



Universidade Federal do Rio de Janeiro

UFRJ

Felipe Chen, Gabriel de Março, Letícia Freire, Chris Ciafrino, Lucas Melick e Vinícius Lettiéri

2022 and 2023 ICPC World Finals - Luxor

April 18th, 2024

Contest (1)

```
.bashrc
13 lines
b() { g++ -DLOC -O2 -std=c++20 -Wall -W -Wfatal-errors \
    -Wconversion -Wshadow -Wlogical-op \
    -Wfloat-equal -o $1 $1.cpp }
d() { b $@ -O0 -g -D_GLIBCXX_DEBUG -fsanitize=address,undefined)
run() { $@ && echo start >&2 && time ./ $2)
hash-cpp() { sed -n $2' '$3' p' $1 | sed '/^#w/d' | cpp -dD -P \
    -fpreprocessed | tr -d '[:space:]' | md5sum | cut -c-6)
# Other compilation flags:
# -Wformat=2 -Wshift-overflow=2 -Wcast-qual
# -Wcast-align -Wduplicated-cond
# -D_GLIBCXX_DEBUG_PEDANTIC -D_FORTIFY_SOURCE=2
# -fno-sanitize-recover -fstack-protector
# Print optimization info: -fopt-info-all
```

Mathematics (2)

2.1 Recurrences

If $a_n = c_1a_{n-1} + \dots + c_ka_{n-k}$, and r_1, \dots, r_k are distinct roots of $x^k - c_1x^{k-1} - \dots - c_k$, there are d_1, \dots, d_k s.t.

$$a_n = d_1r_1^n + \dots + d_kr_k^n.$$

Non-distinct roots r become polynomial factors, e.g. $a_n = (d_1n + d_2)r^n$.

2.2 Trigonometry

$$\sin(v + w) = \sin v \cos w + \cos v \sin w$$
$$\cos(v + w) = \cos v \cos w - \sin v \sin w$$

$$\tan(v + w) = \frac{\tan v + \tan w}{1 - \tan v \tan w}$$
$$\sin v + \sin w = 2 \sin \frac{v + w}{2} \cos \frac{v - w}{2}$$
$$\cos v + \cos w = 2 \cos \frac{v + w}{2} \cos \frac{v - w}{2}$$

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

where V, W are lengths of sides opposite angles v, w .

$$a \cos x + b \sin x = r \cos(x - \phi)$$
$$a \sin x + b \cos x = r \sin(x + \phi)$$

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

2.3 Geometry

2.3.1 Triangles

Side lengths: a, b, c

Semiperimeter: $p = \frac{a + b + c}{2}$

Area: $A = \sqrt{p(p - a)(p - b)(p - c)}$

.bashrc

Circumradius: $R = \frac{abc}{4A}$

Inradius: $r = \frac{A}{p}$

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

Pick's: A polygon on an integer grid strictly containing i lattice points and having b lattice points on the boundary has area $i + \frac{b}{2} - 1$. (Nothing similar in higher dimensions)

2.3.2 Quadrilaterals

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

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

For cyclic quadrilaterals the sum of opposite angles is 180° , $ef = ac + bd$, and $A = \sqrt{(p - a)(p - b)(p - c)(p - d)}$.

2.3.3 Spherical coordinates



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

2.4 Derivatives/Integrals

$$\frac{d}{dx} \arcsin x = \frac{1}{\sqrt{1 - x^2}}$$
$$\frac{d}{dx} \arccos x = -\frac{1}{\sqrt{1 - x^2}}$$
$$\frac{d}{dx} \tan x = 1 + \tan^2 x$$
$$\frac{d}{dx} \arctan x = \frac{1}{1 + x^2}$$
$$\int \tan ax = -\frac{\ln |\cos ax|}{a}$$
$$\int x \sin ax = \frac{\sin ax - ax \cos ax}{a^2}$$
$$\int e^{-x^2} = \frac{\sqrt{\pi}}{2} \text{erf}(x)$$
$$\int xe^{ax} dx = \frac{e^{ax}}{a^2} (ax - 1)$$

2.4.1 XOR sum

$$\bigoplus_{x=0}^{n-1} x = \{0, n - 1, 1, n\}[n \bmod 4]$$
$$\bigoplus_{x=l}^{r-1} x = \bigoplus_{a=0}^{r-1} a \oplus \bigoplus_{b=0}^{l-1} b$$

2.5 Sums

$$c^a + c^{a+1} + \dots + c^b = \frac{c^{b+1} - c^a}{c - 1}, c \neq 1$$

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

2.6 Series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots, (-\infty < x < \infty)$$
$$\ln(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \dots, (-1 < x \leq 1)$$
$$\sqrt{1 + x} = 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{2x^3}{32} - \frac{5x^4}{128} + \dots, (-1 \leq x \leq 1)$$
$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots, (-\infty < x < \infty)$$
$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots, (-\infty < x < \infty)$$

2.7 Probability theory

Let X be a discrete random variable with probability $p_X(x)$ of assuming the value x . It will then have an expected value (mean) $\mu = \mathbb{E}(X) = \sum_x x p_X(x)$ and variance $\sigma^2 = V(X) = \mathbb{E}(X^2) - (\mathbb{E}(X))^2 = \sum_x (x - \mathbb{E}(X))^2 p_X(x)$ where σ is the standard deviation. If X is instead continuous it will have a probability density function $f_X(x)$ and the sums above will instead be integrals with $p_X(x)$ replaced by $f_X(x)$.

Expectation is linear:

$$\mathbb{E}(aX + bY) = a\mathbb{E}(X) + b\mathbb{E}(Y)$$

For independent X and Y ,

$$V(aX + bY) = a^2V(X) + b^2V(Y).$$

2.7.1 Discrete distributions

Binomial distribution

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

$$p(k) = \binom{n}{k} p^k (1 - p)^{n-k}$$

$$\mu = np, \sigma^2 = np(1 - p)$$

$\text{Bin}(n, p)$ is approximately $\text{Po}(np)$ for small p .

2.8 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 $p_{ij} = \Pr(X_n = i | X_{n-1} = j)$, and $\mathbf{p}^{(n)} = \mathbf{P}^n \mathbf{p}^{(0)}$ is the probability distribution for X_n (i.e., $p_i^{(n)} = \Pr(X_n = i)$), where $\mathbf{p}^{(0)}$ is the initial distribution.

π 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 . π_j / π_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, π_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). $\lim_{k \rightarrow \infty} \mathbf{P}^k = \mathbf{1P}$.

A Markov chain is an absorbing chain if

- 1. there is at least one absorbing state and
- 2. it is possible to go from any state to at least one absorbing state in a finite number of steps.

A Markov chain is an A-chain if the states can be partitioned into two sets \mathbf{A} and \mathbf{G} , such that all states in \mathbf{A} are absorbing ($p_{ii} = 1$), and all states in \mathbf{G} leads to an absorbing state in \mathbf{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)

pb-ds.h

Description: Faster/better hash maps, taken from teapot. And set (not multiset!) with support for finding the n'th element, and finding the index of an element.

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

```
<bits/extc++.h>
ef3b3c, 25 lines

const size_t HXOR = std::mt19937_64(time(0))();
template<class T> struct SafeHash {
    size_t operator()(const T& x) const {
        return std::hash<T>()(x ^ T(HXOR));
    }
};
template <typename K, typename V, typename Hash = SafeHash<K>>
using hash_map = __gnu_pbds::gp_hash_table<K, V, Hash>;
template <typename K, typename Hash = SafeHash<K>>
using hash_set = hash_map<K, __gnu_pbds::null_type, Hash>;
template <typename K, typename V, typename Comp = std::less<K>>
using ordered_map = __gnu_pbds::tree<K, V, Comp,
    __gnu_pbds::rb_tree_tag,
    __gnu_pbds::tree_order_statistics_node_update>;
template <typename K, typename Comp = std::less<K>>
using ordered_set = ordered_map<K, __gnu_pbds::null_type, Comp>;
void example() {
    ordered_set<int> t, t2; t.insert(8);
    auto it = t.insert(10).first;
    assert(it == t.lower_bound(9));
    assert(t.order_of_key(10) == 1); // num strictly smaller
    assert(t.order_of_key(11) == 2);
    assert(*t.find_by_order(0) == 8);
    t.join(t2); // assuming T < T2 or T > T2, merge t2 into t
}
```

dsu-rollback.h

Description: Disjoint-set data structure with undo.

Usage: int t = uf.time(); ...; uf.rollback(t);

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

```
7ddfd1d, 21 lines

struct RollbackUF {
    vector<int> e; vector<pair<int,int>> 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 st.size(); }
    void rollback(int t) { // hash-1
        for (int i = time(); i --> t;)
            e[st[i].first] = st[i].second;
        st.resize(t);
    } // hash-1 = 30bb61
    bool unite(int a, int b) { // hash-2
        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;
    } // hash-2 = a7445e
};
```

monotonic-queue.h

Description: Supports pop and push queue-like, and add function adds a constant to all elements currently in the queue.

Time: $\mathcal{O}(1)$

```
0dfbe6, 18 lines

template<typename T, typename Comp> struct monotonic_queue {
    int lo, hi; T S; // hash-1
    deque<pair<T, T>> q;
    monotonic_queue() : lo(0), hi(0), S(0) {}
    void push(T val) {
```

```
while(!q.empty() && Comp()(val, q.back().first + S))
    q.pop_back();
q.emplace_back(val - S, hi++);
}
void pop() {
    if (!q.empty() && q.front().second == lo++) q.pop_front();
} // hash-1 = d8c3c1
void add(T val) { S += val; }
T get_val() const { return q.front().first + S; }
int size() const { return hi-lo; }
};
template<typename T> using min_queue = monotonic_queue<T,
    less_equal<T>>;
template<typename T> using max_queue = monotonic_queue<T,
    greater_equal<T>>;
```

point-context.h

Description: Examples of Segment Tree

228ba0, 20 lines

```
struct seg_node {
    int val, int mi, ma;
    seg_node() : mi(INT_MAX), ma(INT_MIN), val(0) {}
    seg_node(int x) : mi(x), ma(x), val(x) {}
    void merge(const seg_node& l, const seg_node& r) {
        val = l.val + r.val;
        mi = min(l.mi, r.mi), ma = max(l.ma, r.ma);
    }
    void update(int x) { mi = ma = val = x; }
    bool acc_min(int& acc, int x) const {
        if (x >= mi) return true;
        if (acc > mi) acc = mi;
        return false;
    }
    bool acc_max(int& acc, int x) const {
        if (x <= ma) return true;
        if (acc < ma) acc = ma;
        return false;
    }
};
```

rec-lazy-segtree.h

Description: Segment Tree with Lazy update (half-open interval).

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

22037e, 48 lines

```
template<class T> struct segtree_range { // hash-1
    int N; vector<T> ts;
    segtree_range() {}
    segtree_range(int M) : segtree_range(vector<T>(M, T(0))) {}
    template<class Q> segtree_range(const vector<Q>& A) {
        const int N_ = int(A.size());
        N = (1 << __lg(2*N_-1)); ts.resize(2*N);
        for (int i = 0; i < N_; ++i) at(i) = T(A[i]);
        build();
    } // hash-1 = 37f0ae
    T& at(int a) { return ts[a + N]; }
    void build() { for (int a = N; --a; ) merge(a); }
    inline void push(int a) { ts[a].push(ts[2*a], ts[2*a+1]); }
    inline void merge(int a) { ts[a].merge(ts[2*a], ts[2*a+1]); }
    T query(int v, int l, int r, int a, int b) { // hash-2
        if (l >= b || r <= a) return T();
        if (l >= a && r <= b) return ts[v];
        int m = (l + r)/2; push(v); T t;
        t.merge(query(2*v, l, m, a, b), query(2*v+1, m, r, a, b));
        return t;
    } // hash-2 = eb3198
    T query(int a, int b) { return query(1, 0, N, a, b); }
    template<class F, class... Args> void update(int v, int l,
        int r, int a, int b, F f, Args&&... args) { // hash-3
        if (l >= b || r <= a) return;
```

```

    if (l >= a && r <= b && (ts[v].*f)(args...)) return;
    int m = (l + r)/2; push(v);
    update(2*v, l, m, a, b, f, args...);
    update(2*v+1, m, r, a, b, f, args...);
    merge(v);
} // hash-3 = 02f5df
template<class F, class... Args>
void update(int a, int b, F f, Args&&... args) {
    update(l, 0, N, a, b, f, args...);
}
template<class F, class... Args> int find_first(int v, int l,
    int r, int a, int b, F f, Args&&... args) { // hash-4
    if (l >= b || r <= a || !(ts[v].*f)(args...)) return -1;
    if (l + 1 == r) return l;
    int m = (l + r)/2; push(v);
    int cur = find_first(2*v, l, m, a, b, f, args...);
    if (cur == -1)
        cur = find_first(2*v+1, m, r, a, b, f, args...);
    return cur;
} // hash-4 = 68aa30
template<class F, class... Args>
int find_first(int a, int b, F f, Args&&... args) {
    return find_first(l, 0, N, a, b, f, args...);
}
};

```

lazy-context.h

Description: Examples of Segment Tree with Lazy update 3ded8f, 172 lines

```

template<typename T = int64_t> struct seg_node { // hash-1
    T val, lz_add, lz_set;
    int sz; bool to_set;
    seg_node(T n = 0) : val(n), lz_add(0), lz_set(0), sz(1),
        to_set(0) {}
    void push(seg_node& l, seg_node& r) {
        if (to_set) {
            l.assign(lz_set), r.assign(lz_set);
            lz_set = 0; to_set = false;
        }
        if (lz_add != 0) {
            l.add(lz_add), r.add(lz_add), lz_add = 0;
        }
    }
    void merge(const seg_node& l, const seg_node& r) {
        sz = l.sz + r.sz; val = l.val + r.val;
    }
    bool add(T v) { // update range a[i] <- a[i] + v
        val += v * sz; lz_add += v; return true;
    }
    bool assign(T v) { // update range a[i] <- v
        val = v * sz; lz_add = 0;
        lz_set = v; to_set = true; return true;
    }
    T get_sum() const { return val; } // sum a[l, r)
}; // hash-1 = 42f915
// update range a[i] <- a[i] + b * (i - s) + c
// assuming b and c are non zero, be careful
// get sum a[l, r)
template<typename T = int64_t> struct seg_node { // hash-2
    T sum, lzB, lzC;
    int sz, idx;
    seg_node(int id = 0, T v = 0, int s = 0, T b = 0, T c = 0) :
        sum(v), lzB(b), lzC(c - s * b), idx(id), sz(1) {}
    void push(seg_node& l, seg_node& r) {
        l.add(lzB, lzC), r.add(lzB, lzC);
        lzB = lzC = 0;
    }
    void merge(const seg_node& l, const seg_node& r) {
        idx = min(l.idx, r.idx), sz = l.sz + r.sz;

```

```

        sum = l.sum + r.sum;
    }
    T sum_idx(T n) const { return n * (n + 1) / 2; }
    bool add(T b, T c) {
        sum += b * (sum_idx(idx + sz) - sum_idx(idx)) + sz * c;
        lzB += b, lzC += c; return true;
    }
    T get_sum() const { return sum; }
}; // hash-2 = ca218b
// update range a[i] <- b * a[i] + c
// get sum a[l, r)
struct seg_node { // hash-3
    int sz, ll sum, lzB, lzC;
    seg_node() : sz(1), sum(0), lzB(1), lzC(0) {}
    seg_node(ll v) : sz(1), sum(v), lzB(1), lzC(0) {}
    void push(seg_node& l, seg_node& r) {
        l.add(lzB, lzC), r.add(lzB, lzC);
        lzB = 1, lzC = 0;
    }
    void merge(const seg_node& l, const seg_node& r) {
        sz = l.sz + r.sz, sum = l.sum + r.sum;
    }
    bool add(ll b, ll c) {
        sum = (b * sum + c * sz), lzB = (lzB * b);
        lzC = (lzC * b + c); return true;
    }
    ll get_sum() const { return sum; }
}; // hash-3 = 6e5cd5
// update range a[i] <- min(a[i], b);
// update range a[i] <- max(a[i], b);
// get val a[i]
struct seg_node { // hash-4
    int mn, mx;
    int lz0, lz1;
    seg_node() : mn(INT_MAX), mx(INT_MIN), lz0(INT_MAX), lz1(
        INT_MIN) {}
    void push(seg_node& l, seg_node& r) {
        l.minimize(lz0), l.maximize(lz1);
        r.minimize(lz0), r.maximize(lz1);
        lz0 = INT_MAX, lz1 = INT_MIN;
    }
    void merge(const seg_node& l, const seg_node& r) {
        mn = min(l.mn, r.mn), mx = max(l.mx, r.mx);
    }
    bool minimize(int val) {
        mn = lz0 = min(lz0, val);
        mx = lz1 = min(lz0, lz1); return true;
    }
    bool maximize(int val) {
        mx = lz1 = max(lz1, val);
        mn = lz0 = max(lz0, lz1); return true;
    }
    pair<int, int> get() const { return {mx, mn}; }
}; // hash-4 = a6c9e9

```

```

template<typename T> struct lazy_t { // hash-5
    T a, b, c;
    lazy_t() : a(0), b(-INF), c(+INF) {}
    lazy_t(T a, T b, T c) : a(a), b(b), c(c) {}
    void add(T val) {
        a += val, b += val, c += val;
    }
    void upd_min(T val) {
        if (b > val) b = val;
        if (c > val) c = val;
    }
    void upd_max(T val) {
        if (b < val) b = val;
        if (c < val) c = val;
    }

```

```

    }
}; // hash-5 = 9fb833
template<typename T = int64_t> struct seg_node { // hash-6
    T mi, mi2, ma, ma2, sum;
    lazy_t<T> lz;
    seg_node() : mi(INF), mi2(INF), ma(-INF), ma2(-INF), sum(0),
        cnt_mi(0), cnt_ma(0), sz(0), lz() {}
    seg_node(T n) : mi(n), mi2(INF), ma(n), ma2(-INF), sum(n),
        cnt_mi(1), cnt_ma(1), sz(1), lz() {}
    void push(seg_node& l, seg_node& r) {
        if (!l.can_apply(lz) || !r.can_apply(lz)) return;
        lz = lazy_t<T>();
    }
    bool can_apply(const lazy_t<T>& f) {
        if (!add(f.a) || !upd_max(f.b) || !upd_min(f.c)) return
            false;
        return true;
    } // hash-6 = 7b66a0
    void merge(const seg_node& l, const seg_node& r) { // hash-7
        mi = min(l.mi, r.mi);
        mi2 = min((l.mi == mi) ? l.mi2 : l.mi, (r.mi == mi) ? r.mi2
            : r.mi);
        cnt_mi = ((l.mi == mi) ? l.cnt_mi : 0) + ((r.mi == mi) ? r.
            cnt_mi : 0);
        ma = max(l.ma, r.ma);
        ma2 = max((l.ma == ma) ? l.ma2 : l.ma, (r.ma == ma) ? r.ma2
            : r.ma);
        cnt_ma = ((l.ma == ma) ? l.cnt_ma : 0) + ((r.ma == ma) ? r.
            cnt_ma : 0);
        sum = l.sum + r.sum;
        sz = l.sz + r.sz;
    } // hash-7 = 936019
    bool add(T v) { // a-i = a-i + v // hash-8
        if (v) {
            mi += v;
            if (mi2 < INF) mi2 += v;
            ma += v;
            if (ma2 > -INF) ma2 += v;
            sum += sz * v;
            lz.add(v);
        }
        return true;
    } // hash-8 = 83626c
    bool upd_max(T v) { // a-i = max(a-i, v) // hash-9
        if (v > -INF) {
            if (v >= mi2) return false;
            else if (v > mi) {
                if (ma == mi) ma = v;
                if (ma2 == mi) ma2 = v;
                sum += cnt_mi * (v - mi);
                mi = v;
                lz.upd_max(v);
            }
        }
        return true;
    } // hash-9 = 877a43
    bool upd_min(T v) { // a-i = min(a-i, v) // hash-10
        if (v < INF) {
            if (v <= ma2) return false;
            else if (v < ma) {
                if (ma == mi) mi = v;
                if (mi2 == ma) mi2 = v;
                sum -= cnt_ma * (ma - v);
                ma = v;
                lz.upd_min(v);
            }
        }
        return true;
    }

```

```
    } // hash-10 = 514d75
    T get_sum() const { return sum; } // sum a[l, r)
};
```

sparse-segtree.h
Description: Sparse Segment Tree with point update. Doesnt allocate storage for nodes with no data. Use BumpAllocator for better performance!

```
const int SZ = 1<<19;
template<class T> struct node_t {
    T delta = 0; node_t<T>* c[2];
    node_t() { c[0] = c[1] = nullptr; }
    void upd(int pos, T v, int L = 0, int R = SZ-1) { // add v //
        hash-1
        if (L == pos && R == pos) { delta += v; return; }
        int M = (L + R)>>1;
        if (pos <= M) {
            if (!c[0]) c[0] = new node_t();
            c[0]->upd(pos, v, L, M);
        } else {
            if (!c[1]) c[1] = new node_t();
            c[1]->upd(pos, v, M+1, R);
        }
        delta = 0;
        for (int i = 0; i < 2; ++i) if (c[i]) delta += c[i]->delta;
    } // hash-1 = 8c7b01
    T query(int lx, int rx, int L = 0, int R = SZ-1) { // query
        sum of segment // hash-2
        if (rx < L || R < lx) return 0;
        if (lx <= L && R <= rx) return delta;
        int M = (L + R)>>1; T res = 0;
        if (c[0]) res += c[0]->query(lx, rx, L, M);
        if (c[1]) res += c[1]->query(lx, rx, M+1, R);
        return res;
    } // hash-2 = 6aa202
    void upd(int pos, node_t *a, node_t *b, int L = 0, int R = SZ
        -1) { // hash-3
        if (L != R) {
            int M = (L + R)>>1;
            if (pos <= M) {
                if (!c[0]) c[0] = new node_t();
                c[0]->upd(pos, a ? a->c[0] : nullptr, b ? b->c[0] :
                    nullptr, L, M);
            } else {
                if (!c[1]) c[1] = new node_t();
                c[1]->upd(pos, a ? a->c[1] : nullptr, b ? b->c[1] :
                    nullptr, M+1, R);
            }
        }
        delta = (a ? a->delta : 0)+(b ? b->delta : 0);
    } // hash-3 = 514618
};
```

segtree-2d.h
Description: 2D Segment Tree. For query with manhattan distance $\leq D$, do $nx = x + y$ and $ny = x - y$. Update on (nx, ny) and query on $((nx - d, ny - d), (nx + d, ny + d))$
Time: $\mathcal{O}(N \log^2 N)$ of memory, $\mathcal{O}(\log^2 N)$ per query

```
"sparse-segtree.h"
template<class T> struct Node {
    node_t<T> seg; Node* c[2];
    Node() { c[0] = c[1] = nullptr; }
    void upd(int x, int y, T v, int L = 0, int R = SZ-1) { //add v
        if (L == x && R == x) { seg.upd(y,v); return; }
        int M = (L+R)>>1;
        if (x <= M) {
            if (!c[0]) c[0] = new Node();
            c[0]->upd(x,y,v,L,M);
        } else {
```

```
            if (!c[1]) c[1] = new Node();
            c[1]->upd(x,y,v,M+1,R);
        }
        seg.upd(y,v); // only for addition
        // seg.upd(y,c[0]?&c[0]->seg:nullptr,c[1]?&c[1]->seg:
            nullptr);
    }
    T query(int x1, int x2, int y1, int y2, int L = 0, int R = SZ
        -1) { // query sum of rectangle
        if (x1 <= L && R <= x2) return seg.query(y1,y2);
        if (x2 < L || R < x1) return 0;
        int M = (L+R)>>1; T res = 0;
        if (c[0]) res += c[0]->query(x1, x2, y1, y2, L, M);
        if (c[1]) res += c[1]->query(x1, x2, y1, y2, M+1, R);
        return res;
    }
};
```

persistent-segtree.h
Description: Sparse (N can be up to $1e18$) persistent segment tree.

```
template<typename T> struct seg_tree {
    T idnt = 0; // identity value
    T f(T l, T r) { return l + r; }
    struct node {
        int l = 0, r = 0;
        T x;
        node(T _x) : x(_x) {}
    };
    int N; vector<node> t;
    seg_tree(int _N) : N(_N) { t.push_back(node(idnt)); }
    int cpy(int v) {
        t.push_back(t[v]);
        return int(t.size()) - 1;
    }
    int upd(int v, int p, T x, int a = 0, int b = -1) { // hash-1
        b = ~b ? b : N - 1;
        int u = cpy(v);
        if (a == b) {
            t[u].x += x;
            return u;
        }
        int m = (a + b) / 2;
        if (p <= m) t[u].l = upd(t[v].l, p, x, a, m);
        else t[u].r = upd(t[v].r, p, x, m+1, b);
        t[u].x = f(t[t[u].l].x, t[t[u].r].x);
        return u;
    } // hash-1 = 9c27d1
    T get(int v, int l, int r, int a = 0, int b = -1) { // hash-2
        b = ~b ? b : N - 1;
        if (!v || l > b || r < a) return idnt;
        if (a >= l && b <= r) return t[v].x;
        int m = (a + b) / 2;
        return f(get(t[v].l, l, r, a, m), get(t[v].r, l, r, m+1, b));
    } // hash-2 = 7d1e05
    T get_kth(int l, int r, int k, int a = 0, int b = -1) { //
        hash-3
        b = ~b ? b : N - 1;
        if (a == b) return a;
        int cnt = t[t[r].l].x - t[t[l].l].x;
        int m = (a + b) / 2;
        if (k <= cnt) return get_kth(t[l].l, t[r].l, k, a, m);
        return get_kth(t[l].x, t[r].r, k-cnt, m+1, b);
    } // hash-3 = 732fd0
};
```

rmq.h
Description: Range Minimum/Maximum Queries on an array. Returns $\min(V[a], V[a + 1], \dots V[b])$ in constant time. Returns a pair that holds the answer, first element is the value and the second is the index.
Usage: `rmq.t<pair<int, int>> rmq(values);`
// values is a vector of pairs {val(i), index(i)}
`rmq.query(inclusive, exclusive);`
`rmq.t<pair<int, int>, greater<pair<int, int>>> rmq(values)`
//max query
Time: $\mathcal{O}(|V| \log |V| + Q)$

```
template<typename T, typename Cmp=less<T>>
struct rmq_t : private Cmp {
    int N = 0;
    vector<vector<T>> table;
    const T& min(const T& a, const T& b) const { return Cmp::
        operator()(a, b) ? a : b; }
    rmq_t() {}
    rmq_t(const vector<T>& values) : N(int(values.size())), table
        (__lg(N) + 1) {
        table[0] = values;
        for (int a = 1; a < int(table.size()); ++a) {
            table[a].resize(N - (1 << a) + 1);
            for (int b = 0; b + (1 << a) <= N; ++b)
                table[a][b] = min(table[a-1][b], table[a-1][b+(1<<(a-1)
                    )]);
        }
    }
    T query(int a, int b) const {
        int lg = __lg(b - a);
        return min(table[lg][a], table[lg][b - (1 << lg) ]);
    }
};
```

fenwick-tree.h
Description: Computes partial sums $a[0] + a[1] + \dots + a[pos - 1]$, and updates single elements $a[i]$, taking the difference between the old and new value.
Time: Both operations are $\mathcal{O}(\log N)$.

```
template<typename T> struct FT {
    vector<T> s;
    FT(int n) : s(n) {}
    FT(const vector<T>& A) : s(A) {
        const int N = int(s.size());
        for (int a = 0; a < N; ++a)
            if ((a | (a + 1)) < N) s[a | (a + 1)] += s[a];
    }
    void update(int pos, T dif) { // a[pos] += dif
        for (; pos < (int)s.size(); pos |= pos + 1) s[pos] += dif;
    }
    T query(int pos) { // sum of values in [0, pos)
        T res = 0;
        for (; pos > 0; pos &= pos - 1) res += s[pos-1];
        return res;
    }
    // min pos st sum of [0, pos] >= sum. Returns n if no sum
    int lower_bound(T 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 <= (int)s.size() && s[pos + pw-1] < sum)
                pos += pw, sum -= s[pos-1];
        return pos;
    }
};
```


fenwick-tree-2d.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 $\mathcal{O}(\log N)$.)

"fenwick-tree.h" aebdbc, 25 lines

```
template<typename T> struct FT2 {
    vector<vector<int>>> ys; vector<FT<T>> ft;
    FT2(int limx) : ys(limx) {}
    void fakeUpdate(int x, int y) {
        for (; x < (int)ys.size(); x |= x + 1) ys[x].push_back(y);
    }
    void init() {
        for(auto &v : ys){
            sort(v.begin(), v.end());
            v.resize(unique(v.begin(),v.end()) - v.begin());
            ft.emplace_back(v.size());
        }
    }
    int ind(int x, int y) {
        return (int)(lower_bound(ys[x].begin(), ys[x].end(), y) -
            ys[x].begin()); }
    void update(int x, int y, T dif) {
        for (; x < ys.size(); x |= x + 1)
            ft[x].update(ind(x, y), dif);
    }
    T query(int x, int y) {
        T sum = 0;
        for (; x; x &= x - 1) sum += ft[x-1].query(ind(x-1, y));
        return sum;
    }
};
```

mo.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: $\mathcal{O}(N\sqrt{Q})$

5ef29d, 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
```

```
vector<int> mo(vector<pair<int, int>> Q) { // hash-1
    int L = 0, R = 0, blk = 350; // ~N/sqrt(Q)
    vector<int> s(int(Q.size())), res = s;
    #define K(x) pair<int, int>(x.first/blk, x.second ^ -(x.first/
        blk & 1))
    iota(s.begin(), s.end(), 0);
    sort(s.begin(), s.end(), [&](int s, int t){ return K(Q[s]) <
        K(Q[t]); });
    for (int qi : s) {
        auto 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;
} // hash-1 = d9247c
```

```
vector<int> moTree(vector<array<int, 2>> Q, vector<vector<int
    >>& ed, int root=0){ // hash-2
    int N = int(ed.size()), pos[2] = {}, blk = 350; // ~N/sqrt(Q)
    vector<int> s(int(Q.size())), 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(s.begin(), s.end(), 0);
sort(s.begin(), s.end(), [&](int s, int t){ return K(Q[s]) <
    K(Q[t]); });
for (int qi : s) for (int end = 0; end < 2; ++end) {
    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;
} // hash-2 = bbf891
```

line-container.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: $\mathcal{O}(\log N)$

cd3f16, 27 lines

```
struct Line { // hash-1
    mutable ll k, m, p;
    bool operator<(const Line& o) const { return k < o.k; }
    bool operator<(ll x) const { return p < x; }
}; // hash-1 = 7e3ecf
struct LineContainer : multiset<Line, less<>> {
    static const ll inf = LLONG_MAX; //for doubles 1/0
    ll div(ll a, ll b) { //for doubles a/b
        return a / b - ((a ^ b) < 0 && a % b); }
    bool isect(iterator x, iterator y) { // hash-2
        if (y == end()) { x->p = inf; return false; }
        if (x->k == y->k) x->p = x->m > y->m ? inf : -inf;
        else x->p = div(y->m - x->m, x->k - y->k);
        return x->p >= y->p;
    } // hash-2 = ea7809
    void add(ll k, ll m) { // hash-3
        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));
        // hash-3 = 08625f
        ll query(ll x) { // hash-4
            assert(!empty()); auto l = *lower_bound(x);
            return l.k * x + l.m;
        } // hash-4 = d21e2f
};
```

matrix.h

Description: Basic operations on square matrices.

Usage: Matrix<int> A(N, vector<int>(N));

447637, 28 lines

```
template <typename T> struct Matrix : vector<vector<T>>> {
    using vector<vector<T>>>::vector;
    using vector<vector<T>>>::size;
    int h() const { return int(size()); }
    int w() const { return int((*this)[0].size()); }
```

```
Matrix operator*(const Matrix& r) const {
    assert(w() == r.h()); Matrix res(h(), vector<T>(r.w()));
    for(int i = 0; i < h(); ++i) for(int j = 0; j < r.w(); ++j)
        for (int k = 0; k < w(); ++k)
            res[i][j] += (*this)[i][k] * r[k][j];
    return res;
}
friend auto operator*(const Matrix<T>& A, const vector<T>& b) {
    int N = int(A.size()), M = int(A[0].size());
    vector<T> y(N);
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < M; ++j) y[i] += A[i][j] * b[j];
    return y;
}
Matrix& operator*=(const Matrix& r){return *this= *this * r;}
Matrix pow(ll n) const {
    assert(h() == w()); assert(n >= 0);
    Matrix x = *this, r(h(), vector<T>(w()));
    for (int i = 0; i < h(); ++i) r[i][i] = T(1);
    while (n) { if (n & 1) r *= x; x *= x; n >>= 1; }
    return r;
}
};
```

range-color.h

Description: RangeColor structure, supports point queries and range updates, if C isn’t int32_t change freq to map

Time: $\mathcal{O}(\lg(L) * Q)$

3d860e, 35 lines

```
template<class T, class C> struct RangeColor { // hash-1
    struct Node{
        T lo, hi; C color;
        bool operator<(const Node &n) const { return hi < n.hi; }
    };
    C minInf; set<Node> st; vector<T> freq;
    RangeColor(T first, T last, C maxColor, C iniColor = C(0)) :
        minInf(first - T(1)), freq(maxColor + 1) {
        freq[iniColor] = last - first + T(1);
        st.insert({first, last, iniColor});
    } // hash-1 = b9833d
    C query(T i) { //get color in position i
        return st.upper_bound({T(0), i - T(1), minInf})->color;
    }
    void upd(T a, T b, C x) { //set x in [a, b] // hash-2
        auto p = st.upper_bound({T(0), a - T(1), minInf});
        assert(p != st.end());
        T lo = p->lo, hi = p->hi; C old = p->color;
        freq[old] -= (hi - lo + T(1)); p = st.erase(p);
        if (lo < a)
            freq[old] += (a-lo), st.insert({lo, a-T(1), old});
        if (b < hi)
            freq[old] += (hi-b), st.insert({b+T(1), hi, old});
        while ((p != st.end()) && (p->lo <= b)) {
            lo = p->lo, hi = p->hi; old = p->color;
            freq[old] -= (hi - lo + T(1));
            if (b < hi){
                freq[old] += (hi - b); st.erase(p);
                st.insert({b + T(1), hi, old});
                break;
            } else p = st.erase(p);
        }
        freq[x] += (b - a + T(1)); st.insert({a, b, x});
    } // hash-2 = 7575b8
    T countColor(C x){ return freq[x]; }
};
```

implicit-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: $\mathcal{O}(\log N)$

```
8c89db, 76 lines
struct node { // hash-1
    int val, p, sz; bool rev;
    array<node*, 2> c(nullptr, nullptr);
    node(int k) : val(k), p(rng()), sz(0), rev(false) {}
    ~node() {
        delete c[0];
        delete c[1];
    }
}; // hash-1 = 0055b5
inline int sz(node *t) { // hash-2
    return (!t ? 0 : t->sz);
} // hash-2 = 99838d
inline void push(node *t) { // hash-3
    if (!t) return;
    if (t->rev) {
        swap(t->c[0], t->c[1]);
        if (t->c[0]) t->c[0]->rev ^= t->rev;
        if (t->c[1]) t->c[1]->rev ^= t->rev;
        t->rev = 0;
    }
} // hash-3 = 910227
inline void pull(node *t) { // hash-4
    if (!t) return;
    push(t); push(t->c[0]); push(t->c[1]);
    t->sz = sz(t->c[0]) + sz(t->c[1]) + 1;
} // hash-4 = 85cde9
inline void split(node *t, node *&a, node *&b, int k) { //k on left
    push(t);
    if (!t) a = b = nullptr;
    else if (k <= sz(t->c[0])) {
        split(t->c[0], a, t->c[0], k);
        b = t;
    } else {
        split(t->c[1], t->c[1], b, k-1-sz(t->c[0]));
        a = t;
    }
    pull(t);
}
inline void merge(node *&t, node *a, node *b) { // hash-5
    push(a); push(b);
    if (!a) t = b;
    else if (!b) t = a;
    else if (a->p <= b->p) {
        merge(a->c[1], a->c[1], b);
        t = a;
    } else {
        merge(b->c[0], a, b->c[0]);
        t = b;
    }
    pull(t);
} // hash-5 = 34473a
inline void add(node *&t, node *a, int k) { // hash-6
    push(t);
    if (!t) t = a;
    else if (a->p >= t->p) {
        split(t, a->c[0], a->c[1], k);
        t = a;
    } else if (sz(t->c[0]) >= k) add(t->c[0], a, k);
    else add(t->c[1], a, k-1-sz(t->c[0]));
    pull(t);
} // hash-6 = 58c383
void del(node *&t, int k) { // hash-7
    push(t);
```

```
if (!t) return;
if (sz(t->c[0]) == k) merge(t, t->c[0], t->c[1]);
else if (sz(t->c[0]) > k) del(t->c[0], k);
else del(t->c[1], k);
pull(t);
} // hash-7 = a097c3
inline void dump_treap(node *t) { // hash-8
    if (!t) return;
    push(t);
    dump_treap(t->c[0]);
    cerr << t->val << ' ';
    dump_treap(t->c[1]);
} // hash-8 = 8675f9
```

Numerical (4)

poly-interpolate.h

Description: Given n points $(x[i], y[i])$, computes an $n-1$ -degree polynomial p that passes through them: $p(x) = a[0] * x^0 + \dots + a[n-1] * x^{n-1}$. For numerical precision, pick $x[k] = c * \cos(k/(n-1) * \pi), k = 0 \dots n-1$.
Time: $\mathcal{O}(n^2)$

```
97a266, 12 lines
typedef vector<double> vd;
vd interpolate(vd x, vd y, int n) {
    vd res(n), temp(n);
    for(int k = 0; k < n-1; ++k) for(int i = k+1; i < n; ++i)
        y[i] = (y[i] - y[k]) / (x[i] - x[k]);
    double last = 0; temp[0] = 1;
    for(int k = 0; k < n; ++k) for(int i = 0; i < n; ++i) {
        res[i] += y[k] * temp[i]; swap(last, temp[i]);
        temp[i] -= last * x[k];
    }
    return res;
}
```

lagrange.h

Description: Lagrange interpolation over a finite field and some combo stuff
Time: $\mathcal{O}(N)$

```
"../number-theory/modular-arithmetic.h", "../number-theory/preparator.h" ad3879, 13 lines
template<typename T> struct interpolator_t {
    vector<T> S;
    interpolator_t(int N): S(N) {}
    T interpolate(const vector<T>& y, T x) { // hash-1
        int N = int(y.size()); int sgn = (N & 1 ? 1 : -1);
        T res = 0, P = 1; S[N-1] = 1;
        for (int i = N-1; i > 0; --i) S[i-1] = S[i] * (x-i);
        for (int i = 0; i < N; ++i, sgn *= -1, P *= (x-i+1)) {
            res += y[i] * sgn * P * S[i] * ifact[i] * ifact[N-1-i];
        }
        return res;
    } // hash-1 = 2fb780
};
```

berlekamp-massey.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/modular-arithmetic.h" 66d78a, 17 lines
template <typename num>
vector<num> BerlekampMassey(const vector<num>& s) {
    int n = int(s.size()), L = 0, m = 0; num b = 1;
    vector<num> C(n), B(n), T; C[0] = B[0] = 1;
    for(int i = 0; i < n; i++) { +m; // hash-1
```

```
num d = s[i];
for (int j = 1; j <= L; j++) d += C[j] * s[i-j];
if (d == 0) continue;
T = C; num coef = d / b;
for (int j = m; j < n; j++) C[j] -= coef * B[j-m];
if (2 * L > i) continue;
L = i + 1 - L; B = T; b = d; m = 0;
} // hash-1 = ebf5ec
C.resize(L+1); C.erase(C.begin());
for (auto& x : C) x = -x;
return C;
}
```

linear-recurrence.h

Description: Bostan-Mori algorithm. Generates the k 'th term of an n -order linear recurrence $S[i] = \sum_j S[i-j-1]tr[j]$, given $S[0 \dots n-1]$ and $tr[0 \dots n-1]$. Faster than matrix multiplication. Useful together with Berlekamp-Massey.
Usage: linear_rec({0, 1}, {1, 1}, k) // k 'th Fibonacci number
Time: $\mathcal{O}(n \log n \log k)$

```
"../number-theory/modular-arithmetic.h" aa7314, 16 lines
template<typename T>
T linear_rec(const vector<T>& S, const vector<T>& tr, ll K) {
    const int N = int(tr.size());
    vector<T> qs(N+1); qs[0] = 1;
    for (int i = 0; i < N; ++i) qs[i+1] = -tr[i];
    auto fs = fft.convolve(S, qs); fs.resize(N);
    for (; K; K /= 2) { // hash-1
        auto qneg = qs;
        for (int i = 1; i <= N; i += 2) qneg[i] = -qneg[i];
        fs = fft.convolve(fs, qneg); qs = fft.convolve(qs, qneg);
        for (int i = 0; i < N; ++i)
            fs[i] = fs[2 * i + (K & 1)], qs[i] = qs[2 * i];
        qs[N] = qs[2*N]; fs.resize(N), qs.resize(N+1);
    } // hash-1 = 641c09
    return fs[0];
}
```

integrate.h

Description: Simple integration of a function over an interval using Simpson's rule. The error should be proportional to h^4 , although in practice you will want to verify that the result is stable to desired precision when epsilon changes.

```
7bb98e, 7 lines
template<class F>
double quad(double a, double b, F& f, const int n = 1000) {
    double h = (b-a) / 2 / n, v = f(a) + f(b);
    for(int i = 1; i < n*2; ++i)
        v += f(a+i*h) * (i&1 ? 4 : 2);
    return v * h / 3;
}
```

integrate-adaptive.h

Description: Fast integration using an adaptive Simpson's rule.
Usage: double sphereVolume = quad(-1, 1, [](double x) { return quad(-1, 1, [&](double y) { return quad(-1, 1, [&](double z) { return x*x + y*y + z*z < 1; }}});});

```
cfcad2, 13 lines
typedef double d;
#define S(a,b) (f(a) + 4*f((a+b) / 2) + f(b)) * (b-a) / 6
template <class F>
d rec(F& f, d a, d b, d eps, d S) {
    d c = (a + b) / 2, S1 = S(a, c), S2 = S(c, b), T = S1 + S2;
    if (abs(T - S) <= 15 * eps || b - a < 1e-10)
        return T + (T - S) / 15;
    return rec(f, a, c, eps / 2, S1) + rec(f, c, b, eps / 2, S2);
}
```

```
template<class F>
d quad(d a, d b, F f, d eps = 1e-8) {
    return rec(f, a, b, eps, S(a, b));
}
```

gaussian-elimination.h

Time: $\mathcal{O}(\min(N, M)NM)$

../data-structures/matrix.h" 059077, 61 lines

```
template<typename T> struct gaussian_elimination {
    int N, M; Matrix<T> A, E;
    vector<int> pivot; int rank, nullity, sgn;
    gaussian_elimination(const Matrix<T>& A_) : A(A_) { // hash-1
        N = A.size(), M = A[0].size(), E=Matrix<T>(N, vector<T>(N));
        for (int i = 0; i < N; ++i) E[i][i] = 1;
        rank = 0, nullity = M, sgn = 0; pivot.assign(M, -1);
        for (int col = 0, row = 0; col < M && row < N; ++col) {
            int sel = -1;
            for (int i = row; i < N; ++i) if (A[i][col] != 0) {
                sel = i; break;
            }
            if (sel == -1) continue;
            if (sel != row) {
                sgn += 1;
                swap(A[sel], A[row]); swap(E[sel], E[row]);
            }
            for (int i = 0; i < N; ++i) {
                if (i == row) continue;
                T c = A[i][col] / A[row][col];
                for (int j = col; j < M; ++j)
                    A[i][j] -= c*A[row][j];
                for (int j = 0; j < N; ++j)
                    E[i][j] -= c*E[row][j];
            }
            pivot[col] = row++; ++rank, --nullity;
        }
        // hash-1 = d52725
        auto solve(vector<T> b, bool reduced=false) const { // hash-2
            if (reduced == false) b = E * b;
            vector<T> x(M);
            for (int j = 0; j < M; ++j) {
                if (pivot[j] == -1) continue;
                x[j] = b[pivot[j]] / A[pivot[j]][j];
                b[pivot[j]] = 0;
            }
            for (int i = 0; i < N; ++i)
                if (b[i] != 0) return make_pair(false, x);
            return make_pair(true, x);
        }
        // hash-2 = ed96b5
        auto kernel_basis() const { // hash-3
            vector<vector<T>> basis; vector<T> e(M);
            for (int j = 0; j < M; ++j) {
                if (pivot[j] != -1) continue;
                e[j] = 1; auto y = solve(A * e, true).second;
                e[j] = 0, y[j] = -1; basis.push_back(y);
            }
            return basis;
        }
        // hash-3 = 126a4f
        auto inverse() const { // hash-4
            assert(N == M); assert(rank == N);
            Matrix<T> res(N, vector<T>(N));
            vector<T> e(N);
            for (int i = 0; i < N; ++i) {
                e[i] = 1; auto x = solve(e).second;
                for (int j = 0; j < N; ++j) res[j][i] = x[j];
                e[i] = 0;
            }
            return res;
        }
        // hash-4 = a87fb3
    };
};
```

linear-solver-z2.h

Description: Solves $Ax = b$ over \mathbb{F}_2 . If there are multiple solutions, one is returned arbitrarily. Returns true, or false if no solutions. Last column of a is b . c is the rank.

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

```
typedef bitset<2010> bs;
bool gauss(vector<bs> a, bs& ans, int n) {
    int m = int(a.size()), c = 0;
    bs pos; pos.set();
    for (int j = n-1; i; j >= 0; --j) {
        for (i = c; i < m; ++i)
            if (a[i][j]) break;
        if (i == m) continue;
        swap(a[c], a[i]);
        i = c++; pos[j] = 0;
        for (int k = 0; k < m; ++k)
            if (a[k][j] && k != i) a[k] ^= a[i];
    } ans = pos;
    for (int i = 0; i < m; ++i) {
        int ac = 0;
        for (int j = 0; j < n; ++j) {
            if (!a[i][j]) continue;
            if (!pos[j]) pos[j] = 1, ans[j] = ac^a[i][n];
            ac ^= ans[j];
        }
        if (ac != a[i][n]) return false;
    }
    return true;
}
```

simplex.h

Description: Solves a general linear maximization problem: maximize $c^T x$ subject to $Ax \leq b$, $x \geq 0$.

Time: $\mathcal{O}(NM * \#pivots)$, where a pivot may be e.g. an edge relaxation. $\mathcal{O}(2^n)$ in the general case. WARNING: segfaults on empty (size 0) max cx st $Ax \leq b$, $x \geq 0$ do 2 phases; 1st check feasibility; 2nd check boundedness and ans

```
using dbl = double; using vd = vector<dbl>; // hash-1
vd simplex(vector<vd> A, vd b, vd c) { const dbl E = 1e-9;
    int n = A.size(), m = A[0].size() + 1, r = n, s = m-1;
    auto D = vector<vd>(n+2, vd(m+1));
    vector<int> ix = vector<int>(n + m);
    for (int i = 0; i < n + m; ++i) ix[i] = i;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m-1; ++j) D[i][j] = -A[i][j];
        D[i][m-1] = 1; D[i][m] = b[i];
        if (D[r][m] > D[i][m]) r = i;
    } // hash-1 = 0a7467
    for (int j = 0; j < m-1; ++j) D[n][j] = c[j];
    D[n+1][m-1] = -1; int z = 0;
    for (dbl d;;) { // hash-2
        if (r < n) { swap(ix[s], ix[r + m]);
            D[r][s] = 1.0/D[r][s];
            for (int j = 0; j <= m; ++j) if (j != s)
                D[r][j] *= -D[r][s];
            for (int i = 0; i <= n+1; ++i) if (i != r) {
                for (int j = 0; j <= m; ++j)
                    if (j != s) D[i][j] += D[r][j] * D[i][s];
                D[i][s] *= D[r][s];
            }
        }
        r = -1; s = -1;
        for (int j = 0; j < m; ++j) if (s < 0 || ix[s] > ix[j])
            if (D[n+1][j]>E || D[n+1][j]>-E && D[n][j]>E) s = j;
        if (s < 0) break;
        for (int i = 0; i < n; ++i) if (D[i][s] < -E) {
            if (r<0 || (d = D[r][m]/D[r][s]-D[i][m]/D[i][s]) < -E
```

```
        || d < E && ix[r+m] > ix[i+m]) r = i;
    }
    if (r < 0) return vd(); // unbounded
} // hash-2 = dd3d9c
if (D[n+1][m] < -E) return vd(); // infeasible
vd x(m-1);
for (int i=m; i < n+m; ++i) if (ix[i]<m-1) x[ix[i]]=D[i-m][m];
dbl result = D[n][m]; return x; // ans: D[n][m]
}
```

tridiagonal.h

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

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-1} \end{pmatrix} = \begin{pmatrix} d_0 & p_0 & 0 & 0 & \cdots & 0 \\ q_0 & d_1 & p_1 & 0 & \cdots & 0 \\ 0 & q_1 & d_2 & p_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & q_{n-3} & d_{n-2} & p_{n-2} \\ 0 & 0 & \cdots & 0 & q_{n-2} & d_{n-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \end{pmatrix}.$$

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 tr nor the check for $\text{diag}[i] == 0$ is needed.

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

be9642, 23 lines

```
typedef double T;
vector<T> tridiagonal(vector<T> diag, const vector<T> &super,
    const vector<T> &sub, vector<T> b) {
    int n = b.size(); vector<int> tr(n);
    for (int i = 0; i < n-1; ++i)
        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;
}
```

4.1 Fourier transforms

root-of-unity.h

Description: implementation of the root of unity using complex numbers.

800d6f, 16 lines

```
template<typename dbl> struct cplx { // hash-1
    dbl x, y; using P = cplx;
    cplx(dbl x_ = 0, dbl y_ = 0) : x(x_), y(y_) {}
    friend P operator+(P a, P b) { return P(a.x+b.x, a.y+b.y); }
    friend P operator-(P a, P b) { return P(a.x-b.x, a.y-b.y); }
    friend P operator*(P a, P b) { return P(a.x*b.x - a.y*b.y, a.x*b.y + a.y*b.x); }
    friend P conj(P a) { return P(a.x, -a.y); }
```



```
friend P inv(P a) { dbl n = (a.x*a.x+a.y*a.y); return P(a.x/n
,-a.y/n); }
};
template <typename T> struct root_of_unity {};
template <typename dbl> struct root_of_unity<cplx<dbl>> {
    static cplx<dbl> f(int k) {
        static const dbl PI = acos(-1); dbl a = 2*PI/k;
        return cplx<dbl>(cos(a),sin(a));
    }
}; // hash-1 = 800d6f
```

root-of-unity-zp.h
Description: implementation of the root of unity in Z_p .

```
"/number-theory/modular-arithmetic.h" f86a44, 15 lines
template <typename T> struct root_of_unity {};
// (MOD 3) := (M1:897581057),(M3:985661441),(M5:935329793)
using M0 = modnum<998244353U>; // hash-1
constexpr unsigned primitive_root(unsigned M) {
    if (M == 880803841U) return 26U; // (M2)
    else if (M == 943718401U) return 7U; // (M4)
    else if (M == 918552577U) return 5U; // (M6)
    else return 3U;
}
template<unsigned MOD> struct root_of_unity<modnum<MOD>> {
    static constexpr modnum<MOD> g0 = primitive_root(MOD);
    static modnum<MOD> f(int K) {
        assert((MOD-1)%K == 0); return g0.pow((MOD-1)/K);
    }
}; // hash-1 = de8785
```

fast-fourier-transform.h
Description: For convolution of complex numbers or more than two vectors: FFT, multiply pointwise, FFT inverse back.
Time: $\mathcal{O}(N \log N)$ with $N = |A| + |B|$ ($\sim 1s$ for $N = 2^{22}$)

```
1cadbd, 53 lines
inline int nxt_pow2(int s) { return 1 << (s > 1 ? 32 -
__builtin_clz(s-1) : 0); }
template<typename T> struct FFT {
    vector<T> rt; vector<int> rev;
    FFT() : rt(2, T(1)) {}
    void init(int N) { // hash-1
        N = nxt_pow2(N);
        if (N > int(rt.size())) {
            rev.resize(N); rt.reserve(N);
            for (int a = 0; a < N; ++a)
                rev[a] = (rev[a/2] | ((a&1)*N)) >> 1;
            for (int k = int(rt.size()); k < N; k *= 2) {
                rt.resize(2*k);
                T z = root_of_unity<T>::f(2*k);
                for (int a = k/2; a < k; ++a)
                    rt[2*a] = rt[a], rt[2*a+1] = rt[a] * z;
            }
        } // hash-1 = fcd279
        void fft(vector<T>& xs, bool inverse) const { // hash-2
            int N = int(xs.size());
            int s = __builtin_ctz(int(rev.size())/N);
            if (inverse) reverse(xs.begin() + 1, xs.end());
            for (int a = 0; a < N; ++a) {
                if (a < (rev[a] >> s)) swap(xs[a], xs[rev[a] >> s]);
            }
            for (int k = 1; k < N; k *= 2) {
                for (int a = 0; a < N; a += 2*k) {
                    int u = a, v = u + k;
                    for (int b = 0; b < k; ++b, ++u, ++v) {
                        T z = rt[b + k] * xs[v];
                        xs[v] = xs[u] - z, xs[u] = xs[u] + z;
                    }
                }
            }
        }
    };
};
```

```

}
if (inverse)
    for (int a = 0; a < N; ++a) xs[a] = xs[a] * inv(T(N));
} // hash-2 = e2ea4a
vector<T> convolve(vector<T> as, vector<T> bs) { // hash-3
    int N = int(as.size()), M = int(bs.size());
    int K = N + M - 1, S = nxt_pow2(K); init(S);
    if (min(N, M) <= 64) {
        vector<T> res(K);
        for (int u = 0; u < N; ++u) for (int v = 0; v < M; ++v)
            res[u + v] = res[u + v] + as[u] * bs[v];
        return res;
    } else {
        as.resize(S), bs.resize(S);
        fft(as, false); fft(bs, false);
        for (int i = 0; i < S; ++i) as[i] = as[i] * bs[i];
        fft(as, true); as.resize(K); return as;
    }
} // hash-3 = e4b903
};
```

convolutions.h
Description: implementation of crt (naive and garner) to convolute polynomials under arbitrary Fields.
Time: $\mathcal{O}(N \log N)$ with $N = |A| + |B|$ ($\sim 1s$ for $N = 2^{22}$)

```
6ff60b, 38 lines
// T = {unsigned, unsigned long long, modnum<M>}
// Remark: need to satisfy |poly| * mod^2 < \prod_{i} M_i
template<class T, unsigned M0, unsigned M1, unsigned M2,
        unsigned M3, unsigned M4>
T garner(modnum<M0> a0, modnum<M1> a1, modnum<M2> a2, modnum<M3>
> a3, modnum<M4> a4) { // hash-1
    static const modnum<M1> INV_M0_M1 = modnum<M1>(M0).inv();
    static const modnum<M2> INV_M0M1_M2 = (modnum<M2>(M0) * M1).
        inv();
    // static const modnum<M3> INV_M0M1M2M3 = (modnum<M3>(M0) *
        M1 * M2).inv();
    // static const modnum<M4> INV_M0M1M2M3M4 = (modnum<M4>(M0)
        * M1 * M2 * M3).inv();
    const modnum<M1> b1 = INV_M0_M1 * (a1 - a0.x);
    const modnum<M2> b2 = INV_M0M1_M2 * (a2 - (modnum<M2>(b1.x) *
        M0 + a0.x));
    // const modnum<M3> b3 = INV_M0M1M2M3 * (a3 - ((modnum<M3>(
        b2.x) * M1 + b1.x) * M0 + a0.x));
    // const modnum<M4> b4 = INV_M0M1M2M3M4 * (a4 - (((modnum<M4>
        >(b3.x) * M2 + b2.x) * M1 + b1.x) * M0 + a0.x));
    return (T(b2.x) * M1 + b1.x) * M0 + a0.x;
    // return (((T(b4.x) * M3 + b3.x) * M2 + b2.x) * M1 + b1.x) *
        M0 + a0.x;
} // hash-1 = b93180
// results must be in [-448002610255888384, 448002611254132736]
vector<ll> convolve(const vector<ll>& as, const vector<ll>& bs)
{ // hash-2
    static constexpr unsigned M0 = M0::M, M1 = M1::M;
    static const modnum<M1> INV_M0_M1 = modnum<M1>(M0).inv();
    if (as.empty() || bs.empty()) return {};
    const int len_as = int(as.size()), len_bs = int(bs.size());
    vector<modnum<M0>> as0(len_as), bs0(len_bs);
    for (int i = 0; i < len_as; ++i) as0[i] = as[i];
    for (int i = 0; i < len_bs; ++i) bs0[i] = bs[i];
    const vector<modnum<M0>> cs0 = FFT0.convolve(as0, bs0);
    vector<modnum<M1>> as1(len_as), bs1(len_bs);
    for (int i = 0; i < len_as; ++i) as1[i] = as[i];
    for (int i = 0; i < len_bs; ++i) bs1[i] = bs[i];
    const vector<modnum<M1>> cs1 = FFT1.convolve(as1, bs1);
    vector<ll> cs(len_as + len_bs - 1);
    for (int i = 0; i < len_as + len_bs - 1; ++i) {
        const modnum<M1> d1 = INV_M0_M1 * (cs1[i] - cs0[i].x);
        cs[i] = (d1.x > M1 - d1.x)
```

```

? (-1ULL - (static_cast<unsigned ll>(M1 - 1U - d1.x) * M0
+ (M0 - 1U - cs0[i].x)))
: (static_cast<unsigned ll>(d1.x) * M0 + cs0[i].x);
}
return cs;
} // hash-2 = a3bae7
```

fast-subset-transform.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: $\mathcal{O}(N \log N)$

```
5b9574, 16 lines
void FST(vector<int> &a, bool inv) {
    for (int n = a.size(), step = 1; step < n; step *= 2) {
        for (int i = 0; i < n; i += 2 * step) for(int j = i; j < i +
            step; ++j) {
            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(auto &x : a) x /= a.size(); // XOR only
    }
    vector<int> conv(vector<int> a, vector<int> b) {
        FST(a, 0); FST(b, 0);
        for(int i = 0; i < a.size(); ++i) a[i] *= b[i];
        FST(a, 1); return a;
    }
}
```

sum-of-powers.h
Description: Computes monomials and sum of powers product certain polynomials. Check "General purpose numbers" section for more info. (Monomials) $pw(x) = x^d$ for a fixed d . $\sum_{x=0}^{\infty} r^x f(x)$. (degree of $f \leq d$). $\sum_{x=0}^{N-1} r^x f(x)$. (degree of $f \leq d$).

```
"/number-theory/modular-arithmetic.h", "/lagrange.h" 85dfa0, 33 lines
vector<num> get_monomials(int N, long long d) { // hash-1
    vector<int> pfac(N);
    for (int i = 2; i < N; ++i) pfac[i] = i;
    for (int p = 2; p < N; ++p) if (pfac[p] == p)
        for(int m = 2*p; m < N; m += p) if(pfac[m] > p) pfac[m]=p;
    vector<num> pw(N);
    for (int i = 0; i < N; ++i)
        if (i <= 1 || pfac[i] == i) pw[i] = num(i).pow(d);
        else pw[i] = (pw[pfac[i]] * pw[i / pfac[i]]);
    return pw;
} // hash-1 = 966124
num sum_of_power_limit(num r, int d, const vector<num>& fs) {
    // hash-2
    interpolator_t<num> M(d + 2); num s = 1; auto gs = fs;
    for (int x = 0; x <= d; ++x, s *= r) gs[x] *= s;
    num ans = 0, cur_sum = 0; s = 1;
    for (int x = 0; x <= d; ++x, s *= -r) {
        cur_sum += choose(d+1, x) * s; ans += cur_sum * gs[d-x];
    } ans *= (1 - r).pow(-(d + 1));
    return ans;
} // hash-2 = d5b811
num sum_of_power(num r, int d, vector<num>& fs, ll N) { // hash
-3
    if (r == 0) return (0 < N) ? fs[0] : 0;
    interpolator_t<num> M(d + 2);
    vector<num> gs(d + 2); gs[0] = 0; num s = 1;
    for (int x = 0; x <= d; ++x, s *= r)
        gs[x + 1] = gs[x] + s * fs[x];
    if (r == 1) return M.interpolate(gs, N);
```

```
const num c = sum_of_power_limit(r, d, fs);
const num r_inv = r.inv(); num w = 1;
for (int x = 0; x <= d + 1; ++x, w *= r_inv)
    gs[x] = w * (gs[x] - c);
return c + r.pow(N) * M.interpolate(gs, N);
} // hash-3 = 62a62d
```

power-series.h

```
"fast-fourier-transform.h", ".../number-theory/mod-sqrt.h",
".../number-theory/preparator.h"
6c18f3, 131 lines

using num = modnum<998244353U>; FFT<num> fft_data;
using P = vector<num>;
void shrink(P& P){ while(!P.empty()&&!P.back())P.pop_back();}
P& operator+=(P& a, const P& b) { // hash-1
    int N = int(a.size()), M = int(b.size());
    a.resize(max(N, M));
    for (int i = 0; i < M; ++i) a[i] += b[i];
    return a;
} // hash-1 = 2bef29
P operator+(const P& a,const P& b){P r=a;return r+=b;}
P& operator -=(P& a, const P& b) { // hash-2
    int N = int(a.size()), M = int(b.size());
    a.resize(max(N, M));
    for (int i = 0; i < M; ++i) a[i] -= b[i];
    return a;
} // hash-2 = 6c3395
P operator-(const P& a,const P& b) {P r=a;return r-=b;}
P operator*(const P& a, const P& b) {
    return fft.convolve(a,b); }
P& operator*=(P& a, const P& b){ return a*=b; }
// inverse series mod x^N
P inverse(const P& P, int N) { // hash-3
    if (P.empty()) return {};
    P b({P[0]}), ret = {P[0].inv()};
    for (int len = 1; len < N; len *= 2) {
        for (int i = len; i < min(2*len, int(P.size())); ++i)
            b.push_back(P[i]);
        (ret *= P[2] - b * ret).resize(2*len);
    }
    ret.resize(N); return ret;
} // hash-3 = e62bca
P operator/(P a, P b) { // hash-4
    int s = int(a.size()) - int(b.size()) + 1;
    if (s <= 0) return {};
    reverse(a.begin(), a.end()); reverse(b.begin(), b.end());
    a.resize(s); b.resize(s); a *= inverse(std::move(b), s);
    a.resize(s); reverse(a.begin(), a.end());
    return a;
} // hash-4 = 9a630a
P& operator/=(P& a, const P& b){return a = a / b;}
P& operator%=(P& a, const P& b) { // hash-5
    if (a.size() >= b.size()) {
        P c = (a/b) * b; a.resize(b.size()-1);
        for (int i = 0; i < int(a.size()); ++i)
            a[i] = a[i]-c[i];
    }
    shrink(a); return a;
} // hash-5 = 871616
P operator%(const P& a, const P& b) {P r=a; return r%=b;}
P deriv(P P) { // hash-6
    int N = int(P.size());
    if (N == 0) return {};
    P f(N-1);
    for (int x = 1; x < N; ++x) f[x-1] = x * P[x];
    return f;
} // hash-6 = 8c328d
P integ(P P) { // hash-7
    int N = int(P.size());
    if (N == 0) return {};
```

```
P f(N+1); f[1] = 1;
for (int x = 0; x < N; ++x) f[x+1] = invs[x+1] * P[x];
return f;
} // hash-7 = bbdd46
// ln(P) mod x^N
P log(const P& P, int N) { // hash-8
    if (P.empty()) return {};
    P f = integ(deriv(P) * inverse(P, N));
    f.resize(N); return f;
} // hash-8 = 76ed43
// exp(P) mod x^N
P exp(P P, int N) { // hash-9
    if (P.empty()) return {1};
    P b({P[0]+1}), ret = {1};
    for (int len = 1; len < N; len *= 2) {
        for (int i = len; i < min(2*len, int(P.size())); ++i)
            b.push_back(P[i]);
        (ret *= (b - log(ret, 2*len))).resize(2*len);
    }
    ret.resize(N); return ret;
} // hash-9 = 71a111
P pow(P a, ll e, int N) { // hash-10
    P b(N);
    if (!e) { b[0] = 1; return b; }
    int p = 0;
    while (e*p < N && a[p].x == 0) ++p;
    if (1LL*e*p >= N) return b;
    num mu = a[p].pow(e), di = a[p].inv();
    int M = e*p; P c(N-M);
    for (int x = 0; x < N-M; ++x) c[x] = a[x+p] * di;
    c = log(c, N-M);
    for (auto& x : c) x = x * e;
    c = exp(c, N-M);
    for (int x = 0; x < int(c.size()); ++x) b[x+M] = c[x]*mu;
    return b;
} // hash-10 = af6c8e
P eval(const P& a, const vector<num>& x) { // hash-11
    int N = int(x.size());
    if (!N) return {};
    vector<P> up(2*N), down(2*N);
    for (int i = 0; i < N; ++i) up[i+N]=P({0-x[i],1});
    for (int i = N-1; i >= 1; --i) up[i]=up[2*i]*up[2*i+1];
    down[1] = a % up[1];
    for (int i = 2; i < 2*N; ++i) down[i]=down[i/2]*up[i];
    vector<num> y(N);
    for (int i = 0; i < N; ++i)
        y[i]= down[i+N].empty() ? 0 : down[i+N][0];
    return y;
} // hash-11 = 28f80c
P interp(const vector<num>&x, const vector<num>&y){ // hash-12
    int N = int(x.size()); assert(N);
    vector<P> up(2*N), down(2*N);
    for (int i = 0; i < N; ++i) up[i+N]=P({0-x[i],1});
    for (int i = N-1; i >= 1; --i) up[i]=up[2*i]*up[2*i+1];
    vector<num> a = eval(deriv(up[1]), x);
    for (int i = 0; i < N; ++i) down[i+N]=P({y[i]*a[i].inv()});
    for (int i = N-1; i >= 1; --i)
        down[i]=down[2*i]*up[2*i+1] + down[2*i+1]*up[2*i];
    return down[1];
} // hash-12 = d81a89
P taylor_shift(P a, int N, int K) { // hash-13
    P P(N), Q = a;
    P[0] = 1;
    for (int i = 1; i < N; ++i) P[i] += P[i-1] * K;
    for (int i = 1; i < N; ++i) P[i] *= ifact[i];
    reverse(P.begin(), P.end());
    for (int i = 1; i < N; ++i) Q[i] *= fact[i];
    P *= Q;
    for (int i = 0; i < N; ++i) P[i] = P[N-1+i] * ifact[i];
```

```
P.resize(N);
return P;
} // hash-13 = d89033
```

4.1.1 LP formulation

Standard form: maximize $\mathbf{c}^T \mathbf{x}$ subject to $A\mathbf{x} \leq \mathbf{b}$ and $\mathbf{x} \geq 0$.
Dual LP: minimize $\mathbf{b}^T \mathbf{y}$ subject to $A^T \mathbf{y} \geq \mathbf{c}$ and $\mathbf{y} \geq 0$.
 $\bar{\mathbf{x}}$ and $\bar{\mathbf{y}}$ are optimal if and only if for all $i \in [1, n]$, either $\bar{x}_i = 0$ or $\sum_{j=1}^m A_{ji} \bar{y}_j = c_i$ holds and for all $i \in [1, m]$ either $\bar{y}_i = 0$ or $\sum_{j=1}^n A_{ij} \bar{x}_j = b_j$ holds.

1. In case of minimization, let $c'_i = -c_i$
2. $\sum_{1 \leq i \leq n} A_{ji} x_i \geq b_j \rightarrow \sum_{1 \leq i \leq n} -A_{ji} x_i \leq -b_j$
3. $\sum_{1 \leq i \leq n} A_{ji} x_i = b_j$
 - $\sum_{1 \leq i \leq n} A_{ji} x_i \leq b_j$
 - $\sum_{1 \leq i \leq n} A_{ji} x_i \geq b_j$
4. If x_i has no lower bound, replace x_i with $x_i - x'_i$

4.1.2 Generating functions

A list of generating functions for useful sequences:

$(1, 1, 1, 1, 1, \dots)$	$\frac{1}{1-z}$
$(1, -1, 1, -1, 1, \dots)$	$\frac{1}{1+z}$
$(1, 0, 1, 0, 1, 0, \dots)$	$\frac{1}{1-z^2}$
$(1, 0, \dots, 0, 1, 0, 1, 0, \dots, 0, 1, 0, \dots)$	$\frac{1}{1-z^2}$
$(1, 2, 3, 4, 5, 6, \dots)$	$\frac{1}{(1-z)^2}$
$(1, \binom{m+1}{m}, \binom{m+2}{m}, \binom{m+3}{m}, \dots)$	$\frac{1}{(1-z)^{m+1}}$
$(1, c, \binom{c+1}{2}, \binom{c+2}{3}, \dots)$	$\frac{1}{(1-z)^c}$
$(1, c, c^2, c^3, \dots)$	$\frac{1}{1-cz}$
$(0, 1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \dots)$	$\ln \frac{1}{1-z}$

A neat manipulation trick is:

$$\frac{1}{1-z} G(z) = \sum_n \sum_{k \leq n} g_k z^n$$

Number theory (5)

5.1 Modular arithmetic

```
modular-arithmetic.h
Description: Operators for modular arithmetic.
"mod-inv.h"
1fc315, 31 lines

template<unsigned M_> struct modnum {
    static constexpr unsigned M = M_; using num = modnum;
    using ll = long long; using ull = unsigned long long;
    unsigned x;
    num& norm(unsigned a){x = a<M ? a:a-M;return *this;}
    constexpr modnum() : x(0U) {}
    constexpr modnum(unsigned a) : x(a % M) {}
    constexpr modnum(ull a) : x(a % M) {}
    constexpr modnum(int a) : x(((a % = int(M))<0) ? (a+int(M)):a) {}
    constexpr modnum(ll a) : x(((a % = ll(M))<0) ? (a+ll(M)):a) {}
    explicit operator int() const { return x; }
```

```
num& operator+=(const num& a){ return norm(x+a.x); }
num& operator-=(const num& a){ return norm(x-a.x+M); }
num& operator*=(const num& a){ return norm(unsigned((ull(x)*a
.x) % M)); }
num& operator/=(const num& a){ return (*this *= a.inv());}
template<typename T> friend num operator+(T a, const num& b){
    return (num(a) += b); }
template<typename T> friend num operator-(T a, const num& b){
    return (num(a) -= b); }
template<typename T> friend num operator*(T a, const num& b){
    return (num(a) *= b); }
template<typename T> friend num operator/(T a, const num& b){
    return (num(a) /= b); }

num operator+() const { return *this; }
num operator-() const { return num() - *this; }
num pow(ll e) const {
    if (e < 0) { return inv().pow(-e); } num b = x, xe = 1U;
    for (; e >= 1) { if (e & 1) xe *= b; b *= b; }
    return xe;
}
num inv() const { return minv(x, M); }
friend num inv(const num& a) { return a.inv(); }
explicit operator bool() const { return x; }
friend bool operator==(const num& a, const num& b){return a.x
== b.x;}
friend bool operator!=(const num& a, const num& b){return a.x
!= b.x;}
};
```

mod-inv.h
Description: Find x such that $ax \equiv 1(\text{mod } m)$. The inverse only exist if a and m are coprimes.

```
template<typename T> T minv(T a, T m) {
    a %= m; assert(a);
    return a == 1 ? 1 : T(m - 1l(minv(m, a)) * m / a);
}
```

mod-sum.h
Description: Sums of mod'ed arithmetic progressions.
 $\text{modsum}(to, c, k, m) = \sum_{i=0}^{to-1} (ki + c) \% m$. divsum is similar but for floored division.
Time: $\log(m)$, with a large constant.

```
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) { // hash-1
    ull res = k / m * sumsq(to) + c / m * to;
    k %= m; c %= m;
    if (k) {
        ull to2 = (to * k + c) / m;
        res += to * to2;
        res -= divsum(to2, m-1 - c, m, k) + to2;
    }
    return res;
} // hash-1 = ae0cdb
ll modsum(ull to, ll c, ll k, ll m) { // hash-2
    c = ((c % m) + m) % m; k = ((k % m) + m) % m;
    return to * c + k * sumsq(to) - m * divsum(to, c, k, m);
} // hash-2 = 5daf3e
```

mod-mul.h
Description: Calculate $a \cdot b \text{ mod } c$ (or $a^b \text{ mod } c$) for $0 \leq a, b \leq c \leq 7.2 \cdot 10^{18}$.
Time: $\mathcal{O}(1)$ for modmul , $\mathcal{O}(\log b)$ for modpow

```
typedef unsigned long long ull;
ull modmul(ull a, ull b, ull M) { // hash-1
    ll ret = a * b - M * ull(1.L / M * a * b);
    return ret + M * (ret < 0) - M * (ret >= (ll)M);
}
```

```
} // hash-1 = e9309c
ull modpow(ull b, ull e, ull mod) { // hash-2
    ull ans = 1;
    for (; e; b = modmul(b, b, mod), e /= 2)
        if (e & 1) ans = modmul(ans, b, mod);
    return ans;
} // hash-2 = 100b91
```

mod-sqrt.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

```
int jacobi(ll a, ll m) { // hash-1
    int s = 1;
    if (a < 0) a = a % m + m;
    for (; m > 1; ) {
        a %= m; if (a == 0) return 0;
        const int r = __builtin_ctzll(a);
        if ((r & 1) && ((m + 2) & 4)) s = -s;
        a >>= r; if (a & m & 2) s = -s;
        swap(a, m);
    } return s;
} // hash-1 = aacc12
vector<ll> mod_sqrt(ll a, ll p) { // hash-2
    if (p == 2) return {a & 1};
    const int j = jacobi(a, p);
    if (j == 0) return {0};
    if (j == -1) return {};
    ll b, d;
    while (true) {
        b = xrand() % p; d = (b * b - a) % p;
        if (d < 0) d += p;
        if (jacobi(d, p) == -1) break;
    }
    ll f0 = b, f1 = 1, g0 = 1, g1 = 0, tmp;
    for (ll e = (p + 1) >> 1; e; e >>= 1) {
        if (e & 1) {
            tmp = (g0 * f0 + d * ((g1 * f1) % p)) % p;
            g1 = (g0 * f1 + g1 * f0) % p; g0 = tmp;
        }
        tmp = (f0 * f0 + d * ((f1 * f1) % p)) % p;
        f1 = (2 * f0 * f1) % p; f0 = tmp;
    }
    return (g0 < p-g0) ? vector<ll>{g0,p-g0} : vector<ll>{p-g0,g0};
} // hash-2 = b22da7
```

mod-range.h
Description: $\min x \geq 0$ s.t. $l \leq ((ax) \text{ mod } m) \leq r, m > 0, a \geq 0$.

```
template<typename T> T mod_range(T m, T a, T l, T r) {
    l = max(l, T(0)); r = min(r, m - 1);
    if (l > r) return -1;
    a %= m;
    if (a == 0) return (l > 0) ? -1 : 0;
    const T k = (l + a - 1) / a;
    if (a * k <= r) return k;
    const T y = mod_range(a, m, a * k - r, a * k - 1);
    return (y == -1) ? -1 : ((m * y + r) / a);
}
```

5.2 Primality

sieve.h
Description: Prime sieve for generating all primes up to a certain limit. $\text{pfac}[i]$ is the lowest prime factor of i . Also useful if you need to compute any multiplicative function.
Time: $\mathcal{O}(N)$

```
vector<int> run_sieve(int N) {
```

```
vector<int> pfac(N+1), primes, mu(N+1,-1), phi(N+1);
primes.reserve(N+1); mu[1] = phi[1] = 1;
for (int i = 2; i <= N; ++i) {
    if (!pfac[i])
        pfac[i] = i, phi[i] = i-1, primes.push_back(i);
    for (int p : primes) {
        if (p > N/i) break;
        pfac[p * i] = p; mu[p * i] *= mu[i];
        phi[p * i] = phi[i] * phi[p];
        if (i % p == 0) {
            mu[p * i] = 0; phi[p * i] = phi[i] * p;
            break;
        }
    }
} return primes;
}
```

miller-rabin.h
Description: Deterministic Miller-Rabin primality test. Guaranteed to work for numbers up to 2^{64} ; for larger numbers, extend A randomly.
Time: 7 times the complexity of $a^b \text{ mod } c$.

```
"mod-mul.h" bbee97, 12 lines
bool isPrime(ull n) { // hash-1
    if (n < 2 || n % 6 % 4 != 1) return (n | 1) == 3;
    vector<ull> A = {2, 325, 9375, 28178, 450775, 9780504,
        1795265022};
    ull s = __builtin_ctzll(n-1), d = n >> s; // hash-1 = 8cef21
    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;
}
```

pollard-rho.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.

```
"mod-mul.h", "extended-euclid.h", "miller-rabin.h" 6bf31f, 17 lines
ull pollard(ull n) { // hash-1
    auto f = [n](ull x, ull k) { return modmul(x, x, n) + k; };
    ull x = 0, y = 0, t = 30, prd = 2, i = 1, q;
    while (t++ % 40 || gcd(prd, n) == 1) {
        if (x == y) x = ++i, y = f(x, i);
        if ((q = modmul(prd, max(x,y) - min(x,y), n))) prd = q;
        x = f(x, i), y = f(f(y, i), i);
    }
    return gcd(prd, n);
} // hash-1 = 8875ba
vector<ull> factor(ull n) { // hash-2
    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(), r.begin(), r.end());
    return l;
} // hash-2 = 04f1a4
```

5.3 Divisibility

extended-euclid.h
Description: Finds two integers x and y , such that $ax + by = \text{gcd}(a, b)$. If a and b are coprime, then x is the inverse of $a \pmod b$.
Time: $\mathcal{O}(\log \min(a, b))$

```
template<typename T>
T egcd(T a, T b, T &x, T &y) {
    if (!a) { x = 0, y = 1; return b; }
```

```
    T g = egcd(b % a, a, y, x);
    x -= y * (b/a); return g;
}
```

division-lemma.h

Description: This lemma let us exploit the fact tha the sequence (harmonic on integer division) has at most $2\sqrt{N}$ distinct elements, so we can iterate through every possible value of $\lfloor \frac{N}{i} \rfloor$, using the fact that the greatest integer j satisfying $\lfloor \frac{N}{i} \rfloor = \lfloor \frac{N}{j} \rfloor$ is $\lfloor \frac{N}{\lfloor \frac{N}{i} \rfloor} \rfloor$. This one computes the $\sum_{i=1}^N \lfloor \frac{N}{i} \rfloor$.

Time: $\mathcal{O}(\sqrt{N})$	b2c1ab, 15 lines
<pre>int res = 0; for (int a = 1, b; a <= N; a = b + 1) { // floor b = N / (N / a); // quotient (N/a) and there are (b - a + 1) elements int l = b - a + 1, r = a + b; // l * r / 2 = sum(i, j) if (l & 1) r /= 2; else l /= 2; res += l * r * (N / a); } // [1, N), need to deal with case where a = N separately for (int a = 1, b; a < N; a = b + 1) { // ceil const int k = (N - 1) / a + 1; // quotient k b = (N - 1) / (k - 1); int cnt = b - a + 1; // occur cnt times on interval [a, b] }</pre>	

divisors.h

Description: Generate all factors of n given it's prime factorization.

Time: $\mathcal{O}\left(\frac{\sqrt{N}}{\log N}\right)$	5de75c, 14 lines
<pre>"prime-factors.h"</pre>	
<pre>template<typename T> vector<T> get_divisors(T N) { auto factors = prime_factorize(N); vector<T> ans; ans.reserve(int(sqrtl(N) + 1)); auto dfs = [&](auto&& self, auto& ans, T val, int d)->void { auto& [P, E] = factors[d]; if (d == int(factors.size())) ans.push_back(val); else { T X = 1; for (int pw = 0; pw <= E; ++pw, X *= P) self(self, ans, val * X, d + 1); } }; dfs(dfs, ans, 1, 0); return ans; }</pre>	

phi-function.h

	da7671, 6 lines
<pre>const int n = int(1e5)*5; vector<int> phi(n); void calculatePhi() { for(int i = 0; i < n; ++i) phi[i] = i&1 ? i : i/2; for(int i = 3; i < n; i += 2) if (phi[i] == i) for(int j = i; j < n; j += i) phi[j] -= phi[j]/i; }</pre>	

discrete-log.h

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

Time: $\mathcal{O}(\sqrt{m})$	62fc5e, 15 lines
<pre>"extended-euclid.h"</pre>	
<pre>template<typename T> T modLog(T a, T b, T m) { T k = 1, it = 0, g; // hash-1 while ((g = gcd(a, m)) != 1) { if (b == k) return it; if (b % g) return -1; }</pre>	

```
    b /= g; m /= g; ++it; k = k * a / g % m;
} // hash-1 = 416572
T n = sqrtl(m) + 1, f = 1, j = 1; // hash-2
unordered_map<T, T> A;
while (j <= n)
    f = f * a % m, A[f * b % m] = j++;
for(int i = 1; i <= n; ++i) if (A.count(k = k * f % m))
    return n * i - A[k] + it;
return -1; // hash-2 = 25c7bf
}
```

prime-counting.h

Description: Count the number of primes up to N . Also useful for sum of primes.

Time: $\mathcal{O}(N^{3/4}/\log N)$	513314, 21 lines
<pre>struct primes_t { vector<ll> dp, w; ll pi(ll N) { // hash-1 dp.clear(), w.clear(); const int sqrtN = int(sqrt(N)); for (ll a = 1, b; a <= N; a = b+1) b = N / (N / a), w.push_back(N/a); auto get = [&](ll x) { if (x <= sqrtN) return int(x-1); return int(w.size() - N/x); }; // hash-1 = a785c9 reverse(w.begin(), w.end()); dp.reserve(w.size()); for (auto& x : w) dp.push_back(x-1); // hash-2 for (ll i = 2; i*i <= N; ++i) { if (dp[i-1] == dp[i-2]) continue; for (int j = int(w.size()-1; w[j] >= i*i; --j) dp[j] -= dp[get(w[j]/i)] - dp[i-2]; } return dp.back(); // hash-2 = c6b848 } };</pre>	

5.4 Chinese remainder theorem

chinese-remainder.h

Description: Chinese Remainder Theorem. $\text{crt}(a, m, b, n)$ 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: $\mathcal{O}(\log(\text{LCM}(m)))$	ece59a, 7 lines
<pre>"extended-euclid.h"</pre>	
<pre>pair<ll, ll> crt(ll a, ll m, ll b, ll n) { if (n > m) swap(a, b), swap(m, n); ll x, y, g = egcd(m, n, x, y); if ((a - b) % g != 0) return {0, -1}; x = (b - a) % n * x % n / g * m + a; return {x + (x < 0 ? m*n/g : 0), m*n/g}; }</pre>	

5.5 Fractions

fractions.h

Description: Template that helps deal with frtions.

	596163, 28 lines
<pre>template<typename num> struct fraction_t { num p, q; using fr = fraction_t; // hash-1 fraction_t() : p(0), q(1) {} fraction_t(num _n, num _d = 1): p(_n), q(_d){ num g = gcd(p, q); p /= g, q /= g; if (q < 0) p *= -1, q *= -1; assert(q != 0); } // hash-1 = 2306e9 friend bool operator<(const fr& l, const fr& r){ // hash-2 return l.p*r.q < r.p*l.q;} friend bool operator==(const fr& l, const fr& r){return l.p == r.p && l.q == r.q;}</pre>	

```
friend bool operator!=(const fr& l, const fr& r){return !(l
    == r);}
friend fr operator+(const fr& l, const fr& r){
    num g = gcd(l.q, r.q);
    return fr(r.q / g * l.p + l.q / g * r.p, l.q / g * r.q);
} // hash-2 = 604bf5
friend fr operator-(const fr& l, const fr& r) { // hash-3
    num g = gcd(l.q, r.q);
    return fr( r.q / g * l.p - l.q / g * r.p, l.q / g * r.q);
}
friend fr operator*(const fr& l, const fr& r){
    return fr(l.p*r.p, l.q*r.q);}
friend fr operator/(const fr& l, const fr& r){
    return l*fr(r.q,r.p);} // hash-3 = 9d4fb0
friend fr& operator+=(fr& l, const fr& r){return l=l+r;}
friend fr& operator-=(fr& l, const fr& r){return l=l-r;}
template<class T> friend fr& operator*=(fr& l, const T& r){
    return l=l*r;}
template<class T> friend fr& operator/=(fr& l, const T& r){
    return l=l/r;}
};
```

continued-fractions.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 ∞ ; if x is the root of a degree 2 polynomial the a 's eventually become cyclic.

Time: $\mathcal{O}(\log N)$	3dfe17, 18 lines
<pre>typedef double dbl; // for N ~ 1e7; long double for N ~ 1e9 pair<ll, ll> approximate(dbl x, ll N) { ll LP = 0, LQ = 1, P = 1, Q = 0, inf = ll(1e18); dbl 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) { // hash-1 // 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 - (dbl)NP / (dbl)NQ) < abs(x - (dbl)P / (dbl)Q)) ? make_pair(NP, NQ) : make_pair(P, Q); } if (abs(y = 1/(y - (dbl)a)) > 3*N) return {NP, NQ}; LP = P; P = NP; LQ = Q; Q = NQ; } // hash-1 = bfee6a }</pre>	

frac-binary-search.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: $\text{fracBS}(\{ \}(\text{Frac } f) \{ \text{return } f.p >= 3*f.q; \}, 10);$ // $\{1, 3\}$

Time: $\mathcal{O}(\log(N))$	74ffd8, 20 lines
<pre>struct Frac { ll p, q; }; template<class F> Frac fracBS(F f, ll N) { bool dir = 1, A = 1, B = 1; Frac left{0, 1}, right{1, 1}; //right{1,0} to search (0,N] assert(!f(left)); assert(f(right)); while (A B) { // hash-1 ll adv = 0, step = 1; // move right if dir, else left for (int si = 0; step; (step *= 2) >= si) { adv += step; Frac mid{left.p * adv + right.p, left.q * adv + right.q}; if (abs(mid.p) > N mid.q > N dir == !f(mid)) { adv -= step; si = 2; }</pre>	


```
    }
}
right.p += left.p * adv; right.q += left.q * adv;
dir = !dir; swap(left, right);
A = B; B = !!adv;
} // hash-1 = dedbc8
return dir ? right : left;
}
```

5.5.1 Pythagorean Triples

The Pythagorean triples are uniquely generated by

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

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

5.6 Primes

$p = 962592769$ is such that $2^{21} \mid p - 1$, which may be useful. For hashing use 970592641 (31-bit number), 31443539979727 (45-bit), 3006703054056749 (52-bit). There are 78498 primes less than 1 000 000.

5.6.1 Prime counting function $(\pi(x))$

The prime counting function is asymptotic to $\frac{x}{\log x}$, by the prime number theorem.

x	10	10 ²	10 ³	10 ⁴	10 ⁵	10 ⁶	10 ⁷	10 ⁸
$\pi(x)$	4	25	168	1.229	9.592	78.498	664.579	5.761.455

5.6.2 Sum of primes

For any multiplicative f :

$S(n, p) = S(n, p - 1) - f(p) \cdot (S(n/p, p - 1) - S(p - 1, p - 1))$

5.6.3 Moebius Function

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

Moebius Inversion:

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

Other useful formulas/forms:

$$\sum_{d|n} \phi(d) = n$$
$$\sum_{\gcd(i,n)=1} i = n \frac{\phi(n)}{2}$$
$$\sum_{a=1}^n \sum_{b=1}^n [\gcd(a, b) = 1] = \sum_{d=1}^n \mu(d) \lfloor \frac{n}{d} \rfloor^2$$
$$\sum_{a=1}^n \sum_{b=1}^n \gcd(a, b) = \sum_{d=1}^n d \sum_{d|x} \lfloor \frac{n}{x} \rfloor^2 \mu(\frac{x}{d})$$
$$\sum_{a=1}^n \sum_{b=a}^n \gcd(a, b) = \sum_{d=1}^n \sum_{d|x} \phi(\frac{x}{d}) d$$
$$\sum_{a=1}^n \sum_{b=1}^n \text{lcm}(a, b) = \sum_{d=1}^n \mu(d) d \sum_{d|x} x (\lfloor \frac{n}{x} \rfloor + 1)^2$$

dirichlet-convolution int-perm lucas

$$\sum_{a=1}^n \sum_{b=a+1}^n \text{lcm}(a, b) = \sum_{d=1}^n \sum_{d|x \wedge x > d} \phi(\frac{x}{d}) \frac{x^2}{2d}$$
$$\sum_{a \in S} \sum_{b \in S} \gcd(a, b) = \sum_{d=1}^n (\sum_{x|d} \frac{d}{x} \mu(x)) (\sum_{d|v} \text{freq}[v])^2$$
$$\sum_{a \in S} \sum_{b \in S} \text{lcm}(a, b) = \sum_{d=1}^n \frac{1}{d} (\sum_{x|d} x \mu(x)) (\sum_{d|v} v \cdot \text{freq}[v])^2$$
$$\sum_{d|n} \mu(d) = [n = 1] \text{ (very useful)}$$
$$g(n) = \sum_{n|d} f(d) \Leftrightarrow f(n) = \sum_{n|d} \mu(d/n) g(d)$$
$$g(n) = \sum_{1 \leq m \leq n} f(\lfloor \frac{n}{m} \rfloor) \Leftrightarrow f(n) = \sum_{1 \leq m \leq n} \mu(m) g(\lfloor \frac{n}{m} \rfloor)$$

5.6.4 Dirichlet Convolution

Given a function $f(x)$, let

$$(f * g)(x) = \sum_{d|x} g(d) f(x/d)$$

If the partial sums $s_{f * g}(n), s_g(n)$ can be computed in $O(1)$ and $s_f(1 \dots n^{2/3})$ can be computed in $O(n^{2/3})$ then all $s_f(\frac{n}{d})$ can as well. Use

$$s_{f * g}(n) = \sum_{d=1}^n g(d) s_f(n/d).$$

$$\implies s_f(n) = \frac{s_{f * g}(n) - \sum_{d=2}^n g(d) s_f(n/d)}{g(1)}$$

- 1. If $f(x) = \mu(x)$ then $g(x) = 1, (f * g)(x) = (x == 1)$, and $s_f(n) = 1 - \sum_{i=2}^n s_f(n/i)$
- 2. If $f(x) = \phi(x)$ then $g(x) = 1, (f * g)(x) = x$, and $s_f(n) = \frac{n(n+1)}{2} - \sum_{i=2}^n s_f(n/i)$

dirichlet-convolution.h

Description: Dirichlet convolution. Change f, gs and fgs accordingly. This example calculates $\phi(N)$.

Time: $\mathcal{O}(N^{\frac{2}{3}})$

```
template<typename T, typename V> struct mertens {
    V N; T inv; // ~ N^{2/3}
    vector<V> fs; vector<T> psum;
    unordered_map<V, T> mapa;
    V f(V x) { return fs[x]; } // hash-1
    T gs(V x) { return x; }
    T fgs(V x) { return T(x) * (x + 1) / 2; }
    mertens(V M, const vector<V>& F) : N(M+1), fs(F), psum(M+1) {
        inv = gs(1);
        for (V a = 0; a + 1 < N; ++a)
            psum[a + 1] = f(a + 1) + psum[a];
    } // hash-1 = ceee32
    T query(V x) { // hash-2
        if (x < N) return psum[x];
        if (mapa.find(x) != mapa.end()) return mapa[x];
        T ans = fgs(x);
        for (V a = 2, b; a <= x; a = b + 1)
            b = x / (x/a), ans -= (gs(b) - gs(a-1)) * query(x/a);
        return mapa[x] = (ans / inv);
    } // hash-2 = 43d0b6
```

};

Combinatorial (6)

6.1 Permutations

6.1.1 Factorial

n	1	2	3	4	5	6	7	8	9	10
$n!$	1	2	6	24	120	720	5040	40320	362880	3628800
n	11	12	13	14	15	16	17			
$n!$	4.0e7	4.8e8	6.2e9	8.7e10	1.3e12	2.1e13	3.6e14			
n	20	25	30	40	50	100	150	171		
$n!$	2e18	2e25	3e32	8e47	3e64	9e157	6e262	>DBL_MAX		

int-perm.h

Description: Permutation -> integer conversion. (Not order preserving.)

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

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

6.1.2 Binomials

- Sum of every element in the n -th row of pascal triangle is 2^n .
- The product of the elements in each row is $\frac{(n+1)^n}{n!}$
- $\sum_{k=0}^n \binom{n}{k}^2 = \binom{2n}{n}$
- In a row p where p is a prime number, all the terms in that row except the 1s are multiples of p
- To count odd terms in row n , convert n to binary. Let x be the number of 1s in the binary representation. Then the number of odd terms will be 2^x
- Every entry in row $2^n - 1$ is odd

$$\binom{n}{k} = \frac{n}{k} \binom{n-1}{k-1}$$
$$\sum_{k=0}^n \binom{n}{k} k = n 2^{n-1}$$
$$\sum_{m=0}^n \binom{n}{j} \binom{n-m}{k-j} = \binom{n+1}{k+1}$$
$$\binom{n}{k} \binom{n-v}{v} = \binom{n}{v} \binom{n-v}{k}$$
$$\sum_{j=0}^m \binom{m}{j}^2 = \binom{2m}{m}$$
$$(x + y)^n = \sum_{k=0}^n \binom{n}{k} x^{n-k} y^k$$
$$\sum_{j=0}^k \binom{m}{j} \binom{n-m}{k-j} = \binom{n}{k}$$
$$\sum_{r=0}^m m \binom{n+r}{r} = \binom{n+m+1}{m+1}$$
$$\sum_{m=k}^n \binom{m}{k} = \binom{n+1}{k+1}$$
$$2 \sum_{i=L}^R \binom{n}{i} - \binom{n}{L} - \binom{n}{R} = \sum_{i=L+1}^R \binom{n+1}{i}$$

lucas.h

Description: Lucas' thm: 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 $\binom{n}{m} \equiv \prod_{i=0}^k \binom{n_i}{m_i} \pmod{p}$. fact and ifact must hold pre-computed factorials / inverse factorials, e.g. from ModInv.h.

Time: $\mathcal{O}(\log_p m)$

rolling-binomial.h

Description: $\binom{n}{k} \pmod m$ in time proportional to the difference between (n, k) and the previous (n, k) .

```
..../number-theory/preparator.h" de359e, 12 lines
struct Bin {
    int N = 0, K = 0; ll r = 1;
    void m(int a, int b) { r = r * a % mod * invs[b] % mod; }
    ll choose(int n, int k) {
        if (k > n || k < 0) return 0;
        while(N < n) ++N, m(N, N - K);
        while(K < k) ++K, m(N - K + 1, K);
        while(K > k) m(K, N - K + 1), --K;
        while(N > n) m(N - K, N), --N;
        return r;
    }
};
```

multinomial.h

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

```
864cdb, 7 lines
lint multinomial(vector<int>& v) {
    lint c = 1, m = v.empty() ? 1 : v[0];
    for (int i = 1 < v.size(); ++i)
        for (int j = 0; j < v[i]; ++j)
            c = c * ++m / (j+1);
    return c;
}
```

6.1.3 The twelvefold way (from Stanley)

How many functions $f: N \rightarrow X$ are there?

N	X	Any f	Injective	Surjective
dist.	dist.	x^n	$\frac{x!}{(x-n)!}$	$x!\{n\}_x$
indist.	dist.	$\binom{x+n-1}{n}$	$\binom{x}{n}$	$\binom{n-1}{n-x}$
dist.	indist.	$\{1\}^n + \dots + \{x\}^n$	$[n \leq x]$	$\{n\}_x$
indist.	indist.	$p_1(n) + \dots p_x(n)$	$[n \leq x]$	$p_x(n)$

Where $\binom{a}{b} = \frac{1}{b!}(a)_b$, $p_x(n)$ is the number of ways to partition the integer n using x summand and $\{n\}_x$ is the number of ways to partition a set of n elements into x subsets (aka Stirling number of the second kind).

6.1.4 Burnside

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

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

6.1.5 Cycles

Let the number of n -permutations whose cycle lenghts all belong to the set S be denoted by $g_S(n)$

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

6.1.6 Involutions

An involution is a permutation with maximum cycle length 2, and it is its own inverse.

$$a(n) = a(n-1) + (n-1)a(n-2), a(0) = a(1) = 1.$$

1, 1, 2, 4, 10, 26, 76, 232, 764, 2620, 9496, 35696, 140152

6.1.7 Derangements

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

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

6.2 Partitions and subsets

6.2.1 Partition function

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

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

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

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

partitions.h

```
3af1e7, 14 lines
vector<int64_t> prep(int N) {
    vector<int64_t> dp(N); dp[0] = 1;
    for (int n = 1; n < N; ++n) {
        int64_t sum = 0;
        for (int k = 0, l = 1, m = n - 1; ;) {
            sum += dp[m]; if ((m -= (k += 1)) < 0) break;
            sum += dp[m]; if ((m -= (l += 2)) < 0) break;
            sum -= dp[m]; if ((m -= (k += 1)) < 0) break;
            sum -= dp[m]; if ((m -= (l += 2)) < 0) break;
        }
        if ((sum %= M) < 0) sum += M;
        dp[n] = sum;
    } return dp;
}
```

6.3 General purpose numbers

6.3.1 Bernoulli numbers

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

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

Sums of powers:

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

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.

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

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

$$c(8, k) = 8, 0, 5040, 13068, 13132, 6769, 1960, 322, 28, 1$$

$$c(n, 2) = 0, 0, 1, 3, 11, 50, 274, 1764, 13068, 109584, \dots$$

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. $B(n) = 1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, \dots$

$$\mathcal{B}_{n+1} = \sum_{k=0}^n \binom{n}{k} \mathcal{B}_k$$

Also possible to calculate using Stirling numbers of the second kind,

$$B_n = \sum_{k=0}^n S(n, k)$$

If p is prime:

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

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 : $(n-2)! / ((d_1-1)! \dots (d_n-1)!)$ # forests with exactly k rooted trees:

$$\binom{n}{k} k \cdot n^{n-k-1}$$

.

6.3.7 Catalan numbers

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

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

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

6.3.8 Super Catalan numbers

The number of monotonic lattice paths of a $n \times n$ grid that do not touch the diagonal.

$$S(n) = \frac{3(2n-3)S(n-1) - (n-3)S(n-2)}{n}$$
$$S(1) = S(2) = 1$$

1, 1, 3, 11, 45, 197, 903, 4279, 20793, 103049, 518859

6.3.9 Motzkin numbers

Number of ways of drawing any number of nonintersecting chords among n points on a circle. Number of lattice paths from $(0, 0)$ to $(n, 0)$ never going below the x -axis, using only steps NE, E, SE.

$$M(n) = \frac{3(n-1)M(n-2) + (2n+1)M(n-1)}{n+2}$$
$$M(0) = M(1) = 1$$

1, 1, 2, 4, 9, 21, 51, 127, 323, 835, 2188, 5798, 15511, 41835, 113634

6.3.10 Narayana numbers

Number of lattice paths from $(0,0)$ to $(2n,0)$ never going below the x -axis, using only steps NE and SE, and with k peaks.

$$N(n,k) = \frac{1}{n} \binom{n}{k} \binom{n}{k-1}$$
$$N(n,1) = N(n,n) = 1$$
$$\sum_{k=1}^n N(n,k) = C_n$$

1, 1, 1, 1, 3, 1, 1, 6, 6, 1, 1, 10, 20, 10, 1, 1, 15, 50

6.3.11 Schroder numbers

Number of lattice paths from $(0, 0)$ to (n, n) using only steps N,NE,E, never going above the diagonal. Number of lattice paths from $(0, 0)$ to $(2n, 0)$ using only steps NE, SE and double east EE, never going below the x -axis. Twice the Super Catalan number, except for the first term.

1, 2, 6, 22, 90, 394, 1806, 8558, 41586, 206098

6.3.12 Triangles

Given rods of length 1, ..., n ,

$$T(n) = \frac{1}{24} \left\{ \begin{array}{ll} n(n-2)(2n-5) & n \text{ even} \\ (n-1)(n-3)(2n-1) & n \text{ odd} \end{array} \right\}$$

is the number of distinct triangles (positive are) that can be constructed, i.e., the # of 3-subsets of $[n]$ s.t. $x \leq y \leq z$ and $z \neq x + y$.

6.4 Fibonacci

$$Fib(x+y) = Fib(x+1)Fib(y) + Fib(x)Fib(y-1)$$

$$Fib(n+1)Fib(n-1) - Fib(n)^2 = (-1)^n$$

$$Fib(2n-1) = Fib(n)^2 - Fib(n-1)^2$$

$$\sum_{i=0}^n Fib(i) = Fib(n+2) - 1$$

$$\sum_{i=0}^n Fib(i)^2 = Fib(n)Fib(n+1)$$

$$\sum_{i=0}^n Fib(i)^3 = \frac{Fib(n)Fib(n+1)^2 - (-1)^n Fib(n-1) + 1}{2}$$

6.5 Linear Recurrences

$$F_i = \sum_{j=1}^K C_j F_{i-j} + D$$
$$\begin{bmatrix} 0 & 1 & 0 & 0 & \vdots & 0 & 0 \\ 0 & 0 & 1 & 0 & \vdots & 0 & 0 \\ C_K & C_{K-1} & C_{K-2} & C_{K-3} & \vdots & C_1 & 1 \end{bmatrix} \begin{bmatrix} F_0 \\ F_1 \\ F_2 \\ \vdots \\ F_{K-1} \\ D \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ \vdots \\ F_K \\ D \end{bmatrix}$$

6.6 Game Theory

A game can be reduced to Nim if it is a finite impartial game. Nim and its variants include:

6.6.1 Nim

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

6.6.2 Misère Nim

Regular Nim, except that the last player to move *loses*. Play regular Nim until there is only one pile of size larger than 1, reduce it to 0 or 1 such that there is an odd number of piles. The second player wins (a_1, \dots, a_n) if 1) there is a pile $a_i > 1$ and $\bigoplus_{i=1}^n a_i = 0$ or 2) all $a_i \leq 1$ and $\bigoplus_{i=1}^n a_i = 1$.

6.6.3 Staircase Nim

Stones are moved down a staircase and only removed from the last pile. $(x_i)_{i=1}^n$ is an L -position if $(x_{2i-1})_{i=1}^{n/2}$ is (i.e. only look at odd-numbered piles).

nim-product.h

Description: Product of nimbers is associative, commutative, and distributive over addition (xor). Forms finite field of size 2^{2^k} . Application: Given 1D coin turning games G_1, G_2 $G_1 \times G_2$ is the 2D coin turning game defined as follows. If turning coins at x_1, x_2, \dots, x_m is legal in G_1 and y_1, y_2, \dots, y_n is legal in G_2 , then turning coins at all positions (x_i, y_j) is legal assuming that the coin at (x_m, y_n) goes from heads to tails. Then the Grundy function $g(x, y)$ of $G_1 \times G_2$ is $g_1(x) \times g_2(y)$.

Time: 64² xors per multiplication, memorize to speed up.

```
ull nim_prod[64][64];
ull nim_prod2(int i, int j) { // hash-1
    if (nim_prod[i][j]) return nim_prod[i][j];
    if ((i & j) == 0) return nim_prod[i][j] = 1ull << (i|j);
    int a = (i&j) & ~(i&j);
    return nim_prod[i][j] = nim_prod2(i ^ a, j) ^ nim_prod2((i ^
        a) | (a-1), (j ^ a) | (i & (a-1)));
} // hash-1 = 65b711
void all_nim_prod() { // hash-2
    for (int i = 0; i < 64; i++)
        for (int j = 0; j < 64; j++)
            if ((i & j) == 0) nim_prod[i][j] = 1ull << (i|j);
            else {
                int a = (i&j) & ~(i&j);
                nim_prod[i][j] = nim_prod[i ^ a][j] ^ nim_prod[(i ^ a)
                    | (a-1)][(j ^ a) | (i & (a-1))];
            }
} // hash-2 = f46d0f
ull get_nim_prod(ull x, ull y) { // hash-3
    ull res = 0;
    for (int i = 0; i < 64 && (x >> i); ++i)
        if ((x >> i) & 1)
            for (int j = 0; j < 64 && (y >> j); ++j)
                if ((y >> j) & 1) res ^= nim_prod2(i, j);
    return res;
} // hash-3 = c5b053
```

Graph (7)

7.1 Fundamentals

euler-walk.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: $\mathcal{O}(V + E)$

```
vector<int> eulerWalk(vector<vector<pii>>& gr, int nedges, int
    src=0) {
    int n = gr.size();
    vector<int> D(n), its(n), eu(nedges), ret, s = {src};
    D[src]++; // to allow Euler paths, not just cycles
    while (!s.empty()) { // hash-1
        int x = s.back(), y, e, &it = its[x], end = int(gr[x].size
            ());
        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);
    } // hash-1 = f50118
    for(auto &x : D) if (x < 0 || int(ret.size()) != nedges+1)
        return {};
    return {ret.rbegin(), ret.rend()};
}
```

7.2 DFS algorithms

tarjan.h

Description: Finds all strongly connected components in a directed graph.
Usage: scc_t s(g); s.solve([&](const vector<int>& cc) {...}); visits all components in reverse topological order.
Time: $\mathcal{O}(E + V)$

5a01bd, 29 lines

```
struct scc_t {
    int n, t, scc_num;
    vector<vector<int>>> adj;
    vector<int> low, id, stk, in_stk, cc_id;
    scc_t(const auto& g) : n(int(g.size())), t(0), scc_num(0),
        adj(g), low(n,-1), id(n,-1), in_stk(n, false), cc_id(n) {}
    template<class F> void dfs(int cur, F f) { // hash-1
        id[cur] = low[cur] = t++;
        stk.push_back(cur); in_stk[cur] = true;
        for (int nxt : adj[cur])
            if (id[nxt] == -1)
                dfs(nxt, f), low[cur] = min(low[cur], low[nxt]);
            else if (in_stk[nxt])
                low[cur] = min(low[cur], id[nxt]);
        if (low[cur] == id[cur]) {
            vector<int> cc; cc.reserve(stk.size());
            while (true) {
                int v = stk.back(); stk.pop_back();
                in_stk[v] = false;
                cc.push_back(v); cc_id[v] = scc_num;
                if (v == cur) break;
            } f(cc); scc_num++;
        } // hash-1 = b8aa46
    }
    template<class F> void solve(F f) {
        stk.reserve(n);
        for (int r = 0; r < n; ++r) if (id[r] == -1) dfs(r, f);
    }
};
```

bcc.h

Description: Finds all biconnected components in an undirected graph. In a biconnected component there are at least two distinct paths between any two nodes or the component is a bridge. Note that a node can be in several components. *blockcut* constructs the block cut tree of given graph. The first nodes represents the blocks, the others represents the articulation points.

Usage: int e_id = 0; vector<pair<int, int>> g(N);
 for (auto [a,b] : edges) {
 g[a].emplace_back(b, e_id);
 g[b].emplace_back(a, e_id++); }
 bcc_t b(g); b.solve([&](const vector<int>& edges_id) {...});
Time: $\mathcal{O}(E + V)$

f05480, 50 lines

```
struct bcc_t {
    int n, t;
    vector<vector<pii>>> adj;
    vector<int> low, id, stk, is_art;
    bcc_t(const vector<vector<pii>>> &g) : n(int(g.size())),
        t(0), adj(g), low(n,-1), id(n,-1), is_art(n) {}
    template<class F> void dfs(int cur, int e_par, F f){
        id[cur] = low[cur] = t++;
        stk.push_back(e_par); int c = 0;
        for (auto [nxt, e_id] : adj[cur]) {
            if (id[nxt] == -1) { // hash-1
                dfs(nxt, e_id, f);
                low[cur] = min(low[cur], low[nxt]); c++;
                if (low[nxt] < id[cur]) continue;
                is_art[cur] = true;
                auto top = find(stk.rbegin(), stk.rend(), e_id);
                vector<int> cc(stk.rbegin(), next(top));
                f(cc); stk.resize(stk.size()-cc.size());
            }
        }
    }
};
```

```
    }
    else if (e_id != e_par) {
        low[cur] = min(low[cur], id[nxt]);
        if (id[nxt] < id[cur]) stk.push_back(e_id);
        // hash-1 = ea7101
    } if (e_par == -1) is_art[cur] = (c > 1) ? true : false;
}
template<class F> void solve(F f) {
    stk.reserve(n);
    for(int r = 0; r < n; ++r) if(id[r] == -1) dfs(r,-1,f);
}
auto blockcut(const vector<pii>& &edges){
    vector<vector<int>>> cc; vector<int> cc_id(n,-1);
    solve( [&](const vector<int>& &c) { // hash-2
        set<int> vc;
        for(int e : c){
            auto [a, b] = edges[e];
            cc_id[a] = cc_id[b] = int(cc.size());
            vc.insert(a); vc.insert(b);
        } cc.emplace_back(vc.begin(), vc.end());
    } ); // hash-2 = 4e2783
    for(int a = 0; a < n; a++) if(is_art[a])
        cc_id[a] = int(cc.size()), cc.push_back({a});
    int bcc_num = int(cc.size()); // hash-3
    vector<vector<int>>> tree(bcc_num);
    for(int c = 0; c < bcc_num && 1<int(cc[c].size()); ++c)
        for(int a : cc[c]) if(is_art[a]) {
            tree[c].push_back(cc_id[a]);
            tree[cc_id[a]].push_back(c);
        } return make_tuple(cc_id, cc, tree);
    } // hash-3 = 4b5ad9
};
```

2sat.h

Description: Calculates a valid assignment to boolean variables a, b, c,... to a 2-SAT problem, so that an expression of the type $(a \vee b) \wedge (\neg a \vee c) \wedge (d \vee \neg b) \wedge \dots$ becomes true, or reports that it is unsatisfiable. Negated variables are represented by bit-inversions ($\sim x$).

Usage: TwoSat ts(number of boolean variables);
 ts.either(0, ~3); // Var 0 is true or var 3 is false
 ts.set.value(2); // Var 2 is true
 ts.at.most.one({0, ~1, 2}); // ≤ 1 of vars 0, ~1 and 2 are true
 ts.solve(); // Returns true iff it is solvable
 ts.values[0..N-1] holds the assigned values to the vars
Time: $\mathcal{O}(N + E)$, where N is the number of boolean variables, and E is the number of clauses.

"tarjan.h" 74e46b, 36 lines

```
struct TwoSat { // hash-1
    int N;
    vector<vector<int>>> gr;
    vector<int> values; // 0 = false, 1 = true
    TwoSat(int n = 0) : N(n), gr(2*n) {}
    int add_var() { // (optional)
        gr.emplace_back(); gr.emplace_back();
        return N++;
    } // hash-1 = cc8a86
    void either(int f, int j) { // hash-2
        f = max(2*f, -1-2*f); j = max(2*j, -1-2*j);
        gr[f].push_back(j^1); gr[j].push_back(f^1);
    } // hash-2 = 516db0
    void implies(int f, int j) { either(~f, j); }
    void set_value(int x) { either(x, x); }
    void at_most_one(const vector<int>& li) { // (optional)
        if (int(li.size()) <= 1) return;
        int cur = ~li[0];
        for (int i = 2; i < int(li.size()); ++i) {
            int next = add_var();
            either(cur, ~li[i]); either(cur, next);
            either(~li[i], next); cur = ~next;
        }
    }
};
```

```
    } either(cur, ~li[1]);
}
bool solve() { // hash-3
    scc_t s(gr);
    s.solve([&](const vector<int>& &v){ return; });
    values.assign(N, -1);
    for (int i = 0; i < N; ++i)
        if (s.cc_id[2*i] == s.cc_id[2*i+1]) return 0;
    for (int i = 0; i < N; ++i)
        if (s.cc_id[2*i] < s.cc_id[2*i+1]) values[i] = false;
        else values[i] = true;
    return 1;
} // hash-3 = 91930f
};
```

7.3 Trees

centroid-decomposition.h

8b1d29, 41 lines

```
struct centroid_tree { // hash-1
    int N;
    vector<pair<int, int>>> max_sub;
    vector<vector<int>>> dist, adj;
    vector<int> depth, par;
    vector<bool> deleted;
    centroid_tree(const auto& G) : N(G.size()), adj(G), depth(N),
        par(N,-1), deleted(N), dist(_lgl(N)+1, vector(N,0)),
        max_sub(N) {
        rec(0, -1, 0);
    } // hash-1 = 05eea5
    int dfs(int cur, int prv) { // hash-2
        int sz = 1;
        max_sub[cur] = {0, -1};
        for (int nxt : adj[cur]) {
            if (nxt == prv || deleted[nxt]) continue;
            int cur_sz = dfs(nxt, cur);
            sz += cur_sz;
            max_sub[cur] = max(max_sub[cur], {cur_sz, nxt});
        }
        return sz;
    } // hash-2 = 57a49c
    void layer(int cur, int prv, int lvl) { // hash-3
        for (int nxt : adj[cur]) {
            if (nxt == prv || deleted[nxt]) continue;
            dist[lvl][nxt] = dist[lvl][cur] + 1;
            layer(nxt, cur, lvl);
        }
    } // hash-3 = 6e1d27
    void rec(int cur, int prv, int d) { // hash-4
        int sz = dfs(cur, prv);
        while(max_sub[cur].first > sz/2) cur = max_sub[cur].second;
        depth[cur] = d;
        par[cur] = prv;
        layer(cur, prv, d);
        deleted[cur] = true;
        for (int nxt : adj[cur]) {
            if (deleted[nxt]) continue;
            rec(nxt, cur, d + 1);
        }
    } // hash-4 = 24f90d
};
```

heavylight.h

Description: Compress Tree: Given a subset S of nodes, computes the compress tree and returns a list of edges representing a tree rooted at 0.
Time: $\mathcal{O}((\log N)^2)$

d3dcdc, 77 lines

```
template<bool use_edges> struct hld_t {
    int N, T{};
```

```

vector<vector<int>>> adj;
vector<int> sz, depth, chain, par, in, out, preorder;
hld_t() {}
hld_t(const vector<vector<int>>& G, int r = 0) : N(int(G.size()
)),
adj(G), sz(N), depth(N), chain(N), par(N), in(N), out(N),
preorder(N) { dfs_sz(r); chain[r] = r; dfs_hld(r); }
void dfs_sz(int cur) { // hash-1
sz[cur] = 1;
for (auto& nxt : adj[cur]) {
par[nxt] = cur; depth[nxt] = 1 + depth[cur];
adj[nxt].erase(find(adj[nxt].begin(), adj[nxt].end(), cur
));
dfs_sz(nxt); sz[cur] += sz[nxt];
if (sz[nxt] > sz[adj[cur][0]]) swap(nxt, adj[cur][0]);
}
} // hash-1 = 87f84a
void dfs_hld(int cur) { // hash-2
in[cur] = T++; preorder[in[cur]] = cur;
for (auto& nxt : adj[cur]) {
chain[nxt] = (nxt == adj[cur][0] ? chain[cur] : nxt);
dfs_hld(nxt);
} out[cur] = T;
} // hash-2 = b2413a
int lca(int a, int b) { // hash-3
while (chain[a] != chain[b]) {
if (in[a] < in[b]) swap(a, b);
a = par[chain[a]];
} return (in[a] < in[b] ? a : b);
} // hash-3 = 672231
bool is_ancestor(int a, int b) { return in[a] <= in[b] && in[
b] < out[a]; }
int climb(int a, int k) { // hash-4
if (depth[a] < k) return -1;
int d = depth[a] - k;
while (depth[chain[a]] > d) a = par[chain[a]];
return preorder[in[a] - depth[a] + d];
} // hash-4 = 268fc4
int kth_on_path(int a, int b, int K) { // hash-5
int m = lca(a, b);
int x = depth[a] - depth[m], y = depth[b] - depth[m];
if (K > x + y) return -1;
return (x > K ? climb(a, K) : climb(b, x + y - K));
} // hash-5 = 34f729
// bool is true if path should be reversed (only for
noncommutative operations)
const auto& get_path(int a, int b) const { // hash-6
static vector<tuple<bool, int, int>> L, R;
L.clear(); R.clear();
while (chain[a] != chain[b])
if (depth[chain[a]] > depth[chain[b]]) {
L.push_back({true, in[chain[a]], in[a] + 1});
a = par[chain[a]];
} else {
R.push_back({false, in[chain[b]], in[b] + 1});
b = par[chain[b]];
}
}
if (depth[a] > depth[b])
L.push_back({true, in[b] + use_edges, in[a] + 1});
else R.push_back({false, in[a] + use_edges, in[b] + 1});
L.insert(L.end(), R.rbegin(), R.rend());
return L;
} // hash-6 = 5b653a
auto get_subtree(int a) const {
return make_pair(in[a] + use_edges, in[a] + sz[a]);
}
auto compress(vector<int> s) { // hash-7
auto cmp = [&](int a, int b){ return in[a] < in[b]; };
sort(s.begin(), s.end(), cmp); int m = int(s.size())-1;

```

```

for (int i = 0; i < m; ++i)
s.push_back(lca(s[i], s[i+1]));
sort(s.begin(), s.end(), cmp);
s.erase(unique(s.begin(), s.end(), s.end()));
vector<pii> ret = { {0, s[0]} };
for (int i = 0; i + 1 < int(s.size()); ++i)
ret.emplace_back(lca(s[i], s[i+1]), s[i+1]);
return ret;
} // hash-7 = a035c4
};

```

tree-isomorphism.h

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

a4f6c1, 38 lines

```

struct tree_t { // hash-1
vector<int> cen, sz;
vector<vector<int>>> adj;
tree_t(vector<vector<int>>& g):cen(2), sz(g.size()), adj(g){}
int dfs_sz(int v, int p) {
sz[v] = 1;
for (int u : adj[v]) if (u != p)
sz[v] += dfs_sz(u, v);
return sz[v];
} // hash-1 = f1f64f
int dfs(int tsz, int v, int p) { // hash-2
for (int u : adj[v]) if (u != p) {
if (2*sz[u] <= tsz) continue;
return dfs(tsz, u, v);
} return cen[0] = v;
} // hash-2 = d35be1
void find_cenroid(int v) { // hash-3
int tsz = dfs_sz(v, -1);
cen[1] = dfs(tsz, v, -1);
for (int u : adj[cen[0]]) if (2*sz[u] == tsz)
cen[1] = u;
} // hash-3 = 6d03ed
int hash_it(int v, int p = -1) { // hash-4
static map<vector<int>, int> val;
vector<int> offset;
for (int u : adj[v]) if (u != p)
offset.push_back(hash_it(u, v));
sort(offset.begin(), offset.end());
if (!val.count(offset)) val[offset] = int(val.size());
return val[offset];
} // hash-4 = f8d4d1
ll get_hash(int v = 0) { // hash-5
find_cenroid(v);
ll x = hash_it(cen[0]), y=hash_it(cen[1]);
if (x > y) swap(x, y);
return (x << 30) + y;
} // hash-5 = 387158
};

```

7.4 Network flow

dinitz.h

Description: Flow algorithm with complexity $\mathcal{O}(VE \log U)$ where $U = \max|cap|$. $\mathcal{O}(\min(E^{1/2}, V^{2/3})E)$ if $U = 1$; $\mathcal{O}(\sqrt{VE})$ for bipartite matching. To obtain each partition A and B of the cut look at lvl , for $v \in A$, $lvl[v] > 0$, for $u \in B$, $lvl[u] = 0$.

de8d6b, 44 lines

```

template<typename T = int> struct Dinitz { // hash-1
struct edge_t { int to, rev; T c, f; };
vector<vector<edge_t>>> adj;
vector<int> lvl, ptr, q;
Dinitz(int n) : lvl(n), ptr(n), q(n), adj(n) {}
inline void addEdge(int a, int b, T c, T rcap = 0) {
adj[a].push_back({b, (int)adj[b].size(), c, 0});
adj[b].push_back({a, (int)adj[a].size() - 1, rcap, 0});
} // hash-1 = 5a0d77

```

```

T dfs(int v, int t, T f) { // hash-2
if (v == t || !f) return f;
for (int &i = ptr[v]; i < int(adj[v].size()); ++i) {
edge_t &e = adj[v][i];
if (lvl[e.to] == lvl[v] + 1)
if (T p = dfs(e.to, t, min(f, e.c - e.f)) {
e.f += p, adj[e.to][e.rev].f -= p;
return p;
}
} return 0;
} // hash-2 = 8ffe6b
T maxflow(int s, int t) { // hash-3
T flow = 0; q[0] = s;
for (int L = 0; L < 31; ++L) do { // consider L = 30
lvl = ptr = vector<int>(q.size());
int qi = 0, qe = lvl[s] = 1;
while (qi < qe && !lvl[t]) {
int v = q[qi++];
for (edge_t &e : adj[v])
if (!lvl[e.to] && (e.c - e.f) >> (30 - L))
q[qi++] = e.to, lvl[e.to] = lvl[v] + 1;
}
while (T p = dfs(s, t, numeric_limits<T>::max()/4)) flow+=p;
} while (lvl[t]);
return flow;
} // hash-3 = db2141
bool leftOfMinCut(int v) { return bool(lvl[v] != 0); }
auto minCut(int s, int t) { // hash-4
T cost = maxflow(s, t);
vector<edge_t> cut;
for (int i = 0; i < int(adj.size()); ++i) for (edge_t &e :
adj[i])
if (lvl[i] && !lvl[e.to]) cut.push_back(e);
return make_pair(cost, cut);
} // hash-4 = 1843d5
};

```

min-cost-max-flow.h

Description: Min-cost max-flow. Assumes there is no negative cycle.

Time: $\mathcal{O}(F(V + E)\log V)$, being F the amount of flow.

36dde8, 57 lines

```

template<class flow_t, class cost_t> struct min_cost {
static constexpr flow_t FLOW_EPS = flow_t(1e-10);
static constexpr flow_t FLOW_INF = numeric_limits<flow_t>::
max();
static constexpr cost_t COST_EPS = cost_t(1e-10);
static constexpr cost_t COST_INF = numeric_limits<cost_t>::
max();
int n, m{}; vector<int> ptr, nxt, zu;
vector<flow_t> capa; vector<cost_t> cost;
min_cost(int N) : n(N), ptr(n, -1), dist(n), vis(n), pari(n) {}
void addEdge(int u, int v, flow_t w, cost_t c) {
nxt.push_back(ptr[u]); zu.push_back(v); capa.push_back(w);
cost.push_back(c); ptr[u] = m++;
nxt.push_back(ptr[v]); zu.push_back(u); capa.push_back(0);
cost.push_back(-c); ptr[v] = m++;
}
vector<cost_t> pot, dist; vector<bool> vis; vector<int> pari;
vector<flow_t> flows; vector<cost_t> slopes;
// You can pass t = -1 to find a shortest
void shortest(int s, int t) { //path to each vertex. // hash-1
using E = pair<cost_t, int>;
priority_queue<E, vector<E>, greater<E>> que;
for(int u = 0; u < n; ++u) {dist[u]=COST_INF; vis[u]=false;}
for (que.emplace(dist[s] = 0, s); !que.empty(); ) {
const cost_t c = que.top().first;
const int u = que.top().second; que.pop();
if (vis[u]) continue;
vis[u] = true; if (u == t) return;

```



```

    for (int i = ptr[u]; ~i; i = nxt[i]) if (capa[i] >
        FLOW_EPS) {
        const int v = zu[i];
        const cost_t cc = c + cost[i] + pot[u] - pot[v];
        if (dist[v] > cc) { que.emplace(dist[v]=cc, v); pari[v]=i; }
    }
} // hash-1 = 89f16a
auto run(int s, int t, flow_t limFlow = FLOW_INF) { // hash-2
    pot.assign(n, 0); flows = {0}; slopes.clear();
    while (true) {
        bool upd = false;
        for (int i = 0; i < m; ++i) if (capa[i] > FLOW_EPS) {
            const int u = zu[i ^ 1], v = zu[i];
            const cost_t cc = pot[u] + cost[i];
            if (pot[v] > cc + COST_EPS) { pot[v] = cc; upd = true; }
        } if (!upd) break;
    }
    flow_t flow = 0; cost_t tot_cost = 0;
    while (flow < limFlow) {
        shortest(s, t); flow_t f = limFlow - flow;
        if (!vis[t]) break;
        for (int u = 0; u < n; ++u) pot[u] += min(dist[u], dist[t]);
        for (int v = t; v != s; ) { const int i = pari[v];
            if (f > capa[i]) { f = capa[i]; } v = zu[i ^ 1];
        }
        for (int v = t; v != s; ) { const int i = pari[v];
            capa[i] -= f; capa[i ^ 1] += f; v = zu[i ^ 1];
        }
        flow += f; tot_cost += f * (pot[t] - pot[s]);
        flows.push_back(flow); slopes.push_back(pot[t] - pot[s]);
    } return make_pair(flow, tot_cost);
} // hash-2 = 285527
};

```

7.5 Matching

hopcroft-karp.h

Description: Fast bipartite matching algorithm. Graph g should be a list of neighbors of the left partition, and $btoa$ should be a vector full of -1's of the same size as the right partition. Returns the size of the matching. $btoa[i]$ will be the match for vertex i on the right side, or -1 if it's not matched.

Usage: vector<int> btoa(m, -1); hopcroftKarp(g, btoa);

Time: $\mathcal{O}(\sqrt{VE})$ d9a55d, 35 lines

```

using vi = vector<int>;
bool dfs(int a, int L, const vector<vi> &g, vi &btoa, vi &A, vi
    &B) { // hash-1
    if (A[a] != L) return 0;
    A[a] = -1;
    for (auto &b : g[a]) if (B[b] == L + 1) {
        B[b] = 0;
        if (btoa[b] == -1 || dfs(btoa[b], L+1, g, btoa, A, B))
            return btoa[b] = a, 1;
    }
    return 0;
} // hash-1 = 2beb8d
int hopcroftKarp(const vector<vi> &g, vi &btoa) { // hash-2
    int res = 0;
    vector<int> A(g.size()), B(int(btoa.size()), cur, next;
    for (;) {
        fill(A.begin(), A.end(), 0), fill(B.begin(), B.end(), 0);
        cur.clear();
        for (auto &a : btoa) if (a != -1) A[a] = -1;
        for (int a = 0; a < g.size(); ++a) if (A[a] == 0) cur.
            push_back(a);
        for (int lay = 1;; ++lay) {
            bool islast = 0; next.clear();
            for (auto &a : cur) for (auto &b : g[a]) {

```

```

                if (btoa[b] == -1) B[b] = lay, islast = 1;
                else if (btoa[b] != a && !B[b])
                    B[b] = lay, next.push_back(btoa[b]);
            }
            if (islast) break;
            if (next.empty()) return res;
            for (auto &a : next) A[a] = lay;
            cur.swap(next);
        }
        for (int a = 0; a < int(g.size()); ++a)
            res += dfs(a, 0, g, btoa, A, B);
    }
} // hash-2 = ea37fa

```

bipartite-matching.h

Description: Fast Kuhn! Simple maximum cardinality bipartite matching algorithm. Better than hopcroftKarp in practice. Worst case is $\mathcal{O}(VE)$ on an hairy tree. Shuffling the edges and vertices ordering should break some worst-case inputs.

Time: $\Omega(VE)$ 1b4d72, 31 lines

```

struct bm_t {
    int N, M, T;
    vector<vector<int>> adj;
    vector<int> match, seen;
    bm_t(int a, int b) : N(a), M(a+b), T(0), adj(M),
        match(M, -1), seen(M, -1) {}
    void add_edge(int a, int b) { adj[a].push_back(b + N); }
    bool dfs(int cur) { // hash-1
        if (seen[cur] == T) return false;
        seen[cur] = T;
        for (int nxt : adj[cur]) if (match[nxt] == -1) {
            match[nxt] = cur, match[cur] = nxt;
            return true;
        }
        for (int nxt : adj[cur]) if (dfs(match[nxt])) {
            match[nxt] = cur, match[cur] = nxt;
            return true;
        }
        return false;
    } // hash-1 = 85af97
    int solve() { // hash-2
        int res = 0;
        for (int cur = 1; cur; ) {
            cur = 0; ++T;
            for (int i = 0; i < N; ++i) if (match[i] == -1)
                cur += dfs(i);
            res += cur;
        }
        return res;
    } // hash-2 = ff3448
};

```

weighted-matching.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 $\text{cost}[N][M]$, where $\text{cost}[i][j] = \text{cost}$ for $L[i]$ to be matched with $R[j]$ and returns (min cost, match), where $L[i]$ is matched with $R[\text{match}[i]]$. Negate costs for max cost.

Time: $\mathcal{O}(N^2M)$ f0ea90, 28 lines

```

pair<int, vector<int>> hungarian(const vector<vector<int>> &a) {
    if (a.empty()) return {0, {}};
    int n = a.size() + 1, m = a[0].size() + 1;
    vector<int> u(n), v(m), p(m), ans(n - 1);
    for (int i = 1; i < n; ++i) { // hash-1
        p[0] = i; int j0 = 0; // add "dummy" worker 0
        vector<int> dist(m, INT_MAX), pre(m, -1);
        vector<bool> done(m + 1);

```

```

        do {
            done[j0] = true;
            int i0 = p[j0], j1, delta = INT_MAX;
            for (int j = 1; j < m; ++j) 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;
            }
            for (int j = 0; j < m; ++j)
                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;
        }
    } // hash-1 = 6dbf35
    for (int j = 1; j < m; ++j) if (p[j]) ans[p[j]-1] = j-1;
    return {-v[0], ans}; // min cost
}

```

general-matching.h

Description: Maximum Matching for general graphs (undirected and non bipartite) using Edmond's Blossom Algorithm.

Time: $\mathcal{O}(EV^2)$ e5db8e, 47 lines

```

struct blossom_t { // hash-1
    int N, M; vector<vector<int>> adj;
    vector<int> match, ts, ps; vector<array<int, 2>> fs;
    blossom_t(auto& G) : N(int(G.size())), M(0), adj(G), match(N
        , -1), ts(N, -1), ps(N, -1), fs(N, {-1, -1}) {}
    int root(int a) {
        return (ts[a] != M || !~ps[a]) ? a : (ps[a] = root(ps[a]));
    } // hash-1 = abfc73
    void rematch(int a, int b) { // hash-2
        const int w = match[a]; match[a] = b; auto [x, y] = fs[a];
        if (~w && match[w] == a) {
            if (~y) rematch(x, y), rematch(y, x);
            else match[w] = x, rematch(x, w);
        }
    } // hash-2 = b389e7
    bool augment(int src) { // hash-3
        vector<int> bfs = {src}; bfs.reserve(N);
        ts[src] = M; ps[src] = -1; fs[src] = {-1, -1};
        for (int z = 0; z < int(bfs.size()); ++z) {
            int cur = bfs[z];
            for (int nxt : adj[cur]) if (nxt != src) {
                if (match[nxt] == -1) {
                    match[nxt] = cur; rematch(cur, nxt); return true;
                }
                if (ts[nxt] == M) {
                    int a = root(cur), b = root(nxt), m = src;
                    if (a == b) continue;
                    while (a != src || b != src) {
                        if (b != src) swap(a, b);
                        if (fs[a][0] == cur && fs[a][1] == nxt) { m = a; break; }
                        fs[a] = {cur, nxt}; a = root(fs[match[a]][0]);
                    }
                    for (const int r : {root(cur), root(nxt)})
                        for (int v = r; v != m; v = root(fs[match[v]][0]))
                            ts[v] = M, ps[v] = m, bfs.push_back(v);
                }
            } else if (ts[match[nxt]] != M) {
                fs[nxt] = {-1, -1}; ts[match[nxt]] = M;
                ps[match[nxt]] = nxt; fs[match[nxt]] = {cur, -1};
                bfs.push_back(match[nxt]);
            }
        }
    } return false;
} // hash-3 = e2482c

```



```
int run() {
    for(int v = 0; v < N; ++v) if(!~match[v]) M += augment(v);
    return M;
}
};
```

max-independent-set.h

Description: To obtain a maximum independent set of a graph, find a max clique of the complement. If the graph is bipartite, see MinimumVertex-Cover.

7.6 Heuristics

maximal-cliques.h

Description: Runs a callback for all maximal cliques in a graph (given as a symmetric bitset matrix; self-edges not allowed). Possible optimization: on the top-most recursion level, ignore 'cands', and go through nodes in order of increasing degree, where degrees go down as nodes are removed.

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

```
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 cands = P & ~eds[q];
    for(int i = 0; i < eds.size(); ++i) if (cands[i]) {
        R[i] = 1; cliques(eds, f, P & eds[i], X & eds[i], R);
        R[i] = P[i] = 0; X[i] = 1;
    }
}
```

chromatic-number.h

Description: Compute the chromatic number of a graph. Minimum number of colors needed to paint the graph in a way s.t. if two vertices share an edge, they must have distinct colors.

Time: $\mathcal{O}\left(N2^N\right)$

```
template<class T> int min_colors(int N, const T& gr) {
    vector<int> adj(N);
    for (int a = 0; a < N; ++a) // hash-1
        for (int b = a + 1; b < N; ++b) {
            if (!gr[a][b]) continue;
            adj[a] |= (1 << b); adj[b] |= (1 << a);
        } // hash-1 = 1612da
    static vector<unsigned> dp(1 << N), buf(1 << N), w(1 << N);
    for (int mask = 0; mask < (1 << N); ++mask) { // hash-2
        bool ok = true;
        for (int i = 0; i < N; ++i) if (mask & 1 << i)
            if (adj[i] & mask) ok = false;
        if (ok) dp[mask]++;
        buf[mask] = 1;
        w[mask] = __builtin_popcount(mask) % 2 == N % 2 ? 1 : -1;
    } // hash-2 = 4efbee
    for (int i = 0; i < N; ++i) // hash-3
        for (int mask = 0; mask < (1 << N); ++mask)
            if (!(mask & 1 << i)) dp[mask^(1 << i)] += dp[mask];
    for (int colors = 1; colors <= N; ++colors) {
        unsigned S = 0;
        for (int mask = 0; mask < (1 << N); ++mask)
            S += (buf[mask] * dp[mask]) * w[mask];
        if (S) return colors;
    } assert(false); // hash-3 = 1e2316
}
```

cycle-counting.h

Description: Counts 3 and 4 cycles

Time: $\mathcal{O}\left(E\sqrt{E}\right)$

```
using vi = vector<int>;
int count_cycles(const vector<vi>& adj, const vi& deg) {
    const int N = int(adj.size());
    vi idx(N), loc(N); iota(idx.begin(), idx.end(), 0);
    sort(idx.begin(), idx.end(), [&](const int& a, const int& b)
        { return deg[a] < deg[b]; });
    for (int i = 0; i < N; ++i) loc[idx[i]] = i;
    vector<vi> gr(N);
    for (int a = 0; a < N; ++a) for (int b : adj[a])
        if (loc[a] < loc[b]) gr[a].push_back(b);
    int cycle3 = 0, cycle4 = 0; // hash-1
    {
        vector<bool> seen(N, false);
        for (int a = 0; a < N; ++a) {
            for (int b : gr[a]) seen[b] = true;
            for (int b : gr[a]) for (int c : gr[b])
                if (seen[c]) cycle3 += 1;
            for (int b : gr[a]) seen[b] = false;
        }
    }
    vi cnt(N);
    for (int a = 0; a < N; ++a) {
        for (int b : adj[a]) for (int c : gr[b])
            if (loc[a] < loc[c]) {
                cycle4 += cnt[c];
                cnt[c]++;
            }
        for (int b : adj[a]) for (int c : gr[b]) cnt[c] = 0;
    }
    return cycle3; // hash-1 = 586f3b
}
```

edge-coloring.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: $\mathcal{O}(NM)$

```
vector<int> misra_gries(int N, vector<pair<int, int>> eds) {
    const int M = int(eds.size());
    vector<int> cc(N + 1), ret(M), fan(N), free(N), loc;
    for (auto e : eds) ++cc[e.first], ++cc[e.second];
    int u, v, ncols = *max_element(cc.begin(), cc.end()) + 1;
    vector<vector<int>> adj(N, vi(ncols, -1)); // hash-1
    for (auto 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;
        } // hash-1 = ed5ec6
        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++);
    }
    for (int i = 0; i < M; ++i)
        for (tie(u, v) = eds[i]; adj[u][ret[i]] != v;) ++ret[i];
}
```

```
return ret;
}
```

7.7 Other

manhattan-mst.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: $\mathcal{O}(N\log N)$

```
<dsu.h>
de8170, 24 lines
typedef Point<int> P;
pair<vector<array<int, 3>>, int> manhattanMST(vector<P> ps) {
    vector<int> id(ps.size()); iota(id.begin(), id.end(), 0);
    vector<array<int, 3>> edges; // hash-1
    for(int k = 0; k < 4; ++k) {
        sort(id.begin(), id.end(), [&](int i, int j) {
            return (ps[i]-ps[j]).x < (ps[j]-ps[i]).y;});
        map<int, int> sweep;
        for(auto& 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;
        }
        if (k & 1) for(auto& p : ps) p.x = -p.x;
        else for(auto& p : ps) swap(p.x, p.y);
    } // hash-1 = b02b12
    sort(edges.begin(), edges.end());
    UF uf(ps.size()); int cost = 0;
    for (auto e: edges) if (uf.unite(e[1], e[2])) cost += e[0];
    return {edges, cost};
}
```

directed-mst.h

Description: Edmonds' algorithm for finding the weight of the 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/dsu-rollback.h"
84af6a, 56 lines
struct Edge { int a, b; ll w; };
struct Node {
    Edge key; // hash-1
    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;
} // hash-1 = 245213
void pop(Node& a) { a->prop(); a = merge(a->l, a->r); }
pair<ll,vi> dmst(int n, int r, vector<Edge>& g) { // hash-2
    RollbackUF uf(n);
    vector<Node*> heap(n);
    for (Edge e : g) heap[e.b] = merge(heap[e.b], new Node{e});
    ll res = 0;
    vector<int> 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>>> cycs;
for(int s = 0; s < n; ++s) {
    int u = s, qi = 0, w;
    while (seen[u] < 0) {
        if (!heap[u]) return make_pair(-1L, vector<int>());
        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;
            do cyc = at, time = f.time(),
            do cyc = merge(cyc, heap[w = path[--qi]]);
            while (u == path[qi]);
            cycs.push_back({s, u, time, {&Q[qi], &Q[end]}});
        }
    }
    for(int i = 0; i < qi; ++i) in[uf.find(Q[i].b)] = Q[i];
}
for (auto& [u,t,comp] : cycs) { // 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;
}
for(int i = 0; i < n; ++i) par[i] = in[i].a;
return {res, par};
} // hash-2 = 51cafa

```

7.8.3 Construct minimum vertex cover from maximum matching on bipartite graph

1. Redirect every edge $e = (x, y) \in M$, $x \rightarrow y$ otherwise. $cycs.push_front(\{u, time, \{&Q[qi], &Q[end]\}\});$

```

}
for(int i = 0; i < qi; ++i) in[uf.find(Q[i].b)] = Q[i];
}
for (auto& [u,t,comp] : cycs) { // 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;
}
for(int i = 0; i < n; ++i) par[i] = in[i].a;
return {res, par};
} // hash-2 = 51cafa

```

2. DFS from unmatched vertices in X .

7.8.2 Konig's theorem

7.8.4 Minimum weight edge cover

In any bipartite graph the following holds: Maximum matching and minimum vertex cover have the same cardinality. Minimum edge cover, minimum path cover and maximum independent set have the same cardinality $N - |M|$.

7.8.5 Maximum/Minimum flow with lower bound / Circulation problem

1. Construct super source S and sink T .

3. $x \in X$ is chosen if x is unvisited.

2. Connect $v \rightarrow v'$ with weight $2\mu(v)$, where $\mu(v)$ is the cost of the cheapest edge incident to v .

7.8 Theorems

7.8.1 Euler's theorem

Let V , A and F be the number of vertices, edges and faces of connected planar graph, $V - A + F = 2$.

4. For each edge $(x, y) \in E$, connect $x \rightarrow y$ with capacity $u - l$.

3. For each vertex v , denote by $in(v)$ the difference between the sum of incoming lower bounds and the sum of outgoing lower bounds.
4. If $in(v) > 0$, connect $S \rightarrow v$ with capacity $in(v)$, otherwise, connect $v \rightarrow T$ with capacity $-in(v)$.

- To maximize, connect $t \rightarrow s$ with capacity ∞ (skip this in circulation problem), and let f be the maximum flow from S to T . If $f \neq \sum_{v \in V, in(v) > 0} in(v)$, there's no solution. Otherwise, the maximum flow from s to t is the answer.

- To minimize, let f be the maximum flow from S to T . Connect $t \rightarrow s$ with capacity ∞ and let the flow from S to T be f' . If $f + f' \neq \sum_{v \in V, in(v) > 0} in(v)$, there's no solution. Otherwise, f' is the answer.

5. The solution of each edge e is $l_e + f_e$, where f_e corresponds to the flow of edge e on the graph.

7.8.6 Minimum cost cyclic flow

1. Construct super source S and sink T
2. For each edge (x, y, c) , connect $x \rightarrow y$ with $(cost, cap) = (c, 1)$ if $c > 0$, otherwise connect $y \rightarrow x$ with $(cost, cap) = (-c, 1)$
3. For each edge with $c < 0$, sum these cost as K , then increase $d(y)$ by 1, decrease $d(x)$ by 1
4. For each vertex v with $d(v) > 0$, connect $S \rightarrow v$ with $(cost, cap) = (0, d(v))$
5. For each vertex v with $d(v) < 0$, connect $v \rightarrow T$ with $(cost, cap) = (0, -d(v))$
6. Flow from S to T , the answer is the cost of the flow $C + K$

7.8.7 Maximum density induced subgraph

1. Binary search on answer, suppose we're checking answer T
2. Construct a max flow model, let K be the sum of all weights
3. Connect source $s \rightarrow v$, $v \in G$ with capacity K
4. For each edge (u, v, w) in G , connect $u \rightarrow v$ and $v \rightarrow u$ with capacity w
5. For $v \in G$, connect it with sink $v \rightarrow t$ with capacity $K + 2T - (\sum_{e \in E(v)} w(e)) - 2w(v)$
6. T is a valid answer if the maximum flow $f < K|V|$

7.8.8 Project selection problem

1. If $p_v > 0$, create edge (s, v) with capacity p_v ; otherwise, create edge (v, t) with capacity $-p_v$.
2. Create edge (u, v) with capacity w with w being the cost of choosing u without choosing v .
3. The mincut is equivalent to the maximum profit of a subset of projects.

7.8.9 0/1 quadratic programming

$$\sum_x c_x x + \sum_y c_y \bar{y} + \sum_{xy} c_{xy} x \bar{y} + \sum_{xyx'y'} c_{xyx'y'} (x \bar{y} + x' \bar{y}')$$

can be minimized by the mincut of the following graph:

1. Create edge (x, t) with capacity c_x and create edge (s, y) with capacity c_y .

7.8.10 Eulerian Cycles

7.8.11 Number of Spanning Trees

The number of Eulerian cycles in a directed graph G is: Define Laplacian Matrix as $L = D - A$, D being a Diagonal Matrix with $D_{ii} = \deg(i)$ and A an Adjacency Matrix. Create an $N \times N$ spanning tree mat, and for each edge $a \rightarrow b \in G$, do $mat[a][b]--, mat[b][b]++$ (and $mat[b][a]--, mat[a][a]++$ 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).

7.8.12 Tutte Matrix

- A graph has a perfect matching iff the Tutte matrix has a non-zero determinant.

- The rank of the Tutte matrix is equal to twice the size of the maximum matching. The maximum cost matching can be found by polynomial interpolation.

Geometry (8)

8.1 Geometry

all-geometry.h

Description: Geometry Library

a88326, 638 lines

```

template <class T> int sgn(T x) { return (x > 0) - (x < 0); }
template <class T> // hash-1
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)); }

```

```

};
using P = Point<double>; // hash-1 = 96024b
// signed distance between point p and line (a, b)
template<class P> // hash-2
double lineDist(const P& a, const P& b, const P& p) {
    return (double)(b-a).cross(p-a)/(b-a).dist();
} // hash-2 = f6bf6b
// shortest distance between point p and line segment (s, e)
double segDist(P& s, P& e, P& p) { // hash-3
    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;
} // hash-3 = ae751a
// intersection between two segments: Returns either the
// intersection point or the line segment (s,e)
// if there is infinitely many. Empty vector if no intersection

template<class P> // hash-4
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 {s.begin(), s.end()};
} // hash-4 = f6be16
// Same as above but returns only true or false
template<class P> // hash-5
bool segmentIntersectionQ(P s1, P e1, P s2, P e2) {
    if (e1 == s1) {
        if (e2 == s2) return e1 == e2;
        swap(s1, s2); swap(e1, e2);
    }
    P v1 = e1-s1, v2 = e2-s2, d = s2-s1;
    auto a = v1.cross(v2), a1 = d.cross(v1), a2 = d.cross(v2);
    if (a == 0) { // parallel
        auto b1 = s1.dot(v1), c1 = e1.dot(v1),
            b2 = s2.dot(v1), c2 = e2.dot(v1);
        return !a1 && max(b1, min(b2, c2)) <= min(c1, max(b2, c2));
    }
    if (a < 0) { a = -a; a1 = -a1; a2 = -a2; }
    return (0 <= a1 && a1 <= a && 0 <= a2 && a2 <= a);
} // hash-5 = 1ff4ba
// {1, P} if there is a intersection, {-1, (0,0)} if
// there is infinitely, {0,(0,0)} otherwise.
template<class P> // hash-6
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};
} // hash-6 = a01f81
// Projects point P onto line ab. Refl=true to get reflection
// of point P across line ab instead.
template<class P> // hash-7
P lineProj(P a, P b, P p, bool refl=false) {
    P v = b - a;
    return p - v.perp()*(1+refl)*v.cross(p-a)/v.dist2();
} // hash-7 = b5562d
// 1/0/-1 <=> left/on line/right
template<class P> // hash-8
int sideOf(P s, P e, P p) { return sgn(s.cross(e, p)); }
template<class P>

```

```

int sideOf(const P& s, const P& e, const P& p, double eps) {
    auto a = (e-s).cross(p-s);
    double l = (e-s).dist()*eps;
    return (a > l) - (a < -l);
} // hash-8 = 3af81c
// True iff P lies on the line segment (s, e)
template<class P> bool onSegment(P s, P e, P p) { // hash-9
    return p.cross(s, e) == 0 && (s - p).dot(e - p) <= 0;
} // hash-9 = c597e8
// Linear transformation which takes line p0p1 to q0q1
P linearTransformation(const P& p0, const P& p1,
    const P& q0, const P& q1, const P& r) { // hash-10
    P dp = p1-p0, dq = q1-q0, num(dp.cross(dq), dp.dot(dq));
    return q0 + P((r-p0).cross(num), (r-p0).dot(num))/dp.dist2();
} // hash-10 = 45ea01
// Useful utilities for dealing with angles of rays from origin
template<class P> // hash-11
bool sameDir(P s, P t) {
    return s.cross(t) == 0 && s.dot(t) > 0;
}
template<class P> // checks 180 <= s..t < 360?
bool isReflex(P s, P t) {
    auto c = s.cross(t);
    return c ? (c < 0) : (s.dot(t) < 0);
}
// operator < (s,t) for angles in [base, base+2pi)
template<class P>
bool angleCmp(P base, P s, P t) {
    int r = isReflex(base, s) - isReflex(base, t);
    return r ? (r < 0) : (0 < s.cross(t));
}
// is x in [s,t] taken ccw? 1/0/-1 for in/border/out
template<class P>
int angleBetween(P s, P t, P x) {
    if (sameDir(x, s) || sameDir(x, t)) return 0;
    return angleCmp(s, x, t) ? 1 : -1; } // hash-11 = 6edd25
// hash-12
int half(P p) { return p.y != 0 ? sgn(p.y) : -sgn(p.x); }
bool angle_cmp(P a, P b) { int A = half(a), B = half(b);
    return A == B ? a.cross(b) > 0 : A < B; } // hash-12 = 9a6894
// out is the pair of points at which two circles intersect
// hash-13
bool circleInter(P a, P b, double r1, double r2, pair<P, P>* out) {
    if (a == b) { assert(r1 != r2); return false; }
    P vec = b - a;
    double d2 = vec.dist2(), sum = r1+r2, dif = r1-r2,
        p = (d2 + r1*r1 - r2*r2)/(d2*2), h2 = r1*r1 - p*p*d2;
    if (sum*sum < d2 || dif*dif > d2) return false;
    P mid = a + vec*p, per = vec.perp() * sqrt(fmax(0, h2) / d2);
    *out = {mid + per, mid - per};
    return true;
} // hash-13 = c64785
// Finds the external tangents of two circles, or internal if
// r2 is negated, .first and .second gives the tangency point
// at circle 1 and 2 respec.
template<class P> // hash-14
vector<pair<P, P>> tangents(P c1, double r1, P c2, double r2) {
    P d = c2 - c1;
    double dr = r1 - r2, d2 = d.dist2(), h2 = d2 - dr * dr;
    if (d2 == 0 || h2 < 0) return {};
    vector<pair<P, P>> out;
    for (double sign : {-1, 1}) {
        P v = (d * dr + d.perp() * sqrt(h2) * sign) / d2;
        out.push_back({c1 + v * r1, c2 + v * r2});
    }
    if (h2 == 0) out.pop_back();
    return out;
} // hash-14 = b0153d
// radius of the circle going through points A, B and C

```

```

double ccRadius(const P& A, const P& B, const P& C) { // hash-15
    return (B-A).dist()*(C-B).dist()*(A-C).dist() /
        abs((B-A).cross(C-A))/2;
}
// Center of the circle above
P ccCenter(const P& A, const P& B, const P& C) {
    P b = C-A, c = B-A;
    return A + (b*c.dist2()-c*b.dist2()).perp()/b.cross(c)/2;
} // hash-15 = c98102
// min enclosing circle ~ O(N)
pair<P, double> mec(vector<P> ps) { // hash-16
    shuffle(ps.begin(), ps.end(), mt19937(time(0)));
    P o = ps[0];
    double r = 0, EPS = 1 + 1e-8;
    for(int i = 0; i < ps.size(); ++i)
        if ((o - ps[i]).dist() > r * EPS) {
            o = ps[i], r = 0;
            for(int j = 0; j < i; ++j) if ((o-ps[j]).dist()>r*EPS){
                o = (ps[i] + ps[j]) / 2;
                r = (o - ps[i]).dist();
                for(int k = 0; k < j; ++k)
                    if ((o - ps[k]).dist() > r * EPS) {
                        o = ccCenter(ps[i], ps[j], ps[k]);
                        r = (o - ps[i]).dist();
                    }
            }
        }
    return {o, r};
} // hash-16 = 8ab87f
// Intersection between circle and line ab, returns 0/1/2
// intersections
template<class P> // hash-17
vector<P> circleLine(P c, double r, P a, P b) {
    double h2 = r*r - a.cross(b,c)*a.cross(b,c)/(b-a).dist2();
    if (h2 < 0) return {};
    P p = lineProj(a, b, c), h = (b-a).unit() * sqrt(h2);
    if (h2 == 0) return {p};
    return {p - h, p + h};
} // hash-17 = debf86
template<class P> // intersection area of two circles // hash-18
double circleCircleArea(P c, double cr, P d, double dr) {
    if (cr < dr) swap(c, d), swap(cr, dr);
    auto A = [&](double r, double h) {
        return r*r*acos(h/r)-h*sqrt(r*r-h*h);
    };
    auto l = (c - d).dist(), a = (l*l + cr*cr - dr*dr)/(2*l);
    if (l - cr - dr >= 0) return 0; // far away
    if (l - cr + dr <= 0) return M_PI*dr*dr;
    if (l - cr >= 0) return A(cr, a) + A(dr, l-a);
    else return A(cr, a) + M_PI*dr*dr - A(dr, a-l);
} // hash-18 = 8bf2b6
// area of intersection between a circle and a ccw polygon
#define arg(p, q) atan2(p.cross(q), p.dot(q)) // (conc or conv)
double circlePoly(P c, double r, vector<P> ps) { // hash-19
    auto tri = [&](P p, P q) {
        auto r2 = r * r / 2;
        P d = q - p;
        auto a = d.dot(p)/d.dist2(), b = (p.dist2()-r*r)/d.dist2();
        auto det = a * a - b;
        if (det <= 0) return arg(p, q) * r2;
        auto s = max(0., -a-sqrt(det)), t = min(1., -a+sqrt(det));
        if (t < 0 || 1 <= s) return arg(p, q) * r2;
        P u = p + d * s, v = p + d * t;
        return arg(p,u) * r2 + u.cross(v)/2 + arg(v,q) * r2;
    };
    auto sum = 0.0;
    for (int i = 0; i < ps.size(); ++i)
        sum += tri(ps[i] - c, ps[(i + 1) % ps.size()] - c);
    return sum;
}

```

```

} // hash-19 = c6ad22
// True if P lies within the polygon. If strict=true returns
// false for points on the boundary.
template<class P> // hash-20
bool inPolygon(vector<P> &p, P a, bool strict = true) {
    int cnt = 0, n = p.size();
    for(int i = 0; i < n; ++i) {
        P q = p[(i + 1) % n];
        if (onSegment(p[i], q, a)) return !strict; // change to
        // -1 if u need to detect points in the boundary
        //or: if (segDist(p[i], q, a) <= eps) return !strict;
        cnt ^= ((a.y<p[i].y) - (a.y<q.y)) * a.cross(p[i], q) > 0;
    }
    return cnt;
} // hash-20 = f9442d
template<class T> // hash-21
T polygonArea2(vector<Point<T>> &v) {
    T a = v.back().cross(v[0]);
    for(int i = 0; i < v.size()-1; ++i)
        a += v[i].cross(v[i+1]);
    return a;
} // hash-21 = 96da89
template<class T>
Point<T> polygonCentroid(vector<Point<T>> &v){ // hash-22
    Point<T> cent(0,0); T area = 0;
    for(int i = 0; i < v.size(); ++i) {
        int j = (i+1) % (v.size()); T a = cross(v[i], v[j]);
        cent += a * (v[i] + v[j]);
        area += a;
    }
    return cent/area/(T)3;
} // not tested!!! // hash-22 = 218bc9
// Center of mass of a polygon
P polygonCenter(const vector<P>& v) { // hash-23
    P res(0, 0); double A = 0;
    for (int i = 0, j = v.size() - 1; i < v.size(); j = ++i) {
        res = res + (v[i] + v[j]) * v[j].cross(v[i]);
        A += v[j].cross(v[i]);
    }
    return res / A / 3;
} // hash-23 = 26a00f
// Returns vertices of the polygon to the left of the cut (s,e)
vector<P> polygonCut(const vector<P>& poly, P s, P e){
    vector<P> res; // hash-24
    for(int i = 0; i < poly.size(); ++i) {
        P cur = poly[i], prev = i ? poly[i-1] : poly.back();
        bool side = s.cross(e, cur) < 0;
        if (side != (s.cross(e, prev) < 0))
            res.push_back(lineInter(s, e, cur, prev).second);
        if (side) res.push_back(cur);
    }
    return res;
} // hash-24 = 120d5b
// Union of several polygons. O(N^2).
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;
    for(int i = 0; i < poly.size(); ++i)
        for(int v = 0; v < poly[i].size(); ++v) {
            P A = poly[i][v], B = poly[i][(v + 1) % poly[i].size()];
            vector<pair<double, int>> segs = {{0, 0}, {1, 0}};
            for(int j = 0; j < poly.size(); ++j) if (i != j) {
                for(int u = 0; u < poly[j].size(); ++u) {
                    P C = poly[j][u], D = poly[j][(u+1) % poly[j].size()];
                    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(segs.begin(), segs.end());
                for(auto& s : segs) s.first = min(max(s.first, 0.0), 1.0);
                double sum = 0; int cnt = segs[0].second;
                for(int j = 1; j < segs.size(); ++j) {
                    if (!cnt) sum += segs[j].first - segs[j - 1].first;
                    cnt += segs[j].second;
                }
                ret += A.cross(B) * sum;
            } return ret / 2;
        }
    } // no collinear points allowed
    vector<P> convexHull(vector<P> pts) { // hash-25
        if (pts.size() <= 1) return pts;
        sort(pts.begin(), pts.end());
        vector<P> h(pts.size()+1);
        int s = 0, t = 0;
        for (int it = 2; it--; s = --t, reverse(pts.begin(), pts.end()))
            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])};
    } // hash-25 = 3612d7
    // Two points with the max distance on convex hull (ccw, no
    array<P, 2> hullDiameter(vector<P> S) { // duplicate/collinear
        int n = S.size(), j = n < 2 ? 0 : 1; // hash-26
        pair<ll, array<P, 2>> res({0, {S[0], S[0]}});
        for(int i = 0; i < j; ++i)
            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.second;
    } // hash-26 = ebd594
    // Checks whether P lies inside a convex hull.
    bool inHull(const vector<P> &l, P p, bool strict = true) {
        int a = 1, b = l.size() - 1, r = !strict; // hash-27
        if (l.size() < 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;
            (sideOf(l[0], l[c], p) > 0 ? b : a) = c;
        }
        return sgn(l[a].cross(l[b], p)) < r;
    } // hash-27 = d6ec66
    // O(N^4M) // hash-28
    vector<P> minkowski_sum(vector<P> A, vector<P> B) {
        if (int(A.size()) > int(B.size())) swap(A, B);
        if (A.empty()) return {};
        if (int(A.size()) == 1) {
            for (auto& b : B) b = b + A.front();
            return B;
        }
        rotate(A.begin(), min_element(A.begin(), A.end()), A.end());
        rotate(B.begin(), min_element(B.begin(), B.end()), B.end());
        A.push_back(A[0]); A.push_back(A[1]);
        B.push_back(B[0]); B.push_back(B[1]);
        const int N = int(A.size()), M = int(B.size());

```

```

        vector<P> ans; ans.reserve(N+M);
        for (int i = 0, j = 0; i+2 < N || j+2 < M; ) {
            ans.push_back(A[i] + B[j]);
            auto sgn = (A[i+1] - A[i]).cross(B[j+1] - B[j]);
            i += (sgn >= 0); j += (sgn <= 0);
        }
        return ans;
    } // hash-28 = 71a25a
    // Intersection between a line and a convex polygon (given ccw)
    .
    #define cmp(i,j) sgn(dir.perp().cross(poly[(i)%n]-poly[(j)%n]))
    #define extr(i) cmp(i + 1, i) >= 0 && cmp(i, i - 1 + n) < 0
    int extrVertex(vector<P>& poly, P dir) { // hash-29
        int n = poly.size(), left = 0, right = n;
        if (extr(0)) return 0;
        while (left + 1 < right) {
            int m = (left + right) / 2;
            if (extr(m)) return m;
            int ls = cmp(left + 1, left), ms = cmp(m + 1, m);
            (ls<ms || (ls==ms && ls==cmp(left, m)) ? right : left)=m;
        }
        return left;
    } // hash-29 = 99da02
    #define cmpL(i) sgn(a.cross(poly[i], b))
    array<int, 2> lineHull(P a, P b, vector<P>& poly) { // hash-30
        int endA = extrVertex(poly, (a - b).perp());
        int endB = extrVertex(poly, (b - a).perp());
        if (cmpL(endA) < 0 || cmpL(endB) > 0)
            return {-1, -1};
        array<int, 2> res;
        for(int i = 0; i < 2; ++i) {
            int left = endB, right = endA, n = poly.size();
            while ((left + 1) % n != right) {
                int m = ((left + right + (left < right ? 0 : n))/2)%n;
                (cmpL(m) == cmpL(endB) ? left : right) = m;
            }
            res[i] = (left + !cmpL(right)) % 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) % poly.size()) {
                case 0: return {res[0], res[0]};
                case 2: return {res[1], res[1]};
            }
        return res;
    } // hash-30 = 95f4b6
    // Halfplane intersection area
    #define eps 1e-8
    using P = Point<double>;
    struct line_t { // hash-31
        P P1, P2;
        // Right hand side of the ray P1 -> P2
        explicit line_t(P a = P(), P b = P()) : P1(a), P2(b) {};
        P intpo(line_t y) {
            pair<int, P> r = lineInter(P1, P2, y.P1, y.P2);
            assert(r.first == 1);
            return r.second;
        }
        P dir() { return P2 - P1; }
        bool contains(P x) {
            return (P2 - P1).cross(x - P1) < eps;
        }
        bool out(P x) { return !contains(x); }
    }; // hash-31 = f8b605
    bool mycmp(P a, P b) { // hash-32
        // return atan2(a.y, a.x) < atan2(b.y, b.x);
        if (a.x * b.x < 0) return a.x < 0;
        if (abs(a.x) < eps) {

```



```

    if (abs(b.x) < eps) return a.y > 0 && b.y < 0;
    if (b.x < 0) return a.y > 0;
    if (b.x > 0) return true;
}
if (abs(b.x) < eps) {
    if (a.x < 0) return b.y < 0;
    if (a.x > 0) return false;
}
return a.cross(b) > 0;
} // hash-32 = 71c36e
bool cmp(line_t a, line_t b){return mycmp(a.dir(), b.dir());}
double intersection_area(vector<line_t> b) { // hash-33
    sort(b.begin(), b.end(), cmp);
    int n = int(b.size());
    int q = 1, h = 0, i;
    vector<line_t> c(b.size() + 10);
    for (i = 0; i < n; i++) {
        while (q < h && b[i].out(c[h].intpo(c[h - 1]))) h--;
        while (q < h && b[i].out(c[q].intpo(c[q + 1]))) q++;
        c[++h] = b[i];
        if (q < h && abs(c[h].dir().cross(c[h - 1].dir())) < eps) {
            if (c[h].dir().dot(c[h - 1].dir()) > 0) {
                h--;
                if (b[i].out(c[h].P1)) c[h] = b[i];
            } else {
                // The area is either 0 or infinite. If you have a
                // bounding box, then the area is definitely 0.
                return 0;
            }
        }
    }
    while (q < h-1 && c[q].out(c[h].intpo(c[h - 1]))) h--;
    while (q < h-1 && c[h].out(c[q].intpo(c[q + 1]))) q++;
    // Intersection is empty. This is sometimes different from
    // the case when the intersection area is 0.
    if (h - q <= 1) return 0;
    c[h + 1] = c[q];
    vector<P> s;
    for (i = q; i <= h; i++) s.push_back(c[i].intpo(c[i+1]));
    s.push_back(s[0]);
    double ans = 0; int m = int(s.size());
    for (i = 0; i+1 < m; i++) ans += s[i].cross(s[i+1]);
    return ans/2;
} // hash-33 = e9dda6
// Closes pair of points. O(N log N)
pair<P, P> closest(vector<P> v) { // hash-34
    assert(v.size() > 1);
    set<P> S;
    sort(v.begin(), v.end(), [](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;
} // hash-34 = 2b4d6a
// Rectangle union area
struct seg_node{ // hash-35
    int val, cnt, lz;
    seg_node(int n = INF, int c = 0): val(n), cnt(c), lz(0) {}
    void push(seg_node& l, seg_node& r){
        if(lz){
            l.add(lz); r.add(lz); lz = 0;
        }
    }

```

```

    }
    void merge(const seg_node& l, const seg_node& r){
        if(l.val < r.val) val = l.val, cnt = l.cnt;
        else if(l.val > r.val) val = r.val, cnt = r.cnt;
        else val = l.val, cnt = l.cnt + r.cnt;
    }
    void add(int n){
        val += n; lz += n;
    }
    int get_sum(){ return (val ? 0 : cnt); }
};
// x1 y1 x2 y2
ll solve(const vector<array<int, 4>>&v){
    vector<int>ys;
    for(auto& [a, b, c, d] : v){
        ys.push_back(b); ys.push_back(d);
    }
    sort(ys.begin(), ys.end());
    ys.erase(unique(ys.begin(), ys.end()), ys.end());
    vector<array<int, 4>>e;
    for(auto [a, b, c, d] : v){
        b = int(lower_bound(ys.begin(), ys.end(), b) - ys.begin());
        d = int(lower_bound(ys.begin(), ys.end(), d) - ys.begin());
        e.push_back({a, b, d, 1}); e.push_back({c, b, d, -1});
    }
    sort(e.begin(), e.end()); int m = (int)ys.size();
    segtree_range<seg_node>seg(m-1);
    for(int i=0;i<m-1;i++)seg.at(i)=seg_node(0, ys[i+1]-ys[i]);
    seg.build();
    int last = INT_MIN, total = ys[m-1] - ys[0]; ll ans = 0;
    for(auto [x, y1, y2, c] : e){
        ans+=(ll)(total-seg.query(0,m-1).get_sum()*(x-last);
        last = x; seg.update(y1, y2, &seg_node::add, c);
    }
    return ans;
} // hash-35 = 27be94
// Delaunay Triangulation. O(N^2).Each circumcircle contains
// none of the input points. If any three points are colinear
// or any four are on the same circle, behavior is undefined.
template<class P, class F> // hash-36
void delaunay(vector<P>& ps, F trifun) {
    if (ps.size() == 3){ int d = (ps[0].cross(ps[1], ps[2]) < 0);
        trifun(0,1+d,2-d); }
    vector<P3> p3;
    for(auto p : ps) p3.emplace_back(p.x, p.y, p.dist2());
    if(ps.size()>3) for(auto t: hull13d(p3)) if((p3[t.b]-p3[t.a]).
        cross(p3[t.c]-p3[t.a]).dot(P3(0,0,1)) < 0)
        trifun(t.a, t.c, t.b);
} // hash-36 = 0f6a1a
// GEOMETRY 3D
template<class T> struct Point3D { // hash-37
    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);
    } // hash-37 = f914db
    T dist2() const { return x*x + y*y + z*z; } // hash-38
    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;
    }
}; // hash-38 = c9d029
// polyhedron formula. Faces should point outwards.
template<class V, class L>
double signed_poly_volume(const V &p, const L &trilist) {
    double v = 0;
    for(auto &i : trilist) v += p[i.a].cross(p[i.b]).dot(p[i.c]);
    return v / 6;
}
// ConvexHull3D
typedef Point3D<double> P3;
struct PR { // hash-39
    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;
}; // hash-39 = cf7c9e
struct F { P3 q; int a, b, c; };
vector<F> hull13d(const vector<P3>& A) { // hash-40
    assert(A.size() >= 4);
    vector<vector<PR>> E(A.size(),vector<PR>(A.size(),{-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 * -1;
        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);
    };
    for(int i=0;i<4;i++)for(int j=i+1;j<4;j++)for(int k=j+1;k<4;k
        ++){
        mf(i, j, k, 6 - i - j - k); // hash-40 = 8a3c0b
    }
    for(int i=4; i<A.size(); ++i) { // hash-41
        for(int j=0; j<FS.size(); ++j) {
            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 = FS.size();
        for(int j=0; j<nw; j++) {
            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(auto &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;
}; // hash-41 = 52653c

```


Strings (9)

kmp.h

Description: failure[x] computes the length of the longest prefix of s that ends at x, other than s[0...x] itself (abacaba -> -1,0,0,1,0,1,2,3). Can be used to find all occurrences of a pattern in a text.

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

```
vector<int> prefix_function(const string& S) {
    vector<int> fail = {-1}; fail.reserve(S.size());
    for (int i = 0; i < int(S.size()); ++i) {
        int j = fail.back();
        while (j != -1 && S[i] != S[j]) j = fail[j];
        fail.push_back(j+1);
    }
    return fail;
}
```

z-algorithm.h

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

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

```
vector<int> Z(const string& S) { // hash-1
    vector<int> z(S.size()); int l = -1, r = -1;
    for(int i = 1; i < int(S.size()); ++i) {
        z[i] = i >= r ? 0 : min(r - i, z[i - l]);
        while (i + z[i] < int(S.size()) && S[i + z[i]] == S[z[i]])
            z[i]++;
        if (i + z[i] > r) l = i, r = i + z[i];
    } return z;
} // hash-1 = 049391
vector<int> get_prefix(string a, string b) {
    string str = a + '@' + b; vector<int> k = Z(str);
    return vector<int>(k.begin() + int(a.size())+1, k.end());
}
```

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: $\mathcal{O}(N)$

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

min-rotation.h

Description: Finds the lexicographically smallest rotation of a string.
Usage: rotate(v.begin(), v.begin()+min.rotation(v), v.end());
Time: $\mathcal{O}(N)$

```
int min_rotation(string s) {
    int a=0, N=s.size(); s += s;
    for(int b = 0; b < N; ++b) for(int i = 0; i < N; ++i) {
        if (a+i == b || s[a+i] < s[b+i]) {b += max(0, i-1); break;}
        if (s[a+i] > s[b+i]) { a = b; break; }
    } return a;
}
```

rolling-hash.h

Description: Self-explanatory methods for string hashing.

```
db642a, 36 lines
// Arithmetic mod 2^64-1. 2x slower than mod 2^64
typedef uint64_t ull;
struct H { // hash-1
    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(); }
}; // hash-1 = bf6be7
static const H C = (1l)1e11+3; // (order ~ 3e9; random also ok)
struct HashInterval { // hash-2
    vector<H> ha, pw;
    HashInterval(string& str) : ha(int(str.size()+1), pw(ha) {
        pw[0] = 1;
        for(int i = 0; i < int(str.size()); ++i)
            ha[i+1] = ha[i] * C + str[i],
            pw[i+1] = pw[i] * C;
    }
    H operator()(int a, int b) { // hash [a, b)
        return ha[b] - ha[a] * pw[b - a];
    }
}; // hash-2 = 2ddba2
vector<H> get_hashes(string& str, int length) { // hash-3
    if (int(str.size()) < length) return {};
    H h = 0, pw = 1;
    for(int i = 0; i < length; ++i)
        h = h * C + str[i], pw = pw * C;
    vector<H> ret = {h};
    for(int i = length; i < int(str.size()); ++i)
        ret.push_back(h = h * C + str[i] - pw * str[i-length]);
    return ret;
} // hash-3 = d21d2a
H hash_all(string& s){H h{}; for(char c:s) h=h*C+c;return h;}
```

tandem-repeats.h

Description: Find all (i,p) such that s.substr(i,p) == s.substr(i+p,p). No two intervals with the same period intersect or touch.

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

```
"suffix-array.h"
355eb2, 13 lines
vector<array<int, 3>> run(int N, string S) {
    suffix_array_t A(S.begin(), S.end());
    reverse(S.begin(), S.end());
    suffix_array_t B(S.begin(), S.end());
    vector<array<int, 3>> ans;
    for (int p = 1; 2*p <= N; ++p)
        for (int i = 0, prv = -1; i+p <= N; i += p) {
            int l = i-B.get_lcp(N-i-p,N-i), r = i-p+A.get_lcp(i,i+p);
            if (l > r || l == prv) continue;
            ans.push_back({prv = l, r, p});
        }
    return ans;
}
```

aho-corasick.h

```
61f5e3, 36 lines
const int sigma = 26; // hash-1
array<int, sigma> init;
for (int i = 0; i < sigma; i++) init[i] = -1;
vector<array<int, sigma>> trie(1, init);
vector<int> out(1, -1), parent(n, -1), ids(n);
for (int i = 0; i < n; i++) {
    int cur = 0;
    for (char ch : s[i]) {
```

```
int c = ch - 'a';
if (trie[cur][c] == -1) {
    trie[cur][c] = (int)trie.size();
    trie.push_back(init); out.push_back(-1);
}
cur = trie[cur][c];
}
if (out[cur] == -1) out[cur] = i;
ids[i] = out[cur];
} // hash-1 = c8c436
vector<int> bfs,f(trie.size()); bfs.reserve(trie.size());
for (int c = 0; c < sigma; c++) // hash-2
    if (trie[0][c] == -1) trie[0][c] = 0;
    else bfs.push_back(trie[0][c]);
for (int z = 0; z < (int)bfs.size() ; z++) {
    int cur = bfs[z];
    for (int c = 0; c < sigma; c++) {
        if (trie[cur][c] == -1)
            trie[cur][c] = trie[f[cur]][c];
        else {
            int nxt = trie[cur][c];
            int fail = trie[f[cur]][c];
            if (out[nxt] == -1) out[nxt] = out[fail];
            else parent[out[nxt]] = out[fail];
            f[nxt] = fail; bfs.push_back(nxt);
        }
    }
} // hash-2 = 2188a7
```

suffix-array.h

Description: Builds suffix array for a string, first element is the size of the string. The lcp function calculates longest common prefixes for neighbour-strings in suffix array. The returned vector is of size n+1.

Time: $\mathcal{O}(N \log N)$ where N is the length of the string for creation of the SA. $\mathcal{O}(N)$ for longest common prefixes.

```
"../data-structures/rmq.h"
a306fe, 46 lines
struct suffix_array_t { // hash-1
    int N, H; vector<int> sa, invsa, lcp;
    rmq_t<pair<int, int>> rmq;
    bool cmp(int a, int b) { return invsa[a+H] < invsa[b+H]; }
    void ternary_sort(int a, int b) {
        if (a == b) return;
        int md = sa[a+rng()] % (b-a), lo = a, hi = b;
        for (int i = a; i < b; ++i) if (cmp(sa[i], md))
            swap(sa[i], sa[lo++]);
        for (int i = b-1; i >= lo; --i) if (cmp(md, sa[i]))
            swap(sa[i], sa[--hi]);
        ternary_sort(a, lo);
        for (int i = lo; i < hi; ++i) invsa[sa[i]] = hi-1;
        if (hi-lo == 1) sa[lo] = -1;
        ternary_sort(hi, b);
    }
    suffix_array_t() {} // hash-1 = 310a12
    template<typename I> // hash-2
    suffix_array_t(I begin, I end): N(int(end-begin)+1), sa(N) {
        vector<int> v(begin, end); v.push_back(INT_MIN);
        invsa = v; iota(sa.begin(), sa.end(), 0);
        H = 0; ternary_sort(0, N);
        for(H = 1; H <= N; H *= 2) for(int j=0, i=j; i!=N; i=j)
            if (sa[i] < 0) {
                while (j < N && sa[j] < 0) j += -sa[j];
                sa[i] = -(j - i);
            } else {j = invsa[sa[i]] + 1; ternary_sort(i, j);}
        for (int i = 0; i < N; ++i) sa[invsa[i]] = i;
        lcp.resize(N-1); int K = 0;
        for (int i = 0; i < N-1; ++i) {
            if(invsa[i] > 0) while(v[i+K] == v[sa[invsa[i]-1]+K])++K;
            lcp[invsa[i]-1] = K; K = max(K - 1, 0);
        }
    }
}
```

```
vector<pair<int, int>> lcp_index(N-1);
for (int i = 0; i < N-1; ++i) lcp_index[i] = {lcp[i], 1+i};
rmq = rmq_t<pair<int, int>>(std::move(lcp_index));
} // hash-2 = ef483d
auto rmq_query(int a, int b) const {return rmq.query(a,b);}
auto get_split(int a, int b) const {return rmq.query(a,b-1);}
int get_lcp(int a, int b) const { // hash-3
    if (a == b) return N - a;
    a = invsa[a], b = invsa[b];
    if (a > b) swap(a, b);
    return rmq_query(a, b).first;
} // hash-3 = 6877f0
};
```

Suffix Automaton

Number of occurrences

For each state not created by cloning, initialize $cnt(v) = 1$. Then, just do a dfs to calculate $cnt(v)$, with $cnt(link(v)) + = cnt(v)$

Number of different substrings

Is the number of paths in the automaton starting at the root. $d(v) = 1 + \sum_{v \rightarrow w} d(w)$

Total length of different substrings

Is the sum of children answers and paths starting at each children. $ans(v) = \sum_{v \rightarrow w} d(w) + ans(w)$

Lexicographically K -th substring

Is the K -th lexicographically path, so you can search using the number of paths from each state

Smallest cyclic shift

Construct for string $S + S$. Greedily search the minimal character.

First occurence position

When we create a new state cur do $first(pos) = len(cur) - 1$. When we clone q as $clone$ do $first(clone) = first(q)$. Answer is $first(v) - size(P) + 1$, where v is the state of string P

All occurence positions

From $first(v)$ do a dfs using suffix link, from $link(u)$ go to u .

suffix-automaton.h

```
Description: Suffix automaton
defb60, 33 lines

template<int offset = 'a'> struct array_state { // hash-1
    array<int, 26> as;
    array_state() { fill(begin(as), end(as), ~0); }
    int& operator[](char c) { return as[c - offset]; }
    int count(char c) { return (~as[c - offset] ? 1 : 0); }
}; // hash-1 = e60fd5
template<typename C, typename state = map<C, int>> struct
    suffix_automaton { // hash-2
    struct node_t {
        int len, link; int64_t cnt; state next;
    };
    int N, cur; vector<node_t> nodes;
    suffix_automaton() : N(1), cur(0), nodes{node_t{0, -1, 0, {}}}{}
```

```
node_t& operator[](int v) { return nodes[v]; };
void append(C c) {
    int v = cur; cur = N++;
    nodes.push_back(node_t{nodes[v].len + 1, 0, 1, {}});
    for (; ~v && !nodes[v].next.count(c); v = nodes[v].link)
        nodes[v].next[c] = cur;
    if (~v) {
        const int u = nodes[v].next[c];
        if (nodes[v].len + 1 == nodes[u].len) {
            nodes[cur].link = u;
        } else {
            const int clone = N++;
            nodes.push_back(nodes[u]);
            nodes[clone].len = nodes[v].len + 1;
            nodes[u].link = nodes[cur].link = clone;
            for (; ~v && nodes[v].next[c] == u; v = nodes[v].link)
                nodes[v].next[c] = clone;
        }
    }
}; // hash-2 = f4569b
```

Various (10)

Intervals

interval-container.h

Description: Add and remove intervals from a set of disjoint intervals. Will merge the added interval with any overlapping intervals in the set when adding. Intervals are [inclusive, exclusive). Time: $\mathcal{O}(\log N)$

```
edge47, 20 lines

set<pii>::iterator addInterval(set<pii> &is, int L, int R) {
    if (L == R) return is.end(); // hash-1
    auto it = is.lower_bound({L, R}), before = it;
    while (it != is.end() && it->first <= R) {
        R = max(R, it->second);
        before = it = is.erase(it);
    }
    if (it != is.begin() && (--it)->second >= L) {
        L = min(L, it->first); R = max(R, it->second);
        is.erase(it);
    }
    return is.insert(before, {L, R});
} // hash-1 = 52519d
void removeInterval(set<pii> &is, int L, int R) { // hash-2
    if (L == R) return;
    auto it = addInterval(is, L, R);
    auto r2 = it->second;
    if (it->first == L) is.erase(it);
    else (int&)it->second = L;
    if (R != r2) is.emplace(R, r2);
} // hash-2 = 0594c1
```

interval-cover.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: $\mathcal{O}(N \log N)$

```
template<class T>
vector<int> cover(pair<T, T> G, vector<pair<T, T>> I) {
    vector<int> S(I.size()), R;
    iota(S.begin(), S.end(), 0);
    sort(S.begin(), S.end(), [&](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 = {cur, -1};
while (at < I.size() && I[S[at]].first <= cur) {
    mx = max(mx, {I[S[at]].second, S[at]});
    at++;
}
if (mx.second == -1) return {};
cur = mx.first; R.push_back(mx.second);
} return R;
}
```

constant-intervals.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: $\mathcal{O}(k \log \frac{n}{k})$

```
753a4c, 17 lines

template<class F, class G, class T> // hash-1
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);
    }
} // hash-1 = 69b73b
template<class F, class G> // hash-2
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);
} // hash-2 = 856088
```

Misc. algorithms

ternary-search.h

Description: Find the smallest i in $[a, b]$ that maximizes $f(i)$, assuming that $f(a) < \dots < f(i) \geq \dots \geq f(b)$. To reverse which of the sides allows non-strict inequalities, change the < marked with (A) to <=, and reverse the loop at (B). To minimize f , change it to >, also at (B). If you are dealing with real numbers, you'll need to pick $m_1 = (2a + b)/3.0$ and $m_2 = (a + 2b)/3.0$. Consider setting a constant number of iterations for the search, usually [200, 300] iterations are sufficient for problems with error limit as 10^{-6} . Usage: int ind = ternSearch(0,n-1,[&](int i){return a[i];}); Time: $\mathcal{O}(\log(b - a))$

```
35ef73, 11 lines

template<class F> int ternSearch(int a, int b, F f) {
    assert(a <= b);
    while (b - a >= 5) {
        int mid = (a + b) / 2;
        if (f(mid) < f(mid+1)) a = mid; // (A)
        else b = mid+1;
    }
    for(int i = a+1; i <= b; ++i)
        if (f(a) < f(i)) a = i; // (B)
    return a;
}
```

count-triangles.h

Description: Counts $x, y \geq 0$ such that $Ax + By \leq C$. fbb812, 6 lines

```
11 count_triangle(11 B, 11 C) {
    if (C < 0) return 0;
    if (A > B) swap(A, B);
    11 p = C / B, k = B / A, d = (C - p * B) / A;
    return count_triangle(B - k * A, A, C - A * (k * p + d + 1))
        + (p + 1) * (d + 1) + k * p * (p + 1) / 2;
}
```

```
}

floyd-cycle.h
Description: Detect loop in a list. Consider using mod template to avoid overflow.
Time:  $\mathcal{O}(n)$ 
```

```
template<class F> pair<int,int> find(int x0, F f) {
    int t = f(x0), h = f(t), mu = 0, lam = 1;
    while (t != h) t = f(t), h = f(f(h));
    h = x0;
    while (t != h) t = f(t), h = f(h), ++mu;
    h = f(t);
    while (t != h) h = f(h), ++lam;
    return {mu, lam};
}
```

10.3 Dynamic programming

```
divide-and-conquer-dp.h
Description: Given  $a[i] = \min_{l \circ (i) \leq k < h(i)}(f(i, k))$  where the (minimal) optimal  $k$  increases with  $i$ , computes  $a[i]$  for  $i = L..R - 1$ .
Time:  $\mathcal{O}((N + (hi - lo)) \log N)$ 
```

```
struct DP { // hash-1
    vector<int>a, freq;
    vector<ll>old, cur;
    ll cnt; int lcur, rcur;
    DP(const vector<int>&a, int n): a(_a), freq(n), old(n+1,
        linf), cur(n+1, linf), cnt(0), lcur(0), rcur(0){}
    int lo(int ind) { return 0; }
    int hi(int ind) { return ind; }
    void add(int k, int c){ cnt += freq[a[k]]++; }
    void del(int k, int c){ cnt -= --freq[a[k]]; }
    ll C(int l, int r){
        while(lcur > l) add(--lcur, 0);
        while(rcur < r) add(rcur++, 1);
        while(lcur < l) del(lcur++, 0);
        while(rcur > r) del(--rcur, 1);
        return cnt;
    } // hash-1 = 7cabd1
    ll f(int ind, int k) { return old[k] + C(k, ind); } // hash-2
    void store(int ind, int k, ll v) { cur[ind] = 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);
        for(int k = max(LO, lo(mid)); k <= min(HI, hi(mid)); ++k)
            best = min(best, make_pair(f(mid, k), k));
        store(mid, best.second, best.first);
        rec(L, mid, LO, best.second);
        rec(mid+1, R, best.second, HI);
    }
}; // hash-2 = 52b5d3
```

```
knuth-dp.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:  $\mathcal{O}(N^2)$ 
```

```
digit-dp.h
Description: Compute how many # between 1 and  $N$  have  $K$  distinct dig-
its in the base  $L$  without leading zeros;
```

```
Usage: auto hex.to_dec = [&](char c) -> int {
    return ('A' <= c && c <= 'F' ? (10 + c - 'A') : (c - '0'));
};
digit.dp<modnum<int(1e9) + 7>, hex.to_dec>(N, K);
Time:  $\mathcal{O}(NK)$ 
```

```
template<typename T, class F> T digit_dp(const string& S, int K
    , F& L) {
    const int base = 16, len = int(S.size());
    vector<bool> w(base);
    vector<vector<T>> dp(len + 1, vector<T>(base + 2));
    int cnt = 0;
    for (int d = 0; d < len; ++d) {
        // adding new digit to numbers with prefix < s
        for (int x = 0; x <= base; ++x) {
            dp[d + 1][x] += dp[d][x] * x;
            dp[d + 1][x + 1] += dp[d][x] * (base - x);
        }
        // adding strings whith prefix only 0's and last digit != 0
        if (d) dp[d + 1][1] += (base - 1);
        // adding prefix equal to s and last digit < s, first digit
        // cannot be 0
        for (int x = 0; x < L(S[d]); ++x) {
            if (d == 0 && x == 0) continue;
            if (w[x]) dp[d + 1][cnt] += 1;
            else dp[d + 1][cnt + 1] += 1;
        }
        // marking if the last digit appears in the prefix of s
        if (w[L(S[d])] == false)
            w[L(S[d])] = true, cnt++;
    }
    // adding string k
    dp[len][cnt] += 1; return dp[len][K];
}
```

```
knapsack-bounded-costs.h
Description: You are given  $N$  types of items, you have  $cnt[i]$  items of  $i$ -th
type, and each item of  $i$ -th type  $weight[i]$  and  $cost[i]$ . What is the maximal
cost you can get by picking some items weighing exactly  $X$  in total?
Time:  $\mathcal{O}(N \cdot W)$ 
```

```
"../data-structures/monotonic-queue.h"
auto solve(vi weight, vi cost, vi cnt, int X) {
    vector<int> dp(X+1, 0); int N = int(weight.size());
    vector<max_monotonic_queue<int>> M(X+1);
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < min(X+1, weight[i]); ++j)
            M[j] = max_monotonic_queue<int>();
        for (int j = 0; j <= X; ++j) {
            auto& que = M[j % weight[i]];
            if (que.size() > cnt[i]) que.pop();
            que.add(cost[i]);
            que.push(dp[j]);
            dp[j] = que.get_val();
        }
    } return dp[X];
}
```

```
two-max-equal-sum.h
Description: Two maximum equal sum disjoint subsets,  $s[i] = 0$  if  $v[i]$  wasn't
selected,  $s[i] = 1$  if  $v[i]$  is in the first subset and  $s[i] = 2$  if  $v[i]$  is in the second
subset
Time:  $\mathcal{O}(n * S)$ 
```

```
auto twoMaxEqualSumDS(const vector<int> &v){
    int sum=accumulate(v.begin(), v.end(), 0), n=int(v.size());
    vector<int> old(2*sum + 1, INT_MIN/2), dp(2*sum + 1), s(n);
    vector<vector<int>> rec(n, vector<int>(2*sum + 1));
    int i; old[sum] = 0;
    for(i = 0; i < n; ++i, swap(old, dp))
```

```
    for(int a, b, d = v[i]; d <= 2*sum - v[i]; d++){
        dp[d] = max(old[d], a = old[d - v[i]] + v[i]);
        dp[d] = max(dp[d], b = old[d + v[i]]);
        rec[i][d] = dp[d] == a ? 1 : dp[d] == b ? 2 : 0;
    }
    for(int j = i-1, d = sum; j >= 0; --j)
        d+=(s[j] = rec[j][d]) ? s[j] == 2 ? v[j] : - v[j] : 0;
    return make_pair(old[sum], s);
}
```

10.4 Optimization tricks

- 10.4.1 Bit hacks
- $c = x \& -x$, $r = x + c$; $((r \wedge x) >> 2) / c$ | r is the next number after x with the same number of bits set.
 - $\text{rep}(b, 0, K) \text{ rep}(i, 0, (1 \ll K))$ if $(i \& 1 \ll b)$ $D[i] += D[i \wedge (1 \ll b)]$; computes all sums of subsets.

10.5 Bit Twiddling Hack

```
hacks.h
// Iterate on non-empty submasks of a bitmask.
for (int s = m; s > 0; s = (m & (s - 1)))
// Iterate on non-zero bits of a bitmask B. // hash-1
for (int j = B.Find_next(0); j < MAXV; j = B.Find_next(j))
ll next_perm(ll v) { // compute next perm i.e.
    ll t = v | (v-1); // 00111,01011,01101,10011 ...
    return (t + 1) | (((~t & ~t) - 1)>>(__builtin_ctz(v) + 1));
} // hash-1 = 2392b3
template<typename F> // All subsets of size k of {0..N-1}
void iterate_k_subset(ll N, ll k, F f){ // hash-2
    ll mask = (1ll << k) - 1;
    while (! (mask & 1ll << N)) { f(mask);
        ll t = mask | (mask-1);
        mask = (t+1) | (((~t & ~t)-1)>>(__builtin_ctz1l(mask)+1));
    }
} // hash-2 = 87802c
template<typename F> // All subsets of set // hash-3
void iterate_mask_subset(ll set, F f){ ll mask = set;
    do f(mask), mask = (mask-1) & set;
    while (mask != set);
} // hash-3 = 3b9428
```

10.6 Random Numbers

```
random-numbers.h
Description: An example on the usage of generator and distribution. Use
shuffle instead of random shuffle.
```

```
mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().
    count());
shuffle(permutation.begin(), permutation.end(), rng);
uniform_int_distribution<int> uid(1, 100); //[1, 100]
unsigned xrand() {
    static unsigned x = 314159265, y = 358979323, z = 846264338,
        w = 327950288;
    unsigned t = x ^ x << 11; x = y; y = z; z = w; return w = w ^
        w >> 19 ^ t ^ t >> 8;
}
```

10.7 Test Hash

```
test-hash.h
Description: Just to test the hash - 4bbda0
```

```
string hash = "good";
```