



Jagiellonian University in Krakow

# Jagiellonian teapots

Antoni Długosz, Kacper Paciorek, Kacper Topolski

2024-08-08

1 Contest

2 Mathematics

3 Data structures

4 Numerical

5 Number theory

6 Combinatorial

7 Graph

8 Geometry

9 Strings

10Various

Contest (1)

```
tldr
bits/stdc++.h, namespace std
fwd, rep, all, sz, st, nd, pii, vi, ll
#ifdef LOC
auto SS = signal(6, [](int) { *(int *)0 = 0; }); //
better stacktrace
auto &operator<<(auto &out, pair<auto, auto> a) //
print pair
auto &operator<<(auto &out, auto a) // print collection
void dump(auto... x) { ( ( cerr << x << " , " ) , ... )
<< '\n'; }
#define debug(x...) cerr << "[" #x "]: ", dump(x)
#else
#define debug(...) 0
#endif
int32_t main() ios_base::sync_with_stdio(0), cin.tie(0)
mt19937_64 gen(seed); uniform_int_distribution<T> distr
(a, b);
auto my_rand = bind(distr, gen); // my_rand() -> x \in
[a, b]
#pragma GCC optimize("Ofast,unroll-loops")
#pragma GCC target("popcnt,avx,tune=native")
//#pragma GCC optimize("trapv") kills the program on
integer overflows
// bitset features: _Find_first(), _Find_next(i) (finds
AFTER i, not incl)
// c = x&x, r = x+c; (((r^x) >> 2)/c) | r = next
number after x with same popcnt
#define each(a, x) for (auto &a : (x))
#define x / y, first / second
#define mp make_pair
```

```
.bashrc
compile() {
  g++ -std=c++20 -Wall -Wfatal-errors -Wconversion -
  DLOC -fsanitize=address,undefined -g -o$1 $1.
  cpp
}
libhash() {
  cat $1.cpp | cpp -dD -P -fpreprocessed | tr -d '[:
  space:]' | md5sum |cut -c-6
}
```

Mathematics (2)

2.1 Equations

$$\begin{aligned} ax + by &= e & x &= \frac{ed - bf}{ad - bc} \\ cx + dy &= f & y &= \frac{af - ec}{ad - bc} \end{aligned}$$

1 In general, given an equation  $Ax = b$ , the solution to a variable  $x_i$  is given by

1

$$x_i = \frac{\det A'_i}{\det A}$$

1

3 where  $A'_i$  is  $A$  with the  $i$ 'th column replaced by  $b$ .

6

2.2 Trigonometry

7

$$\begin{aligned} \sin(v \pm w) &= \sin v \cos w \pm \cos v \sin w & \cos(v \pm w) &= \cos v \cos w \mp \sin v \sin w \\ \sin v + \sin w &= 2 \sin \frac{v+w}{2} \cos \frac{v-w}{2} & \cos v + \cos w &= 2 \cos \frac{v+w}{2} \cos \frac{v-w}{2} \\ |\sin \frac{x}{2}| &= \sqrt{\frac{1 - \cos x}{2}} & |\cos \frac{x}{2}| &= \sqrt{\frac{1 + \cos x}{2}} \\ \tan(v \pm w) &= \frac{\tan v \pm \tan w}{1 \mp \tan v \tan w} & |\tan \frac{x}{2}| &= \sqrt{\frac{1 - \cos x}{1 + \cos x}} \end{aligned}$$

15

17

$$(V + W) \tan(v - w)/2 = (V - W) \tan(v + w)/2$$

where  $V, W$  are lengths of sides opp angles  $v, w$

$$\begin{aligned} a \cos x + b \sin x &= r \cos(x - \phi) \\ a \sin x + b \cos x &= r \sin(x + \phi) \end{aligned}$$

where  $r = \sqrt{a^2 + b^2}, \phi = \text{atan2}(b, a)$ .

2.3 Geometry

2.3.1 Triangles

$$\begin{aligned} \text{Side lengths: } a, b, c \quad \text{Semiperimeter: } p &= \frac{a + b + c}{2} \\ \text{Area: } A &= \sqrt{p(p - a)(p - b)(p - c)} \\ \text{Inradius: } r &= \frac{A}{p} \quad \text{Circumradius: } R = \frac{abc}{4A} \end{aligned}$$

Length of median (divides triangle into two equal-area triangles):  $m_a = \frac{1}{2}\sqrt{2b^2 + 2c^2 - a^2}$   
Length of bisector (divides angles in two):

$$s_a = \sqrt{bc \left[ 1 - \left( \frac{a}{b + c} \right)^2 \right]}$$

Law of sines:  $\frac{\sin \alpha}{a} = \frac{\sin \beta}{b} = \frac{\sin \gamma}{c} = \frac{1}{2R}$   
Law of cosines:  $a^2 = b^2 + c^2 - 2bc \cos \alpha$

Law of tangents:  $\frac{a + b}{a - b} = \frac{\tan \frac{\alpha + \beta}{2}}{\tan \frac{\alpha - \beta}{2}}$

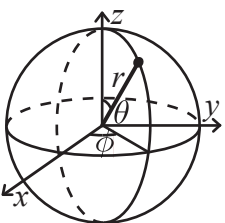
2.3.2 Quadrilaterals

With side lengths  $a, b, c, d$ , diagonals  $e, f$ , diagonals angle  $\theta$ , area  $A$  and magic flux  $F = b^2 + d^2 - a^2 - c^2$ :

$$4A = 2ef \cdot \sin \theta = F \tan \theta = \sqrt{4e^2 f^2 - F^2}$$

For cyclic quadrilaterals  $ef = ac + bd$ , and  $A = \sqrt{(p - a)(p - b)(p - c)(p - d)}$ .

2.3.3 Spherical coordinates



$$\begin{aligned} x &= r \sin \theta \cos \phi \\ y &= r \sin \theta \sin \phi \\ z &= r \cos \theta \\ r &= \sqrt{x^2 + y^2 + z^2} \\ \theta &= \text{acos}(z / \sqrt{x^2 + y^2 + z^2}) \\ \phi &= \text{atan2}(y, x) \end{aligned}$$

2.4 Derivatives/Integrals

$$\begin{aligned} \int \sqrt{a^2 + x^2} dx &= \frac{x}{2} \sqrt{a^2 + x^2} + \frac{a^2}{2} \ln(x + \sqrt{a^2 + x^2}) & \int \sqrt{a^2 - x^2} dx &= \frac{x}{2} \sqrt{a^2 - x^2} + \frac{a^2}{2} \arcsin \frac{x}{|a|} \\ \int \frac{dx}{\sqrt{a^2 - x^2}} &= \arcsin \frac{x}{|a|} = -\arccos \frac{x}{|a|} & \int \frac{dx}{\sqrt{a^2 + x^2}} &= \ln(x + \sqrt{a^2 + x^2}) \\ \text{Sub } s &= \tan(x/2) \text{ to get: } dx = \frac{2s}{1 + s^2}, \quad \sin x = \frac{2s}{1 + s^2}, \quad \cos x = \frac{1 - s^2}{1 + s^2} \\ \int_a^b f(x)g(x)dx &= [F(x)g(x)]_a^b - \int_a^b F(x)g'(x)dx & (\text{Integration by parts}) \\ \int \tan ax &= -\frac{\ln |\cos ax|}{a} & \int x \sin ax &= \frac{\sin ax - ax \cos ax}{a^2} \\ \int e^{-x^2} &= \frac{\sqrt{\pi}}{2} \text{erf}(x), \quad \int xe^{ax} dx = \frac{e^{ax}}{a^2}(ax - 1) & \frac{d}{dx} \tan x &= 1 + \tan^2 x, \quad \frac{d}{dx} \arctan x = \frac{1}{1 + x^2} \\ \text{Curve length: } \int_a^b \sqrt{1 + (f'(x))^2} dx & & \text{When } X(t), Y(t) : \int_a^b \sqrt{(X'(t))^2 + (Y'(t))^2} dt & \\ \text{Solid of revolution vol: } \pi \int_a^b (f(x))^2 dx & & \text{Surface area: } 2\pi \int_a^b |f(x)| \sqrt{1 + (f'(x))^2} dx & \end{aligned}$$

2.5 Sums

$$\begin{aligned} 1^2 + 2^2 + 3^2 + \dots + n^2 &= \frac{n(2n + 1)(n + 1)}{6} \\ 1^3 + 2^3 + 3^3 + \dots + n^3 &= \frac{n^2(n + 1)^2}{4} \\ 1^4 + 2^4 + 3^4 + \dots + n^4 &= \frac{n(n + 1)(2n + 1)(3n^2 + 3n - 1)}{30} \end{aligned}$$

2.6 Series

$$\begin{aligned} \ln(1 + x) &= x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \dots, \quad (-1 < x \leq 1) \\ \sqrt{1 + x} &= 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{2x^3}{32} - \frac{5x^4}{128} + \dots, \quad (-1 \leq x \leq 1) \end{aligned}$$

2.6.1 Discrete distributions

Binomial distribution

The number of successes in  $n$  independent yes/no experiments, each which yields success with probability  $p$  is  $\text{Bin}(n, p)$ ,  $n = 1, 2, \dots$ ,  $0 \leq p \leq 1$ .

$$\begin{aligned} p(k) &= \binom{n}{k} p^k (1 - p)^{n - k} \\ \mu &= np, \sigma^2 = np(1 - p) \\ \text{Bin}(n, p) &\text{ is approximately } \text{Po}(np) \text{ for small } p. \end{aligned}$$

First success distribution

The number of trials needed to get the first success in independent yes/no experiments, each wich yields success with probability  $p$  is  $\text{Fs}(p)$ ,  $0 \leq p \leq 1$ .

$$\begin{aligned} p(k) &= p(1 - p)^{k - 1}, \quad k = 1, 2, \dots \\ \mu &= \frac{1}{p}, \sigma^2 = \frac{1 - p}{p^2} \end{aligned}$$

Poisson distribution

The number of events occurring in a fixed period of time  $t$  if these events occur with a known average rate  $\kappa$  and independently of the time since the last event is  $\text{Po}(\lambda)$ ,  $\lambda = t\kappa$ .

$$p(k) = e^{-\lambda} \frac{\lambda^k}{k!}, k = 0, 1, 2, \dots$$

$$\mu = \lambda, \sigma^2 = \lambda$$

2.6.2 Continuous distributions

Uniform distribution

If the probability density function is constant between  $a$  and  $b$  and 0 elsewhere it is  $\text{U}(a, b)$ ,  $a < b$ .

$$\begin{aligned} f(x) &= \begin{cases} \frac{1}{b - a} & a < x < b \\ 0 & \text{otherwise} \end{cases} \\ \mu &= \frac{a + b}{2}, \sigma^2 = \frac{(b - a)^2}{12} \end{aligned}$$

Exponential distribution

The time between events in a Poisson process is  $\text{Exp}(\lambda)$ ,  $\lambda > 0$ .

$$\begin{aligned} f(x) &= \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases} \\ \mu &= \frac{1}{\lambda}, \sigma^2 = \frac{1}{\lambda^2} \end{aligned}$$

Normal distribution

Most real random values with mean  $\mu$  and variance  $\sigma^2$  are well described by  $\mathcal{N}(\mu, \sigma^2)$ ,  $\sigma > 0$ .

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x - \mu)^2}{2\sigma^2}}$$

If  $X_1 \sim \mathcal{N}(\mu_1, \sigma_1^2)$  and  $X_2 \sim \mathcal{N}(\mu_2, \sigma_2^2)$  then

$$aX_1 + bX_2 + c \sim \mathcal{N}(\mu_1 + \mu_2 + c, a^2\sigma_1^2 + b^2\sigma_2^2)$$

2.7 Markov chains

A *Markov chain* is a discrete random process with the property that the next state depends only on the current state. Let  $X_1, X_2, \dots$  be a sequence of random variables generated by the Markov process. Then there is a transition matrix  $\mathbf{P} = (p_{ij})$ , with  $p_{ij} = \text{Pr}(X_n = i | X_{n - 1} = j)$ , and  $\mathbf{p}^{(n)} = \mathbf{P}^n \mathbf{p}^{(0)}$  is the probability distribution for  $X_n$  (i.e.,  $p_i^{(n)} = \text{Pr}(X_n = i)$ ), where  $\mathbf{p}^{(0)}$  is the initial distribution.

$\pi$  is a stationary distribution if  $\pi = \pi \mathbf{P}$ . If the Markov chain is *irreducible* (it is possible to get to any state from any state), then  $\pi_i = \frac{1}{\mathbb{E}(T_i)}$  where  $\mathbb{E}(T_i)$  is the expected time between two visits in state  $i$ .  $\pi_j / \pi_i$  is the expected number of visits in state  $j$  between two visits in state  $i$ .

For a connected, undirected and non-bipartite graph, where the transition probability is uniform among all neighbors,  $\pi_i$  is proportional to node  $i$ 's degree.

A Markov chain is *ergodic* if the asymptotic distribution is independent of the initial distribution. A finite Markov chain is ergodic iff it is irreducible and *aperiodic* (i.e., the gcd of cycle lengths is 1).  $\lim_{k \rightarrow \infty} \mathbf{P}^k = \mathbf{1}\pi$ .

A Markov chain is an **A-chain** if the states can be partitioned into two sets **A** and **G**, such that all states in **A** are absorbing ( $p_{ii} = 1$ ), and all states in **G** leads to an absorbing state in **A**. The probability for absorption in state  $i \in \mathbf{A}$ , when the initial state is  $j$ , is  $a_{ij} = p_{ij} + \sum_{k \in \mathbf{G}} a_{ik} p_{kj}$ . The expected time until absorption, when the initial state is  $i$ , is  $t_i = 1 + \sum_{k \in \mathbf{G}} p_{ki} t_k$ .

Data structures (3)

## PBDS.h

**Description:** Policy Based Data Structures 460200, 17 lines

```
// Order Statistics Tree: Caution: Not a multiset!
#include <bits/extc++.h>
using namespace __gnu_pbds;
template <class T> using Tree = tree<T, null_type, less
    <T>, rb_tree_tag,
    tree_order_statistics_node_update>;
Tree<int> t, t2;
auto it = t.insert(10).first; // it == t.upper_bound(9)
;
t.order_of_key(10); // # of entries strictly
    smaller than key
t.join(t2); // fast only if max(T) <
    min(T2) or min(T) > max(T2)
// Hash Table: faster but can lead to MLE (1.5x worse
    performance), initial capacity must = 2^k
struct chash { // large odd number for C
    const uint64_t C = 1l(4e18 * acos(0)) | 7l;
    ll operator (l) (ll x) const { return __builtin_bswap64
        (x * C); }
};
gp_hash_table<ll, int, chash> h({}, {}, {}, {}, {}, {1 <<
    16}); // cc_hash_table also exists if needed
```

## PersistentSegmentTreePointUpdate.h

**Description:** sparse (N can be up to 1e18) persistent segment tree supporting point updates and range queries. Ranges are inclusive

**Time:**  $\mathcal{O}(\log N)$  7927d0, 49 lines

```
struct PSegmentTree { // default: update set_pos, query
    sum
    typedef int val;
    val idnt = 0; // identity value
    val f(val l, val r) {
        return l + r; // implement this!
    }
    struct node {
        int l = 0, r = 0;
        val x;
        node(val x) : x(x) {
        }
    };
    int N;
    vector<node> t;
    PSegmentTree(int N) : N(N) {
        t.push_back(node(idnt)); // 0th node is the root of an
            empty tree
        // t.reserve() in case of memory issues
    }
    int cpy(int v) {
        t.push_back(t[v]);
        return sz(t) - 1;
    }
    // creates lgN +- eps new nodes
    int upd(int v, int p, val x, int a = 0, int b = -1) {
        b = ~b ? b : N - 1;
        int u = cpy(v);
        if (a == b) {
            t[u].x = x; // change something here if not
                swapping values
            return u;
        }
        int c = (a + b) / 2;
        if (p <= c)
            t[u].l = upd(t[v].l, p, x, a, c);
        else
            t[u].r = upd(t[v].r, p, x, c + 1, b);
        t[u].x = f(t[t[u].l].x, t[t[u].r].x);
        return u;
    }
    // doesn't create new nodes
    val get(int v, int l, int r, int a = 0, int b = -1) {
        b = ~b ? b : N - 1;
        if (!v || l > b || r < a)
            return idnt;
        if (a >= l && b <= r)
            return t[v].x;
        int c = (a + b) / 2;
        return f(get(t[v].l, l, r, a, c), get(t[v].r, l, r,
            c + 1, b));
    }
};
```

## UnionFindRollback.h

**Description:** Disjoint-set data structure with undo. If undo is not needed, skip st, time() and rollback().

**Usage:** int t = uf.time(); ...; uf.rollback(t);  
**Time:**  $\mathcal{O}(\log(N))$  de4ad0, 21 lines

```
struct RollbackUF {
    vi e; vector<pii> st;
    RollbackUF(int n) : e(n, -1) {}
    int size(int x) { return -e[find(x)]; }
    int find(int x) { return e[x] < 0 ? x : find(e[x]); }
    int time() { return sz(st); }
    void rollback(int t) {
        for (int i = time(); i --> t;)
            e[st[i].first] = st[i].second;
        st.resize(t);
    }
    bool join(int a, int b) {
        a = find(a), b = find(b);
        if (a == b) return false;
        if (e[a] > e[b]) swap(a, b);
        st.push_back({a, e[a]});
        st.push_back({b, e[b]});
        e[a] += e[b]; e[b] = a;
        return true;
    }
};
```

## DequeRollback.h

**Description:** Deque-like undoing on data structures with amortized  $\mathcal{O}(\log n)$  overhead for operations. Maintains a deque of objects alongside a data structure that contains all of them. The data structure only needs to support insertions and undoing of last insertion using the following interface: - insert(...) - insert an object to DS - time() - returns current version number - rollback(t) - undo all operations after t Assumes time() == 0 for empty DS. 13639a, 38 lines

```
struct DequeUndo {
    // Argument for insert(...) method of DS.
    using T = tuple<int, int>;
    DataStructure ds; // Configure DS type here.
    vector<T> elems[2];
    vector<pii> his = {{0,0}};
    // Push object to front or back of deque, depending
        on side arg.
    void push(T val, bool side) {
        elems[side].pb(val);
        doPush(0, side);
    }
    // Pop object from front or back of deque, depending
        on side arg.
    void pop(int side) {
        auto &A = elems[side], &B = elems[!side];
        int cnt[2] = {};
        if (A.empty()) {
            assert(!B.empty());
            auto it = B.begin() + sz(B)/2 + 1;
            A.assign(B.begin(), it);
            B.erase(B.begin(), it);
            reverse(all(A)); his.resize(1);
            cnt[0] = sz(A); cnt[1] = sz(B);
        } else {
            do {
                cnt[his.back().y ^ side]++;
                his.pop_back();
            } while (cnt[0]*2 < cnt[1] && cnt[0] < sz(A)
                );
        }
        cnt[0]--; A.pop_back();
        ds.rollback(his.back().x);
        for (int i : {1, 0})
            while (cnt[i]) doPush(--cnt[i], i^side);
    }
    void doPush(int i, bool s) {
        apply([&](auto... x) { ds.insert(x...); }, elems[s].
            rbegin(i));
        his.pb({ds.time(), s});
    }
};
```

## LineContainer.h

**Description:** Container where you can add lines of the form  $kx+m$ , and query maximum values at points  $x$ . Useful for dynamic programming (“convex hull trick”).

**Time:**  $\mathcal{O}(\log N)$  8ec1c7, 30 lines

```
struct Line {
    mutable ll k, m, p;
```

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

## Treap.h

**Description:** A short self-balancing tree. It acts as a sequential container with log-time splits/joins, and is easy to augment with additional data.

**Time:**  $\mathcal{O}(\log N)$  cfcece5, 55 lines

```
struct Node {
    Node *l = 0, *r = 0;
    int val, y, c = 1;
    Node(int val) : val(val), y(rand()) {}
    void recalc();
};
int cnt(Node* n) { return n ? n->c : 0; }
void Node::recalc() { c = cnt(l) + cnt(r) + 1; }
pair<Node*, Node*> split(Node* n, int k) {
    if (!n) return {}; /* pushdown() for lazy if needed */
    if (cnt(n->l) >= k) { // "n->val >= k" for
        lower_bound(k)
        auto pa = split(n->l, k);
        n->l = pa.nd;
        n->recalc();
        return {pa.st, n};
    } else {
        auto pa = split(n->r, k - cnt(n->l) - 1); // and
            just "k"
        n->r = pa.st;
        n->recalc();
        return {n, pa.nd};
    }
}
Node* merge(Node* l, Node* r) {
    if (!l) return r;
    if (!r) return l; /* pushdown() */
    if (l->y > r->y) {
        l->r = merge(l->r, r);
        l->recalc();
        return l;
    } else {
        r->l = merge(l, r->l);
        r->recalc();
        return r;
    }
}
Node* ins(Node* t, Node* n, int pos) {
    auto pa = split(t, pos);
    return merge(merge(pa.st, n), pa.nd);
}
template<class F> void each(Node* n, F f) {
    if (n) { /*pushdown()*/ each(n->l, f); f(n->val);
        each(n->r, f); }
}
// Example application: move the range [l, r) to index
    k
void move(Node* k, int l, int r, int k) {
    Node a, *b, *c;
    tie(a,b) = split(t, l); tie(b,c) = split(b, r - 1);
    if (k <= l) t = merge(ins(a, b, k), c);
    else t = merge(a, ins(c, b, k - r));
}
```

## FenwickTree.h

**Description:** Computes partial sums  $a[0] + a[1] + \dots + a[\text{pos} - 1]$ , and updates single elements  $a[i]$ , taking the difference between the old and new value.

**Time:** Both operations are  $\mathcal{O}(\log N)$ . e62fac, 22 lines

```
struct FT {
    vector<ll> s;
    FT(int n) : s(n) {}
    void update(int pos, ll dif) { // a[pos] += dif
        for (; pos < sz(s); pos |= pos + 1) s[pos] += dif;
    }
    ll query(int pos) { // sum of values in [0, pos)
        ll res = 0;
        for (; pos > 0; pos &= pos - 1) res += s[pos-1];
        return res;
    }
    int lower_bound(ll sum) { // min pos st sum of [0, pos)
        // >= sum
        // Returns n if no sum is >= sum, or -1 if empty
        sum is.
        if (sum <= 0) return -1;
        int pos = 0;
        for (int pw = 1 << 25; pw; pw >= 1) {
            if (pos + pw <= sz(s) && s[pos + pw-1] < sum)
                pos += pw, sum -= s[pos-1];
        }
        return pos;
    }
};
```

## FenwickTree2d.h

**Description:** Computes sums  $a[i,j]$  for all  $i < I, j < J$ , and increases single elements  $a[i,j]$ . Requires that the elements to be updated are known in advance (call fakeUpdate() before init()).

**Time:**  $\mathcal{O}(\log^2 N)$ . (Use persistent segment trees for  $\mathcal{O}(\log N)$ .)

\*FenwickTree.h" 157f07, 22 lines

```
struct FT2 {
    vector<vi> ys; vector<FT> ft;
    FT2(int limx) : ys(limx) {}
    void fakeUpdate(int x, int y) {
        for (; x < sz(ys); x |= x + 1) ys[x].push_back(y);
    }
    void init() {
        for (vi& v : ys) sort(all(v)), ft.emplace_back(sz(v)
            );
    }
    int ind(int x, int y) {
        return (int)(lower_bound(all(ys[x]), y) - ys[x].
            begin()); }
    void update(int x, int y, ll dif) {
        for (; x < sz(ys); x |= x + 1)
            ft[x].update(ind(x, y), dif);
    }
    ll query(int x, int y) {
        ll sum = 0;
        for (; x; x &= x - 1)
            sum += ft[x-1].query(ind(x-1, y));
        return sum;
    }
};
```

## WaveletTree.h

**Description:** Wavelet tree. Supports fast kth order statistics on ranges (no updates).

**Time:**  $\mathcal{O}(\log N)$  276fc5, 35 lines

```
struct WaveletTree {
    vector<vi> seq, left;
    int len;
    WaveletTree() {}
    // time and space:  $\mathcal{O}((n+\text{maxVal}) \log \text{maxVal})$ 
    // Values are expected to be in  $[0;\text{maxVal})$ .
    WaveletTree(const vi& elems, int maxVal) {
        for (len = 1; len < maxVal; len *= 2);
        seq.resize(len*2); left.resize(len*2);
        seq[1] = elems; build(1, 0, len);
    }
    void build(int i, int b, int e) {
        if (i >= len) return;
        int m = (b+e) / 2;
        left[i].push_back(0);
        for(auto &x : seq[i]) {
            left[i].push_back(left[i].back() + (x < m));
            seq[i*2 + (x >= m)].push_back(x);
        }
    }
```

```
    build(i*2, b, m); build(i*2+1, m, e);
} // Find k-th (0 indexed) smallest element in [begin
    yend)
int kth(int begin, int end, int k, int i=1) {
    if (i >= len) return seq[i][0];
    int x = left[i][begin], y = left[i][end];
    if (k < y-x) return kth(x, y, k, i*2);
    return kth(begin-x, end-y, k-y+x, i*2+1);
} // Count number of elements >= vb and < ve
int count(int begin, int end, int vb, int ve, int i =
    1, int b = 0, int e = -1) {
    if (e < 0) e = len;
    if (b >= ve || vb >= e) return 0;
    if (b >= vb && e <= ve) return end-begin;
    int m = (b+e) / 2; int x = left[i][begin], y = left
        [i][end];
    return count(x, y, vb, ve, i*2, b, m) + count(begin
        -x, end-y, vb, ve, i*2+1,m,e);
}
};
```

## RMQ.h

**Description:** Range Minimum Queries on an array. Returns min(V[a], V[a + 1], ... V[b - 1]) in constant time.

**Usage:** RMQ rmq(values);

rmq.query(inclusive, exclusive);

**Time:**  $\mathcal{O}(|V| \log |V| + Q)$

4a9db2, 16 lines

```
template<class T>
struct RMQ {
    vector<vector<T>> jmp;
    RMQ(const vector<T>& V) : jmp(1, V) {
        for (int pw = 1, k = 1; pw * 2 <= sz(V); pw *= 2,
            ++k) {
            jmp.emplace_back(sz(V) - pw * 2 + 1);
            rep(j,sz(jmp[k]))
                jmp[k][j] = min(jmp[k - 1][j], jmp[k - 1][j +
                    pw]);
        }
    }
    T query(int a, int b) {
        assert(a < b); // or return inf if a == b
        int dep = 31 - __builtin_clz(b - a);
        return min(jmp[dep][a], jmp[dep][b - (1 << dep)]);
    }
};
```

## MoQueries.h

**Description:** Answer interval or tree path queries by finding an approximate TSP through the queries, and moving from one query to the next by adding/removing points at the ends. If values are on tree edges, change step to add/remove the edge (a,c) and remove the initial add call (but keep in).

**Time:**  $\mathcal{O}(N\sqrt{Q})$

1957f4, 49 lines

```
void add(int ind, int end) { ... } // add a[ind] (end =
    0 or 1)
void del(int ind, int end) { ... } // remove a[ind]
int calc() { ... } // compute current answer
vi mo(vector<pii> Q) {
    int L = 0, R = 0, blk = 350; // ~N/sqrt(Q)
    vi s(sz(Q)), res = s;
#define K(x) pii(x.first/blk, x.second ^ -(x.first/blk
        & 1))
    iota(all(s), 0);
    sort(all(s), [&](int s, int t){ return K(Q[s]) < K(Q[
        t]); });
    for (int qi : s) {
        pii q = Q[qi];
        while (L > q.first) add(--L, 0);
        while (R < q.second) add(R++, 1);
        while (L < q.first) del(L++, 0);
        while (R > q.second) del(--R, 1);
        res[qi] = calc();
    }
    return res;
}
vi moTree(vector<array<int, 2>> Q, vector<vi>& ed, int
    root=0){
    int N = sz(ed), pos[2] = {}, blk = 350; // ~N/sqrt(Q)
    vi s(sz(Q)), res = s, I(N), L(N), R(N), in(N), par(N)
        ;
    add(0, 0), in[0] = 1;
    auto dfs = [&](int x, int p, int dep, auto& f) ->
        void {
            par[x] = p;
            L[x] = N;
            if (dep) I[x] = N++;
            for (int y : ed[x]) if (y != p) f(y, x, !dep, f);
        }
};
```

```
    if (!dep) I[x] = N++;
    R[x] = N;
};
dfs(root, -1, 0, dfs);
#define K(x) pii(I[x][0] / blk, I[x][1] ^ -(I[x][0] /
    blk & 1))
    iota(all(s), 0);
    sort(all(s), [&](int s, int t){ return K(Q[s]) < K(Q[
        t]); });
    for (int qi : s) fwd(end,0,2) {
        int &a = pos[end], b = Q[qi][end], i = 0;
#define step(c) { if (in[c]) { del(a, end); in[a] = 0;
        } \
            else { add(c, end); in[c] = 1; } a =
            c; }
        while (!L[b] <= L[a] && R[a] <= R[b])
            I[i++] = b, b = par[b];
        while (a != b) step(par[a]);
        while (i--) step(I[i]);
        if (end) res[qi] = calc();
    }
    return res;
}
};
```

## Numerical (4)

### 4.1 Polynomials and recurrences

#### Polynomial.h

5307ee, 17 lines

```
struct Poly {
    vector<double> a;
    double operator()(double x) const {
        double val = 0;
        for (int i = sz(a); i--;) (val += x) += a[i];
        return val;
    }
    void diff() {
        fwd(1,1,sz(a)) a[i-1] = i*a[i];
        a.pop_back();
    }
    void divroot(double x0) {
        double b = a.back(), c; a.back() = 0;
        for(int i=sz(a)-1; i--;) c = a[i], a[i] = a[i+1]*x0
            +b, b=c;
        a.pop_back();
    }
}
};
```

## PolyRoots.h

**Description:** Finds the real roots to a polynomial.

**Usage:** polyRoots({{2,-3,1}},-1e9,1e9) // solve

$x^2-3x+2 = 0$

**Time:**  $\mathcal{O}(n^2 \log(1/\epsilon))$

"Polynomial.h" 2c892f, 23 lines

```
vector<double> polyRoots(Poly p, double xmin, double
    xmax) {
    if (sz(p.a) == 2) { return {-p.a[0]/p.a[1]}; }
    vector<double> ret;
    Poly der = p;
    der.diff();
    auto dr = polyRoots(der, xmin, xmax);
    dr.push_back(xmin-1);
    dr.push_back(xmax+1);
    sort(all(dr));
    rep(i,sz(dr)-1) {
        double l = dr[i], h = dr[i+1];
        bool sign = p(l) > 0;
        if (sign ^ (p(h) > 0)) {
            fwd(it,0,60) { // while (h - l > 1e-8)
                double m = (l + h) / 2, f = p(m);
                if ((f <= 0) ^ sign) l = m;
                else h = m;
            }
            ret.push_back((l + h) / 2);
        }
    }
    return ret;
}
};
```

## PolyInterpolate.h

**Description:** 1. Interpolate set of points (i, vec[i]) and return it evaluated at x; 2. Given n points (x, f(x)) compute n-1-degree polynomial f that passes through them;

**Time:**  $\mathcal{O}(n)$  and  $\mathcal{O}(n^2)$

8dba48, 33 lines

```
template<class T>
T polyExtend(vector<T>& vec, T x) {
```

```
    int n = sz(vec);
    vector<T> fac(n, 1), suf(n, 1);
    fwd(i, 1, n) fac[i] = fac[i-1] * i;
    for (int i=n; --i;) suf[i-1] = suf[i]*(x-i);
    T pref = 1, ret = 0;
    rep(i, n) {
        T d = fac[i] * fac[n-i-1] * ((n-i)%2*2-1);
        ret += vec[i] * suf[i] * pref / d;
        pref *= x-i;
    }
    return ret;
}
template<class T>
vector<T> polyInterp(vector<pair<T, T>> P) {
    int n = sz(P);
    vector<T> ret(n), tmp(n);
    T last = 0;
    tmp[0] = 1;
    rep(k, n-1) fwd(i, k+1, n)
        P[i].y = (P[i].y-P[k].y) / (P[i].x-P[k].x);
    rep(k, n) rep(i, n) {
        ret[i] += P[k].y * tmp[i];
        swap(last, tmp[i]);
        tmp[i] -= last * P[k].x;
    }
    return ret;
}
};
```

## BerlekampMassey.h

**Description:** Recovers any  $n$ -order linear recurrence relation from the first  $2n$  terms of the recurrence. Useful for guessing linear recurrences after brute-forcing the first terms. Should work on any field, but numerical stability for floats is not guaranteed. Output will have size  $\leq n$ .

**Usage:** berlekampMassey({0, 1, 1, 3, 5, 11}) // {1, 2}

**Time:**  $\mathcal{O}(N^2)$

"./number-theory/ModPow.h" 641c59, 20 lines

```
vector<ll> berlekampMassey(vector<ll> s) {
    int n = sz(s), L = 0, m = 0;
    vector<ll> C(n), B(n), T;
    C[0] = B[0] = 1;
    ll b = 1;
    rep(i,n) { ++m;
        ll d = s[i] % mod;
        fwd(j,1,L+1) d = (d + C[j] * s[i - j]) % mod;
        if (!d) continue;
        T = C; ll coef = d * modpow(b, mod-2) % mod;
        fwd(j,m,n) C[j] = (C[j] - coef * B[j - m]) % mod;
        if (2 * L > i) continue;
        L = i + 1 - L; B = T; b = d; m = 0;
    }
    C.resize(L + 1); C.erase(C.begin());
    for (ll x : C) x = (mod - x) % mod;
    return C;
}
};
```

## LinearRecurrence.h

**Description:** Generates the  $k$ 'th term of an  $n$ -order linear recurrence  $S[i] = \sum_j S[i - j - 1]tr[j]$ , given  $S[0 \dots \geq n - 1]$  and  $tr[0 \dots n - 1]$ . Faster than matrix multiplication. Useful together with Berlekamp–Massey.

**Usage:** linearRec({0, 1}, {1, 1}, k) //  $k$ 'th Fibonacci number

**Time:**  $\mathcal{O}(n^2 \log k)$

1868dd, 26 lines

```
typedef vector<ll> Poly;
ll linearRec(Poly S, Poly tr, ll k) {
    int n = sz(tr);
    auto combine = [&](Poly a, Poly b) {
        Poly res(n * 2 + 1);
        rep(i,n+1) rep(j,n+1)
            res[i + j] = (res[i + j] + a[i] * b[j]) % mod;
        for (int i = 2 * n; i > n; --i) rep(j,n)
            res[i - 1 - j] = (res[i - 1 - j] + res[i] * tr[j]
                ) % mod;
        res.resize(n + 1);
        return res;
    };
    Poly pol(n + 1), e(pol);
    pol[0] = e[1] = 1;
    for (++k; k; k /= 2) {
        if (k % 2) pol = combine(pol, e);
        e = combine(e, e);
    }
    ll res = 0;
    rep(i,n) res = (res + pol[i + 1] * S[i]) % mod;
    return res;
}
};
```

}

## PolynomialPotepa.h

**Description:** Poynomials. Implement  $\mathbb{Z}_p$ , or modify to use  $\mathbb{I}$  modulo mod.

**Time:** see below

267aa1, 251 lines

```
using Poly = vector<Zp>;
// Cut off trailing zeroes; time: O(n)
void norm(Poly &P) {
    while (!P.empty()i) && !P.back().x)
        P.pop_back();
}
// Evaluate polynomial at x; time: O(n)
Zp eval(const Poly &P, Zp x) {
    Zp n = 0, y = 1;
    each(a, P) n += a * y, y *= x;
    return n;
}
// Add polynomial; time: O(n)
Poly &operator+=(Poly &l, const Poly &r) {
    l.resize(max(sz(l), sz(r)));
    rep(i, sz(r)) l[i] += r[i];
    norm(l);
    return l;
}
Poly operator+(Poly l, const Poly &r) { return l += r;
}
// Subtract polynomial; time: O(n)
Poly &operator-=(Poly &l, const Poly &r) {
    l.resize(max(sz(l), sz(r)));
    rep(i, sz(r)) l[i] -= r[i];
    norm(l);
    return l;
}
Poly operator-(Poly l, const Poly &r) { return l -= r;
}
// Multiply by polynomial; time: O(n lg n)
Poly &operator*=(Poly &l, const Poly &r) {
    if (min(sz(l), sz(r)) < 50) {
        // Naive multiplication
        Poly p(sz(l) + sz(r));
        rep(i, sz(l)) rep(j, sz(r)) p[i + j] += l[i] * r[
            j];
        l.swap(p);
    } else {
        // FFT multiplication
    }
    norm(l);
    return l;
}
Poly operator*(Poly l, const Poly &r) { return l *= r;
}
// Compute inverse series mod  $x^n$ ; O(n lg n) Requires P
(0) != 0.
Poly invert(const Poly &P, int n) {
    assert(!P.empty()i) && P[0].x);
    Poly tmp{P[0]}, ret = {P[0].inv()};
    for (int i = 1; i < n; i *= 2) {
        fwd(j, i, min(i * 2, sz(P))) tmp.push_back(P[j]);
        (ret *= Poly{2} - tmp * ret).resize(i * 2);
    }
    ret.resize(n);
    return ret;
}
// Floor division by polynomial; O(n lg n)
Poly &operator/=(Poly &l, Poly r) {
    norm(l);
    norm(r);
    int d = sz(l) - sz(r) + 1;
    if (d <= 0)
        return l.clear(), l;
    reverse(all(l));
    reverse(all(r));
    l.resize(d);
    l *= invert(r, d);
    l.resize(d);
    reverse(all(l));
    return l;
}
Poly operator/(Poly l, const Poly &r) { return l /= r;
}
// Remainder modulo a polynomial; O(n lg n)
Poly operator%(const Poly &l, const Poly &r) { return l
    - r * (l / r); }
Poly &operator%=(Poly &l, const Poly &r) { return l -=
    r * (l / r); }
// Compute  $a^e$  mod  $x^n$ , where a is polynomial;
// time: O(n log n log e)
Poly pow(Poly a, ll e, int n) {
    Poly t = {1};
```

```
double det(vector<vector<double>>& a) {
    int n = sz(a); double res = 1;
    rep(i,n) {
        int b = i;
        fwd(j,i+1,n) if (fabs(a[j][i]) > fabs(a[b][i])) b =
            j;
    }
}
```



```
    if (i != b) swap(a[i], a[b]), res *= -1;
    res *= a[i][i];
    if (res == 0) return 0;
    fwd(j,i+1,n) {
        double v = a[j][i] / a[i][i];
        if (v != 0) fwd(k,i+1,n) a[j][k] -= v * a[i][k];
    }
    return res;
}
```

## IntDeterminant.h

**Description:** Calculates determinant using modular arithmetics. Modulos can also be removed to get a pure-integer version.

**Time:**  $\mathcal{O}(N^3)$  139eec, 18 lines

```
const ll mod = 12345;
ll det(vector<vector<ll>>& a) {
    int n = sz(a); ll ans = 1;
    rep(i,n) {
        fwd(j,i+1,n) {
            while (a[j][i] != 0) { // gcd step
                ll t = a[i][i] / a[j][i];
                if (t) fwd(k,i,n)
                    a[i][k] = (a[i][k] - a[j][k] * t) % mod;
                swap(a[i], a[j]);
                ans *= -1;
            }
            ans = ans * a[i][i] % mod;
            if (!ans) return 0;
        }
        return (ans + mod) % mod;
    }
}
```

## SolveLinear.h

**Description:** Solves  $A * x = b$ . If there are multiple solutions, an arbitrary one is returned. Returns rank, or -1 if no solutions. Data in  $A$  and  $b$  is lost.

**Time:**  $\mathcal{O}(n^2m)$  11d015, 38 lines

```
typedef vector<double> vd;
const double eps = 1e-12;
int solveLinear(vector<vd>& A, vd& b, vd& x) {
    int n = sz(A), m = sz(x), rank = 0, br, bc;
    if (n) assert(sz(A[0]) == m);
    vi col(m); iota(all(col), 0);
    rep(i,n) {
        double v, bv = 0;
        fwd(r,i,n) fwd(c,i,m)
            if ((v = fabs(A[r][c])) > bv)
                br = r, bc = c, bv = v;
        if (bv <= eps) {
            fwd(j,i,n) if (fabs(b[j]) > eps) return -1;
            break;
        }
        swap(A[i], A[br]);
        swap(b[i], b[br]);
        swap(col[i], col[bc]);
        rep(j,n) swap(A[j][i], A[j][bc]);
        bv = 1 / A[i][i];
        fwd(j,i+1,n) {
            double fac = A[j][i] * bv;
            b[j] -= fac * b[i];
            fwd(k,i+1,m) A[j][k] -= fac * A[i][k];
        }
        rank++;
    }
    x.assign(m, 0);
    for (int i = rank; i--;) {
        b[i] /= A[i][i];
        x[col[i]] = b[i];
        rep(j,i) b[j] -= A[j][i] * b[i];
    }
    return rank; // (multiple solutions if rank < m)
}
```

## SolveLinear2.h

**Description:** To get all uniquely determined values of  $x$  back from SolveLinear, make the following changes:

"SolveLinear.h" acb9c0, 7 lines

```
rep(j,n) if (j != i) // instead of fwd(j,i+1,n)
// ... then at the end:
x.assign(m, undefined);
rep(i,rank) {
    fwd(j,rank,m) if (fabs(A[i][j]) > eps) goto fail;
    x[col[i]] = b[i] / A[i][i];
}
```

fail;; }

## SolveLinearBinary.h

**Description:** Solves  $Ax = b$  over  $F_2$ . If there are multiple solutions, one is returned arbitrarily. Returns rank, or -1 if no solutions. Destroys  $A$  and  $b$ .

**Time:**  $\mathcal{O}(n^2m)$  d99ddb, 34 lines

```
typedef bitset<1000> bs;
int solveLinear(vector<bs>& A, vi& b, bs& x, int m) {
    int n = sz(A), rank = 0, br;
    assert(m <= sz(x));
    vi col(m); iota(all(col), 0);
    rep(i,n) {
        for (br=i; br<n; ++br) if (A[br].any()) break;
        if (br == n) {
            fwd(j,i,n) if(b[j]) return -1;
            break;
        }
        int bc = (int)A[br]._Find_next(i-1);
        swap(A[i], A[br]);
        swap(b[i], b[br]);
        swap(col[i], col[bc]);
        rep(j,n) if (A[j][i] != A[j][bc]) {
            A[j].flip(i); A[j].flip(bc);
        }
        fwd(j,i+1,n) if (A[j][i]) {
            b[j] ^= b[i];
            A[j] ^= A[i];
        }
        rank++;
    }
    x = bs();
    for (int i = rank; i--;) {
        if (!b[i]) continue;
        x[col[i]] = 1;
        rep(j,i) b[j] ^= A[j][i];
    }
    return rank; // (multiple solutions if rank < m)
}
```

## MatrixInverse.h

**Description:** Invert matrix  $A$ . Returns rank; result is stored in  $A$  unless singular (rank < n). Can easily be extended to prime moduli; for prime powers, repeatedly set  $A^{-1} = A^{-1}(2I - AA^{-1}) \pmod{p^k}$  where  $A^{-1}$  starts as the inverse of  $A \pmod p$ , and  $k$  is doubled in each step.

**Time:**  $\mathcal{O}(n^3)$  731feb, 35 lines

```
int matInv(vector<vector<double>>& A) {
    int n = sz(A); vi col(n);
    vector<vector<double>> tmp(n, vector<double>(n));
    rep(i,n) tmp[i][i] = 1, col[i] = i;
    rep(i,n) {
        int r = i, c = i;
        fwd(j,i,n) fwd(k,i,n)
            if (fabs(A[j][k]) > fabs(A[r][c]))
                r = j, c = k;
        if (fabs(A[r][c]) < 1e-12) return i;
        A[i].swap(A[r]); tmp[i].swap(tmp[r]);
        rep(j,n)
            swap(A[j][i], A[j][c]), swap(tmp[j][i], tmp[j][c]);
        swap(col[i], col[c]);
        double v = A[i][i];
        fwd(j,i+1,n) {
            double f = A[j][i] / v;
            A[j][i] = 0;
            fwd(k,i+1,n) A[j][k] -= f * A[i][k];
            rep(k,n) tmp[j][k] -= f * tmp[i][k];
        }
        fwd(j,i+1,n) A[i][j] /= v;
        rep(j,n) tmp[i][j] /= v;
        A[i][i] = 1;
    }
    for (int i = n-1; i > 0; --i) rep(j,i) {
        double v = A[j][i];
        rep(k,n) tmp[j][k] -= v * tmp[i][k];
    }
    rep(i,n) rep(j,n) A[col[i]][col[j]] = tmp[i][j];
    return n;
}
```

## Tridiagonal.h

**Description:**  $x = \text{tridiagonal}(d,p,q,b)$  solves the equation system

$$\begin{pmatrix} b_0 & & & & & & \\ b_1 & d_0 & p_0 & 0 & \cdots & 0 & \\ b_2 & 0 & q_1 & d_1 & p_1 & 0 & \cdots & 0 \\ b_3 & & & q_2 & d_2 & p_2 & \cdots & 0 \\ \vdots & & & \vdots & \vdots & \vdots & \ddots & \vdots \\ \vdots & 0 & 0 & \cdots & q_{n-3} & d_{n-2} & p_{n-2} & \\ b_{n-1} & 0 & 0 & \cdots & 0 & q_{n-2} & d_{n-1} & \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ \vdots \\ \vdots \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ \vdots \\ \vdots \\ x_{n-1} \end{pmatrix}.$$

This is useful for solving problems on the type

$$a_i = b_i a_{i-1} + c_i a_{i+1} + d_i, 1 \leq i \leq n,$$

where  $a_0, a_{n+1}, b_i, c_i$  and  $d_i$  are known.  $a$  can then be obtained from

$$\{a_i\} = \text{tridiagonal}(\{1, -1, -1, \dots, -1, 1\}, \{0, c_1, c_2, \dots, c_n\}, \{b_1, b_2, \dots, b_n, 0\}, \{a_0, d_1, d_2, \dots, d_n, a_{n+1}\}).$$

Fails if the solution is not unique.

If  $|d_i| > |p_i| + |q_{i-1}|$  for all  $i$ , or  $|d_i| > |p_{i-1}| + |q_i|$ , or the matrix is positive definite, the algorithm is numerically stable and neither  $\text{tr}$  nor the check for  $\text{diag}[i] == 0$  is needed.

**Time:**  $\mathcal{O}(N)$  059430, 26 lines

```
typedef double T;
vector<T> tridiagonal(vector<T> diag, const vector<T>&
    super,
    const vector<T>& sub, vector<T> b) {
    int n = sz(b); vi tr(n);
    rep(i,n-1) {
        if (abs(diag[i]) < 1e-9 * abs(super[i])) { // diag[
            i] == 0
                b[i+1] -= b[i] * diag[i+1] / super[i];
                if (i+2 < n) b[i+2] -= b[i] * sub[i+1] / super[i
                    ];
                diag[i+1] = sub[i]; tr[++i] = 1;
            } else {
                diag[i+1] -= super[i] * sub[i] / diag[i];
                b[i+1] -= b[i] * sub[i] / diag[i];
            }
        }
        for (int i = n; i--;) {
            if (tr[i]) {
                swap(b[i], b[i-1]);
                diag[i-1] = diag[i];
                b[i] /= super[i-1];
            } else {
                b[i] /= diag[i];
                if (i) b[i-1] -= b[i] * super[i-1];
            }
        }
        return b;
    }
```

## 4.4 Fourier transforms

### FastFourierTransform.h

**Description:**  $\text{fft}(a)$  computes  $\hat{f}(k) = \sum x a[x] \exp(2\pi i \cdot kx / N)$  for all  $k$ .  $N$  must be a power of 2. Useful for convolution:  $\text{conv}(a, b) = c$ , where  $c[x] = \sum a[i]b[x-i]$ . For convolution of complex numbers or more than two vectors: FFT, multiply pointwise, divide by  $n$ , reverse(start+1, end), FFT back. Rounding is safe if  $(\sum a_i^2 + \sum b_i^2) \log_2 N < 9 \cdot 10^{14}$  (in practice  $10^{16}$ ; higher for random inputs). Otherwise, use NTT/FFT-Mod.

**Time:**  $\mathcal{O}(N \log N)$  with  $N = |A| + |B|$  ( $\sim 1s$  for  $N=35 \cdot 2^{22}$  lines

```
typedef complex<double> C;
typedef vector<double> vd;
void fft(vector<C>& a) {
    int n = sz(a), L = 31 - __builtin_clz(n);
    static vector<complex<long double>> R(2, 1);
    static vector<C> rt(2, 1); // (^ 10% faster if
        double)
    for (static int k = 2; k < n; k *= 2) {
        R.resize(n); rt.resize(n);
        auto x = polar(1.0/L, acos(-1.0/L) / k);
        fwd(i,k,2*k) rt[i] = R[i] = i&1 ? R[i/2] * x : R[i
            /2];
    }
    vi rev(n);
    rep(i,n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
    rep(i,n) if (i < rev[i]) swap(a[i], a[rev[i]]);
    for (int k = 1; k < n; k *= 2)
        for (int i = 0; i < n; i += 2 * k) rep(j,k) {
            C z = rt[j+k] * a[i+j+k]; // (25% faster if hand-
                rolled)
            a[i + j + k] = a[i + j] - z;
```

```
            a[i + j] += z;
        }
    }
    vd conv(const vd& a, const vd& b) {
        if (a.empty() || b.empty()) return {};
        vd res(sz(a) + sz(b) - 1);
        int L = 32 - __builtin_clz(sz(res)), n = 1 << L;
        vector<C> in(n), out(n);
        copy(all(a), begin(in));
        rep(i,sz(b)) in[i].imag(b[i]);
        fft(in);
        for (C& x : in) x *= x;
        rep(i,n) out[i] = in[-i & (n - 1)] - conj(in[i]);
        fft(out);
        rep(i,sz(res)) res[i] = imag(out[i]) / (4 * n);
        return res;
    }
```

## FastFourierTransformMod.h

**Description:** Higher precision FFT, can be used for convolutions modulo arbitrary integers as long as  $N \log_2 N \cdot \text{mod} < 8.6 \cdot 10^{14}$  (in practice  $10^{16}$  or higher). Inputs must be in  $[0, \text{mod})$ .

**Time:**  $\mathcal{O}(N \log N)$ , where  $N = |A| + |B|$  (twice as slow as NTT or FFT)

"FastFourierTransform.h" ffe1eb, 22 lines

```
typedef vector<ll> vl;
template<int M> vl convMod(const vl &a, const vl &b) {
    if (a.empty() || b.empty()) return {};
    vl res(sz(a) + sz(b) - 1);
    int B=32-__builtin_clz(sz(res)), n=1<<B, cut=int(sqrt
        (M));
    vector<C> L(n), R(n), outs(n), outl(n);
    rep(i,sz(a)) L[i] = C((int)a[i] / cut, (int)a[i] %
        cut);
    rep(i,sz(b)) R[i] = C((int)b[i] / cut, (int)b[i] %
        cut);
    fft(L), fft(R);
    rep(i,n) {
        int j = -i & (n - 1);
        outl[j] = (L[i] + conj(L[j])) * R[i] / (2.0 * n);
        outs[j] = (L[i] - conj(L[j])) * R[i] / (2.0 * n) /
            1i;
    }
    fft(outl), fft(outs);
    rep(i,sz(res)) {
        ll av = ll(real(outl[i])+.5), cv = ll(imag(outs[i])
            +.5);
        ll bv = ll(imag(outl[i])+.5) + ll(real(outs[i])+.5)
            ;
        res[i] = ((av % M * cut + bv) % M * cut + cv) % M;
    }
    return res;
}
```

## NumberTheoreticTransform.h

**Description:**  $\text{ntt}(a)$  computes  $\hat{f}(k) = \sum x a[x]g^{xk}$  for all  $k$ , where  $g = \text{root}^{(\text{mod}-1)/N}$ .  $N$  must be a power of 2. Useful for convolution modulo specific nice primes of the form  $2^a b + 1$ , where the convolution result has size at most  $2^a$ . For arbitrary modulo, see FFTMod.  $\text{conv}(a, b) = c$ , where  $c[z] = \sum a[i]b[x-i]$ . For manual convolution: NTT the inputs, multiply pointwise, divide by  $n$ , reverse(start+1, end), NTT back. Inputs must be in  $[0, \text{mod})$ .

**Time:**  $\mathcal{O}(N \log N)$

".../number-theory/ModPow.h" 2e0a0e, 34 lines

```
const ll mod = (119 << 23) + 1, root = 62; // =
    998244353
// For  $p < 2^{30}$  there is also e.g.  $5 << 25, 7 << 26,$ 
     $479 << 21$ 
// and  $483 << 21$  (same root). The last two are  $> 10^{9.}$ 
// int128: (2147483641LL<<32) - but 2x11 & crt is
    faster.
typedef vector<ll> vl;
void ntt(vl &a) {
    int n = sz(a), L = 31 - __builtin_clz(n);
    static vl rt(2, 1);
    for (static int k = 2, s = 2; k < n; k *= 2, s++) {
        rt.resize(n);
        ll z[] = {1, modpow(root, mod >> s)};
        fwd(i,k,2*k) rt[i] = rt[i / 2] * z[i & 1] % mod;
    }
    vi rev(n);
    rep(i,n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
```

```

rep(i,n) if (i < rev[i]) swap(a[i], a[rev[i]]);
for (int k = 1; k < n; k *= 2)
    for (int i = 0; i < n; i += 2 * k) rep(j,k) {
        ll z = rt[j+k] * a[i+j+k] % mod, &ai = a[i+j];
        a[i+j+k] = ai - z + (z > ai ? mod : 0);
        ai += (ai + z >= mod ? z - mod : z);
    }
vl conv(const vl &a, const vl &b) {
    if (a.empty() || b.empty()) return {};
    int s = sz(a) + sz(b) - 1, B = 32 - __builtin_clz(s),
        n = 1 << B;
    ll inv = modpow(n, mod - 2);
    vl L(a), R(b), out(n);
    L.resize(n), R.resize(n);
    ntt(L), ntt(R);
    rep(i,n) out[-i & (n - 1)] = (ll)L[i] * R[i] % mod *
        inv % mod;
    ntt(out);
    return {out.begin(), out.begin() + s};
}

```

## FastSubsetTransform.h

**Description:** Transform to a basis with fast convolutions of the form  $c[z] = \sum_{z=x \oplus y} a[x] \cdot b[y]$ , where  $\oplus$  is one of AND, OR, XOR. The size of  $a$  must be a power of two.

**Time:**  $\mathcal{O}(N \log N)$  790905, 16 lines

```

void FST(vi& a, bool inv) {
    for (int n = sz(a), step = 1; step < n; step *= 2) {
        for (int i = 0; i < n; i += 2 * step) fwd(j,i,i+step) {
            int &u = a[j], &v = a[j + step]; tie(u, v) =
                inv ? pii(v - u, u) : pii(v, u + v); // AND
            inv ? pii(v, u - v) : pii(u + v, u); // OR
            pii(u + v, u - v); // XOR
        }
        if (inv) for (int& x : a) x /= sz(a); // XOR only
    }
    vi conv(vi a, vi b) {
        FST(a, 0); FST(b, 0);
        rep(i, sz(a)) a[i] ^= b[i];
        FST(a, 1); return a;
    }
}

```

## Number theory (5)

### 5.1 Modular arithmetic

#### ModLog.h

**Description:** Returns the smallest  $x > 0$  s.t.  $a^x = b \pmod{m}$ , or  $-1$  if no such  $x$  exists. `modLog(a,1,m)` can be used to calculate the order of  $a$ .

**Time:**  $\mathcal{O}(\sqrt{m})$  e593f3, 11 lines

```

ll modLog(ll a, ll b, ll m) {
    ll n = (ll) sqrt(m) + 1, e = 1, f = 1, j = 1;
    unordered_map<ll, ll> A;
    while (j <= n && (e = f * e % m) != b % m)
        A[e * b % m] = j++;
    if (e == b % m) return j;
    if (__gcd(m, e) == __gcd(m, b))
        fwd(i, 2, n+2) if (A.count(e = e * f % m))
            return n * i - A[e];
    return -1;
}

```

## ModSum.h

**Description:** Sums of mod'ed arithmetic progressions.

`modsum(to, c, k, m) = \sum_{i=0}^{to-1} (ki + c) \% m`. `divsum` is similar but for floored division.

**Time:**  $\log(m)$ , with a large constant. 5c5bc5, 16 lines

```

typedef unsigned long long ull;
ull sumsq(ull to) { return to / 2 * ((to-1) | 1); }
ull divsum(ull to, ull c, ull k, ull m) {
    ull res = k / m * sumsq(to) + c / m * to;
    k %= m; c %= m;
    if (!k) return res;
    ull to2 = (to * k + c) / m;
    return res + (to - 1) * to2 - divsum(to2, m-1 - c, m, k);
}
ll modsum(ull to, ll c, ll k, ll m) {
    c = ((c % m) + m) % m;
    k = ((k % m) + m) % m;
}

```

```

return to * c + k * sumsq(to) - m * divsum(to, c, k, m);
}

```

## ModSqrt.h

**Description:** Tonelli-Shanks algorithm for modular square roots. Finds  $x$  s.t.  $x^2 = a \pmod{p}$  ( $-x$  gives the other solution).

**Time:**  $\mathcal{O}(\log^2 p)$  worst case,  $\mathcal{O}(\log p)$  for most  $p$

**"ModPow.h"** 19a793, 24 lines

```

ll sqrt(ll a, ll p) {
    a %= p; if (a < 0) a += p;
    if (a == 0) return 0;
    assert(modpow(a, (p-1)/2, p) == 1); // else no solution
    if (p % 4 == 3) return modpow(a, (p+1)/4, p);
    // a^(n+3)/8 or 2^(n+3)/8 * 2^(n-1)/4 works if p % 8 == 5
    ll s = p - 1, n = 2;
    int r = 0, m;
    while (s % 2 == 0)
        ++r, s /= 2;
    while (modpow(n, (p - 1) / 2, p) != p - 1) ++n;
    ll x = modpow(a, (s + 1) / 2, p);
    ll b = modpow(a, s, p), g = modpow(n, s, p);
    for (; r = m) {
        ll t = b;
        for (m = 0; m < r && t != 1; ++m)
            t = t * t % p;
        if (m == 0) return x;
        ll gs = modpow(g, 1LL << (r - m - 1), p);
        g = gs * gs % p;
        x = x * gs % p;
        b = b * g % p;
    }
}

```

### 5.2 Primality

#### FastEratosthenes.h

**Description:** Prime sieve for generating all primes smaller than LIM.

**Time:** LIM=1e9  $\approx$  1.5s 3dcf2f, 20 lines

```

const int LIM = 1e6;
bitset<LIM> isPrime;
vi eratosthenes() {
    const int S = (int)round(sqrt(LIM)), R = LIM / 2;
    vi pr = {2}, sieve(S+1); pr.reserve((int)(LIM/log(LIM) * 1.1));
    vector<pii> cp;
    for (int i = 3; i <= S; i += 2) if (!sieve[i]) {
        cp.push_back({i, i * i / 2});
        for (int j = i * i; j <= S; j += 2 * i) sieve[j] = 1;
    }
    for (int L = 1; L <= R; L += S) {
        array<bool, S> block{};
        for (auto &[p, idx] : cp)
            for (int i=idx; i < S+L; idx = (i+p)) block[i-L] = 1;
        rep(i,min(S, R - L))
            if (!block[i]) pr.push_back((L + i) * 2 + 1);
    }
    for (int i : pr) isPrime[i] = 1;
    return pr;
}

```

## MillerRabin.h

**Description:** Deterministic Miller-Rabin primality test. Guaranteed to work for numbers up to  $7 \cdot 10^{18}$ ; for larger numbers, use Python and extend A randomly.

**Time:** 7 times the complexity of  $a^b \pmod{c}$ .

**"ModMulLL.h"** 60dcd1, 12 lines

```

bool isPrime(ull n) {
    if (n < 2 || n % 6 % 4 != 1) return (n | 1) == 3;
    ull A[] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022},
        s = __builtin_ctzll(n-1), d = n >> s;
    for (ull a : A) { // ^count trailing zeroes
        ull p = modpow(a%n, d, n), i = s;
        while (p != 1 && p != n-1 && a % n && i--)
            p = modmul(p, p, n);
        if (p != n-1 && i != s) return 0;
    }
    return 1;
}

```

## Factor.h

**Description:** Pollard-rho randomized factorization algorithm. Returns prime factors of a number, in arbitrary order (e.g. 2299 -> {11, 19, 11}).

**Time:**  $\mathcal{O}(n^{1/4})$ , less for numbers with small factors.

**"ModMulLL.h", "MillerRabin.h"** d8d98d, 18 lines

```

ull pollard(ull n) {
    ull x = 0, y = 0, t = 30, prd = 2, i = 1, q;
    auto f = [&](ull x) { return modmul(x, x, n) + i; };
    while (t++ % 40 || __gcd(prd, n) == 1) {
        if (x == y) x = ++i, y = f(x);
        if ((q = modmul(prd, max(x,y) - min(x,y), n))) prd
            = q;
        x = f(x), y = f(f(y));
    }
    return __gcd(prd, n);
}
vector<ull> factor(ull n) {
    if (n == 1) return {};
    if (isPrime(n)) return {n};
    ull x = pollard(n);
    auto l = factor(x), r = factor(n / x);
    l.insert(l.end(), all(r));
    return l;
}

```

### 5.3 Divisibility

#### euclid.h

**Description:** Finds two integers  $x$  and  $y$ , such that  $ax + by = \gcd(a, b)$ . If you just need gcd, use the built in `__gcd` instead. If  $a$  and  $b$  are coprime, then  $x$  is the inverse of  $a \pmod{b}$ .

**"ModMulLL.h"** 3d3aed, 6 lines

```

ll euclid(ll a, ll b, ll &x, ll &y) {
    if (!b) return x = 1, y = 0, a;
    ll d = euclid(b, a % b, y, x);
    return y -= a/b * x, d;
}

```

## CRT.h

**Description:** Chinese Remainder Theorem.

`crt(a, m, b, n)` computes  $x$  such that  $x \equiv a \pmod{m}$ ,  $x \equiv b \pmod{n}$ . If  $|a| < m$  and  $|b| < n$ ,  $x$  will obey  $0 \leq x < \text{lcm}(m, n)$ . Assumes  $mn < 2^{62}$ .

**Time:**  $\log(n)$

**"euclid.h"** 04d93a, 7 lines

```

ll crt(ll a, ll m, ll b, ll n) {
    if (n > m) swap(a, b), swap(m, n);
    ll x, y, g = euclid(m, n, x, y);
    assert((a - b) % g == 0); // else no solution
    x = (b - a) % n * x % n / g * m + a;
    return x < 0 ? x + m*n/g : x;
}

```

## phiFunction.h

**Description:** Euler's  $\phi$  function is defined as  $\phi(n) := \#$  of positive integers  $\leq n$  that are coprime with  $n$ .  $\phi(1) = 1$ ,  $p$  prime  $\Rightarrow \phi(p^k) = (p - 1)p^{k-1}$ ,  $m, n$  coprime  $\Rightarrow \phi(mn) = \phi(m)\phi(n)$ . If  $n = p_1^{k_1} p_2^{k_2} \dots p_r^{k_r}$  then  $\phi(n) = (p_1 - 1)p_1^{k_1-1} \dots (p_r - 1)p_r^{k_r-1}$ .  $\phi(n) = n \cdot \prod_{p|n} (1 - 1/p)$ .  $\sum_{d|n} \phi(d) = n$ ,  $\sum_{1 \leq k \leq n, \gcd(k,n)=1} k = n\phi(n)/2$ ,  $n > 1$

**Euler's thm:**  $a, n$  coprime  $\Rightarrow a^{\phi(n)} \equiv 1 \pmod{n}$ .

**Fermat's little thm:**  $p$  prime  $\Rightarrow a^{p-1} \equiv 1 \pmod{p} \forall a$ .

**Time:**  $\mathcal{O}(n^{2/3})$  b3ee8e, 34 lines

```

constexpr int MOD = 998244353;
vi phi(1e7 + 1);
void calcPhi() {
    iota(all(phi), 0);
    fwd(i, 2, sz(phi)) if (phi[i] == i) for (int j = i; j < sz(phi); j += i) phi[j] = phi[j] / i * (i - 1);
}
vector<ll> phiSum; // [k] = sum from 0 to k-1
void calcPhiSum() {
    calcPhi();
    phiSum.resize(sz(phi) + 1);
    rep(i, sz(phi)) phiSum[i + 1] = (phiSum[i] + phi[i]) % MOD;
}
// Get prefix sum of phi(0) + ... + phi(n-1).
// WARNING: Call calcPhiSum first! For MOD > 4*10^9,
// answer will overflow.

```

```

ll getPhiSum(ll n) {
    static unordered_map<ll, ll> big;
    if (n < sz(phiSum))
        return phiSum[n];
    if (big.count(--n))
        return big[n];
    ll ret = (n % 2 ? n % MOD * ((n + 1) / 2 % MOD) : n / 2 % MOD * (n % MOD + 1) % MOD;
    for (ll s, i = 2; i <= n; i = s + 1) {
        s = n / (n / i);
        ret -= (s - i + 1) % MOD * getPhiSum(n / i + 1) % MOD;
    }
    return big[n] = ret = (ret % MOD + MOD) % MOD;
}

```

## Min25.h

**Description:** Calculates prefsums of multiplicative function at each floor( $N/i$ ). `id(N/i)` returns index of  $N/i$  in `keys[]`. Remember about overflows. See example below.

**Time:**  $\mathcal{O}\left(\frac{n^{3/4}}{\log n}\right)$  3e1c86, 50 lines

```

vector<ll> global_primes; // global_primes.back() >
                        sqrt(N); use e.g. sieve of Eratosthenes
template<typename T>
struct Min25 {
    ll N;
    vector<ll> keys, primes;
    Min25(ll N_) : N(N_) {
        for (ll l = 1; l <= N; ++l)
            keys.push_back(l = N / (N / l));
        for (int i = 0; global_primes[i] *
            global_primes[i] <= N; ++i)
            primes.push_back(global_primes[i]);
    }
    ll id(ll x) {
        ll id = x < N / x ? x - 1 : sz(keys) - N / x;
        assert(keys[id] == x);
        return id;
    }
    // f has to be TOTALLY multiplicative, pref(x) is
    // regular prefix sum function of f
    vector<T> overPrimes(auto pref) {
        vector<T> dp(sz(keys));
        rep(i, sz(keys))
            dp[i] = pref(keys[i]) - T(1);
        for (ll p : primes) {
            auto fp = dp[p - 1] - dp[p - 2];
            for (int i = sz(keys) - 1; i >= 0 && p * p
                <= keys[i]; --i)
                dp[i] = dp[i] - (dp[id(keys[i] / p)] -
                    dp[p - 2]) * fp;
        }
        return dp;
    }
    // dp are prefix sums of f over primes, f(prime,
    // power, prime*power) calculates f on primes
    // powers
    void fullSum(vector<T> &dp, auto f) {
        for (ll p : primes | views::reverse) {
            for (int i = sz(keys) - 1; i >= 0 && p * p
                <= keys[i]; --i) {
                for (ll k = 1, q = p; q * p <= keys[i];
                    ++k, q *= p)
                    dp[i] = dp[i] + f(p, k + 1, q * p)
                        + f(p, k, q) * (dp[id(keys[i] / q)] -
                            dp[p - 1]);
            }
        }
        for (auto &v : dp) v = v + T(1);
    }
};
/*vector<ll> example(Min25<ll> &min25) {
    auto primeCnt = min25.overPrimes([](ll x){return x;
    }); // f(x)=1 over primes
    auto primeSum = min25.overPrimes([](ll x){return x
    * (x+1)/2; }); // f(x)=x over primes
    vector<ll> phi;
    rep(i, sz(min25.keys))
        phi.push_back(primeSum[i] - primeCnt[i]); // f(
        x)=x-1 over primes i.e. phi(x)=f(x)
    min25.fullSum(phi, [](int p, int k, ll pk){return
        pk - pk / p; }); // f(x)=phi(x) over all
        numbers
    return phi;
}*/

```

## 5.4 Fractions ContinuedFractions.h

**Description:** Given  $N$  and a real number  $x \geq 0$ , finds the closest rational approximation  $p/q$  with  $p, q \leq N$ . It will obey  $|p/q - x| \leq 1/qN$ .

For consecutive convergents,  $p_{k+1}q_k - q_{k+1}p_k = (-1)^k$ . ( $p_k/q_k$  alternates between  $> x$  and  $< x$ .) If  $x$  is rational,  $y$  eventually becomes  $\infty$ ; if  $x$  is the root of a degree 2 polynomial the  $a$ 's eventually become cyclic.

**Time:**  $\mathcal{O}(\log N)$  dd6c5e, 21 lines

```
typedef double d; // for N ~ 1e7; long double for N ~ 1e9
pair<ll, ll> approximate(d x, ll N) {
    ll LP = 0, LQ = 1, P = 1, Q = 0, inf = LLONG_MAX; d y = x;
    for (;) {
        ll lim = min(P ? (N-LP) / P : inf, Q ? (N-LQ) / Q : inf),
            a = (ll)floor(y), b = min(a, lim),
            NP = b*P + LP, NQ = b*Q + LQ;
        if (a > b) {
            // If b > a/2, we have a semi-convergent that gives us a
            // better approximation; if b = a/2, we *may* have one.
            // Return {P, Q} here for a more canonical approximation.
            return (abs(x - (d)NP / (d)NQ) < abs(x - (d)P / (d)Q)) ?
                make_pair(NP, NQ) : make_pair(P, Q);
        }
        if (abs(y = 1/(y - (d)a)) > 3*N) {
            return {NP, NQ};
        }
        LP = P; P = NP;
        LQ = Q; Q = NQ;
    }
}
```

## FracBinarySearch.h

**Description:** Given  $f$  and  $N$ , finds the smallest fraction  $p/q \in [0, 1]$  such that  $f(p/q)$  is true, and  $p, q \leq N$ . You may want to throw an exception from  $f$  if it finds an exact solution, in which case  $N$  can be removed.

**Usage:** fracBS([f](Frac f) { return f.p>=3\*f.q; }, 10); // {1,3}

**Time:**  $\mathcal{O}(\log(N))$  27ab3e, 25 lines

```
struct Frac { ll p, q; };
template<class F>
Frac fracBS(F f, ll N) {
    bool dir = 1, A = 1, B = 1;
    Frac lo(0, 1), hi(1, 1); // Set hi to 1/0 to search
    (0, N]
    if (f(lo)) return lo;
    assert(f(hi));
    while (A || B) {
        ll adv = 0, step = 1; // move hi if dir, else lo
        for (int si = 0; step; (step *= 2) >= si) {
            adv += step;
            Frac mid(lo.p * adv + hi.p, lo.q * adv + hi.q);
            if (abs(mid.p) > N || mid.q > N || dir == !f(mid)) {
                adv -= step; si = 2;
            }
        }
        hi.p += lo.p * adv;
        hi.q += lo.q * adv;
        dir = !dir;
        swap(lo, hi);
        A = B; B = !adv;
    }
    return dir ? hi : lo;
}
```

## 5.5 Pythagorean Triples

The Pythagorean triples are uniquely generated by

$$a = k \cdot (m^2 - n^2), \quad b = k \cdot (2mn), \quad c = k \cdot (m^2 + n^2),$$

with  $m > n > 0$ ,  $k > 0$ ,  $m \perp n$ , and either  $m$  or  $n$  even.

## 5.6 Primes & primitive roots

(1000002089, {3, 104, }, (1000000000000200011, {6, 105})) There are 78498 primes less than 1 000 000.

## 5.7 Estimates

The number of divisors of  $n$  is at most around 100 for  $n < 5e4$ , 500 for  $n < 1e7$ , 2000 for  $n < 1e10$ , 200 000 for  $n < 1e19$ .

## 5.8 Mobius Function

$$\mu(n) = \begin{cases} 0 & n \text{ is not square free} \\ 1 & n \text{ has even number of prime factors} \\ -1 & n \text{ has odd number of prime factors} \end{cases}$$

Mobius Inversion:

$$g(n) = \sum_{d|n} f(d) \Leftrightarrow f(n) = \sum_{d|n} \mu(d)g(n/d)$$

Other useful formulas/forms:

$$\sum_{d|n} \mu(d) = [n = 1] \text{ (very useful)}$$

$$g(n) = \sum_{1 \leq m \leq n} f(\lfloor \frac{n}{m} \rfloor) \Leftrightarrow f(n) = \sum_{1 \leq m \leq n} \mu(m)g(\lfloor \frac{n}{m} \rfloor)$$

Define Dirichlet convolution as

$$f * g(n) = \sum_{d|n} f(d)g(n/d). \text{ Let } s_f(n) = \sum_{i=1}^n f(i).$$

$$\text{Then } s_f(n) = \frac{s_f * g(n) - \sum_{d=2}^n s_f(\lfloor \frac{n}{d} \rfloor)g(d)}{g(1)}.$$

## Combinatorial (6)

### 6.1 Permutations

#### 6.1.1 Factorial

#### IntPerm.h

**Description:** Permutation -> integer conversion. (Not order preserving.) Integer -> permutation can use a lookup table.

**Time:**  $\mathcal{O}(n)$  044568, 6 lines

```
int permToInt(vi& v) {
    int use = 0, i = 0, r = 0;
    for(int x:v) r = r * ++i + __builtin_popcount(use &
        -(1<<x)),
        use |= 1 << x; // (note: minus, not ~!)
    return r;
}
```

#### 6.1.2 Cycles

Let  $g_S(n)$  be the number of  $n$ -permutations whose cycle lengths all belong to the set  $S$ . Then

$$\sum_{n=0}^{\infty} g_S(n) \frac{x^n}{n!} = \exp \left( \sum_{n \in S} \frac{x^n}{n} \right)$$

#### 6.1.3 Derangements

Permutations of a set such that none of the elements appear in their original position.

$$D(n) = (n-1)(D(n-1)+D(n-2)) = nD(n-1)+(-1)^n = \left\lfloor \frac{n!}{e} \right\rfloor$$

#### 6.1.4 Burnside's lemma

Given a group  $G$  of symmetries and a set  $X$ , the number of elements of  $X$  up to *symmetry* equals

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

where  $X^g$  are the elements fixed by  $g$  ( $g.x = x$ ).

If  $f(n)$  counts “configurations” (of some sort) of length  $n$ , we can ignore rotational symmetry using  $G = \mathbb{Z}_n$  to get

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

## 6.2 Partitions and subsets

$$\prod_{n=1}^{\infty} (1 - x^n) = 1 + \sum_{k=1}^{\infty} (-1)^k \left( x^{k(3k+1)/2} + x^{k(3k-1)/2} \right)$$

### 6.2.1 Partition function

Number of ways of writing  $n$  as a sum of positive integers, disregarding the order of the summands.

$$p(0) = 1, \quad p(n) = \sum_{k \in \mathbb{Z} \setminus \{0\}} (-1)^{k+1} p(n - k(3k - 1)/2)$$

$$p(n) \sim 0.145/n \cdot \exp(2.56\sqrt{n})$$

$\frac{n}{p(n)}$	0	1	2	3	4	5	6	7	8	9	20	50	100
	1	1	2	3	5	7	11	15	22	30	627	~2e5	~2e8

### 6.2.2 Lucas' Theorem

Let  $n, m$  be non-negative integers and  $p$  a prime. Write  $n = n_k p^k + \dots + n_1 p + n_0$  and  $m = m_k p^k + \dots + m_1 p + m_0$ . Then  $\binom{n}{m} \equiv \prod_{i=0}^k \binom{n_i}{m_i} \pmod{p}$ .

### 6.2.3 Binomials

#### multinomial.h

**Description:** Computes  $\binom{k_1 + \dots + k_n}{k_1, k_2, \dots, k_n} = \frac{(\sum k_i)!}{k_1! k_2! \dots k_n!}$ . 62d05k6, 6 lines

```
ll multinomial(vi& v) {
    ll c = 1, m = v.empty() ? 1 : v[0];
    fwd(i, 1, sz(v)) rep(j, v[i])
        c = c * ++m / (j+1);
    return c;
}
```

## 6.3 General purpose numbers

### 6.3.1 Bernoulli numbers

EGF of Bernoulli numbers is  $B(t) = \frac{t}{e^t - 1}$  (FFT-able).

$$B[0, \dots] = [1, -\tfrac{1}{2}, \tfrac{1}{6}, 0, -\tfrac{1}{30}, 0, \tfrac{1}{42}, \dots]$$

Sums of powers:

$$\sum_{i=1}^n n^m = \frac{1}{m+1} \sum_{k=0}^m \binom{m+1}{k} B_k \cdot (n+1)^{m+1-k}$$

Euler-Maclaurin formula for infinite sums:

$$\begin{aligned} \sum_{i=m}^{\infty} f(i) &= \int_m^{\infty} f(x) dx - \sum_{k=1}^{\infty} \frac{B_k}{k!} f^{(k-1)}(m) \\ &\approx \int_m^{\infty} f(x) dx + \frac{f(m)}{2} - \frac{f'(m)}{12} + \frac{f'''(m)}{720} + O(f^{(5)}(m)) \end{aligned}$$

### 6.3.2 Stirling numbers of the first kind

Number of permutations on  $n$  items with  $k$  cycles.

$$c(n, k) = c(n-1, k-1) + (n-1)c(n-1, k), \quad c(0, 0) = 1$$

$$\sum_{k=0}^n c(n, k) x^k = x(x+1) \dots (x+n-1)$$

$$\begin{aligned} c(8, k) &= \\ 8, 0, 5040, 13068, 13132, 6769, 1960, 322, 28, 1 \\ c(n, 2) &= \\ 0, 0, 1, 3, 11, 50, 274, 1764, 13068, 109584, \dots \end{aligned}$$

### 6.3.3 Eulerian numbers

Number of permutations  $\pi \in S_n$  in which exactly  $k$  elements are greater than the previous element.  $k$   $j$ :s s.t.  $\pi(j) > \pi(j+1)$ ,  $k+1$   $j$ :s s.t.  $\pi(j) \geq j$ ,  $k$   $j$ :s s.t.  $\pi(j) > j$ .

$$E(n, k) = (n-k)E(n-1, k-1) + (k+1)E(n-1, k)$$

$$E(n, 0) = E(n, n-1) = 1$$

$$E(n, k) = \sum_{j=0}^k (-1)^j \binom{n+1}{j} (k+1-j)^n$$

### 6.3.4 Stirling numbers of the second kind

Partitions of  $n$  distinct elements into exactly  $k$  groups.

$$S(n, k) = S(n-1, k-1) + kS(n-1, k)$$

$$S(n, 1) = S(n, n) = 1$$

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

### 6.3.5 Bell numbers

Total number of partitions of  $n$  distinct elements.  $B(n) = 1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, \dots$ . For  $p$  prime,

$$B(p^m + n) \equiv mB(n) + B(n+1) \pmod{p}$$

$$B(n) = \sum_{k=0}^n \binom{n}{k} \cdot B(k)$$

### 6.3.6 Labeled unrooted trees

# on  $n$  vertices:  $n^{n-2}$

# on  $k$  existing trees of size  $n_i$ :

$$n_1 n_2 \dots n_k n^{k-2}$$

# with degrees  $d_i$ :

$$(n-2)! / ((d_1-1)! \dots (d_n-1)!)$$

### 6.3.7 Catalan numbers

$$C_n = \frac{1}{n+1} \binom{2n}{n} = \binom{2n}{n} - \binom{2n}{n+1} = \frac{(2n)!}{(n+1)!n!}$$

$$C_0 = 1, \quad C_{n+1} = \frac{2(2n+1)}{n+2} C_n, \quad C_{n+1} = \sum C_i C_{n-i}$$

$$C_n = 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786$$

- sub-diagonal monotone paths in an  $n \times n$  grid.
- strings with  $n$  pairs of parenthesis, correctly nested.
- binary trees with with  $n+1$  leaves (0 or 2 children).
- ordered trees with  $n+1$  vertices.



- ways a convex polygon with  $n + 2$  sides can be cut into triangles by connecting vertices with straight lines.
- permutations of  $[n]$  with no 3-term increasing subseq.

Catalan convolution: find the count of balanced parentheses sequences consisting of  $n + k$  pairs of parentheses where the first  $k$  symbols are open brackets.

$$C^k = \frac{k+1}{n+k+1} \binom{2n+k}{n}$$

### 6.3.8 LGV Lemma

- $G$  - DAG,  $A = \{a_1, \dots, a_n\}$ ,  $B = \{b_1, \dots, b_n\}$  - subsets of vertices,  $w_e$  - edge weights.
- $\omega(P)$  - path weight, the product of edge weights in that path.
- Let  $M_{a,b} = \sum_{P:a \rightarrow b} \omega(P)$  be the sum of path weights over all possible paths from  $a$  to  $b$  (when unit weights, note this is the number of paths).
- Let  $n$ -tuple of paths  $\mathcal{P} = (P_1, \dots, P_n) : A \rightarrow B$  be the set of non-intersecting (by vertices, including also endpoints) paths from  $A$  to  $B$ . There exists  $\sigma(\mathcal{P})$ , such that  $P_i \in a_i \rightarrow b_{\sigma_i}$ .

Lemma:  $\det(M) = \sum_{(P_1, \dots, P_n) : A \rightarrow B} \operatorname{sgn}(\sigma(\mathcal{P})) \prod_{i=1}^n \omega(P_i)$

Particularly useful when only identity permutation is possible.

### 6.4 Other DeBruijn.h

**Description:** Recursive FKM, given alphabet  $[0, k]$  constructs cyclic string of length  $k^n$  that contains every length  $n$  string as substr.

```
7794a7, 13 lines
vi dseq(int k, int n) {
    if (k == 1) return {0};
    vi res, aux(n+1);
    function<void(int,int)> gen = [&](int t, int p) {
        if (t > n) { // consider lyndon word of len p
            if (n%p == 0) FOR(i,1,p+1) res.push_back(aux[i]);
        } else {
            aux[t] = aux[t-p]; gen(t+1,p);
            FOR(i,aux[t-p]+1,k) aux[t] = i, gen(t+1,t);
        }
    };
    gen(1,1); return res;
}
```

### NimProduct.h

**Description:** Nim Product.

```
9bba25, 17 lines
using ull = uint64_t;
ull _nimProd2[64][64];
ull nimProd2(int i, int j) {
    if (_nimProd2[i][j]) return _nimProd2[i][j];
    if ((i & j) == 0) return _nimProd2[i][j] = 1ull << (i | j);
    int a = (i&j) & ~(i&j);
    return _nimProd2[i][j] = nimProd2(i ^ a, j) ^ nimProd2((i ^ a) | (a-1), (j ^ a) | (i & (a-1)));
}
ull nimProd(ull x, ull y) {
    ull res = 0;
    for (int i = 0; (x >> i) && i < 64; i++)
        if ((x >> i) & 1)
            for (int j = 0; (y >> j) && j < 64; j++)
                if ((y >> j) & 1)
```

```
res ^= nimProd2(i, j);
return res;
}
```

### PermGroup.h

**Description:** Schreier-Sims lets you add a permutation to a group, count number of permutations in a group, test whether a permutation is a member of a group. Works well for  $n \leq 15$ , maybe for larger too. Construct PermGroup() and run order() to get order of the group.

```
f705c5, 54 lines
Time: O(n^6)

vi inv(vi v) { vi V(sz(v)); rep(i,sz(v)) V[v[i]]=i; return V; }
vi id(int n) { vi v(n); iota(all(v),0); return v; }
vi operator*(const vi& a, const vi& b) {
    vi c(sz(a)); rep(i,sz(a)) c[i] = a[b[i]];
    return c;
}

struct PermGroup {
    struct Group {
        vi flag;
        vector<vi> gen, sigma;
        Group(int n, int p) : flag(n), sigma(n) {
            flag[p] = 1; sigma[p] = id(n);
        }
    };
    vector<Group> g;
    int n = 0; vector<Group> g;
    PermGroup() {}
    bool check(const vi& cur, int k) {
        if (!k) return 1;
        int t = cur[k];
        return g[k].flag[t] ? check(inv(g[k].sigma[t])*cur, k-1) : 0;
    }
    void updateX(const vi& cur, int k) {
        int t = cur[k]; // if flag, fixes k -> k
        if (g[k].flag[t]) ins(inv(g[k].sigma[t])*cur, k-1);
        else {
            g[k].flag[t] = 1, g[k].sigma[t] = cur;
            for(auto x: g[k].gen) updateX(x*cur, k);
        }
    }
    void ins(const vi& cur, int k) {
        if (check(cur, k)) return;
        g[k].gen.push_back(cur);
        rep(i, n) if (g[k].flag[i]) updateX(cur*g[k].sigma[i], k);
    }
    ll order(vector<vi> gen) {
        if (sz(gen) == 0) return 1;
        n = sz(gen[0]);
        rep(i, n) g.push_back(Group(n, i));
        for(auto a: gen)
            by one
            ll tot = 1; // watch out for overflows, can be up to n!
            rep(i, n) {
                int cnt = 0;
                rep(j, i+1) cnt += g[i].flag[j];
                tot *= cnt;
            }
        return tot;
    }
};
```

### GrayCode.h

**Description:** Gray code: gray(0), ..., gray( $2^n - 1$ ) - permutation in which each two consecutive (cyclically) numbers differ in exactly one bit.

```
3b77ab, 4 lines
unsigned gray(unsigned n) { return n^n>>1; }
unsigned igray(unsigned n) {
    n^n>>1; n^n>>2; n^n>>4; n^n>>8; n^n>>16;
    return n;
}
```

## Graph (7)

### 7.1 Fundamentals

#### BellmanFord.h

**Description:** Calculates shortest paths from  $s$  in a graph that might have negative edge weights. Unreachable nodes get dist = inf; nodes reachable through negative-weight cycles get dist = -inf. Assumes  $V^2 \max |w_i| < 2^{63}$ .

```
Time: O(VE) 5091d0, 23 lines

const ll inf = LLONG_MAX;
struct Ed { int a, b, w, s() { return a < b ? a : -a; } };
void bellmanFord(vector<Node>& nodes, vector<Ed>& eds, int s) {
    nodes[s].dist = 0;
    sort(all(eds), [](Ed a, Ed b) { return a.s() < b.s(); });
    int lim = sz(nodes) / 2 + 2; // /3+100 with shuffled vertices
    rep(i, lim) for (Ed ed : eds) {
        Node cur = nodes[ed.a], &dest = nodes[ed.b];
        if (abs(cur.dist) == inf) continue;
        ll d = cur.dist + ed.w;
        if (d < dest.dist) {
            dest.prev = ed.a;
            dest.dist = d;
        }
    }
    rep(i, lim) for (Ed e : eds) {
        if (nodes[e.a].dist == -inf)
            nodes[e.b].dist = -inf;
    }
}
```

### FloydWarshall.h

**Description:** Calculates all-pairs shortest path in a directed graph that might have negative edge weights. Input is an distance matrix  $m$ , where  $m[i][j] = \text{inf}$  if  $i$  and  $j$  are not adjacent. As output,  $m[i][j]$  is set to the shortest distance between  $i$  and  $j$ , inf if no path, or -inf if the path goes through a negative-weight cycle.

```
Time: O(N^3) 277cec, 12 lines

const ll inf = 1LL << 62;
void floydWarshall(vector<vector<ll>>& m) {
    int n = sz(m);
    rep(i, n) m[i][i] = min(m[i][i], 0LL);
    rep(k, n) rep(i, n) rep(j, n)
        if (m[i][k] != inf && m[k][j] != inf) {
            auto newDist = max(m[i][k] + m[k][j], -inf);
            m[i][j] = min(m[i][j], newDist);
        }
    rep(k, n) if (m[k][k] < 0) rep(i, n) rep(j, n)
        if (m[i][k] != inf && m[k][j] != inf) m[i][j] = -inf;
}
```

### 7.2 Network flow

#### PushRelabel.h

**Description:** Push-relabel using the highest label selection rule and the gap heuristic. Quite fast in practice. To obtain the actual flow, look at positive values only.

```
Time: O(V^2\sqrt{E}) e964dd, 48 lines

struct PushRelabel {
    struct Edge {
        int dest, back;
        ll f, c;
    };
    vector<vector<Edge>> g;
    vector<ll> ec;
    vector<Edge>& cur;
    vector<vi> hs; vi H;
    PushRelabel(int n) : g(n), ec(n), cur(n), hs(2*n), H(n) {}
    void addEdge(int s, int t, ll cap, ll rcap=0) {
        if (s == t) return;
        g[s].push_back({t, sz(g[t]), 0, cap});
        g[t].push_back({s, sz(g[s])-1, 0, rcap});
    }
    void addFlow(Edge& e, ll f) {
        Edge sback = g[e.dest][e.back];
        if (!ec[e.dest] && f) hs[H[e.dest]].push_back(e.dest);
        e.f += f; e.c -= f; ec[e.dest] += f;
        back.f -= f; back.c += f; ec[back.dest] -= f;
    }
    ll calc(int s, int t) {
        int v = sz(g); H[s] = v; ec[t] = 1;
        vi co(2*v); co[0] = v-1;
```

```
rep(i,v) cur[i] = g[i].data();
for (Edge& e : g[s]) addFlow(e, e.c);
for (int hi = 0; ) {
    while (hs[hi].empty()) if (!hi--) return -ec[s];
    int u = hs[hi].back(); hs[hi].pop_back();
    while (ec[u] > 0) // discharge u
        if (cur[u] == g[u].data() + sz(g[u])) {
            H[u] = le9;
            for (Edge& e : g[u]) if (e.c && H[u] > H[e.dest]+1)
                H[u] = H[e.dest]+1, cur[u] = &e;
            if (++co[H[u]], !--co[hi] && hi < v)
                rep(i,v) if (hi < H[i] && H[i] < v)
                    --co[H[i]], H[i] = v + 1;
            hi = H[u];
        } else if (cur[u]->c && H[u] == H[cur[u]->dest]+1)
            addFlow(*cur[u], min(ec[u], cur[u]->c));
        else ++cur[u];
    }
}
bool leftOfMinCut(int a) { return H[a] >= sz(g); }
```

### MinCostKFlowFast.h

**Description:** Min cost K-flow. Supports fast 1st phase distance computation

**Time:**  $O(INIT + F * n \log n)$  INIT  $\leq V * E$  and depends on first dist computation

```
641a19, 70 lines
#include <bits/extc++.h>
struct MCMF {
    const ll INF = 2e18;
    struct edge {
        int from, to, rev; ll cap, cost, flow;
    };
    int N;
    vector<vector<edge>> ed;
    vi seen;
    vector<ll> dist, pi;
    vector<edge*> par;
    MCMF(int N) : N(N), ed(N), seen(N), dist(N), pi(N), par(N) {}
    void addEdge(int from, int to, ll cap, ll cost) {
        if (from == to) return;
        ed[from].push_back(edge{ from, to, sz(ed[to]), cap, cost, 0 });
        ed[to].push_back(edge{ to, from, sz(ed[from])-1, 0, -cost, 0 });
    }
    void path(int s) {
        fill(all(seen), 0); fill(all(dist), INF);
        dist[s] = 0; ll di;
        __gnu_pbds::priority_queue<pair<ll, int>> q;
        vector<decltype(q)::point_iterator> its(N);
        q.push({0, s});
        while (!q.empty()) {
            s = q.top().second; q.pop();
            seen[s] = 1; di = dist[s] + pi[s];
            for (edge& e : ed[s]) if (!seen[e.to]) {
                ll val = di - pi[e.to] + e.cost;
                if (e.cap - e.flow > 0 && val < dist[e.to]) {
                    dist[e.to] = val;
                    par[e.to] = &e;
                    if (its[e.to] == q.end())
                        its[e.to] = q.push({ -dist[e.to], e.to });
                    else
                        q.modify(its[e.to], { -dist[e.to], e.to });
                }
            }
        }
        rep(i, N) pi[i] = min(pi[i] + dist[i], INF);
    }
    pair<ll, ll> maxflow(int s, int t, ll k = -1) {
        if (k == -1) k = INF;
        ll totflow = 0, totcost = 0;
        while (path(s), seen[t]) {
            ll fl = k - totflow;
            for (edge& x = par[t]; x; x = par[x->from])
                fl = min(fl, x->cap - x->flow);
            totflow += fl;
            for (edge& x = par[t]; x; x = par[x->from]) {
                x->flow += fl;
                ed[x->to][x->rev].flow -= fl;
            }
            if (totflow == k) break;
        }
        rep(i, N) for (edge& e : ed[i]) totcost += e.cost * e.flow;
        return {totflow, totcost/2};
    }
}
```

```
// If some costs can be negative, call this before
maxflow:
void setpi(int s) { // (otherwise, leave this out)
fill(all(pi), INF); pi[s] = 0;
int it = N, ch = 1; ll v;
while (ch-- && it--)
    rep(i,N) if (pi[i] != INF)
        for (edge& e : ed[i]) if (e.cap)
            if ((v = pi[i] + e.cost) < pi[e.to])
                pi[e.to] = v, ch = 1;
assert(it >= 0); // negative cost cycle
}
};
```

## Dinic.h

**Description:** Flow algorithm with complexity  $O(VE \log U)$  where  $U = \max|\text{cap}|$ .  $O(\min(E^{1/2}, V^{2/3})E)$  if  $U = 1$ ;  $O(\sqrt{VE})$  for bipartite matching.

```
struct Dinic {
    struct Edge {
        int to, rev;
        ll c, oc;
        ll flow() { return max(oc - c, 0LL); } // if you
            need flows
    };
    vi lvl, ptr, q;
    vector<vector<Edge>> adj;
    Dinic(int n) : lvl(n), ptr(n), q(n), adj(n) {}
    void addEdge(int a, int b, ll c, ll rcap = 0) {
        adj[a].push_back({b, sz(adj[b]), c, c});
        adj[b].push_back({a, sz(adj[a]) - 1, rcap, rcap});
    }
    ll dfs(int v, int t, ll f) {
        if (v == t || !f) return f;
        for (int& i = ptr[v]; i < sz(adj[v]); i++) {
            Edge& e = adj[v][i];
            if (lvl[e.to] == lvl[v] + 1)
                if (ll p = dfs(e.to, t, min(f, e.c))) {
                    e.c -= p, adj[e.to][e.rev].c += p;
                    return p;
                }
        }
        return 0;
    }
    ll calc(int s, int t) {
        ll flow = 0; q[0] = s;
        rep(L,31) do { // 'int L=30' maybe faster for
            random data
            lvl = ptr = vi(sz(q));
            int qi = 0, qe = lvl[s] = 1;
            while (qi < qe && !lvl[t]) {
                int v = q[qi++];
                for (Edge e : adj[v])
                    if (!lvl[e.to] && e.c >> (30 - L))
                        q[qi++] = e.to, lvl[e.to] = lvl[v] + 1;
            }
            while ((ll p = dfs(s, t, LLONG_MAX)) flow += p;
            while (lvl[t]);
            return flow;
        }
        bool leftOfMinCut(int a) { return lvl[a] != 0; }
    };
};
```

## GlobalMinCut.h

**Description:** Find a global minimum cut in an undirected graph, as represented by an adjacency matrix.

**Time:**  $O(V^3)$

```
pair<int, vi> globalMinCut(vector<vi> mat) {
pair<int, vi> best = {INT_MAX, {}};
int n = sz(mat);
vector<vi> co(n);
rep(i,n) co[i] = {i};
fwd(ph,1,n) {
    vi w = mat[0];
    size_t s = 0, t = 0;
    fwd(it,0,n-ph) { // O(V^2) -> O(E log V) with prio.
        queue
        w[t] = INT_MIN;
        s = t, t = max_element(all(w)) - w.begin();
        rep(i,n) w[i] += mat[t][i];
    }
    best = min(best, {w[t] - mat[t][t], co[t]});
    co[s].insert(co[s].end(), all(co[t]));
    rep(i,n) mat[s][i] += mat[t][i];
    rep(i,n) mat[i][s] = mat[s][i];
    mat[0][t] = INT_MIN;
}
};
```

```
return best;
}

GomoryHu.h
Description: Given a list of edges representing an undirected
flow graph, returns edges of the Gomory-Hu tree. The max
flow between any pair of vertices is given by minimum edge
weight along the Gomory-Hu tree path.
Time:  $O(V)$  Flow Computations
"Dinic.h" 1647b0, 13 lines

typedef array<ll, 3> Edge;
vector<Edge> gomoryHu(int N, vector<Edge> ed) {
vector<Edge> tree;
vi par(N);
fwd(1,1,N) {
    Dinic D(N); // Dinic also works
    for (Edge t : ed) D.addEdge(t[0], t[1], t[2], t[2])
        ;
    tree.push_back({i, par[i], D.calc(i, par[i])});
    fwd(j,i+1,N)
        if (par[j] == par[i] && D.leftOfMinCut(j)) par[j]
            = i;
    return tree;
}
};
```

### Flow with demands

Say we want  $d(e) \leq f(e) \leq c(e)$  for each edge. To find an arbitrary flow, add  $s', t'$  and the following edges:

- $\forall v \in V : c'((s', v)) = \sum_w d((u, v)), \quad c'((v, t')) = \sum_w d((v, w)),$
- $\forall (u, v) \in E : c'((u, v)) = c((u, v)) - d((u, v)),$
- $c'((t, s)) = \infty.$

For min flow, replace  $\infty$  with  $L$  and find smallest  $L$  such that flow is saturated.

### 7.3 Matching

#### TurboMatching.h

**Description:** match[v] = vert matched to v or -1, returns num edges in matching

**Time:**  $O(?)$

```
ae2b0d, 33 lines

int matching(vector<vi>& G, vi& match) {
vector<bool> seen; int n = 0, k = 1;
match.assign(sz(G), -1);
auto dfs = [&](auto f, int i) -> int {
    if (seen[i]) { return 0; } seen[i] = 1;
    for (auto e : G[i]) {
        if (match[e] < 0 || f(f, match[e])) {
            match[i] = e; match[e] = i; return 1;
        }
    }
    return 0;
};
while (k) {
    seen.assign(sz(G), 0); k = 0;
    rep(i, sz(G)) if (match[i] < 0) k += dfs(dfs, i);
    n += k;
}
return n;
}

vi vertexCover(vector<vi>& G, vi& match) {
vi ret, col(sz(G)), seen(sz(G));
auto dfs = [&](auto f, int i, int c) -> void {
    if (col[i]) { return; } col[i] = c+1;
    for (auto e : G[i]) seen[e] = 1, f(f, match[e]);
};
rep(i, sz(G)) dfs(dfs, i, 0);
rep(i, sz(G)) if (match[i]<0) aug(aug, i);
rep(i, sz(G)) if (seen[i]==col[i]-1) ret.push_back(i);
return ret;
};
```

## BoskiMatching.h

**Description:** Bosek's algorithm for partially online bipartite maximum matching - white vertices (right side) are fixed, black vertices (left) are added one by one. • match[v] = index of black vertex matched to white vertex v or -1 if unmatched

- Black vertices are indexed in order they were added, from 0.

**Time:**  $O(E\sqrt{V})$

```
struct Matching : vi { // Usage: Matching match(
    num_white);
vector<vi> adj; vi rank, low, pos, vis, seen; int k{0
};
Matching(int n = 0) : vi(n, -1), rank(n) {}
bool add(vi vec) { //match.add(
    indices_of_white_neighbours);
adj.push_back(move(vec));
low.push_back(0); pos.push_back(0); vis.push_back
(0);
if (!adj.back().empty()) {
    int i = k;
    nxt:
    seen.clear();
    if (dfs(sz(adj)-1, ++k-i)) return 1;
    for(auto v: seen) for(auto e: adj[v])
        if (rank[e] < le9 && vis[at(e)] < k)
            goto nxt;
    for(auto v: seen) for(auto w: adj[v])
        rank[w] = low[v] = le9;
    return 0;
} //returns 1 if matching size increased
bool dfs(int v, int g) {
    if (vis[v] < k) vis[v] = k, seen.push_back(v);
    while (low[v] < g) {
        int e = adj[v][pos[v]];
        if (at(e) != v && low[v] == rank[e]) {
            rank[e]++;
            if (at(e) == -1 || dfs(at(e), rank[e]))
                return at(e) = v, 1;
            } else if (++pos[v] == sz(adj[v])) {
                pos[v] = 0, low[v]++;
            }
        }
        return 0; }
};
```

## WeightedMatching.h

**Description:** Given a weighted bipartite graph, matches every node on the left with a node on the right such that no nodes are in two matchings and the sum of the edge weights is minimal. Takes cost[N][M], where cost[i][j] = cost for L[i] to be matched with R[j] and returns (min cost, match), where L[i] is matched with R[match[i]]. Negate costs for max cost. Requires  $N \leq M$ .

**Time:**  $O(N^2M)$

```
deee37, 31 lines

pair<int, vi> hungarian(const vector<vi> &a) {
    if (a.empty()) return {0, {}};
    int n = sz(a) + 1, m = sz(a[0]) + 1;
    vi u(n), v(m), p(m), ans(n - 1);
    fwd(1,1,n) {
        p[0] = i;
        int j0 = 0; // add "dummy" worker 0
        vi dist(m, INT_MAX), pre(m, -1);
        vector<bool> done(m + 1);
        do { // dijkstra
            done[j0] = true;
            int i0 = p[j0], j1, delta = INT_MAX;
            fwd(j,1,m) if (!done[j]) {
                auto cur = a[i0 - 1][j - 1] - u[i0] - v[j];
                if (cur < dist[j]) dist[j] = cur, pre[j] = j0;
                if (dist[j] <= delta) delta = dist[j], j1 = j;
            }
            rep(j,m) {
                if (done[j]) u[p[j]] += delta, v[j] -= delta;
                else dist[j] -= delta;
            }
            j0 = j1;
        } while (p[j0]);
        while (j0) { // update alternating path
            int j1 = pre[j0];
            p[j0] = p[j1], j0 = j1;
        }
    }
    fwd(j,1,m) if (p[j]) ans[p[j] - 1] = j - 1;
    return {-v[0], ans}; // min cost
};
```

## GeneralMatching.h

**Description:** Matching for general graphs using Blossom algorithm.

**Time:**  $O(NM, \text{surprisingly fast in practice})$

```
3f5cfa, 52 lines

vi Blossom(vector<vi> &graph) {
    int n = sz(graph), timer = -1;
    vi mate(n, -1), label(n), parent(n),
        orig(n), aux(n, -1), q;
    auto lca = [&](int x, int y) {
```

```
for (timer++; ; swap(x, y)) {
    if (x == -1) continue;
    if (aux[x] == timer) return x;
    aux[x] = timer;
    x = (mate[x] == -1 ? -1 : orig[parent[mate[x]]]);
}
};
auto blossom = [&](int v, int w, int a) {
    while (orig[v] != a) {
        parent[v] = w; w = mate[v];
        if (label[w] == 1) label[w] = 0, q.push_back(w);
        orig[v] = orig[w] = a; v = parent[w];
    }
};
auto augment = [&](int v) {
    while (v != -1) {
        int pv = parent[v], nv = mate[pv];
        mate[v] = pv; mate[pv] = v; v = nv;
    }
};
auto bfs = [&](int root) {
    fill(all(label), -1);
    iota(all(orig), 0);
    q.clear();
    label[root] = 0; q.push_back(root);
    for (int i = 0; i < (int)q.size(); ++i) {
        int v = q[i];
        for (auto x : graph[v]) {
            if (label[x] == -1) {
                label[x] = 1; parent[x] = v;
                if (mate[x] == -1)
                    return augment(x), 1;
                label[mate[x]] = 0; q.push_back(mate[x]);
            } else if (label[x] == 0 && orig[v] != orig[x])
                {
                    int a = lca(orig[v], orig[x]);
                    blossom(x, v, a); blossom(v, x, a);
                }
        }
    }
    return 0;
};
// Time halves if you start with (any) maximal
// matching.
for (int i = 0; i < n; i++)
    if (mate[i] == -1)
        bfs(i);
return mate;
};
```

## MatroidIntersection.h

**Description:** Find largest subset S of [n] such that S is independent in both matroid A and B, given by their oracles, see example implementations below. Returns vector V such that V[i] = 1 iff i-th element is included in found set;

**Time:**  $O(r^2 \cdot (\text{init} + n \cdot \text{add}))$ , where r is max independent set.

```
fe424f, 149 lines

template<class T, class U>
vector<bool> intersectMatroids(T& A, U& B, int n) {
vector<bool> ans(n);
bool ok = 1;
// NOTE: for weighted matroid intersection find
// shortest augmenting paths
// first by weight change, then by length using
// Bellman-Ford,
// Speedup trick (only for unweighted):
A.init(ans); B.init(ans);
rep(i, n)
    if (A.canAdd(i) && B.canAdd(i))
        ans[i] = 1, A.init(ans), B.init(ans); //End of
        speedup
while (ok) {
    vector<vi> G(n);
    vector<bool> good(n);
    queue<int> que;
    vi prev(n, -1);
    A.init(ans); B.init(ans); ok = 0;
    rep(i, n) if (!ans[i]) {
        if (A.canAdd(i)) que.push(i), prev[i] = -2;
        good[i] = B.canAdd(i);
    }
    rep(i, n) if (ans[i]) {
        ans[i] = 0;
        A.init(ans); B.init(ans);
        rep(j, n) if (i != j && !ans[j]) {
            if (A.canAdd(j)) G[i].push_back(j); // -cost[j]
            if (B.canAdd(j)) G[j].push_back(i); // cost[i]
        }
        ans[i] = 1;
    }
```

```

}
while (!que.empty()) {
    int i = que.front();
    que.pop();
    if (good[i]) { // best found (unweighted =
        shortest path)
        ans[i] = 1;
        while (prev[i] >= 0) { // alternate matching
            ans[i] = prev[i] == 0;
            ans[i] = prev[i] == 1;
        }
        ok = 1; break;
    }
    for(auto j: G[i]) if (prev[j] == -1)
        que.push(j), prev[j] = i;
}
return ans;
}
// Matroid where each element has color
// and set is independent iff for each color c
// #(elements of color c) <= maxAllowed[c].
struct LimOracle {
    vi color; // color[i] = color of i-th element
    vi maxAllowed; // Limits for colors
    vi tmp;
    // Init oracle for independent set S; O(n)
    void init(vector<bool>& S) {
        tmp = maxAllowed;
        rep(i, sz(S)) tmp[color[i]] -= S[i];
    }
    // Check if S+[k] is independent; time: O(1)
    bool canAdd(int k) { return tmp[color[k]] > 0; }
};
// Graphic matroid - each element is edge,
// set is independent iff subgraph is acyclic.
struct GraphOracle {
    vector<pii> elems; // Ground set: graph edges
    int n; // Number of vertices, indexed [0;n-1]
    vi par;
    int find(int i) {
        return par[i] == -1 ? i : par[i] = find(par[i]);
    }
    // Init oracle for independent set S; ~O(n)
    void init(vector<bool>& S) {
        par.assign(n, -1);
        rep(i, sz(S)) if (S[i])
            par[find(elems[i].st)] = find(elems[i].nd);
    }
    // Check if S+[k] is independent; time: ~O(1)
    bool canAdd(int k) {
        return find(elems[k].st) != find(elems[k].nd);
    }
};
// Co-graphic matroid - each element is edge,
// set is independent iff after removing edges
// from graph number of connected components
// doesn't change.
struct CographOracle {
    vector<pii> elems; // Ground set: graph edges
    int n; // Number of vertices, indexed [0;n-1]
    vector<vi> G;
    vi pre, low;
    int cnt;
    int dfs(int v, int p) {
        pre[v] = low[v] = ++cnt;
        for(auto e: G[v]) if (e != p)
            low[v] = min(low[v], pre[e] ? : dfs(e, v));
        return low[v];
    }
    // Init oracle for independent set S; O(n)
    void init(vector<bool>& S) {
        G.assign(n, {});
        pre.assign(n, 0);
        low.resize(n);
        cnt = 0;
        rep(i, sz(S)) if (!S[i]) {
            pii e = elems[i];
            G[e.st].push_back(e.nd);
            G[e.nd].push_back(e.st);
        }
        rep(v, n) if (!pre[v]) dfs(v, -1);
    }
    // Check if S+[k] is independent; time: O(1)
    bool canAdd(int k) {
        pii e = elems[k];
        return max(pre[e.st], pre[e.nd]) != max(low[e.st],
            low[e.nd]);
    }
};
// Matroid equivalent to linear space with XOR
struct XorOracle {

```

```

vector<ll> elems; // Ground set: numbers
vector<ll> base;
// Init for independent set S; O(n+r^2)
void init(vector<bool>& S) {
    base.assign(63, 0);
    rep(i, sz(S)) if (S[i]) {
        ll e = elems[i];
        rep(j, sz(base)) if ((e >> j) & 1) {
            if (!base[j]) {
                base[j] = e;
                break;
            }
            e ^= base[j];
        }
    }
    // Check if S+[k] is independent; time: O(r)
    bool canAdd(int k) {
        ll e = elems[k];
        rep(i, sz(base)) if ((e >> i) & 1) {
            if (!base[i]) return 1;
            e ^= base[i];
        }
        return 0;
    }
};
};

```

## 7.4 DFS algorithms

### Dominators.h

**Description:** Tarjan's dominators in directed graph Returns tree (as array of parents) of immediate dominators idom. idom[root] = root, idom[v] = -1 if v is unreachable from root  
**Time:**  $\mathcal{O}(|E|\log|V|)$

```

vi dominators(vector<vi>& G, int root) {
    int n = sz(G); vector<vi> in(n), bucket(n);
    vi pre(n, -1), anc(n, -1), par(n), best(n);
    vi ord, idom(n, -1), sdom(n, n), rdom(n);
    auto dfs = [&](auto f, int v, int p) -> void {
        if (pre[v] == -1) {
            par[v] = p; pre[v] = sz(ord);
            ord.push_back(v);
            for (auto e : G[v])
                in[e].push_back(v), f(f, e, v);
        }
        auto find = [&](auto f, int v) -> pii {
            if (anc[v] == -1) return {best[v], v};
            int b; tie(b, anc[v]) = f(f, anc[v]);
            if (sdom[b] < sdom[best[v]]) best[v] = b;
            return {best[v], anc[v]};
        };
        rdom[root] = idom[root] = root;
        iota(all(best), 0); dfs(dfs, root, -1);
        rep(i, sz(ord)) {
            int v = ord[sz(ord)-i-1], b = pre[v];
            for (auto e : in[v])
                b = min(b, pre[e] < pre[v] ? pre[e] :
                    sdom[find(find, e).st]);
            for (auto u : bucket[v]) rdom[u] = find(find, u).st;
            sdom[v] = b; anc[v] = par[v];
            bucket[ord[sdom[v]]].push_back(v);
        }
        for (auto v : ord) idom[v] = (rdom[v] == v ?
            ord[sdom[v]] : idom[rdom[v]]);
        return idom;
    };
}

```

### StronglyConnected.h

**Description:** SCC scc(graph), scc[v] = index of SCC of vertex v, scc.comps[i] = vertices of i-th scc, components in reverse topological order  
**Time:**  $\mathcal{O}(|E|+|V|)$

```

struct SCC : vi {
    vector<vi> comps; vi S; SCC() {}
    SCC(vector<vi>& G) : vi(sz(G)-1), S(sz(G)) {
        rep(i, sz(G)) if (!S[i]) dfs(G, i);
    }
    int dfs(vector<vi>& G, int v) {
        int low = S[v] = sz(S); S.push_back(v);
        for (auto e : G[v]) if (at(e) < 0)
            low = min(low, S[e] ? : dfs(G, e));
        if (low == S[v]) {
            comps.push_back({});
            fwd(i, S[v], sz(S)) {
                at(S[i]) = sz(comps)-1;
                comps.back().push_back(S[i]);
            }
            S.resize(S[v]);
        }
        return low;
    };
}

```

### Biconnected.h

**Description:** Biconnected bi(graph), bi[v] = indices of components containing v, bi.verts[i] = vertices of i-th component, bi.edges[i] = edges of i-th component. Bridges <=> components with 2 vertices, Articulation points <=> vertices in > 1 comp, Isolated vertex <=> empty component list

```

Time:  $\mathcal{O}(|E|+|V|)$  893123, 26 lines
struct Biconnected : vector<vi> {
    vector<vi> verts; vector<pii> S;
    vector<vector<pii>> edges;
    Biconnected() {}
    Biconnected(vector<vi>& G) : S(sz(G)) {
        resize(sz(G));
        rep(i, sz(G)) S[i].x ? : dfs(G, i, -1);
        rep(c, sz(verts)) for(auto v : verts[c])
            at(v).push_back(c);
    }
    int dfs(vector<vi>& G, int v, int p) {
        int low = S[v].x = sz(S)-1; S.push_back({v, -1});
        for (auto e : G[v]) if (e != p) {
            if (S[e].x < S[v].x) S.push_back({v, e});
            low = min(low, S[e].x ? : dfs(G, e, v));
        }
        if (p + 1 && low >= S[p].x) {
            verts.push_back({p}); edges.push_back({});
            fwd(i, S[v].x, sz(S)) {
                if (S[i].y == -1)
                    verts.back().push_back(S[i].x);
                else edges.back().push_back(S[i]);
            }
            S.resize(S[v].x);
        }
        return low;
    };
}

```

### 2Sat.h

**Description:** Variables are indexed from 1 and negations are negative indices.

**Usage:** SAT2 sat(variable.cnt), sat.solve(),  
 sat[i] = value of i-th variable (0 or 1),  
 (internally: i.false.id = 2i-2, i.true.id = 2i-1)  
**Time:**  $\mathcal{O}(|\text{SantaClauses}|+|\text{Variables}|)$  067670, 37 lines

```

struct SAT2 : vi {
    vector<vi> G; vi order, flags;
    SAT2(int n = 0) : G(n*2) {}
    void imply(int i, int j) { // Add (i => j) constraint
        i = i*2 ^ i >> 31; j = j*2 ^ j >> 31;
        G[--i].push_back(--j); G[j*1].push_back(i*1);
    } // Add (i v j) constraint
    void either(int i, int j) { imply(-i, j); }
    bool solve() { // Saves assignment in values[]
        assign(sz(G)/2+1, -1); flags.assign(sz(G), 0);
        rep(i, sz(G)) dfs(i);
        while (sz(order)) {
            if (!propag(order.back()^1, 1)) return 0;
            order.pop_back();
        }
        return 1;
    }
    void dfs(int i) {
        if (flags[i]) { return; } flags[i] = 1;
        for (auto e : G[i]) dfs(e);
        order.push_back(i);
    }
    bool propag(int i, bool first) {
        if (!flags[i]) { return 1; } flags[i] = 0;
        if (at(i/2+1) >= 0) return first;
        at(i/2+1) = i&1;
        for (auto e : G[i]) if (!propag(e, 0)) return 0;
        return 1;
    }
    // NEXT PART NOT ALWAYS NEEDED
    int addVar() { G.resize(sz(G)+2); return sz(G)/2; }
    void atMostOneTrue(vi& vars) {
        int y, x = addVar();
        for (auto i : vars) {
            imply(x, y = addVar());
            imply(i, -x); imply(i, x = y);
        }
    };
}

```

### EulerWalk.h

**Description:** Eulerian undirected/directed path/cycle algorithm. Input should be a vector of (dest, global edge index), where for undirected graphs, forward/backward edges have the same index. Returns a list of nodes in the Eulerian path/cycle with src at both start and end, or empty list if no cycle/path exists. To get edge indices back, add .second to s and ret.

```

Time:  $\mathcal{O}(V+E)$  780b64, 15 lines
vi eulerWalk(vector<vector<pii>>& gr, int nedges, int
    src=0) {
    int n = sz(gr);
    vi D(n), its(n), eu(nedges), ret, s = {src};
    D[src]++; // to allow Euler paths, not just cycles
    while (!s.empty()) {
        int x = s.back(), y, e, &it = its[x], end = sz(gr[x]
            );
        if (it == end){ ret.push_back(x); s.pop_back();
            continue; }
        tie(y, e) = gr[x][it++];
        if (!eu[e]) {
            D[x]--, D[y]++;
            eu[e] = 1; s.push_back(y);
        }
    }
    for (int x : D) if (x < 0 || sz(ret) != nedges+1)
        return {};
    return {ret.rbegin(), ret.rend()};
}

```

### KthShortest.h

**Description:** Eppsteins kth shortest path algorithm. Gives kth shortest (not necessarily simple) paths in  $\log k$  time, 64 lines

```

constexpr ll INF = 1e18;
struct Eppstein {
    using T = ll; using Edge = pair<int, T>;
    struct Node {
        int E[2] = {}, s = 0;
        Edge x;
    };
    T shortest; // Shortest path length
    priority_queue<pair<T, int>> Q;
    vector<Node> P[1]; vi h;
    Eppstein(vector<vector<Edge>>& G, int s, int t) {
        int n = sz(G);
        vector<vector<Edge>> H(n);
        rep(i, n) for(auto &e : G[i])
            H[e.st].push_back({i, e.nd});
        vi ord, par(n, -1);
        vector<T> d(n, -INF);
        Q.push({d[t] = 0, t});
        while (!Q.empty()) {
            auto v = Q.top(); Q.pop();
            if (d[v.nd] == v.st) {
                if (d[v.nd] == v.st) {
                    ord.push_back(v.nd);
                    for(auto &e : H[v.nd]) if (v.st-e.nd > d[e.st])
                        Q.push({d[e.st] = v.st-e.nd, e.st});
                    par[e.st] = v.nd;
                }
            }
        }
        if ((shortest = -d[s]) >= INF) return;
        h.resize(n);
        for(auto &v : ord) {
            int p = par[v];
            if (p+1) h[v] = h[p];
            for(auto &e : G[v]) if (d[e.st] > -INF) {
                T k = e.nd - d[e.st] + d[v];
                if (k || e.st != p) h[v] = push(h[v], {e.st, k}
                    );
                else p = -1;
            }
        }
        P[0].x.st = s; Q.push({0, 0});
    }
    int push(int t, Edge x) {
        P.push_back(P[t]);
        if (!P[t] = sz(P)-1).s || P[t].x.nd >= x.nd)
            swap(x, P[t].x);
        if (P[t].s) {
            int i = P[t].E[0], j = P[t].E[1];
            int d = P[i].s > P[j].s;
            int k = push(d ? j : i, x);
            P[t].E[d] = k; // Don't inline k!
        }
        P[t].s++;
        return t;
    }
    ll nextPath() { // next length, -1 if no next path
        if (Q.empty()) return -1;
        auto v = Q.top(); Q.pop();
        for (int i : P[v.nd].E) if (i)
            Q.push({ v.st-P[i].x.nd+P[v.nd].x.nd, i });
        int t = h[P[v.nd].x.st];
        if (t) Q.push({v.st - P[t].x.nd, t });
        return shortest - v.st;
    }
}

```



## 7.5 Coloring

### EdgeColoring.h

**Description:** Given a simple, undirected graph with max degree  $D$ , computes a  $(D + 1)$ -coloring of the edges such that no neighboring edges share a color. ( $D$ -coloring is NP-hard, but can be done for bipartite graphs by repeated matchings of max-degree nodes.)

**Time:**  $\mathcal{O}(NM)$

be7d13, 31 lines

```
vi edgeColoring(int N, vector<pii> eds) {
    vi cc(N + 1), ret(sz(eds)), fan(N), free(N), loc;
    for (pii e : eds) ++cc[e.first], ++cc[e.second];
    int u, v, ncols = *max_element(all(cc)) + 1;
    vector<vi> adj(N, vi(ncols, -1));
    for (pii e : eds) {
        tie(u, v) = e;
        fan[0] = v;
        loc.assign(ncols, 0);
        int at = u, end = u, d, c = free[u], ind = 0, i = 0;
        while (d = free[v], !loc[d] && (v = adj[u][d]) != -1)
            loc[d] = ++ind, cc[ind] = d, fan[ind] = v;
        cc[loc[d]] = c;
        for (int cd = d; at != -1; cd ^= c ^ d, at = adj[at][cd])
            swap(adj[at][cd], adj[end = at][cd ^ c ^ d]);
        while (adj[fan[i]][d] != -1) {
            int left = fan[i], right = fan[++i], e = cc[i];
            adj[u][e] = left;
            adj[left][e] = u;
            adj[right][e] = -1;
            free[right] = e;
        }
        adj[u][d] = fan[i];
        adj[fan[i]][d] = u;
        for (int y : {fan[0], u, end})
            for (int& z = free[y] = 0; adj[y][z] != -1; z++);
    }
    rep(i, sz(eds))
        for (tie(u, v) = eds[i]; adj[u][ret[i]] != v; ++ret[i]);
    return ret;
}
```

### ChordalGraph.h

**Description:** A graph is chordal if any cycle  $C \geq 4$  has a chord i.e. an edge  $(u, v)$  where  $u$  and  $v$  is in the cycle but  $(u, v)$  is not A perfect elimination ordering (PEO) in a graph is an ordering of the vertices of the graph such that,  $\forall v : v$  and its neighbors that occur after  $v$  in the order (later) form a clique. A graph is chordal if and only if it has a perfect elimination ordering. Optimal vertex coloring of the graph: first fit:  $col[i] =$  smallest color that is not used by any of the neighbours earlier in PEO. Max clique = Chromatic number =  $1 + \max$  over number of later neighbours for all vertices. Chromatic polynomial =  $(x - d_1)(x - d_2) \dots (x - d_n)$  where  $d_i =$  number of neighbors of  $i$  later in PEO.

**Time:**  $\mathcal{O}(n + m)$

137fe8, 38 lines

```
vi perfectEliminationOrder(vector<vi>& g) { // 0-indexed, adj list
    int top = 0, n = sz(g);
    vi ord, vis(n), indeg(n);
    vector<vi> bucket(n);
    rep(i, n) bucket[0].push_back(i);
    for (int i = 0; i < n; ++i) {
        while (bucket[top].empty()) --top;
        int u = bucket[top].back();
        bucket[top].pop_back();
        if (vis[u]) continue;
        ord.push_back(u);
        vis[u] = 1;
        ++i;
        for (int v : g[u]) {
            if (vis[v]) continue;
            bucket[++indeg[v]].push_back(v);
            top = max(top, indeg[v]);
        }
    }
    reverse(all(ord));
    return ord;
}

bool isChordal(vector<vi>& g, vi ord) { // ord = perfectEliminationOrder(g)
    int n = sz(g);
    set<pii> edg;
    rep(i, n) for (auto v : g[i]) edg.insert({i, v});
}
```

```
vi pos(n); rep(i, n) pos[ord[i]] = i;
rep(u, n) {
    int mn = n;
    for (auto v : g[u]) if (pos[u] < pos[v]) mn = min(mn, pos[v]);
    if (mn != n) {
        int p = ord[mn];
        for (auto v : g[u]) if (pos[v] > pos[u] && v != p && !edg.count({v, p})) return 0;
    }
}
return 1;
}
```

## 7.6 Heuristics

### MaximalCliques.h

**Description:** Runs a callback for all maximal cliques in a graph (given as a symmetric bitset matrix; self-edges not allowed). Callback is given a bitset representing the maximal clique.

**Time:**  $\mathcal{O}(3^{n/3})$ , much faster for sparse graphs

6effc5, 12 lines

```
typedef bitset<128> B;
template<class F>
void cliques(vector<B>& eds, F f, B P = ~B(), B X={}, B R={}) {
    if (!P.any()) { if (!X.any()) f(R); return; }
    auto q = (P | X)._Find_first();
    auto cand = P & ~eds[q];
    rep(i, sz(eds)) if (cand[i]) {
        R[i] = 1;
        cliques(eds, f, P & eds[i], X & eds[i], R);
        R[i] = P[i] = 0; X[i] = 1;
    }
}
```

### MaximumClique.h

**Description:** Quickly finds a maximum clique of a graph (given as symmetric bitset matrix; self-edges not allowed). Can be used to find a maximum independent set by finding a clique of the complement graph.

**Time:** Runs in about 1s for  $n=155$  and worst case random graphs ( $p=.90$ ). Runs faster for sparse graphs

abd580, 49 lines

```
typedef vector<bitset<200>> vb;
struct Maxclique {
    double limit=0.025, pk=0;
    struct Vertex { int i, d=0; };
    typedef vector<Vertex> vv;
    vb e;
    vv V;
    vector<vi> C;
    vi qmax, q, S, old;
    void init(vv& r) {
        for (auto& v : r) v.d = 0;
        for (auto& v : r) for (auto j : r) v.d += e[v.i][j.i];
    }
    sort(all(r), [](auto a, auto b) { return a.d > b.d; });
    int mxD = r[0].d;
    rep(i, sz(r)) r[i].d = min(i, mxD) + 1;
}

void expand(vv& R, int lev = 1) {
    S[lev] += S[lev - 1] - old[lev];
    old[lev] = S[lev - 1];
    while (sz(R)) {
        if (sz(q) + R.back().d <= sz(qmax)) return;
        q.push_back(R.back().i);
        vv T;
        for (auto v : R) if (e[R.back().i][v.i]) T.push_back({v.i});
        if (sz(T)) {
            if (S[lev]++ / ++pk < limit) init(T);
            int j = 0, mxk = 1, mnk = max(sz(qmax) - sz(q) + 1, 1);
            C[1].clear(), C[2].clear();
            for (auto v : T) {
                int k = 1;
                auto f = [&](int i) { return e[v.i][i]; };
                while (any_of(all(C[k]), f)) k++;
                if (k > mxk) mxk = k, C[mxk + 1].clear();
                if (k < mnk) T[j++] .i = v.i;
                C[k].push_back(v.i);
            }
            if (j > 0) T[j - 1].d = 0;
            fvd(k, mnk, mxk + 1) for (int i : C[k])
                T[j].i = i, T[j++].d = k;
            expand(T, lev + 1);
        } else if (sz(q) > sz(qmax)) qmax = q;
    }
}
```

```
q.pop_back(), R.pop_back();
}
}

vi maxClique() { init(V), expand(V); return qmax; }
Maxclique(vb conn) : e(conn), C(sz(e)+1), S(sz(C)), old(S) {}
rep(i, sz(e)) V.push_back({i});
}
```

## 7.7 Trees

### LCA.h

**Description:** Data structure for computing lowest common ancestors in a tree (with 0 as root). C should be an adjacency list of the tree, either directed or undirected.

**Time:**  $\mathcal{O}(N \log N + Q)$

0f62fb, 21 lines

```
struct LCA {
    int T = 0;
    vi time, path, ret;
    RMQ<int> rmq;
    LCA(vector<vi>& C) : time(sz(C)), rmq((dfs(C, 0, -1), ret)) {}
    void dfs(vector<vi>& C, int v, int par) {
        time[v] = T++;
        for (int y : C[v]) if (y != par) {
            path.push_back(v), ret.push_back(time[v]);
            dfs(C, y, v);
        }
    }
    int lca(int a, int b) {
        if (a == b) return a;
        tie(a, b) = minmax(time[a], time[b]);
        return path[rmq.query(a, b)];
    }
    //dist(a,b){return depth[a] + depth[b] - 2*depth[lca(a,b)];}
};
```

### CompressTree.h

**Description:** Given a rooted tree and a subset  $S$  of nodes, compute the minimal subtree that contains all the nodes by adding all (at most  $|S| - 1$ ) pairwise LCA's and compressing edges. Returns a list of (par, orig.index) representing a tree rooted at 0. The root points to itself.

**Time:**  $\mathcal{O}(|S| \log |S|)$

"LCA.h" 65149a, 21 lines

```
typedef vector<pair<int, int>> vpi;
vpi compressTree(LCA& lca, const vi& subset) {
    static vi rev; rev.resize(sz(lca.time));
    vi li = subset, &T = lca.time;
    auto cmp = [&](int a, int b) { return T[a] < T[b]; };
    sort(all(li), cmp);
    int m = sz(li) - 1;
    rep(i, m) {
        int a = li[i], b = li[i+1];
        li.push_back(lca.lca(a, b));
    }
    sort(all(li), cmp);
    li.erase(unique(all(li)), li.end());
    rep(i, sz(li)) rev[li[i]] = i;
    vpi ret = {pii(0, li[0])};
    rep(i, sz(li) - 1) {
        int a = li[i], b = li[i+1];
        ret.emplace_back(rev[lca.lca(a, b)], b);
    }
    return ret;
}
```

### HLD.h

**Description:** Decomposes a tree into vertex disjoint heavy paths and light edges such that the path from any leaf to the root contains at most  $\log(n)$  light edges. Code does additive modifications and max queries, but can support commutative segtree modifications/queries on paths and subtrees. Takes as input the full adjacency list. VALS\_EDGES being true means that values are stored in the edges, as opposed to the nodes. All values initialized to the segtree default. Root must be 0.

**Time:**  $\mathcal{O}(\log^2 N)$

```
"MyLazyTree.h" // make some sort of tree or whatever you like
80e958, 48 lines
//this tree should support add(l, r, x) -> add on [l, r) and query(l, r)
template <bool VALS_EDGES> struct HLD {
    int N, tim = 0;
```

```
vector<vi> adj;
vi par, siz, depth, rt, pos;
MyLazyTree *tree; // right-opened intervals [l,r),
HLD(vector<vi> adj)
    : N(sz(adj)), adj(adj), par(N, -1), siz(N, 1), depth(N),
    rt(N), pos(N), tree(new Node(0, N)) { dfsSz(0);
    dfsHld(0); }
void dfsSz(int v) {
    if (par[v] != -1) adj[v].erase(find(all(adj[v]), par[v]));
    for (int& u : adj[v]) {
        par[u] = v, depth[u] = depth[v] + 1;
        dfsSz(u);
        siz[v] += siz[u];
        if (siz[u] > siz[adj[v][0]]) swap(u, adj[v][0]);
    }
}
void dfsHld(int v) {
    pos[v] = tim++;
    for (int u : adj[v]) {
        rt[u] = (u == adj[v][0] ? rt[v] : u);
        dfsHld(u);
    }
}
template <class B> void process(int u, int v, B op) {
    for (; rt[u] != rt[v]; v = par[rt[v]]) {
        if (depth[rt[u]] > depth[rt[v]]) swap(u, v);
        op(pos[rt[v]], pos[v] + 1);
    }
    if (depth[u] > depth[v]) swap(u, v);
    op(pos[u] + VALS_EDGES, pos[v] + 1); // return u for lca
}
void modifyPath(int u, int v, int val) {
    process(u, v, [&](int l, int r) { tree->add(l, r, val); });
}
int queryPath(int u, int v) { // Modify depending on problem
    int res = -le9;
    process(u, v, [&](int l, int r) {
        res = max(res, tree->query(l, r));
    });
    return res;
} //queryPoint = return tree->query(pos[v])
int querySubtree(int v) { // modifySubtree is similar
    return tree->query(pos[v] + VALS_EDGES, pos[v] + siz[v]);
}
};
```

### Centroid.h

**Description:** Computes centroid tree for a given (0-indexed) tree, memory  $\mathcal{O}(n \log n)$  •  $child[v]$  = children of  $v$  in centroid tree •  $par[v]$  = parent of  $v$  in centroid tree (-1 for root) •  $depth[v]$  = depth of  $v$  in centroid tree (0 for root) =  $sz(ind[v]) - 1$  •  $size[v]$  = size of centroid subtree of  $v$  •  $ind[v][i]$  = index of vertex  $v$  in  $i$ -th centroid subtree from root, preorder •  $subtree[v]$  = list of vertices in centroid subtree of  $v$  •  $dist[v]$  = distances from  $v$  to vertices in its centroid subtree (in the order of  $subtree[v]$ ) •  $neigh[v]$  = neighbours of  $v$  in its centroid subtree •  $dir[v][i]$  = index of centroid neighbour that is first vertex on path from centroid  $v$  to  $i$ -th vertex of centroid subtree (-1 for centroid)

**Time:**  $\mathcal{O}(n \log n)$

6d2021, 51 lines

```
struct CentroidTree {
    vector<vi> child, ind, dists, subtree, neigh, dir;
    vi par, depth, size;
    int root; // Root centroid
    CentroidTree() {}
    CentroidTree(vector<vi>& G) {
        child(sz(G)), ind(sz(G)), dists(sz(G)), subtree(
            sz(G)),
            neigh(sz(G)), dir(sz(G)), par(sz(G), -2), depth(
                sz(G)), size(sz(G))
            { root = decomp(G, 0, 0); }
        void dfs(vector<vi>& G, int v, int p) {
            size[v] = 1;
            for (auto e : G[v]) if (e != p && par[e] == -2)
                dfs(G, e, v), size[v] += size[e];
        }
        void layer(vector<vi>& G, int v,
            int p, int c, int d) {
            ind[v].push_back(sz(subtree[c]));
            subtree[c].push_back(v);
            dists[c].push_back(d);
            dir[c].push_back(sz(neigh[c]) - 1); // possibly add extra functionalities here
        }
    };
};
```



```

    for(auto e: G[v]) if (e != p && par[e] == -2) {
        if (v == c) neigh[c].push_back(e);
        layer(G, e, v, c, d+1);
    }
}
int decomp(vector<vi>& G, int v, int d) {
    dfs(G, v, -1);
    int p = -1, s = size[v];
loop:
    for(auto e: G[v]) {
        if (e != p && par[e] == -2 &&
            size[e] > s/2) {
            p = v; v = e; goto loop;
        }
    }
    par[v] = -1; size[v] = s; depth[v] = d;
    layer(G, v, -1, v, 0);
    for(auto e: G[v]) if (par[e] == -2) {
        int j = decomp(G, e, d+1);
        child[v].push_back(j);
        par[j] = v;
    }
    return v;
}
};

```

## LinkCutTree.h

**Description:** Represents a forest of unrooted trees. You can add and remove edges (as long as the result is still a forest), and check whether two nodes are in the same tree.

**Time:** All operations take amortized  $\mathcal{O}(\log N)$ [@fb462](#), 90 lines

```

struct Node { // Splay tree. Root's pp contains tree's
    parent.
    Node *p = 0, *pp = 0, *c[2];
    bool flip = 0;
    Node() { c[0] = c[1] = 0; fix(); }
    void fix() {
        if (c[0]) c[0]->p = this;
        if (c[1]) c[1]->p = this;
        // (+ update sum of subtree elements etc. if wanted
        )
    }
    void pushFlip() {
        if (!flip) return;
        flip = 0; swap(c[0], c[1]);
        if (c[0]) c[0]->flip ^= 1;
        if (c[1]) c[1]->flip ^= 1;
    }
    int up() { return p ? p->c[1] == this : -1; }
    void rot(int i, int b) {
        int h = i ^ b;
        Node *x = c[i], *y = b == 2 ? x : x->c[h], *z = b ?
            y : x;
        if ((y->p == p) p->c[up()] = y;
            c[i] = z->c[i ^ 1];
            if (b < 2) {
                x->c[h] = y->c[h ^ 1];
                y->c[h ^ 1] = x;
            }
            z->c[i ^ 1] = this;
            fix(); x->fix(); y->fix();
            if (p) p->fix();
            swap(pp, y->pp);
        }
    }
    void splay() {
        for (pushFlip(); p; ) {
            if (p->p) p->p->pushFlip();
            p->pushFlip(); pushFlip();
            int c1 = up(), c2 = p->up();
            if (c2 == -1) p->rot(c1, 2);
            else p->p->rot(c2, c1 != c2);
        }
    }
    Node* first() {
        pushFlip();
        return c[0] ? c[0]->first() : (splay(), this);
    }
};
struct LinkCut {
    vector<Node> node;
    LinkCut(int N) : node(N) {}
    void link(int u, int v) { // add an edge (u, v)
        assert(!connected(u, v));
        makeRoot(&node[u]);
        node[u].pp = &node[v];
    }
    void cut(int u, int v) { // remove an edge (u, v)
        Node *x = &node[u], *top = &node[v];
        makeRoot(top); x->splay();
        assert(top == (x->pp ? x->c[0]));

```

```

        if (x->pp) x->pp = 0;
        else {
            x->c[0] = top->p = 0;
            x->fix();
        }
    }
    bool connected(int u, int v) { // are u, v in the
        same tree?
        Node* nu = access(&node[u])->first();
        return nu == access(&node[v])->first();
    }
    void makeRoot(Node* u) {
        access(u);
        u->splay();
        if (u->c[0]) {
            u->c[0]->p = 0;
            u->c[0]->flip ^= 1;
            u->c[0]->pp = u;
            u->c[0] = 0;
            u->fix();
        }
    }
    Node* access(Node* u) {
        u->splay();
        while (Node* pp = u->pp) {
            pp->splay(); u->pp = 0;
            if (pp->c[1]) {
                pp->c[1]->p = 0; pp->c[1]->pp = pp; }
            pp->c[1] = u; pp->fix(); u = pp;
        }
        return u;
    }
};

```

## DirectedMST.h

**Description:** Finds a minimum spanning tree/arborescence of a directed graph, given a root node. If no MST exists, returns -1.

**Time:**  $\mathcal{O}(E \log V)$

```

"../data-structures/UnionFindRollback.h" 84db4b, 60 lines
struct Edge { int a, b; ll w; };
struct Node {
    Edge key;
    Node *l, *r;
    ll delta;
    void prop() {
        key.w += delta;
        if (l) l->delta += delta;
        if (r) r->delta += delta;
        delta = 0;
    }
    Edge top() { prop(); return key; }
};
Node *merge(Node *a, Node *b) {
    if (!a || !b) return a ? b :
        a->prop(), b->prop();
    if (a->key.w > b->key.w) swap(a, b);
    swap(a->l, (a->r = merge(b, a->r)));
    return a;
}
void pop(Node*& a) { a->prop(); a = merge(a->l, a->r);
}
pair<ll, vi> dmst(int n, int r, vector<Edge>& g) {
    RollbackUF uf(n);
    vector<Node*> heap(n);
    for (Edge e : g) heap[e.b] = merge(heap[e.b], new
        Node(e));
    ll res = 0;
    vi seen(n, -1), path(n), par(n);
    seen[r] = r;
    vector<Edge> Q(n), in(n, {-1,-1}), comp;
    deque<tuple<int, int, vector<Edge>>> cysc;
    rep(s,n) {
        int u = s, qi = 0, w;
        while (seen[u] < 0) {
            if (!heap[u]) return {-1,{};};
            Edge e = heap[u]->top();
            heap[u]->delta -= e.w, pop(heap[u]);
            Q[qi] = e, path[qi++] = u, seen[u] = s;
            res += e.w, u = uf.find(e.a);
            if (seen[u] == s) {
                Node* cyc = 0;
                int end = qi, time = uf.time();
                do cyc = merge(cyc, heap[w = path[--qi]]);
                while (uf.join(u, w));
                u = uf.find(u), heap[u] = cyc, seen[u] = -1;
                cysc.push_front({u, time, {&Q[qi], &Q[end]});
            }
        }
    }
    rep(i,qi) in[uf.find(Q[i].b)] = Q[i];

```

```

}
for (auto& [u,t,comp] : cysc) { // restore sol (
    optional)
    uf.rollback(t);
    Edge inEdge = in[u];
    for (auto& e : comp) in[uf.find(e.b)] = e;
    in[uf.find(inEdge.b)] = inEdge;
}
rep(i,n) par[i] = in[i].a;
return {res, par};
}

```

## 7.8 Math

### 7.8.1 Number of Spanning Trees

Create an  $N \times N$  matrix  $\text{mat}$ , and for each edge

$a \rightarrow b \in G$ , do  $\text{mat}[a][b]--$ ,  $\text{mat}[b][b]++$  (and

$\text{mat}[b][a]--$ ,  $\text{mat}[a][a]++$  if  $G$  is undirected).

Remove the  $i$ th row and column and take the determinant; this yields the number of directed spanning trees rooted at  $i$  (if  $G$  is undirected, remove any row/column).

### 7.8.2 Erdős–Gallai theorem

A simple graph with node degrees  $d_1 \geq \dots \geq d_n$  exists

iff  $d_1 + \dots + d_n$  is even and for every  $k = 1 \dots n$ ,

$$\sum_{i=1}^k d_i \leq k(k-1) + \sum_{i=k+1}^n \min(d_i, k).$$

## Geometry (s)

### 8.1 Geometric primitives

#### Point.h

**Description:** Class to handle points in the plane. T can be e.g. double or long long. (Avoid int.)

```

int sgn(long long x) { return (x>0) - (x<0); } //
floats compare with eps
template<class T>
struct Point {
    typedef Point P;
    T x, y;
    explicit Point(T x=0, T y=0) : x(x), y(y) {}
    bool operator<(P p) const { return tie(x,y) < tie(p.x
        ,p.y); }
    bool operator==(P p) const { return tie(x,y)==tie(p.x
        ,p.y); }
    P operator+(P p) const { return P(x+p.x, y+p.y); }
    P operator-(P p) const { return P(x-p.x, y-p.y); }
    P operator*(T d) const { return P(x*d, y*d); }
    P operator/(T d) const { return P(x/d, y/d); }
    T dot(P p) const { return x*p.x + y*p.y; }
    T cross(P p) const { return x*p.y - y*p.x; }
    T cross(P a, P b) const { return (a-*this).cross(b-*
        this); }
    T dist2() const { return x*x + y*y; }
    double dist() const { return sqrt((double)dist2()); }
    // angle to x-axis in interval [-pi, pi]
    double angle() const { return atan2(y, x); }
    P unit() const { return *this/dist(); } // makes dist
        ()=1
    P perp() const { return P(-y, x); } // rotates +90
        degrees
    P normal() const { return perp().unit(); }
    // returns point rotated 'a' radians ccw around the
        origin
    P rotate(double a) const {
        return P(x*cos(a)-y*sin(a),x*sin(a)+y*cos(a)); }
    friend ostream& operator<<(ostream& os, P p) {
        return os << "(" << p.x << ", " << p.y << ")"; }
};

```

## lineDistance.h

**Description:** Returns the signed distance between point  $p$  and the line containing points  $a$  and  $b$ . Positive value on left side and negative on right as seen from  $a$  towards  $b$ .  $a==b$  gives nan.  $P$  is supposed to be Point<T> or Point3D<T> where  $T$  is e.g. double or long long. It uses products in intermediate steps so watch out for overflow if using int or long long. Using Point3D will always give a non-negative distance. For Point3D, call .dist on the result of the cross product.

"Point.h" [f6bf6b](#), 4 lines

```

template<class P>
double lineDist(const P& a, const P& b, const P& p) {
    return (double) (b-a).cross(p-a)/(b-a).dist();
}

```

## SegmentDistance.h

**Description:** Returns the shortest distance between point  $p$  and the line segment from point  $s$  to  $e$ .

**Usage:** Point<double> a, b(2,2), p(1,1);  
bool onSegment = segDist(a,b,p) < 1e-10;  
"Point.h" [5c88f4](#), 6 lines

```

typedef Point<double> P;
double segDist(P& s, P& e, P& p) {
    if (s==e) return (p-s).dist();
    auto d = (e-s).dist2(), t = min(d,max(.0, (p-s).dot(e-
        s)));
    return ((p-s)*d-(e-s)*t).dist()/d;
}

```

## SegmentIntersection.h

**Description:** If a unique intersection point between the line segments going from  $s_1$  to  $e_1$  and from  $s_2$  to  $e_2$  exists then it is returned. If no intersection point exists an empty vector is returned. If infinitely many exist a vector with 2 elements is returned, containing the endpoints of the common line segment. The wrong position will be returned if  $P$  is Point<ll> and the intersection point does not have integer coordinates. Products of three coordinates are used in intermediate steps so watch out for overflow if using int or long long.

**Usage:** vector<P> inter = segInter(s1,e1,s2,e2);  
if (sz(inter)==1)  
cout << "segments intersect at " << inter[0] <<  
endl;  
"Point.h", "onSegment.h" [9d57f2](#), 13 lines

```

template<class P> vector<P> segInter(P a, P b, P c, P d
    ) {
    auto oa = c.cross(d, a), ob = c.cross(d, b),
        oc = a.cross(b, c), od = a.cross(b, d);
    // Checks if intersection is single non-endpoint
        point.
    if (sgn(oa) * sgn(ob) < 0 && sgn(oc) * sgn(od) < 0)
        return {(a * ob - b * oa) / (ob - oa)};
    set<P> s;
    if (onSegment(c, d, a)) s.insert(a);
    if (onSegment(c, d, b)) s.insert(b);
    if (onSegment(a, b, c)) s.insert(c);
    if (onSegment(a, b, d)) s.insert(d);
    return {all(s)};
}

```

## lineIntersection.h

**Description:** If a unique intersection point of the lines going through  $s_1,e_1$  and  $s_2,e_2$  exists {1, point} is returned. If no intersection point exists {0, (0,0)} is returned and if infinitely many exists {-1, (0,0)} is returned. The wrong position will be returned if  $P$  is Point<ll> and the intersection point does not have integer coordinates. Products of three coordinates are used in intermediate steps so watch out for overflow if using int or ll.

**Usage:** auto res = lineInter(s1,e1,s2,e2);  
if (res.first == 1)  
cout << "intersection point at " << res.second <<  
endl;  
"Point.h" [a01f81](#), 8 lines

```

template<class P>
pair<int, P> lineInter(P s1, P e1, P s2, P e2) {
    auto d = (e1 - s1).cross(e2 - s2);
    if (d == 0) // if parallel
        return {(s1.cross(e1, s2) == 0), P(0, 0)};
    auto p = s2.cross(e1, e2), q = s2.cross(e2, s1);
    return {1, (s1 * p + e1 * q) / d};
}

```

## sideOf.h

**Description:** Returns where  $p$  is as seen from  $s$  towards  $e$ .  $1/0/-1 \Leftrightarrow$  left/on line/right. If the optional argument  $\text{eps}$  is given 0 is returned if  $p$  is within distance  $\text{eps}$  from the line.  $P$  is supposed to be Point<T> where  $T$  is e.g. double or long long. It uses products in intermediate steps so watch out for overflow if using int or long long.

**Usage:** bool left = sideOf(p1,p2,q)==1;  
"Point.h" [3af81c](#), 9 lines



## PolygonUnion.h

**Description:** Calculates the area of the union of  $n$  polygons (not necessarily convex). The points within each polygon must be given in CCW order. (Epsilon checks may optionally be added to sideOf/sgn, but shouldn't be needed.)

**Time:**  $\mathcal{O}(N^2)$ , where  $N$  is the total number of points

```
"Point.h", "sideOf.h" 19c5c5, 33 lines

typedef Point<double> P;
double rat(P a, P b) { return sgn(b.x) ? a.x/b.x : a.y/b.y; }
double polyUnion(vector<vector<P>>& poly) {
    double ret = 0;
    rep(i, sz(poly)) rep(v, sz(poly[i])) {
        P A = poly[i][v], B = poly[i][(v + 1) % sz(poly[i])]
        vector<pair<double, int>> segs = {{0, 0}, {1, 0}};
        rep(j, sz(poly)) if (i != j) {
            rep(u, sz(poly[j])) {
                P C = poly[j][u], D = poly[j][(u + 1) % sz(poly[j])];
                int sc = sideOf(A, B, C), sd = sideOf(A, B, D);
                if (sc != sd) {
                    double sa = C.cross(D, A), sb = C.cross(D, B);
                    if (min(sc, sd) < 0)
                        segs.emplace_back(sa / (sa - sb), sgn(sc - sd));
                } else if (!sc && !sd && j < i && sgn((B-A).dot(D-C)) > 0) {
                    segs.emplace_back(rat(C - A, B - A), 1);
                    segs.emplace_back(rat(D - A, B - A), -1);
                }
            }
        }
        sort(all(segs));
        for (auto& s : segs) s.first = min(max(s.first, 0.0), 1.0);
        double sum = 0;
        int cnt = segs[0].second;
        fwd(j, 1, sz(segs)) {
            if (!cnt) sum += segs[j].first - segs[j - 1].first;
            cnt += segs[j].second;
        }
        ret += A.cross(B) * sum;
    }
    return ret / 2;
}
```

## ConvexHull.h

**Description:** Returns a vector of the points of the convex hull in counter-clockwise order. Points on the edge of the hull between two other points are not considered part of the hull.

**Time:**  $\mathcal{O}(n \log n)$

```
"Point.h" 310954, 13 lines

typedef Point<ll> P;
vector<P> convexHull(vector<P> pts) {
    if (sz(pts) <= 1) return pts;
    sort(all(pts));
    vector<P> h(sz(pts)+1);
    int s = 0, t = 0;
    for (int it = 2; it--; s = --t, reverse(all(pts)))
        for (P p : pts) {
            while (t >= s + 2 && h[t-2].cross(h[t-1], p) <= 0) t--;
            h[t++] = p;
        }
    return {h.begin(), h.begin() + t - (t == 2 && h[0] == h[1])};
}
```

## HullDiameter.h

**Description:** Returns the two points with max distance on a convex hull (ccw, no duplicate/collinear points).

**Time:**  $\mathcal{O}(n)$

```
"Point.h" 8edf58, 12 lines

typedef Point<ll> P;
array<P, 2> hullDiameter(vector<P> pts) {
    int n = sz(S), j = n < 2 ? 0 : 1;
    pair<ll, array<P, 2>> res({0, {S[0], S[0]}});
    rep(i, j)
        for (; j = (j + 1) % n) {
            res = max(res, {(S[i] - S[j]).dist2(), {S[i], S[j]}});
            if ((S[(j + 1) % n] - S[j]).cross(S[i + 1] - S[i]) >= 0)
                break;
        }
    return res;
}
```

```
        break;
    }
    return res.second;
}
```

## PointInsideHull.h

**Description:** Determine whether a point t lies inside a convex hull (CCW order, with no collinear points). Returns true if point lies within the hull. If strict is true, points on the boundary aren't included.

**Time:**  $\mathcal{O}(\log N)$

```
"Point.h", "sideOf.h", "OnSegment.h" 71446b, 14 lines

typedef Point<ll> P;
bool inHull(const vector<P>& l, P p, bool strict = true) {
    int a = 1, b = sz(l) - 1, r = !strict;
    if (sz(l) < 3) return r && onSegment(l[0], l.back(), p);
    if (sideOf(l[0], l[a], l[b]) > 0) swap(a, b);
    if (sideOf(l[0], l[a], p) >= r || sideOf(l[0], l[b], p) <= -r)
        return false;
    while (abs(a - b) > 1) {
        int c = (a + b) / 2;
        if (sideOf(l[0], l[c], p) > 0 ? b : a) = c;
    }
    return sgn(l[a].cross(l[b], p)) < r;
}
```

## LineHullIntersection.h

**Description:** Line-convex polygon intersection. The polygon must be ccw and have no collinear points. lineHull(line, poly) returns a pair describing the intersection of a line with the polygon:  $\bullet (-1, -1)$  if no collision,  $\bullet (i, -1)$  if touching the corner  $i$ ,  $\bullet (i, i)$  if along side  $(i, i+1)$ ,  $\bullet (i, j)$  if crossing sides  $(i, i+1)$  and  $(j, j+1)$ . In the last case, if a corner  $i$  is crossed, this is treated as happening on side  $(i, i+1)$ . The points are returned in the same order as the line hits the polygon.

**Time:**  $\mathcal{O}(\log n)$

```
"Point.h" 46fb73, 44 lines

template<class P> int extrVertex(vector<P> &poly,
    function<P(P)> dir) {
    int n = sz(poly), lo = 0, hi = n;
    auto cmp = [&](int i, int j) {return sgn(dir(poly[i%n]).cross(poly[i % n] - poly[j % n]));};
    auto extr = [&](int i) {return cmp(i + 1, i) >= 0 && cmp(i, i - 1 + n) < 0;};
    if (extr(0)) return 0;
    while (lo + 1 < hi) {
        int m = (lo + hi) / 2;
        if (extr(m)) return m;
        int ls = cmp(lo + 1, lo), ms = cmp(m + 1, m);
        (ls < ms || (ls == ms && ls == cmp(lo, m)) ? hi : lo) = m;
    }
    return lo;
} //also, use extrVertex<P>(poly, [&](P) {return v.perp();}) for vector v
// to get the first ccw point of a hull with the max projection onto v
#define cmpL(i) sgn(a.cross(poly[i], b))
template<class P> array<int, 2> lineHull(P a, P b, vector<P> &poly) {
    int endA = extrVertex<P>(poly, [&](P) {return b - a;});
    int endB = extrVertex<P>(poly, [&](P) {return a - b;});
    if (cmpL(endA) < 0 || cmpL(endB) > 0) return {-1, -1};
    array<int, 2> res;
    rep(i, 2) {
        int lo = endB, hi = endA, n = sz(poly);
        while ((lo + 1) % n != hi) {
            int m = ((lo + hi + (lo < hi ? 0 : n)) / 2) % n;
            (cmpL(m) == cmpL(endB) ? lo : hi) = m;
        }
        res[i] = (lo + 1 % cmpL(hi)) % n;
        swap(endA, endB);
    }
    if (res[0] == res[1]) return {res[0], -1};
    if (!cmpL(res[0]) && !cmpL(res[1]))
        switch ((res[0] - res[1] + sz(poly) + 1) % sz(poly)) {
            case 0: return {res[0], res[0]};
            case 2: return {res[1], res[1]};
        }
    return res;
}
```

```

}
template<class P> pii getTangentPointOrSide(vector<P>& poly, P p, bool left) {
    int n = sz(poly); //left tangent is earlier on hull
    int i = extrVertex<P>(poly, [&](P q) {return left ? p - q : q - p;});
    return p.cross(poly[i], poly[(i+1)%n]) ? pii(i, i) : pii(i, (i+1)%n);
}
```

## HalfplaneIntersection.h

**Description:** Online half plane intersection. Works both for ll and long double. Bounding box is optional, but needed for distinguishing bounded vs unbounded. Halfplanes are sorted ccw in HPI.s. Time:  $\mathcal{O}(\log n)$  per add.

```
"Point.h" f5db92, 97 lines

using T = ll; // has to fit 2*|pts|**2 (slopes etc)
using P = Point<T>; // only cross needed
using SuperT = __int128_t; // has to fit 6*|pts|**3 (intersections)
const SuperT EPS = 1e-12; // |pts| <= 10^6 for doubles (citation needed)
struct Line { //coords <= 10^9 for abc constructor and <= 10^6 for p,q
    T a, b, c;
    Line(T a_=0, T b_=0, T c_=0): a(a_), b(b_), c(c_) {}
    //ax + by + c >= 0
    Line(P p, P q): a(p.y-q.y), b(q.x-p.x), c(p.cross(q)) {} //p->q ccw
    Line operator-() const {return Line(-a, -b, -c); }
    bool up() const {return a?(a<0):(b>0); }
    P v() const {return P(a,b); }
    P vx() {return P(b,c); } P vy() {return P(a,c); }
    T wek(Line p) const {return v().cross(p.v()); }
    bool operator<(Line b) const {
        if (up() != b.up()) return up() > b.up();
        return wek(b) > 0;
    }
};
bool parallel(Line a, Line b) {return !a.wek(b); }
bool same(Line a, Line b) {
    return parallel(a,b) && !a.vy().cross(b.vy()) && !a.vx().cross(b.vx());
}
T weaker(Line a, Line b) {
    if (abs(a.a) > abs(a.b)) return a.c*abs(b.a) - b.c*abs(a.a);
    return a.c*abs(b.b) - b.c*abs(a.b);
}
array<SuperT, 3> intersect(Line a, Line b) {
    SuperT det = a.wek(b);
    SuperT x = a.vx().cross(b.vx());
    SuperT y = a.vy().cross(b.vy());
    // if (T is floating point) return {x / det, -y / det, 1.0};
    if (det > 0) return {x, -y, det};
    return {-x, y, -det};
}
struct HPI {
    bool empty=0, pek=0;
    set<Line> s;
    typedef set<Line>::iterator iter;
    iter next(iter it){return ++it == s.end() ? s.begin() : it;}
    iter prev(iter it){return it == s.begin() ? --s.end() : --it;}
    bool hide(Line a, Line b, Line c) { // do a,b hide c?
        if (parallel(a,b)) {
            if (weaker(a, -b) < 0) empty = 1;
            return 0;
        }
        if (a.wek(b) < 0) swap(a,b);
        auto [rx, ry, rdet] = intersect(a,b);
        auto v = rx*c.a + ry*c.b + rdet*c.c;
        if (a.wek(c) >= 0 && c.wek(b) >= 0 && v >= -EPS)
            return 1;
        if (a.wek(c) < 0 && c.wek(b) < 0) {
            if (v < -EPS) empty = 1;
            else if (v <= EPS) pek = 1;
        }
        return 0;
    }
    void delAndMove(iter& i, int nxt) {
        iter j = i;
        if (nxt==1) i = next(i);
        else i = prev(i);
        s.erase(j);
    }
    void add(Line l) {
        if (empty) return;
    }
};
```

```

if (l.a == 0 && l.b == 0) {
    if (l.c < 0) empty = 1;
    return;
}
iter it = s.lower_bound(l); //parallel
if (it != s.end() && parallel(*it, l) && it->up() == 1.up()) {
    if (weaker(l, *it)>=0) return;
    delAndMove(it,1);
}
if (it == s.end()) it = s.begin(); // *it>p
while (sz(s) >= 2 && hide(l, *next(it), *it))
    delAndMove(it,1);
if (sz(s)) it = prev(it); // *it<p
while (sz(s) >= 2 && hide(l, *prev(it), *it))
    delAndMove(it,0);
if (sz(s) < 2 || !hide(*it, *next(it), l)) s.insert(l);
}
int type() { //0 = empty, 1=point, 2=segment, 3=halfline
    if (empty) return 0; //4=line, 5=polygon or unbounded
    if (sz(s) <= 4) {
        vector<Line> r(all(s));
        if (sz(r) == 2 && parallel(r[0], r[1]) && weaker(r[0], -r[1])<0)
            return 0;
        rep(i, sz(r)) rep(j, i) if (same(r[i], r[j])) {
            if (sz(r) == 2) return 4; if (sz(r) == 3) return 3;
            if (sz(r) == 4 && same(r[0], r[2]) && same(r[1], r[3])) return 1;
            return 2;
        }
        if (sz(r) == 3 && pek) return 1;
        return 5;
    }
};
```

## 8.4 Misc. Point Set Problems

### ClosestPair.h

**Description:** Finds the closest pair of points.

**Time:**  $\mathcal{O}(n \log n)$

```
"Point.h" ac41a6, 17 lines

typedef Point<ll> P;
pair<P, P> closest(vector<P> v) {
    assert(sz(v) > 1);
    set<P> S;
    sort(all(v), [](P a, P b) { return a.y < b.y; });
    pair<ll, pair<P, P>> ret(LLONG_MAX, {P(), P()});
    int j = 0;
    for (P p : v) {
        P d(1 + (ll)sqrt(ret.first), 0);
        while (v[j].y <= p.y - d.x) S.erase(v[j++]);
        auto lo = S.lower_bound(p - d), hi = S.upper_bound(p + d);
        for (; lo != hi; ++lo)
            ret = min(ret, {(lo - p).dist2(), {lo, p}});
        S.insert(p);
    }
    return ret.second;
}
```

## ManhattanMST.h

**Description:** Given  $N$  points, returns up to  $4*N$  edges, which are guaranteed to contain a minimum spanning tree for the graph with edge weights  $w(p, q) = -p.x - q.x - +p.y - q.y -$ . Edges are in the form (distance, src, dst). Use a standard MST algorithm on the result to find the final MST.

**Time:**  $\mathcal{O}(N \log N)$

```
"Point.h" 381880, 23 lines

typedef Point<int> P;
vector<array<int, 3>> manhattanMST(vector<P> ps) {
    vi id(sz(ps));
    iota(all(id), 0);
    vector<array<int, 3>> edges;
    rep(k, 4) {
        sort(all(id), [&](int i, int j) {
            return (ps[i]-ps[j]).x < (ps[j]-ps[i]).y; });
        map<int, int> sweep;
        for (int i : id) {
            for (auto it = sweep.lower_bound(-ps[i].y);
                it != sweep.end(); sweep.erase(it++)) {
                int j = it->second;
                P d = ps[i] - ps[j];
                if (d.y > d.x) break;
                edges.push_back({d.y + d.x, i, j});
            }
        }
    }
```



```

    }
    sweep[-ps[i].y] = i;
  }
  for (P& p : ps) if (k & 1) p.x = -p.x; else swap(p.x, p.y);
}
return edges;
}

```

## kdTree.h

**Description:** KD-tree (2d, can be extended to 3d)

"Point.h" bac5b0, 63 lines

```

typedef long long T;
typedef Point<T> P;
const T INF = numeric_limits<T>::max();
bool on_x(const P& a, const P& b) { return a.x < b.x; }
bool on_y(const P& a, const P& b) { return a.y < b.y; }
struct Node {
  P pt; // if this is a leaf, the single point in it
  T x0 = -INF, x1 = -INF, y0 = INF, y1 = -INF; // bounds
  Node *first = 0, *second = 0;
  T distance(const P& p) { // min squared distance to a point
    T x = (p.x < x0 ? x0 : p.x > x1 ? x1 : p.x);
    T y = (p.y < y0 ? y0 : p.y > y1 ? y1 : p.y);
    return (P(x,y) - p).dist2();
  }
  Node(vector<P>&& vp) : pt(vp[0]) {
    for (P p : vp) {
      x0 = min(x0, p.x); x1 = max(x1, p.x);
      y0 = min(y0, p.y); y1 = max(y1, p.y);
    }
    if (vp.size() > 1) {
      // split on x if width >= height (not ideal...)
      sort(all(vp), x1 - x0 >= y1 - y0 ? on_x : on_y);
      // divide by taking half the array for each child (not
      // best performance with many duplicates in the middle)
      int half = sz(vp)/2;
      first = new Node({vp.begin(), vp.begin() + half});
      second = new Node({vp.begin() + half, vp.end()});
    }
  };
  struct KDTree {
    Node* root;
    KDTree(const vector<P>&& vp) : root(new Node({all(vp)})) {}
    pair<T, P> search(Node *node, const P& p) {
      if (!node->first) {
        // uncommnet if we should not find the point itself
        // if (p == node->pt) return {INF, P()};
        return make_pair((p - node->pt).dist2(), node->pt);
      }
      Node *f = node->first, *s = node->second;
      T bfirst = f->distance(p), bsec = s->distance(p);
      if (bfirst > bsec) swap(bsec, bfirst), swap(f, s);
      // search closest side first, other side if needed
      auto best = search(f, p);
      if (bsec < best.first)
        best = min(best, search(s, p));
      return best;
    }
    // find nearest point to a point, and its squared distance
    // (requires an arbitrary operator< for Point)
    pair<T, P> nearest(const P& p) {
      return search(root, p);
    }
  };
};

```

## FastDelaunay.h

**Description:** Fast Delaunay triangulation. Each circumcircle contains none of the input points. There must be no duplicate points. If all points are on a line, no triangles will be returned. Should work for doubles as well, though there may be precision issues in 'circ'. Returns triangles in order {t[0][0], t[0][1], t[0][2], t[1][0], ...}, all counter-clockwise.

**Time:**  $\mathcal{O}(n \log n)$

"Point.h" 0f676e, 88 lines

```

typedef Point<ll> P;
typedef struct Quad* Q;
typedef __int128_t lll; // (can be ll if coords are < 2
e4)

```

```

P arb(LLONG_MAX, LLONG_MAX); // not equal to any other point
struct Quad {
  Q rot, o; P p = arb; bool mark;
  P& F() { return r()->p; }
  Q& r() { return rot->rot; }
  Q prev() { return rot->o->rot; }
  Q next() { return r()->prev(); }
} *H;
bool circ(P p, P a, P b, P c) { // is p in the circumcircle?
  lll p2 = p.dist2(), A = a.dist2()-p2,
  B = b.dist2()-p2, C = c.dist2()-p2;
  return p.cross(a,b)*C + p.cross(b,c)*A + p.cross(c,a)
    *B > 0;
}
Q makeEdge(P orig, P dest) {
  Q r = H ? H : new Quad(new Quad{new Quad{new Quad{0}}});
  H = r->o; r->r()->r() = r;
  rep(i,4) r = r->rot; r->p = arb, r->o = i & 1 ? r : r->r();
  r->p = orig; r->F() = dest;
  return r;
}
void splice(Q a, Q b) {
  swap(a->o->rot->o, b->o->rot->o); swap(a->o, b->o);
}
Q connect(Q a, Q b) {
  Q q = makeEdge(a->F(), b->p);
  splice(q, a->next());
  splice(q->r(), b);
  return q;
}
pair<Q,Q> rec(const vector<P>&& s) {
  if (sz(s) <= 3) {
    Q a = makeEdge(s[0], s[1]), b = makeEdge(s[1], s.back());
    if (sz(s) == 2) return { a, a->r() };
    splice(a->r(), b);
    auto side = s[0].cross(s[1], s[2]);
    Q c = side ? connect(b, a) : 0;
    return {side < 0 ? c->r() : a, side < 0 ? c : b->r()};
  }
  #define H(e) e->F(), e->p
  #define valid(e) (e->F().cross(H(base)) > 0)
  Q A, B, ra, rb;
  int half = sz(s) / 2;
  tie(ra, A) = rec(all(s) - half);
  tie(B, rb) = rec(sz(s) - half + all(s));
  while ((B->p.cross(H(A)) < 0 && (A = A->next())) ||
    (A->p.cross(H(B)) > 0 && (B = B->r()->o)));
  Q base = connect(B->r(), A);
  if (A->p == ra->p) ra = base->r();
  if (B->p == rb->p) rb = base;
  #define DEL(e, init, dir) Q e = init->dir; if (valid(e)) \
    \
    while (circ(e->dir->F(), H(base), e->F())) { \
      Q t = e->dir; \
      splice(e, e->prev()); \
      splice(e->r(), e->r()->prev()); \
      e->o = H; H = e; e = t; \
    }
  for (;) {
    DEL(LC, base->r(), o); DEL(RC, base, prev());
    if (!valid(LC) && !valid(RC)) break;
    if (!valid(LC) || (valid(RC) && circ(H(RC), H(LC))))
      base = connect(RC, base->r());
    else
      base = connect(base->r(), LC->r());
  }
  return { ra, rb };
}
vector<P> triangulate(vector<P> pts) {
  sort(all(pts)); assert(unique(all(pts)) == pts.end());
  if (sz(pts) < 2) return {};
  Q e = rec(pts).first;
  vector<Q> q = {e};
  int qi = 0;
  while (e->o->F().cross(e->F(), e->p) < 0) e = e->o;
  #define ADD { Q c = e; do { c->mark = 1; pts.push_back(c->p); \
    \
    q.push_back(c->r()); c = c->next(); } while (c != e);
  ADD; pts.clear();
  while (qi < sz(q)) if (!(e = q[qi++]->mark) ADD;
  return pts;
}

```

## 8.5 3D

### PolyhedronVolume.h

**Description:** Magic formula for the volume of a polyhedron.

Faces should point outwards.

3058c3, 6 lines

```

template<class V, class L>
double signedPolyVolume(const V& p, const L& trilst) {
  double v = 0;
  for (auto i : trilst) v += p[i.a].cross(p[i.b]).dot(
    p[i.c]);
  return v / 6;
}

```

## Point3D.h

**Description:** Class to handle points in 3D space. T can be e.g. double or long long.

8058ae, 32 lines

```

template<class T> struct Point3D {
  typedef Point3D P;
  typedef const P R;
  T x, y, z;
  explicit Point3D(T x=0, T y=0, T z=0) : x(x), y(y), z(z) {}
  bool operator<(R p) const {
    return tie(x, y, z) < tie(p.x, p.y, p.z); }
  bool operator==(R p) const {
    return tie(x, y, z) == tie(p.x, p.y, p.z); }
  P operator+(R p) const { return P(x+p.x, y+p.y, z+p.z); }
  P operator-(R p) const { return P(x-p.x, y-p.y, z-p.z); }
  P operator*(T d) const { return P(x*d, y*d, z*d); }
  P operator/(T d) const { return P(x/d, y/d, z/d); }
  T dot(R p) const { return x*p.x + y*p.y + z*p.z; }
  P cross(R p) const {
    return P(y*p.z - z*p.y, z*p.x - x*p.z, x*p.y - y*p.x);
  }
  T dist2() const { return x*x + y*y + z*z; }
  double dist() const { return sqrt((double)dist2()); }
  //Azimuthal angle (longitude) to x-axis in interval [-pi, pi]
  double phi() const { return atan2(y, x); }
  //Zenith angle (latitude) to the z-axis in interval [0, pi]
  double theta() const { return atan2(sqrt(x*x+y*y), z); }
  P unit() const { return *this/(T)dist(); } //makes dist()=1
  //returns unit vector normal to *this and p
  P normal(P p) const { return cross(p).unit(); }
  //returns point rotated 'angle' radians ccw around axis
  P rotate(double angle, P axis) const {
    double s = sin(angle), c = cos(angle); P u = axis.unit();
    return u*dot(u)*1-c + (*this)*c - cross(u)*s;
  }
};

```

## 3dHull.h

**Description:** Computes all faces of the 3-dimension hull of a point set. \*No four points must be coplanar\*, or else random results will be returned. All faces will point outwards.

**Time:**  $\mathcal{O}(n^2)$

"Point3D.h" 8f7440, 49 lines

```

typedef Point3D<double> P3;
struct PR {
  void ins(int x) { (a == -1 ? a : b) = x; }
  void rem(int x) { (a == x ? a : b) = -1; }
  int cnt() { return (a != -1) + (b != -1); }
  int a, b;
};
struct F { P3 q; int a, b, c; };
vector<F> hull3d(const vector<P3>& A) {
  assert(sz(A) >= 4);
  vector<vector<PR>> E(sz(A), vector<PR>(sz(A), {-1, -1}));
  #define E(x,y) E[f.x][f.y]
  vector<F> FS;
  auto mf = [&](int i, int j, int k, int l) {
    P3 q = (A[j] - A[i]).cross(A[k] - A[i]);
    if (q.dot(A[l]) > q.dot(A[i]))
      q = q - l;
    F f(q, i, j, k);
    E(a,b).ins(k); E(a,c).ins(j); E(b,c).ins(i);
    FS.push_back(f);
  };
};

```

```

rep(i,4) fwd(j,i+1,4) fwd(k,j+1,4)
mf(i, j, k, 6 - i - j - k);
fwd(i,4,sz(A)) {
  rep(j,sz(FS)) {
    F f = FS[j];
    if (f.q.dot(A[i]) > f.q.dot(A[f.a])) {
      E(a,b).rem(f.c);
      E(a,c).rem(f.b);
      E(b,c).rem(f.a);
      swap(FS[j--], FS.back());
      FS.pop_back();
    }
  }
  int nw = sz(FS);
  rep(j,nw) {
    F f = FS[j];
    #define C(a, b, c) if (E(a,b).cnt() != 2) mf(f.a, f.b, i, f.c);
    C(a, b, c); C(a, c, b); C(b, c, a);
  }
  for (F& it : FS) if ((A[it.b] - A[it.a]).cross(
    A[it.c] - A[it.a]).dot(it.q) <= 0) swap(it.c, it.b);
  return FS;
};

```

## sphericalDistance.h

**Description:** Returns the shortest distance on the sphere with radius radius between the points with azimuthal angles (longitude) f1 ( $\phi_1$ ) and f2 ( $\phi_2$ ) from x axis and zenith angles (latitude) t1 ( $\theta_1$ ) and t2 ( $\theta_2$ ) from z axis (0 = north pole). All angles measured in radians. The algorithm starts by converting the spherical coordinates to cartesian coordinates so if that is what you have you can use only the two last rows. dx\*radius is then the difference between the two points in the x direction and d\*radius is the total distance between the points.

1a7, 8 lines

```

double sphericalDistance(double f1, double t1,
  double f2, double t2, double radius) {
  double dx = sin(t2)*cos(f2) - sin(t1)*cos(f1);
  double dy = sin(t2)*sin(f2) - sin(t1)*sin(f1);
  double dz = cos(t2) - cos(t1);
  double d = sqrt(dx*dx + dy*dy + dz*dz);
  return radius*2*asin(d/2);
}

```

## Strings (9)

### KMP.h

**Description:** pi[x] computes the length of the longest prefix of s that ends at x, other than s[0...x] itself (abacaba -> 0010123). Can be used to find all occurrences of a string.

**Time:**  $\mathcal{O}(n)$

ccedb5, 9 lines

```

vi pi(const string& s) {
  vi p(sz(s));
  fwd(i,1,sz(s)) {
    int g = p[i-1];
    while (g && s[i] != s[g]) g = p[g-1];
    p[i] = g + (s[i] == s[g]);
  }
  return p;
}

```

### Zfunc.h

**Description:** z[x] computes the length of the longest common prefix of s[i:] and s, except z[0] = 0. (abacaba -> 0010301)

**Time:**  $\mathcal{O}(n)$

011e28, 14 lines

```

vi Z(const string& S, bool z0n = false) {
  vi z(sz(S));
  int l = -1, r = -1;
  fwd(i,1,sz(S)) { // from below l is a small L
    z[i] = i >= r ? 0 : min(r - i, z[i - l]);
    while (i + z[i] < sz(S) && S[i + z[i]] == S[z[i]
      ])
      z[i]++;
    if (i + z[i] > r)
      l = i, r = i + z[i];
  }
  if (z0n && sz(S))
    z[0] = sz(S);
  return z;
}

```



## Manacher.h

**Description:** For each position in a string, computes  $p[0][i]$  = half length of longest even palindrome around pos  $i$ ,  $p[1][i]$  = longest odd (half rounded down).

**Time:**  $\mathcal{O}(N)$  589844, 13 lines

```
array<vi, 2> manacher(const string& s) {
    int n = sz(s);
    array<vi, 2> p = {vi(n+1), vi(n)};
    rep(z, 2) for (int i=0, l=0, r=0; i < n; i++) {
        int t = r-i+!z;
        if (i<r) p[z][i] = min(t, p[z][l+t]);
        int L = i-p[z][i], R = i+p[z][i]-!z;
        while (L>=1 && R<=n && s[L-1] == s[R+1])
            p[z][i]++, L--, R++;
        if (R>r) l=L, r=R;
    }
    return p;
}
```

## ALCS.h

**Description:** All-substrings common sequences algorithm. Given strings A and B, algorithm computes:  $C(i, j, k) = |LCS(A[i: i], B[j: k])|$  in compressed form; To describe the compression, note that: 1.  $C(i, j, k-1) \leq C(i, j, k) \leq C(i, j, k-1) + 1$  2. If  $j < k$  and  $C(i, j, k) = C(i, j, k-1) + 1$ , then  $C(i, j+1, k) = C(i, j+1, k-1) + 1$  3. If  $j \geq k$ , then  $C(i, j, k) = 0$  This allows us to store just the following:  $ih(i, k) = \min j$  s.t.  $C(i, j, k-1) < C(i, j, k)$

**Time:**  $\mathcal{O}(nm)$  e5b4cf, 58 lines

```
struct ALCS {
    string A, B;
    vector<vi> ih;
    // Precompute compressed matrix; time: O(nm)
    ALCS(string s, string t) : A(s), B(t) {
        int n = sz(A), m = sz(B);
        ih.resize(n+1, vi(m+1));
        iota(all(ih[0]), 0);
        fwd(1, 1, n+1) {
            int iv = 0;
            fwd(j, 1, m+1) {
                if (A[l-1] != B[j-1]) {
                    ih[l][j] = max(ih[l-1][j], iv);
                    iv = min(ih[l-1][j], iv);
                } else {
                    ih[l][j] = iv;
                    iv = ih[l-1][j];
                }
            }
        }
        // Compute |LCS(A[:i], B[:k])|; time: O(k-j)
        // Note: You can precompute data structure
        // to answer these queries in O(log n)
        // or compute all answers for fixed 'i'.
        int operator()(int i, int j, int k) {
            int ret = 0;
            fwd(q, j, k) ret += (ih[i][q+1] <= j);
            return ret;
        }
        // Compute subsequence LCS(A[:i], B[:k]);
        // time: O(k-j)
        string recover(int i, int j, int k) {
            string ret;
            while (i > 0 && j < k) {
                if (ih[i][k-1] <= j) {
                    ret.push_back(B[k]);
                    while (A[--i] != B[k])
                        ;
                }
                reverse(all(ret));
                return ret;
            }
            // Compute LCS'es of given prefix of A,
            // and all prefixes of given suffix of B.
            // Returns vector L of length |B|+1 s.t.
            // L[k] = |LCS(A[:i], B[:k])|; time: O(|B|)
            vi row(int i, int j) {
                vi ret(sz(B)+1);
                fwd(k, j+1, sz(ret)) ret[k] = ret[k-1] + (ih[i][k] <= j);
                return ret;
            }
        };
    }
};
```

## MainLorentz.h

**Description:** Main-Lorentz algorithm for finding all squares in given word; Results are in compressed form: (b, e, l) means that for each  $b \leq i < e$  there is square at position  $i$  of size  $2l$ . Each square is present in only one interval.

**Time:**  $\mathcal{O}(n \lg n)$  5e103f, 46 lines

```
struct Sqr {
    int begin, end, len;
};
vector<Sqr> lorentz(const string &s) {
    vector<Sqr> ans;
    vi pos(sz(s) / 2 + 2, -1);
    fwd(mid, 1, sz(s)) {
        int part = mid & ~(mid-1), off = mid - part;
        int end = min(mid+part, sz(s));
        auto a = s.substr(off, part);
        auto b = s.substr(mid, end-mid);
        string ra(a.rbegin(), a.rend());
        string rb(b.rbegin(), b.rend());
        rep(j, 2) {
            // Set # to some unused character!
            vi z1 = Z(ra, true);
            vi z2 = Z(b + "#" + a, true);
            z1.push_back(0);
            z2.push_back(0);
            rep(c, sz(a)) {
                int l = sz(a) - c;
                int x = c - min(l-1, z1[l]);
                int y = c - max(l - z2[sz(b) + c + 1], j);
                if (x > y)
                    continue;
                int sb = (j ? end - y - 1 * 2 : off + x);
                int se = (j ? end - x - 1 * 2 + 1 : off + y + 1);
                int &p = pos[l];
                if (p != -1 && ans[p].end == sb)
                    ans[p].end = se;
                else
                    p = sz(ans), ans.push_back({sb, se, l});
            }
            a.swap(rb);
            b.swap(ra);
        }
        return ans;
    }
}
```

## Lyndon.h

**Description:** Compute Lyndon factorization for s; Word is simple iff it's strictly smaller than any of it's nontrivial suffixes. Lyndon factorization is division of string into non-increasing simple words. It is unique.

**Time:**  $\mathcal{O}(n)$  fa3adf, 12 lines

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

## MinRotation.h

**Description:** Finds the lexicographically smallest rotation of a string.

**Usage:** rotate(v.begin(), v.begin()+minRotation(v), v.end());

**Time:**  $\mathcal{O}(N)$  70d292, 8 lines

```
int minRotation(string s) {
    int a=0, N=sz(s); s += s;
    rep(b, N) rep(k, N) {
        if (a+k == b || s[a+k] < s[b+k]) {b += max(0, k-1); break;}
        if (s[a+k] > s[b+k]) {a = b; break;}
    }
    return a;
}
```

## SuffixArray.h

**Description:** Builds suffix array for a string. sa[i] is the starting index of the suffix which is  $i$ 'th in the sorted suffix array. The returned vector is of size  $n+1$ , and sa[0] = n. The lcp array contains longest common prefixes for neighbouring strings in the suffix array: lcp[i] = lcp(sa[i], sa[i-1]), lcp[0] = 0. The input string must not contain any zero bytes.

**Time:**  $\mathcal{O}(n \log n)$  9ff92c, 23 lines

```
struct SuffixArray {
    vi sa, lcp;
    SuffixArray(string& s, int lim=256) { // or
        basic_string<int>
        int n = sz(s) + 1, k = 0, a, b;
        vi x(all(s)+1), y(n), ws(max(n, lim)), rank(n);
        sa = lcp = y, iota(all(sa), 0);
        for (int j = 0, p = 0; p < n; j = max(1, j * 2),
            lim = p) {
            p = j, iota(all(y), n - j);
            rep(i, n) if (sa[i] >= j) y[p++] = sa[i] - j;
            fill(all(ws), 0);
            rep(i, n) ws[x[i]]++;
            fwd(i, 1, lim) ws[i] += ws[i-1];
            for (int i = n; i--; ) sa[--ws[x[i]]] = y[i];
            swap(x, y), p = 1, x[sa[0]] = 0;
            fwd(i, 1, n) a = sa[i-1], b = sa[i], x[b] =
                (y[a] == y[b] && y[a+j] == y[b+j]) ? p-1 : p++;
        }
        fwd(i, 1, n) rank[sa[i]] = i;
        for (int i = 0, j; i < n-1; lcp[rank[i++]] = k)
            for (k && k--, j = sa[rank[i]-1];
                s[i+k] == s[j+k]; k++);
    }
};
```

## SuffixTree.h

**Description:** Ukkonen's algorithm for online suffix tree construction. Each node contains indices [l, r] into the string, and a list of child nodes. Suffixes are given by traversals of this tree, joining [l, r] substrings. The root is 0 (has l = -1, r = 0), non-existent children are -1. To get a complete tree, append a dummy symbol – otherwise it may contain an incomplete path (still useful for substring matching, though).

**Time:**  $\mathcal{O}(26N)$  f2f561, 50 lines

```
struct SuffixTree {
    enum { N = 200010, ALPHA = 26 }; // N ~ 2*maxlen+10
    int toi(char c) { return c - 'a'; }
    string a; // v = cur node, q = cur position
    int t[N][ALPHA], l[N], r[N], p[N], s[N], v=0, q=0, m=2;
    void ukkadd(int i, int c) { suff:
        if (r[v]<=q) {
            if (t[v][c]==-1) { t[v][c]=m; l[m]=i;
                p[m++]=v; v=s[v]; q=r[v]; goto suff; }
            v=t[v][c]; q=l[v];
        }
        if (q==-1 || c==toi(a[q])) q++; else {
            l[m+1]=i; p[m+1]=m; l[m]=l[v]; r[m]=q;
            p[m]=p[v]; t[m][c]=m+1; t[m][toi(a[q])]=v;
            l[v]=q; p[v]=m; t[p[m]][toi(a[l[m]])]=m;
            v=s[p[m]]; q=l[m];
            while (q<r[m]) { v=t[v][toi(a[q])]; q+=r[v]-l[v]+1; }
            if (q==r[m]) s[m]=v; else s[m]=m+2;
            q=r[v]-(q-r[m]); m+=2; goto suff;
        }
    }
    SuffixTree(string a) : a(a) {
        fill(r, r+N, sz(a));
        memset(s, 0, sizeof s);
        memset(t, -1, sizeof t);
        fill(t[1], t[1]+ALPHA, 0);
        s[0] = 1; l[0] = l[1] = -1; r[0] = r[1] = p[0] = p[1] = 0;
        rep(i, sz(a)) ukkadd(i, toi(a[i]));
    }
    pii best;
    int lcs(int node, int i1, int i2, int olen) {
        if (l[node] <= i1 && i1 < r[node]) return 1;
        if (l[node] <= i2 && i2 < r[node]) return 2;
        int mask = 0, len = node ? olen + (r[node] - l[node] - 1) : 0;
        rep(c, ALPHA) if (t[node][c] != -1)
            mask |= lcs(t[node][c], i1, i2, len);
        if (mask == 3)
            best = max(best, {len, r[node] - len});
        return mask;
    }
};
```

```
static pii LCS(string s, string t) {
    SuffixTree st(s + (char)('z' + 1) + t + (char)('z' + 2));
    st.lcs(0, sz(s), sz(s) + 1 + sz(t), 0);
    return st.best;
}
```

## Hashing.h

**Description:** Self-explanatory methods for string hashing

```
// Arithmetic mod 2^64-1. 2x slower than mod 2^64 and more
// code, but works on evil test data (e.g. Thue-Morse, where
// ABBA... and BAAB... of length 2^10 hash the same mod 2^64).
// "typedef ull H;" instead if you think test data is random,
// or work mod 10^9+7 if the Birthday paradox is not a problem.
typedef uint64_t ull;
struct H {
    ull x; H(ull x=0) : x(x) {}
    H operator+(H o) { return x + o.x + (x + o.x < x); }
    H operator-(H o) { return *this + ~o.x; }
    H operator*(H o) { auto m = (__uint128_t)x * o.x;
        return H((ull)m + (ull)(m >> 64)); }
    ull get() const { return x + !~x; }
    bool operator==(H o) const { return get() == o.get(); }
    bool operator<(H o) const { return get() < o.get(); }
};
static const H C = (1ll)1e11+3; // (order ~ 3e9; random also ok)
struct HashInterval {
    vector<H> ha, pw;
    HashInterval(string& str) : ha(sz(str)+1), pw(ha) {
        pw[0] = 1;
        rep(i, sz(str))
            ha[i+1] = ha[i] * C + str[i],
            pw[i+1] = pw[i] * C;
    }
    H hashInterval(int a, int b) { // hash [a, b)
        return ha[b] - ha[a] * pw[b-a];
    }
};
vector<H> getHashes(string& str, int length) {
    if (sz(str) < length) return {};
    H h = 0, pw = 1;
    rep(i, length)
        h = h * C + str[i], pw = pw * C;
    vector<H> ret = {h};
    fwd(i, length, sz(str)) {
        ret.push_back(h = h * C + str[i] - pw * str[i-length]);
    }
    return ret;
}
H hashString(string& s) {H h{}; for(char c:s) h=h*C+c; return h;}
```

## AhoCorasick.h

**Description:** Aho-Corasick automaton, used for multiple pattern matching. Initialize with AhoCorasick ac(patterns); the automaton start node will be at index 0. find(word) returns for each position the index of the longest word that ends there, or -1 if none. findAll(–, word) finds all words (up to  $N\sqrt{N}$  many if no duplicate patterns) that start at each position (shortest first). Duplicate patterns are allowed; empty patterns are not. To find the longest words that start at each position, reverse all input. For large alphabets, split each symbol into chunks, with sentinel bits for symbol boundaries.

**Time:** construction takes  $\mathcal{O}(26N)$ , where  $N$  = sum of length of patterns. find(x) is  $\mathcal{O}(N)$ , where  $N$  = length of x. findAll is  $\mathcal{O}(NM)$ .

e98ee6, 66 lines

```
struct AhoCorasick {
    enum { alpha = 26, first = 'A' }; // change this!
    struct Node {
        // (nmatches is optional)
        int back, next[alpha], start = -1, end = -1,
            nmatches = 0;
        Node(int v) { memset(next, v, sizeof(next)); }
    };
    vector<Node> N;
    vi backp;
    void insert(string& s, int j) {
        assert(!s.empty());
```

```

int n = 0;
for (char c : s) {
    int& m = N[n].next[c - first];
    if (m == -1) { n = m = sz(N); N.emplace_back(-1);
    }
    else n = m;
}
if (N[n].end == -1) N[n].start = j;
backp.push_back(N[n].end);
N[n].end = j;
N[n].nmatches++;
}
AhoCorasick(vector<string>& pat) : N(1, -1) {
    rep(i, sz(pat)) insert(pat[i], i);
    N[0].back = sz(N);
    N.emplace_back(0);
    queue<int> q;
    for (q.push(0); !q.empty(); q.pop()) {
        int n = q.front(), prev = N[n].back;
        rep(i, alpha) {
            int &ed = N[n].next[i], y = N[prev].next[i];
            if (ed == -1) ed = y;
            else {
                N[ed].back = y;
                (N[ed].end == -1 ? N[ed].end : backp[N[ed].start])
                = N[y].end;
                N[ed].nmatches += N[y].nmatches;
                q.push(ed);
            }
        }
    }
}
vi find(string word) {
    int n = 0;
    vi res; // ll count = 0;
    for (char c : word) {
        n = N[n].next[c - first];
        res.push_back(N[n].end);
        // count += N[n].nmatches;
    }
    return res;
}
vector<vi> findAll(vector<string>& pat, string word)
{
    vi r = find(word);
    vector<vi> res(sz(word));
    rep(i, sz(word)) {
        int ind = r[i];
        while (ind != -1) {
            res[i - sz(pat[ind]) + 1].push_back(ind);
            ind = backp[ind];
        }
    }
    return res;
}
};

```

## PalindromicTree.h

**Description:** Computes plaindromic tree: for each end position in the string we store longest palindrome ending in that position. link is the suffix palindrome links, eg ababa -> aba. Can be used to compute shortest decomposition of strings to palindromes in  $O(n \log n)$  time - use [DP] lines.

**Time:**  $O(N)$  6cbddf, 39 lines

```

constexpr int ALPHA = 26;
struct PalTree {
    vi txt; //;Node 0=empty pal (root of even), 1="-1"
    pal (of odd)
    vi len{0, -1}; // Lengths of palindromes
    vi link{1, 0}; // Suffix palindrome links, eg [ababa]
    -> [aba]
    vector<array<int, ALPHA>> to({}, {}); // egdes, ex:
    aba -> cabac
    int last{0}; // Current node (max suffix pal)
    vi diff{0, 0}; // [DP] len[i]-len[link[i]]
    vi slink{0, 0}; // [DP] like link but to having
    different 'diff'
    vi series{0, 0}; // [DP] dp for series (groups of pals
    with -diff)
    vi ans{0}; // [DP] ans for prefix
    int ext(int i) {
        while (len[i]+2>sz(txt) || txt[sz(txt)-len[i]-2] !=
        txt.back())
            i = link[i];
        return i;
    }
}
void add(int x) //x in [0,ALPHA), time O(1) or O(lg
n) for DP
txt.push_back(x); last = ext(last);

```

```

if (!to[last][x]) {
    len.push_back(len[last] + 2);
    link.push_back(to[ext(link[last])][x]);
    to[last][x] = sz(to);
    to.push_back({});
    diff.push_back(len.back() - len[link.back()]);
    // [DP]
    slink.push_back(diff.back() == diff[link.back()])
    ? // [DP]
    slink[link.back()] : link.back();
    series.push_back(0); // [DP]
}
last = to[last][x];
ans.push_back(INT_MAX);
// [DP]
for (int i = last; len[i] > 0; i = slink[i]) {
    // [DP]
    series[i] = ans[sz(ans) - len[slink[i]] - diff[i]
    - 1]; // [DP]
    if (diff[i] == diff[link[i]]) // [DP]
        series[i] = min(series[i], series[link[i]]);
    // [DP]
    // For even only palindromes set ans only for even
    sz(txt) // [DP]
    ans.back() = min(ans.back(), series[i] + 1);
    // [DP]
}
}
};

```

## Various (10)

### 10.1 Intervals

#### IntervalCover.h

**Description:** Compute indices of smallest set of intervals covering another interval. Intervals should be [inclusive, exclusive). To support [inclusive, inclusive], change (A) to add || R.empty(). Returns empty set on failure (or if G is empty).

**Time:**  $O(N \log N)$  9e9d8d, 19 lines

```

template<class T>
vi cover(pair<T, T> G, vector<pair<T, T>> I) {
    vi S(sz(I)), R;
    iota(all(S), 0);
    sort(all(S), [&](int a, int b) { return I[a] < I[b];
    });
    T cur = G.first;
    int at = 0;
    while (cur < G.second) { // (A)
        pair<T, int> mx = make_pair(cur, -1);
        while (at < sz(I) && I[S[at]].first <= cur) {
            mx = max(mx, make_pair(I[S[at]].second, S[at]));
            at++;
        }
        if (mx.second == -1) return {};
        cur = mx.first;
        R.push_back(mx.second);
    }
    return R;
}
}

```

## ConstantIntervals.h

**Description:** Split a monotone function on [from, to) into a minimal set of half-open intervals on which it has the same value. Runs a callback g for each such interval.

**Usage:** constantIntervals(0, sz(v), [&](int x){return v[x];}, [&](int lo, int hi, T val){...});

**Time:**  $O(k \log \frac{n}{k})$  753a4c, 19 lines

```

template<class F, class G, class T>
void rec(int from, int to, F& f, G& g, int& i, T& p, T
q) {
    if (p == q) return;
    if (from == to) {
        g(i, to, p);
        i = to; p = q;
    }
    else {
        int mid = (from + to) >> 1;
        rec(from, mid, f, g, i, p, f(mid));
        rec(mid+1, to, f, g, i, p, q);
    }
}

```

```

template<class F, class G>
void constantIntervals(int from, int to, F f, G g) {
    if (to <= from) return;
}

```

```

int i = from; auto p = f(i), q = f(to-1);
rec(from, to-1, f, g, i, p, q);
g(i, to, q);
}

```

### 10.2 Misc. algorithms

#### FastKnapsack.h

**Description:** Given N non-negative integer weights w and a non-negative target t, computes the maximum  $S \leq t$  such that S is the sum of some subset of the weights.

**Time:**  $O(N \max(w_i))$  e74d03, 16 lines

```

int knapsack(vi w, int t) {
    int a = 0, b = 0, x;
    while (b < sz(w) && a + w[b] <= t) a += w[b++];
    if (b == sz(w)) return a;
    int m = *max_element(all(w));
    vi u, v(2*m, -1);
    v[a+m-t] = b;
    fwd(i, b, sz(w)) {
        u = v;
        rep(x, m) v[x+w[i]] = max(v[x+w[i]], u[x]);
        for (x = 2*m; --x > m;) fwd(j, max(0, u[x]), v[x])
            v[x-w[j]] = max(v[x-w[j]], j);
    }
    for (a = t; v[a+m-t] < 0; a--);
    return a;
}
}

```

## FastMod.h

**Description:** Compute  $a \% b$  about 5 times faster than usual, where b is constant but not known at compile time. Returns a value congruent to a (mod b) in the range [0, ~~28~~)a02, 8 lines

```

typedef unsigned long long ull;
struct FastMod {
    ull b, m;
    FastMod(ull b) : b(b), m(-1ULL / b) {}
    ull reduce(ull a) { // a % b + (0 or b)
        return a - (ull)((_uint128_t(m) * a) >> 64) * b;
    }
}
};

```

### 10.3 Dynamic programming

#### KnuthDP.h

**Description:** When doing DP on intervals:  $a[i][j] = \min_{i < k < j} (a[i][k] + a[k][j]) + f(i, j)$ , where the (minimal) optimal k increases with both i and j, one can solve intervals in increasing order of length, and search  $k = p[i][j]$  for  $a[i][j]$  only between  $p[i][j-1]$  and  $p[i+1][j]$ . This is known as Knuth DP. Sufficient criteria for this are if  $f(b, c) \leq f(a, d)$  and  $f(a, c) + f(b, d) \leq f(a, d) + f(b, c)$  for all  $a \leq b \leq c \leq d$ . Consider also: LineContainer (ch. Data structures), monotone queues, ternary search.

**Time:**  $O(N^2)$

## DivideAndConquerDP.h

**Description:** Given  $a[i] = \min_{lo(i) \leq k < hi(i)} (f(i, k))$  where the (minimal) optimal k increases with i, computes  $a[i]$  for  $i = L..R-1$ .

**Time:**  $O((N + (hi - lo)) \log N)$  f816e3, 18 lines

```

struct DP { // Modify at will:
    int lo(int ind) { return 0; }
    int hi(int ind) { return ind; }
    ll f(int ind, int k) { return dp[ind][k]; }
    void store(int ind, int k, ll v) { res[ind] = pii(k, v); }
    void rec(int L, int R, int LO, int HI) {
        if (L >= R) return;
        int mid = (L + R) >> 1;
        pair<ll, int> best(LLONG_MAX, LO);
        fwd(k, max(LO, lo(mid)), min(HI, hi(mid)))
            best = min(best, make_pair(f(mid, k), k));
        store(mid, best.second, best.first);
        rec(L, mid, LO, best.second+1);
        rec(mid+1, R, best.second, HI);
    }
    void solve(int L, int R) { rec(L, R, INT_MIN, INT_MAX); }
}
};

```

## AliensTrick.h

**Description:** Optimize dp where you want "k things with minimal cost". The slope of f(k) must be non increasing. Provide a function g(lambda) that computes the best answer for any k with costs increased by lambda. 71bca3, 8 lines

```

ll aliens(ll k, auto g) { // returns f(k)
    ll l = 0, r = 1e11; // make sure lambda range [1, r)
    is ok (r > max slope etc)
    while (l + 1 < r) {
        ll m = (l + r) / 2;
        (g(m - 1) + k <= g(m) ? l : r) = m;
    }
    return g(l) - l * k; // return l if you want the
    optimal lambda
}

```