <= 4; ++i) {
            s[i] = s[i-1];
            while (s[i] < N && quad(Ps[s[i]+1] - Ps[s[i]]) < i) ++s[i];
        }
    }

    // Runs in O(lg n). Returns 1 for inside, 0 for border, -1 for outside.
    int contains(const P &p) const {    // Tested on ICPC WF J 2016
        if (p.y < Ps[s[0]].y || Ps[s[2]].y < p.y) return -1;
        {   // right hull
            size_t l = s[0], r = s[2];
            while (l < r) {
                size_t m = (l + r) / 2;
                if (Ps[m+1].y < p.y) l = m + 1; else r = m;
            }
            int side = ccw(Ps[l+1], p, Ps[l]);
            if (side < 0) return -1;
            if (side == 0) {
                if (max(Ps[l].x, Ps[l+1].x) <  p.x) return -1;
                if (min(Ps[l].x, Ps[l+1].x) <= p.x) return 0;
            }
        }{  // left hull
            size_t l = s[2], r = s[4];
            while (l < r) {
                size_t m = (l + r) / 2;
                if (Ps[m+1].y > p.y) l = m + 1; else r = m;
            }
            int side = ccw(Ps[l+1], p, Ps[l]);
            if (side < 0) return -1;
            if (side == 0) {
                if (min(Ps[l].x, Ps[l+1].x) >  p.x) return -1;
                if (max(Ps[l].x, Ps[l+1].x) >= p.x) return 0;
            }
        }
        return 1;
    }

    // Returns some i such that Ps[i] * v is maximal.
    size_t extreme(const P &v) const {  // ONLY TESTED LOCALLY
        size_t l, r;
        if (v.x > 0 || (v.x == 0LL && v.y < 0LL))
                l = s[0], r = s[2];
        else    l = s[2], r = s[4];
        while (l < r) {
            size_t m = (l + r) / 2;
            if ((Ps[m+1] - Ps[m]) * v <= 0LL)
                    r = m;
            else    l = m + 1;
        }
        l %= N;
        // When there are two possible outputs, l is one of them. If l is
        // the 'largest' of the two, then this code moves to the 'smallest',
        // if it is not l (then it must be l-1). Only keep if important.
        if (Ps[(N+(l-1))%N] * v >= Ps[l] * v) l = (N+(l-1))%N;
        return l;
    }

    // Returns i such that p -> Ps[i] is right-tangent to Ps, cq j left.
    std::pair<size_t, size_t> tangent(const P &p) const {
//      Tested on ICPC WF J 2016
//      assert(this->contains(p) < 0);
        size_t rt, lt;
        {   // right tangent
            size_t l, r;
            if (p.x < Ps[s[1]].x && (p.y <= Ps[s[0]].y
                    || ccw(Ps[s[1]], p, Ps[s[0]]) < 0LL))
                l = s[0], r = s[1]; else
            if (p.y < Ps[s[2]].y && (p.x >= Ps[s[1]].x
                    || ccw(Ps[s[2]], p, Ps[s[1]]) < 0LL))
                l = s[1], r = s[2]; else
            if (p.x > Ps[s[3]].x && (p.y >= Ps[s[2]].y
                    || ccw(Ps[s[3]], p, Ps[s[2]]) < 0LL))
                l = s[2], r = s[3]; else
            if (p.y > Ps[s[4]].y && (p.x <= Ps[s[3]].x
                    || ccw(Ps[s[4]], p, Ps[s[3]]) < 0LL))
                l = s[3], r = s[4]; else assert(false);
            while (l < r) {
                size_t m = (l + r) / 2;
                if ((Ps[m+1] - Ps[m]) * (p - Ps[m]) >= 0LL
                            || ccw(Ps[m], Ps[m+1], p) < 0)
                        l = m + 1;
                else    r = m;
            }
            rt = l % N;
        }{  // left tangent
            size_t l, r;
            if (p.y >= Ps[s[0]].y && (p.x > Ps[s[1]].x
                    || ccw(Ps[s[1]], p, Ps[s[0]]) < 0LL))
                l = s[0], r = s[1]; else
            if (p.x <= Ps[s[1]].x && (p.y > Ps[s[2]].y
                    || ccw(Ps[s[2]], p, Ps[s[1]]) < 0LL))
                l = s[1], r = s[2]; else
            if (p.y <= Ps[s[2]].y && (p.x < Ps[s[3]].x
                    || ccw(Ps[s[3]], p, Ps[s[2]]) < 0LL))
                l = s[2], r = s[3]; else
            if (p.x >= Ps[s[3]].x && (p.y < Ps[s[4]].y
                    || ccw(Ps[s[4]], p, Ps[s[3]]) < 0LL))
                l = s[3], r = s[4]; else assert(false);
            while (l < r) {
                size_t m = (l + r + 1) / 2;
                if ((Ps[m-1] - Ps[m]) * (p - Ps[m]) >= 0LL
                            || ccw(Ps[m], Ps[m-1], p) > 0)
                        r = m - 1;
                else    l = m;
            }
            lt = l % N;
        }
        return {rt, lt};
    }
    P operator[](int i) const { return Ps[i%N]; }
    P &operator[](int i) { return Ps[i%N]; }
};
```

7.7 Halfspace intersection

Intersection is hard, but testing if the intersection is not empty is easy. Divide the halfplanes into a lower and upper envelope, and consider the associated convex hull structures U and L. Now the function $x \mapsto U(x) - L(x)$ is concave, so we can ternary search for a maximum, which is positive if and only if the halfplanes have a non-empty intersection. Runs in $O(n \log C)$ time.

7.8 Delaunay and Voronoi

The Delaunay triangulation D of a point set S is a triangulation of S such that the interior of the circumcircle of any triangle does not contain any other points of S. Useful properties:

- D contains only $O(n)$ simplices.
- The *nearest neighbour graph* is a subset of the Delaunay triangulation.
- The dual graph of a Delaunay triangulation is a Voronoi diagram (find the centers of the circumcircles, and connect those of adjacent faces with an edge).

Computation: see *Parabolic lifting map*.

7.9 Parabolic lifting map

The parabolic lifting map is the map $p \mapsto p^+, (x, y) \mapsto (x, y, x^2 + y^2)$. This embedding has several useful properties:

- For a point set S let convS^+ be the convex hull of its parabolic lift. A face f of this convex hull is *downward-facing* if no other point in convS^+ is directly below f (w.r.t. the z-axis). Projecting these faces back into $\mathbb{R}^2$ (by discarding all z-coordinates) yields a Delaunay triangulation of S.
- Two sets of points in $\mathbb{R}^2$ can be separated by a circle iff their lifts in $\mathbb{R}^3$ can be separated by a plane (this plane corresponds exactly to the circle).
- More generally, testing whether a point p is in a circle C is equivalent to testing whether p^+ isconvexhull below the induced plane C^+.

8 Strings

8.1 Z-Algorithm

```
void Z_algorithm(const string &s, vi &Z) {
    Z.assign(s.length(), -1);
    int L = 0, R = 0, n = s.length();
    for (int i = 1; i < n; ++i) {
        if (i > R) {
            L = R = i;
            while (R < n && s[R - L] == s[R]) R++;
            Z[i] = R - L; R--;
        } else if (Z[i - L] >= R - i + 1) {
            L = i;
            while (R < n && s[R - L] == s[R]) R++;
            Z[i] = R - L; R--;
        } else Z[i] = Z[i - L];
    }
}
```

8.2 KMP

```
void compute_prefix_function(string &w, vi &pi) {
    pi.assign(w.length(), 0);
    int k = pi[0] = -1;

    for(int i = 1; i < w.length(); ++i) {
        while(k >= 0 && w[k + 1] != w[i]) k = pi[k];
        if(w[k + 1] == w[i]) k++;
        pi[i] = k;
    }
}

void knuth_morris_pratt(string &s, string &w) {
    int q = -1;
    vi pi;
    compute_prefix_function(w, pi);
    for(int i = 0; i < s.length(); ++i) {
        while(q >= 0 && w[q + 1] != s[i]) q = pi[q];
        if(w[q + 1] == s[i]) q++;
        if(q + 1 == w.length()) {
            // Match at position (i - w.length() + 1)
            q = pi[q];
        }
    }
}
```

8.3 Aho-Corasick

```
template <int ALPHABET_SIZE, int (*mp)(char)>
struct AC_FSM {
    struct Node {
        int child[ALPHABET_SIZE], failure = 0, match_par = -1;
        vi match;
        Node() { for (int i = 0; i < ALPHABET_SIZE; ++i) child[i] = -1; }
    };
```

```cpp
    vector<Node> a;
    vector<string> &words;
    AC_FSM(vector<string> &words) : words(words) {
        a.push_back(Node());
        construct_automaton();
    }
    void construct_automaton() {
        for (int w = 0, n = 0; w < words.size(); ++w, n = 0) {
            for (int i = 0; i < words[w].size(); ++i) {
                if (a[n].child[mp(words[w][i])] == -1) {
                    a[n].child[mp(words[w][i])] = a.size();
                    a.push_back(Node());
                }
                n = a[n].child[mp(words[w][i])];
            }
            a[n].match.push_back(w);
        }
        queue<int> q;
        for (int k = 0; k < ALPHABET_SIZE; ++k) {
            if (a[0].child[k] == -1) a[0].child[k] = 0;
            else if (a[0].child[k] > 0) {
                a[a[0].child[k]].failure = 0;
                q.push(a[0].child[k]);
            }
        }
        while (!q.empty()) {
            int r = q.front(); q.pop();
            for (int k = 0, arck; k < ALPHABET_SIZE; ++k) {
                if ((arck = a[r].child[k]) != -1) {
                    q.push(arck);
                    int v = a[r].failure;
                    while (a[v].child[k] == -1) v = a[v].failure;
                    a[arck].failure = a[v].child[k];
                    a[arck].match_par = a[v].child[k];
                    while (a[arck].match_par != -1
                            && a[a[arck].match_par].match.empty())
                        a[arck].match_par = a[a[arck].match_par].match_par;
                }
            }
        }
    }
    void aho_corasick(string &sentence, vvi &matches){
        matches.assign(words.size(), vi());
        int state = 0, ss = 0;
        for (int i = 0; i < sentence.length(); ++i, ss = state) {
            while (a[ss].child[mp(sentence[i])] == -1)
                ss = a[ss].failure;
            state = a[state].child[mp(sentence[i])]
                = a[ss].child[mp(sentence[i])];
            for (ss = state; ss != -1; ss = a[ss].match_par)
                for (int w : a[ss].match)
                    matches[w].push_back(i + 1 - words[w].length());
        }
    }
};
```

8.4 Manacher's Algorithm

```cpp
void manacher(string &s, vi &pal) {
    int n = s.length(), i = 1, l, r;
    pal.assign(2 * n + 1, 0);
    while (i < 2 * n + 1) {
        if ((i&1) && pal[i] == 0) pal[i] = 1;
        l = i / 2 - pal[i] / 2; r = (i-1) / 2 + pal[i] / 2;

        while (l - 1 >= 0 && r + 1 < n && s[l - 1] == s[r + 1])
            --l, ++r, pal[i] += 2;

        for (l = i - 1, r = i + 1; l >= 0 && r < 2 * n + 1; --l, ++r) {
            if (l <= i - pal[i]) break;
            if (l / 2 - pal[l] / 2 > i / 2 - pal[i] / 2)
                pal[r] = pal[l];
            else {  if (l >= 0)
                    pal[r] = min(pal[l], i + pal[i] - r);
                break;
            }
        }
        i = r;
}   }
```

9 Nondeterminism

9.1 Hashing

Possibly `rand()` draws from a small range, verify by checking `RAND_MAX`. Otherwise use mt19937 (see language documentation).

For a proper rolling hash over a string, fix the modulus, and draw the base b uniformly at random from $\{0, 1, \ldots, p-1\}$. Note that when comparing rolling hashes of strings of different lengths, it is useful to hash the empty character to 0, and hash all actual characters to nonzero values.

Some primes:

$$10^3 + \{-9, -3, 9, 13\}, \quad 10^6 + \{-17, 3, 33\}, \quad 10^9 + \{7, 9, 21, 33, 87\}$$

9.2 Timing

```cpp
auto now(){ return chrono::high_resolution_clock::now(); }
using TP = decltype(now()); // time point
auto duration(TP t1, TP t2){
    return chrono::duration_cast<chrono::microseconds>(t2-t1).count();
}
```