

- 1 Contest
- 2 Mathematics
- 3 Data structures
- 4 Numerical
- 5 Number theory
- 6 Combinatorial
- 7 Graph
- 8 Geometry
- 9 Strings
- 10 Various

Contest (1)

template.cpp	23 lines
<pre>#include <bits/stdc++.h> using namespace std; #pragma GCC optimize("O3,unroll-loops") #pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt") #define rep(i, a, b) for(int i = a; i < (b); ++i) #define all(x) begin(x), end(x) #define sz(x) int (size(x)) typedef long long ll; typedef pair<int, int> pii; typedef vector<int> vi; int main() { #ifdef DBG freopen("in.txt", "r", stdin); freopen("out.txt", "w", stdout); #endif cin.tie(0)->sync_with_stdio(0); cin.exceptions(cin.failbit); <i>// Not for weird formats</i> } g++ -g -O2 -std=gnu++20 -static -DDBG X.cpp && ./a.out</pre>	
.bashrc	3 lines
<pre>alias c='g++ -Wall -Wconversion -Wfatal-errors -g -std=c++14 \ -fsanitize=undefined,address' xmodmap -e 'clear lock' -e 'keycode 66=less greater' <i>#caps = ◊</i></pre>	
.vimrc	6 lines
<pre>set cin aw ai is ts=4 sw=4 tm=50 nu noe b g=dark ru cul sy on im jk <esc> im kj <esc> no ; : " Select region and then type :Hash to hash your selection. " Useful for verifying that there aren't mistypes. ca Hash w !cpp -dD -P -fpreprocessed \ tr -d '[:space:]' \ \ md5sum \ cut -c-6</pre>	

hash.sh	3 lines
<pre># Hashes a file, ignoring all whitespace and comments. Use for # verifying that code was correctly typed. cpp -dD -P -fpreprocessed tr -d '[:space:]' md5sum cut -c-6</pre>	
troubleshoot.txt	52 lines
<pre>Pre-submit: Write a few simple test cases if sample is not enough. Are time limits close? If so, generate max cases. Is the memory usage fine? Could anything overflow? Make sure to submit the right file. Wrong answer: Print your solution! Print debug output, as well. Are you clearing all data structures between test cases? Can your algorithm handle the whole range of input? Read the full problem statement again. Do you handle all corner cases correctly? Have you understood the problem correctly? Any uninitialized variables? Any overflows? Confusing N and M, i and j, etc.? Are you sure your algorithm works? What special cases have you not thought of? Are you sure the STL functions you use work as you think? Add some assertions, maybe resubmit. Create some testcases to run your algorithm on. Go through the algorithm for a simple case. Go through this list again. Explain your algorithm to a teammate. Ask the teammate to look at your code. Go for a small walk, e.g. to the toilet. Is your output format correct? (including whitespace) Rewrite your solution from the start or let a teammate do it. Runtime error: Have you tested all corner cases locally? Any uninitialized variables? Are you reading or writing outside the range of any vector? Any assertions that might fail? Any possible division by 0? (mod 0 for example) Any possible infinite recursion? Invalidated pointers or iterators? Are you using too much memory? Debug with resubmits (e.g. remapped signals, see Various). Time limit exceeded: Do you have any possible infinite loops? What is the complexity of your algorithm? Are you copying a lot of unnecessary data? (References) How big is the input and output? (consider scanf) Avoid vector, map. (use arrays/unordered_map) What do your teammates think about your algorithm? Memory limit exceeded: What is the max amount of memory your algorithm should need? Are you clearing all data structures between test cases?</pre>	
<h2>Mathematics (2)</h2>	
<h3>2.1 Equations</h3>	
$ax^2 + bx + c = 0 \Rightarrow x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$	

The extremum is given by $x = -b/2a$.

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

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

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

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

2.2 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. $a_n = (d_1n + d_2)r^n$.

2.3 Trigonometry

$$\begin{aligned} \sin(v + w) &= \sin v \cos w + \cos v \sin w \\ \cos(v + w) &= \cos v \cos w - \sin v \sin w \end{aligned}$$

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

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

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

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

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

2.4 Geometry

2.4.1 Triangles

Side lengths: a, b, c

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

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

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

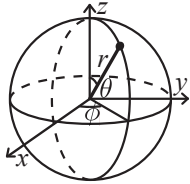
2.4.2 Quadrilaterals

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

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

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

2.4.3 Spherical coordinates



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

2.5 Derivatives/Integrals

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

Integration by parts:

$$\int_a^b f(x)g(x)dx = [F(x)g(x)]_a^b - \int_a^b F(x)g'(x)dx$$

2.6 Sums

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

$$1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$$

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

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

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

2.7 Series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots, (-\infty < x < \infty)$$

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \dots, (-1 < x \leq 1)$$

$$\sqrt{1+x} = 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{2x^3}{32} - \frac{5x^4}{128} + \dots, (-1 \leq x \leq 1)$$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots, (-\infty < x < \infty)$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots, (-\infty < x < \infty)$$

2.8 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 ,

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

2.8.1 Discrete distributions

Binomial distribution

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

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

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

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

First success distribution

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

$$p(k) = p(1-p)^{k-1}, k = 1, 2, \dots$$

$$\mu = \frac{1}{p}, \sigma^2 = \frac{1-p}{p^2}$$

Poisson distribution

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

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

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

2.8.2 Continuous distributions

Uniform distribution

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

$$f(x) = \begin{cases} \frac{1}{b-a} & a < x < b \\ 0 & \text{otherwise} \end{cases}$$

$$\mu = \frac{a+b}{2}, \sigma^2 = \frac{(b-a)^2}{12}$$

Exponential distribution

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

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

$$\mu = \frac{1}{\lambda}, \sigma^2 = \frac{1}{\lambda^2}$$

Normal distribution

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

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

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

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

```
template<class T>
struct SubMatrix {
    vector<vector<T>> p;
    SubMatrix(vector<vector<T>>& v) {
        int R = sz(v), C = sz(v[0]);
        p.assign(R+1, vector<T>(C+1));
        rep(r,0,R) rep(c,0,C)
```

```

        p[r+1][c+1] = v[r][c] + p[r][c+1] + p[r+1][c] - p[r][c];
    }
    T sum(int u, int l, int d, int r) {
        return p[d][r] - p[d][l] - p[u][r] + p[u][l];
    }
};

```

Matrix.h

Description: Basic operations on square matrices.
Usage: Matrix<int, 3> A;
 A.d = {{{{1,2,3}}, {{4,5,6}}, {{7,8,9}}}}};
 vector<int> vec = {1,2,3};
 vec = (A^N) * vec;

c43c7d, 26 lines

```

template<class T, int N> struct Matrix {
    typedef Matrix M;
    array<array<T, N>, N> d{};
    M operator*(const M& m) const {
        M a;
        rep(i,0,N) rep(j,0,N)
            rep(k,0,N) a.d[i][j] += d[i][k]*m.d[k][j];
        return a;
    }
    vector<T> operator*(const vector<T>& vec) const {
        vector<T> ret(N);
        rep(i,0,N) rep(j,0,N) ret[i] += d[i][j] * vec[j];
        return ret;
    }
    M operator^(ll p) const {
        assert(p >= 0);
        M a, b(*this);
        rep(i,0,N) a.d[i][i] = 1;
        while (p) {
            if (p&1) a = a*b;
            b = b*b;
            p >>= 1;
        }
        return a;
    }
};

```

LineContainer.h

Description: Container where you can add lines of the form $kx+m$, and query maximum values at points x . Useful for dynamic programming (“convex hull trick”).
Time: $\mathcal{O}(\log N)$

Sec1c7, 30 lines

```

struct Line {
    mutable ll k, m, p;
    bool operator<(const Line& o) const { return k < o.k; }
    bool operator<(ll x) const { return p < x; }
};

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

```

Matrix LineContainer Treap PersistentLazyTreap

```

        isect(x, erase(y));
    }
    ll query(ll x) {
        assert(!empty());
        auto l = *lower_bound(x);
        return l.k * x + l.m;
    }
};

```

Treap.h

Description: A short self-balancing tree. It acts as a sequential container with log-time splits/joins, and is easy to augment with additional data.
Time: $\mathcal{O}(\log N)$

9556fc, 55 lines

```

struct Node {
    Node *l = 0, *r = 0;
    int val, y, c = 1;
    Node(int val) : val(val), y(rand()) {}
    void recalc();
};

int cnt(Node* n) { return n ? n->c : 0; }
void Node::recalc() { c = cnt(l) + cnt(r) + 1; }

template<class F> void each(Node* n, F f) {
    if (n) { each(n->l, f); f(n->val); each(n->r, f); }
}

pair<Node*, Node*> split(Node* n, int k) {
    if (!n) return {};
    if (cnt(n->l) >= k) { // "n->val >= k" for lower_bound(k)
        auto pa = split(n->l, k);
        n->l = pa.second;
        n->recalc();
        return {pa.first, n};
    } else {
        auto pa = split(n->r, k - cnt(n->l) - 1); // and just "k"
        n->r = pa.first;
        n->recalc();
        return {n, pa.second};
    }
}

```

```

Node* merge(Node* l, Node* r) {
    if (!l) return r;
    if (!r) return l;
    if (l->y > r->y) {
        l->r = merge(l->r, r);
        l->recalc();
        return l;
    } else {
        r->l = merge(l, r->l);
        r->recalc();
        return r;
    }
}

```

```

Node* ins(Node* t, Node* n, int pos) {
    auto pa = split(t, pos);
    return merge(merge(pa.first, n), pa.second);
}

```

```

// Example application: move the range [l, r) to index k
void move(Node& t, int l, int r, int k) {
    Node *a, *b, *c;
    tie(a,b) = split(t, l); tie(b,c) = split(b, r - l);
    if (k <= l) t = merge(ins(a, b, k), c);
    else t = merge(a, ins(c, b, k - r));
}

```

PersistentLazyTreap.h

Description: A persistent treap with lazy propagation. Use a bump allocator for better performance.

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

9c7ce3, 86 lines

```

ll rnd() {
    static uniform_int_distribution<ll> d(MIN<ll>, MAX<ll>);
    return d(randy);
}

struct Node; typedef array<Node *, 2> ann;
int T = 0; // current version. T++ => next op is persistent.
struct Node {
    ll p; // priority
    ann c{}; // children
    int t; // timestamp
    int n; // size
    int v; // value
    ll s; // sum
    int l = 0; // lazy value
    Node(int val, ll p) : p(p), c{}, t(T), v(val), s(val) {}
    Node(int val) : Node(val, rnd()) {}
    //~Node() { rep(i, 0, 2) delete c[i]; } // breaks persistence
    void upd(int x) { l += x; v += x; s += (ll)n * x; } // custom
};
// If persistence not needed, just return n.
// Use bump allocator (w/ free list?) for efficiency.
void cp(Node *&n) {
    if (n && n->t != T) (n = new Node(*n))->t = T;
}
// If lazy prop not needed, just call cp(n) and return n.
Node *push(Node *&n) { // custom; recurse for ST beats
    cp(n); if (n && n->l) { // if (has lazy value)
        rep(i, 0, 2) if (cp(n->c[i]), n->c[i]) n->c[i]->upd(n->l);
        n->l = 0; // reset lazy value
    } return n;
}
void fix(Node *n) { // custom
    n->n = 1; n->s = n->v; // reset sum
    rep(i, 0, 2) if (n->c[i]) {
        n->n += n->c[i]->n;
        n->s += n->c[i]->s; // update n's sum
    }
}
// p determines which side a node goes on (l -> right)
template<class P> void split(Node *n, ann &A, P &&p) {
    if (n) {
        bool d = p(push(n)); split(n->c[!d], A, p);
        n->c[!d] = A[d]; fix(A[d] = n);
    }
}
template<class P> auto split(Node *n, P &&p) {
    ann A{}; split(n, A, p); return make_pair(A[0], A[1]);
}
// 1st node where p is true or nullptr
// Pd: push-down every node
template<bool Pd, class P> Node *upper_bound(Node *n, P &&p) {
    if (n) {
        bool d = p(Pd ? push(n) : n);
        Node *m = upper_bound(n->c[!d], p);
        if (m || !d) n = m; // m || d for last where !p
    } return n;
}
// p for implicit split/upper_bound (splits < i and >= i)
struct impl {
    int i;
    impl(int i) : i(i) {}
    bool operator()(Node *n) {
        int l = n->c[0] ? n->c[0]->n : 0, d = l >= i;
        if (!d) i -= l + 1; return d;
    }
}

```

```

}
};

Node *join(ann A) {
    rep(i, 0, 2) if (!A[i]) return A[!i];
    bool d = A[0]->p < A[1]->p;
    Node *n = push(A[!d]); A[!d] = n->c[d];
    n->c[d] = join(A); fix(n); return n;
}

Node *join(Node *a, Node *b) { return join({ a, b }); }
template<bool Pd, class F> void each(Node *n, const F &&f) {
    if (Pd ? push(n) : n) {
        each(n->c[0], f); f(n); each(n->c[1], f);
    }
}

Node *tree(vector<Node *> A) { // A's sorted, singleton,
    nonlazy
    static stack<Node *> S({ 0 }); Node *r;
    for (Node *a : A) {
        cp(a); for (r = 0; S.top() && a->p < S.top()->p; S.pop())
            fix(r = S.top());
        a->c[0] = r; if (S.top()) S.top()->c[1] = a; S.push(a);
    } for (r = 0; S.top(); S.pop()) fix(r = S.top()); return r;
}
```

FenwickTree.h

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

Time: Both operations are $\mathcal{O}(\log N)$.

```

e62fac, 22 lines

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

FenwickTree2d.h

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

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

```

" FenwickTree.h"
157f07, 22 lines

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

```

        return (int)(lower_bound(all(ys[x]), y) - ys[x].begin()); }
    void update(int x, int y, ll dif) {
        for (; x < sz(ys); x |= x + 1)
            ft[x].update(ind(x, y), dif);
    }
    ll query(int x, int y) {
        ll sum = 0;
        for (; x; x &= x - 1)
            sum += ft[x-1].query(ind(x-1, y));
        return sum;
    }
};
```

RMQ.h

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

Usage: RMQ rmq(values);

rmq.query(inclusive, exclusive);

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

```

510c32, 16 lines

template<class T>
struct RMQ {
    vector<vector<T>> jmp;
    RMQ(const vector<T>& V) : jmp(1, V) {
        for (int pw = 1, k = 1; pw * 2 <= sz(V); pw *= 2, ++k) {
            jmp.emplace_back(sz(V) - pw * 2 + 1);
            rep(j, 0, sz(jmp[k]))
                jmp[k][j] = min(jmp[k - 1][j], jmp[k - 1][j + pw]);
        }
    }
    T query(int a, int b) {
        assert(a < b); // or return inf if a == b
        int dep = 31 - __builtin_clz(b - a);
        return min(jmp[dep][a], jmp[dep][b - (1 << dep)]);
    }
};
```

MoQueries.h

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

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

```

a12ef4, 49 lines

void add(int ind, int end) { ... } // add a[ind] (end = 0 or 1)
void del(int ind, int end) { ... } // remove a[ind]
int calc() { ... } // compute current answer

vi mo(vector<pii> Q) {
    int L = 0, R = 0, blk = 350; // ~N/sqrt(Q)
    vi s(sz(Q)), res = s;
    #define K(x) pii(x.first/blk, x.second ^ -(x.first/blk & 1))
    iota(all(s), 0);
    sort(all(s), [&](int s, int t){ return K(Q[s]) < K(Q[t]); });
    for (int qi : s) {
        pii q = Q[qi];
        while (L > q.first) add(--L, 0);
        while (R < q.second) add(R++, 1);
        while (L < q.first) del(L++, 0);
        while (R > q.second) del(--R, 1);
        res[qi] = calc();
    }
    return res;
}
```

```

vi moTree(vector<array<int, 2>> Q, vector<vi>& ed, int root=0){
    int N = sz(ed), pos[2] = {}, blk = 350; // ~N/sqrt(Q)
    vi s(sz(Q)), res = s, I(N), L(N), R(N), in(N), par(N);
    add(0, 0), in[0] = 1;
```

```

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

Numerical (4)

4.1 Polynomials and recurrences

```

Polynomial.h
c9b7b0, 17 lines

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

PolyRoots.h

Description: Finds the real roots to a polynomial.

Usage: polyRoots({{2,-3,1}},-1e9,1e9) // solve $x^2-3x+2 = 0$

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

```

"Polynomial.h"
b00bfe, 23 lines

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

```

LP Solver(const vvd& A, const v& b, const v& c) :
    m(sz(b)), n(sz(c)), N(n+1), B(m), D(m+2, v(n+2)) {
        rep(i,0,m) rep(j,0,n) D[i][j] = A[i][j];
        rep(i,0,m) { B[i] = n+i; D[i][n] = -1; D[i][n+1] = b[i]; }
        rep(j,0,n) { N[j] = j; D[m][j] = -c[j]; }
        N[n] = -1; D[m+1][n] = 1;

```

```
    }

    void pivot(int r, int s) {
        T *a = D[r].data(), inv = 1 / a[s];
        rep(i,0,m+2) if (i != r && abs(D[i][s]) > eps) {
            T *b = D[i].data(), inv2 = b[s] * inv;
            rep(j,0,n+2) b[j] -= a[j] * inv2;
            b[s] = a[s] * inv2;
        }
        rep(j,0,n+2) if (j != s) D[r][j] *= inv;
        rep(i,0,m+2) if (i != r) D[i][s] *= -inv;
        D[r][s] = inv;
        swap(B[r], N[s]);
    }

    bool simplex(int phase) {
        int x = m + phase - 1;
        for (;;) {
            int s = -1;
            rep(j,0,n+1) if (N[j] != -phase) ltj(D[x]);
            if (D[x][s] >= -eps) return true;
            int r = -1;
            rep(i,0,m) {
                if (D[i][s] <= eps) continue;
                if (r == -1 || MP(D[i][n+1] / D[i][s], B[i])
                    < MP(D[r][n+1] / D[r][s], B[r])) r = i;
            }
            if (r == -1) return false;
            pivot(r, s);
        }
    }

    T solve(vd &x) {
        int r = 0;
        rep(i,1,m) if (D[i][n+1] < D[r][n+1]) r = i;
        if (D[r][n+1] < -eps) {
            pivot(r, n);
            if (!simplex(2) || D[m+1][n+1] < -eps) return -inf;
            rep(i,0,m) if (B[i] == -1) {
                int s = 0;
                rep(j,1,n+1) ltj(D[i]);
                pivot(i, s);
            }
        }
        bool ok = simplex(1); x = vd(n);
        rep(i,0,m) if (B[i] < n) x[B[i]] = D[i][n+1];
        return ok ? D[m][n+1] : inf;
    }
};
```

4.3 Matrices

Determinant.h

Description: Calculates determinant of a matrix. Destroys the matrix.

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

bd5cec, 15 lines

```
double det(vector<vector<double>>& a) {
    int n = sz(a); double res = 1;
    rep(i,0,n) {
        int b = i;
        rep(j,i+1,n) if (fabs(a[j][i]) > fabs(a[b][i])) b = j;
        if (i != b) swap(a[i], a[b]), res *= -1;
        res *= a[i][i];
        if (res == 0) return 0;
        rep(j,i+1,n) {
            double v = a[j][i] / a[i][i];
            if (v != 0) rep(k,i+1,n) a[j][k] -= v * a[i][k];
        }
    }
    return res;
}
```

IntDeterminant.h

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

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

3313dc, 18 lines

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

SolveLinear.h

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

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

44c9ab, 38 lines

```
typedef vector<double> vd;
const double eps = 1e-12;

int solveLinear(vector<vd>& A, vd& b, vd& x) {
    int n = sz(A), m = sz(x), rank = 0, br, bc;
    if (n) assert(sz(A[0]) == m);
    vi col(m); iota(all(col), 0);

    rep(i,0,n) {
        double v, bv = 0;
        rep(r,i,n) rep(c,i,m)
            if ((v = fabs(A[r][c])) > bv)
                br = r, bc = c, bv = v;
        if (bv <= eps) {
            rep(j,i,n) if (fabs(b[j]) > eps) return -1;
            break;
        }
        swap(A[i], A[br]);
        swap(b[i], b[br]);
        swap(col[i], col[bc]);
        rep(j,0,n) swap(A[j][i], A[j][bc]);
        bv = 1/A[i][i];
        rep(j,i+1,n) {
            double fac = A[j][i] * bv;
            b[j] -= fac * b[i];
            rep(k,i+1,m) A[j][k] -= fac*A[i][k];
        }
        rank++;
    }

    x.assign(m, 0);
    for (int i = rank; i--;) {
        b[i] /= A[i][i];
        x[col[i]] = b[i];
        rep(j,0,i) b[j] -= A[j][i] * b[i];
    }
    return rank; // (multiple solutions if rank < m)
}
```

SolveLinear2.h

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

"SolveLinear.h"

08e495, 7 lines

rep(j,0,n) if (j != i) // instead of rep(j,i+1,n)
// ... then at the end:
x.assign(m, undefined);
rep(i,0,rank) {
 rep(j,rank,m) if (fabs(A[i][j]) > eps) goto fail;
 x[col[i]] = b[i] / A[i][i];
fail; }

SolveLinearBinary.h

Description: Solves $Ax = b$ over \mathbb{F}_2 . If there are multiple solutions, one is returned arbitrarily. Returns rank, or -1 if no solutions. Destroys A and b .

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

fa2d7a, 34 lines

typedef bitset<1000> bs;

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

 x = bs();
 for (int i = rank; i--;) {
 if (!b[i]) continue;
 x[col[i]] = 1;
 rep(j,0,i) b[j] ^= A[j][i];
 }
 return rank; // (multiple solutions if rank < m)
}

MatrixInverse.h

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

Time: $\mathcal{O}(n^3)$

ebfff6, 35 lines

int matInv(vector<vector<double>>& A) {
 int n = sz(A); vi col(n);
 vector<vector<double>> tmp(n, vector<double>(n));
 rep(i,0,n) tmp[i][i] = 1, col[i] = i;

 rep(i,0,n) {
 int r = i, c = i;
 rep(j,i,n) rep(k,i,n)
 if (fabs(A[j][k]) > fabs(A[r][c]))
 r = j, c = k;

FastSubsetTransform.h

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

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

```
void FST(vi& a, bool inv) {
    for (int n = sz(a), step = 1; step < n; step *= 2) {
```



```

auto f = [n](ull x) { return modmul(x, x, n) + 1; };
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);
    if ((q = modmul(prd, max(x,y) - min(x,y), n))) prd = q;
    x = f(x), y = f(y);
}

```

```
    }
    return __gcd(prd, n);
}
vector<ull> factor(ull n) {
    if (n == 1) return {};
    if (isPrime(n)) return {n};
    ull x = pollard(n);
    auto l = factor(x), r = factor(n / x);
    l.insert(l.end(), all(r));
    return l;
}
```

5.3 Divisibility

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

```
33ba8f, 5 lines
1l euclid(1l a, 1l b, 1l &x, 1l &y) {
    if (!b) return x = 1, y = 0, a;
    1l d = euclid(b, a % b, y, x);
    return y -= a/b * x, d;
}
```

```
CRT.h
Description: Chinese Remainder Theorem.
crt(a, m, b, n) computes x such that x ≡ a (mod m), x ≡ b (mod n). If
|a| < m and |b| < n, x will obey 0 ≤ x < lcm(m, n). Assumes mn < 2^62.
Time: log(n)
"euclid.h"
04d93a, 7 lines
1l crt(1l a, 1l m, 1l b, 1l n) {
    if (n > m) swap(a, b), swap(m, n);
    1l x, y, g = euclid(m, n, x, y);
    assert((a - b) % g == 0); // else no solution
    x = (b - a) % n * x % n / g * m + a;
    return x < 0 ? x + m*n/g : x;
}
```

5.3.1 Bézout’s identity

For $a \neq 0, b \neq 0$, then $d = \gcd(a, b)$ is the smallest positive integer for which there are integer solutions to

$$ax + by = d$$

If (x, y) is one solution, then all solutions are given by

$$\left(x + \frac{kb}{\gcd(a,b)}, y - \frac{ka}{\gcd(a,b)}\right), \quad k \in \mathbb{Z}$$

```
phiFunction.h
Description: Euler’s φ function is defined as φ(n) := # of positive integers
≤ n that are coprime with n. φ(1) = 1, p prime ⇒ φ(p^k) = (p − 1)p^{k−1},
m, n coprime ⇒ φ(mn) = φ(m)φ(n). If n = p_1^{k_1} p_2^{k_2} ... p_r^{k_r} then φ(n) =
(p_1 − 1)p_1^{k_1−1} ... (p_r − 1)p_r^{k_r−1}. φ(n) = n · ∏_{p|n} (1 − 1/p).
∑_{d|n} φ(d) = n, ∑_{1 ≤ k ≤ n, gcd(k,n)=1} k = nφ(n)/2, n > 1
Euler’s thm: a, n coprime ⇒ a^{φ(n)} ≡ 1 (mod n).
Fermat’s little thm: p prime ⇒ a^{p−1} ≡ 1 (mod p) ∀ a.
cf7d6d, 8 lines
const int LIM = 5000000;
int phi[LIM];

void calculatePhi() {
    rep(i, 0, LIM) phi[i] = i&1 ? i : i/2;
    for (int i = 3; i < LIM; i += 2) if(phi[i] == i)
```

```
    for (int j = i; j < LIM; j += i) phi[j] -= phi[j] / i;
}
```

LinearSieve.h
Description: Finds prime powers and computes a multiplicative function $(f(ab) = f(a)f(b))$ for coprime a, b on $[1, n]$. Customize with $f(p)$ and $f(ip)$ where p is i ’s smallest prime factor. Currently computes ϕ .
Time: $\mathcal{O}(n)$

```
bcd9a, 23 lines
vi P; // primes
bool C[MAXN]; // composite?
int F[MAXN], K[MAXN]; // function value, power of min pf

void sieve(int n) {
    fill_n(C, n, 0); F[1] = 1; P.clear();
    rep(i, 2, n) {
        if (!C[i]) {
            P.push_back(i);
            K[i] = 1;
            F[i] = i - 1; // f(i) for i prime (custom)
        }
        for (int j = 0, x; j < sz(P) && (x = i * P[j]) < n; ++j) {
            C[x] = K[x] = 1;
            if (i % P[j]) F[x] = F[i] * F[P[j]]; // f(i*p), p doesn't
                divide i
            else {
                F[x] = F[i] * P[j]; // f(ip), p = i's min pf (custom)
                K[x] += K[i];
                break;
            }
        }
    }
}
```

5.4 Fractions

ContinuedFractions.h
Description: Given N and a real number $x \geq 0$, finds the closest rational approximation p/q with $p, q \leq N$. It will obey $|p/q - x| \leq 1/qN$. For consecutive convergents, $p_{k+1}q_k - q_{k+1}p_k = (-1)^k$. (p_k/q_k alternates between $> x$ and $< x$.) If x is rational, y eventually becomes ∞ ; if x is the root of a degree 2 polynomial the a ’s eventually become cyclic.
Time: $\mathcal{O}(\log N)$

```
dd65e, 21 lines
typedef double d; // for N ~ 1e7; long double for N ~ 1e9
pair<1l, 1l> approximated(d x, 1l N) {
    1l LP = 0, LQ = 1, P = 1, Q = 0, inf = LLONG_MAX; d y = x;
    for (;;) {
        1l lim = min(P ? (N-LP) / P : inf, Q ? (N-LQ) / Q : inf),
            a = (1l)floor(y), b = min(a, lim),
            NP = b*P + LP, NQ = b*Q + LQ;
        if (a > b) {
            // If b > a/2, we have a semi-convergent that gives us a
            // better approximation; if b = a/2, we *may* have one.
            // Return {P, Q} here for a more canonical approximation.
            return (abs(x - (d)NP / (d)NQ) < abs(x - (d)P / (d)Q)) ?
                make_pair(NP, NQ) : make_pair(P, Q);
        }
        if (abs(y = 1/(y - (d)a)) > 3*N) {
            return {NP, NQ};
        }
        LP = P; P = NP;
        LQ = Q; Q = NQ;
    }
}
```

FracBinarySearch.h

Description: Given f and N , finds the smallest fraction $p/q \in [0, 1]$ such that $f(p/q)$ is true, and $p, q \leq N$. You may want to throw an exception from f if it finds an exact solution, in which case N can be removed.
Usage: `fracBS({}f)(Frac f) { return f.p>=3*f.q; }, 10);` // $\{1, 3\}$
Time: $\mathcal{O}(\log(N))$

```
27ab3e, 25 lines
struct Frac { 1l p, q; };

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

5.5 Pythagorean Triples

The Pythagorean triples are uniquely generated by

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

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

5.6 Primes

$p = 962592769$ 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.

Primitive roots exist modulo any prime power p^a , except for $p = 2, a > 2$, and there are $\phi(\phi(p^a))$ many. For $p = 2, a > 2$, the group $\mathbb{Z}_{2^a}^\times$ is instead isomorphic to $\mathbb{Z}_2 \times \mathbb{Z}_{2^{a-2}}$.

5.7 Estimates

$$\sum_{d|n} d = O(n \log \log n).$$

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

5.8 Mobius Function

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

Mobius Inversion:

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

Other useful formulas/forms:

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

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

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

Combinatorial (6)

6.1 Permutations

6.1.1 Factorial

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

IntPerm.h
Description: Permutation -> integer conversion. (Not order preserving.)
Integer -> permutation can use a lookup table.
Time: $\mathcal{O}(n)$

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

6.1.2 Cycles

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

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

6.1.3 Derangements

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

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

6.1.4 Burnside’s lemma

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

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

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

IntPerm multinomial

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

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

6.2 Partitions and subsets

6.2.1 Partition function

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

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

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

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

6.2.2 Lucas’ Theorem

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

6.2.3 Binomials

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

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

6.3 General purpose numbers

6.3.1 Bernoulli numbers

EGF of Bernoulli numbers is $B(t) = \frac{t}{e^t - 1}$ (FFT-able).
 $B[0, \dots] = [1, -\frac{1}{2}, \frac{1}{6}, 0, -\frac{1}{30}, 0, \frac{1}{42}, \dots]$

Sums of powers:

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

Euler-Maclaurin formula for infinite sums:

$$\sum_{i=m}^\infty f(i) = \int_m^\infty f(x) dx - \sum_{k=1}^\infty \frac{B_k}{k!} f^{(k-1)}(m)$$

$$\approx \int_m^\infty f(x) dx + \frac{f(m)}{2} - \frac{f'(m)}{12} + \frac{f'''(m)}{720} + O(f^{(5)}(m))$$

6.3.2 Stirling numbers of the first kind

Number of permutations on n items with k cycles.

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

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

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

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

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. $B(n) = 1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, \dots$. For p 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 : $(n-2)! / ((d_1-1)! \dots (d_n-1)!)$

6.3.7 Catalan numbers

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

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

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

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

- ```
pair<ll, ll> maxflow(int s, int t) {
 ll totflow = 0, totcost = 0;
 while (path(s, seen[t]) {
 ll fl = INF;
 for (int p, r, x = t; tie(p,r) = par[x], x != s; x = p)
 fl = min(fl, r ? cap[p][x] - flow[p][x] : flow[x][p]);
```

```
totflow += fl;
for (int p,r,x = t; tie(p,r) = par[x], x != s; x = p)
 if (r) flow[p][x] += fl;
 else flow[x][p] -= fl;
}
rep(i,0,N) rep(j,0,N) totcost += cost[i][j] * flow[i][j];
return {totflow, totcost};
}

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

EdmondsKarp.h

**Description:** Flow algorithm with guaranteed complexity  $O(VE^2)$ . To get edge flow values, compare capacities before and after, and take the positive values only.

482fe0, 35 lines

```
template<class T> T edmondsKarp(vector<unordered_map<int, T>&&
 graph, int source, int sink) {
 assert(source != sink);
 T flow = 0;
 vi par(sz(graph)), q = par;

 for (;;) {
 fill(all(par), -1);
 par[source] = 0;
 int ptr = 1;
 q[0] = source;

 rep(i,0,ptr) {
 int x = q[i];
 for (auto e : graph[x]) {
 if (par[e.first] == -1 && e.second > 0) {
 par[e.first] = x;
 q[ptr++] = e.first;
 if (e.first == sink) goto out;
 }
 }
 }
 return flow;
 }
out:
 T inc = numeric_limits<T>::max();
 for (int y = sink; y != source; y = par[y])
 inc = min(inc, graph[par[y]][y]);

 flow += inc;
 for (int y = sink; y != source; y = par[y]) {
 int p = par[y];
 if ((graph[p][y] -= inc) <= 0) graph[p].erase(y);
 graph[y][p] += inc;
 }
}
```

Dinic.h

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

d7f0f1, 42 lines

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

MinCut.h

**Description:** After running max-flow, the left side of a min-cut from  $s$  to  $t$  is given by all vertices reachable from  $s$ , only traversing edges with positive residual capacity.

GlobalMinCut.h

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

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

8b0e19, 21 lines

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

```
rep(i,0,n) mat[s][i] += mat[t][i];
rep(i,0,n) mat[i][s] = mat[s][i];
mat[0][t] = INT_MIN;
}
return best;
}
```

GomoryHu.h

**Description:** Given a list of edges representing an undirected flow graph, returns edges of the Gomory-Hu tree. The max flow between any pair of vertices is given by minimum edge weight along the Gomory-Hu tree path.

**Time:**  $O(V)$  Flow Computations

0418b3, 13 lines

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

7.3 Matching

hopcroftKarp.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:**  $vi\ btoa(m, -1);$  hopcroftKarp( $g, btoa$ );

**Time:**  $O(\sqrt{VE})$

f612e4, 42 lines

```
bool dfs(int a, int L, vector<vi>& g, vi& btoa, vi& A, vi& B) {
 if (A[a] != L) return 0;
 A[a] = -1;
 for (int 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;
}
```

```
int hopcroftKarp(vector<vi>& g, vi& btoa) {
 int res = 0;
 vi A(g.size()), B(btoa.size()), cur, next;
 for (;;) {
 fill(all(A), 0);
 fill(all(B), 0);
 cur.clear();
 for (int a : btoa) if(a != -1) A[a] = -1;
 rep(a,0,sz(g)) if(A[a] == 0) cur.push_back(a);
 for (int lay = 1;; lay++) {
 bool islast = 0;
 next.clear();
 for (int a : cur) for (int 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 (int a : next) A[a] = lay;
 cur.swap(next);
}
rep(a,0,sz(g))
 res += dfs(a, 0, g, btoa, A, B);
}
```

DFSMatching.h

**Description:** Simple 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:** vi btoa(m, -1); dfsMatching(g, btoa);  
**Time:**  $\mathcal{O}(VE)$

522b98, 22 lines

```
bool find(int j, vector<vi>& g, vi& btoa, vi& vis) {
 if (btoa[j] == -1) return 1;
 vis[j] = 1; int di = btoa[j];
 for (int e : g[di])
 if (!vis[e] && find(e, g, btoa, vis)) {
 btoa[e] = di;
 return 1;
 }
 return 0;
}
int dfsMatching(vector<vi>& g, vi& btoa) {
 vi vis;
 rep(i,0,sz(g)) {
 vis.assign(sz(btoa), 0);
 for (int j : g[i])
 if (find(j, g, btoa, vis)) {
 btoa[j] = i;
 break;
 }
 }
 return sz(btoa) - (int)count(all(btoa), -1);
}
```

MinimumVertexCover.h

**Description:** Finds a minimum vertex cover in a bipartite graph. The size is the same as the size of a maximum matching, and the complement is a maximum independent set.

"DFSMatching.h" da4196, 20 lines

```
vi cover(vector<vi>& g, int n, int m) {
 vi match(m, -1);
 int res = dfsMatching(g, match);
 vector<bool> lfound(n, true), seen(m);
 for (int it : match) if (it != -1) lfound[it] = false;
 vi q, cover;
 rep(i,0,n) if (lfound[i]) q.push_back(i);
 while (!q.empty()) {
 int i = q.back(); q.pop_back();
 lfound[i] = 1;
 for (int e : g[i]) if (!seen[e] && match[e] != -1) {
 seen[e] = true;
 q.push_back(match[e]);
 }
 }
 rep(i,0,n) if (!lfound[i]) cover.push_back(i);
 rep(i,0,m) if (seen[i]) cover.push_back(n+i);
 assert(sz(cover) == res);
 return cover;
}
```

WeightedMatching.h

**Description:** Given a weighted bipartite graph, matches every node on the left with a node on the right such that no nodes are in two matchings and the sum of the edge weights is minimal. Takes cost[N][M], where cost[i][j] = cost for L[j] to be matched with R[j] and returns (min cost, match), where L[i] is matched with R[match[i]]. Negate costs for max cost. Requires  $N \leq M$ .  
**Time:**  $\mathcal{O}(N^2M)$

1e0fe9, 31 lines

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

FastHungarian.h

**Description:** Like WeightedMatching.h, but faster for dense graphs. costs should be nxn.  
**Time:**  $\mathcal{O}(N^3)$

0a2c68, 74 lines

```
typedef double T;
const T EPS = 1e-8, INF = numeric_limits<T>::infinity();
struct Hungarian {
 vector<vector<T>> C;
 int n;
 vector<T> lw, lj, slj;
 vi slwj, mw, mj, p;
 vector<bool> cw;
 Hungarian(vector<vector<T>> &costs) :
 C(costs), n(sz(C)), lw(n), lj(n), slj(n), slwj(n),
 mw(n, -1), mj(n, -1), p(n), cw(n) {}
 void reduce() {
 rep(w, 0, n) {
 T m = INF;
 rep(j, 0, n) m = min(C[w][j], m);
 rep(j, 0, n) C[w][j] -= m;
 }
 rep(j, 0, n) {
 T m = INF;
 rep(w, 0, n) m = min(m, C[w][j]);
 rep(w, 0, n) C[w][j] -= m;
 }
 }
 void init() {
 rep(j, 0, n) lj[j] = INF;
```

```
 rep(w, 0, n) rep(j, 0, n) lj[j] = min(lj[j], C[w][j]);
 }
 void match() {
 rep(w, 0, n) rep(j, 0, n)
 if (mw[w] < 0 && mj[j] < 0 && abs(C[w][j] - lw[w] - lj[j]
) <= EPS) {
 mw[w] = j; mj[j] = w;
 }
 }
 int next() {
 rep(w, 0, n) if (mw[w] < 0) return w;
 return -1;
 }
 void init_phase(int w) {
 fill(all(cw), false); fill(all(p), -1); cw[w] = true;
 rep(j, 0, n) { slj[j] = C[w][j] - lw[w] - lj[j]; slwj[j] =
 w; }
 }
 void upd(T s) {
 rep(w, 0, n) lw[w] += s * cw[w];
 rep(j, 0, n) (p[j] < 0 ? slj : lj)[j] -= s;
 }
 void phase() {
 for (;;) {
 int sw = -1, sj = -1; T s = INF;
 rep(j, 0, n)
 if (p[j] < 0 && slj[j] < s) { s = slj[j]; sw = slwj[j];
 sj = j; }
 if (abs(s) > EPS) upd(s);
 p[sj] = sw;
 if (mj[sj] < 0) {
 int j = sj, w = p[sj];
 for (;;) {
 mj[j] = w; swap(mw[w], j); if (j < 0) break;
 w = p[j];
 } return;
 }
 int w = mj[sj]; cw[w] = 1;
 rep(j, 0, n)
 if (p[j] < 0) {
 T sl = C[w][j] - lw[w] - lj[j];
 if (slj[j] > sl) { slj[j] = sl; slwj[j] = w; }
 }
 }
 }
 vi operator()() {
 reduce(); init(); match();
 for (int w = next(); w >= 0; w = next()) {
 init_phase(w); phase();
 } return mw;
 }
};
```

GeneralMatching.h

**Description:** Matching for general graphs. Fails with probability  $N/mod$ .  
**Time:**  $\mathcal{O}(N^3)$

"../numerical/MatrixInverse-mod.h" cb1912, 40 lines

```
vector<pii> generalMatching(int N, vector<pii>& ed) {
 vector<vector<ll>> mat(N, vector<ll>(N)), A;
 for (pii pa : ed) {
 int a = pa.first, b = pa.second, r = rand() % mod;
 mat[a][b] = r, mat[b][a] = (mod - r) % mod;
 }

 int r = matInv(A = mat), M = 2*N - r, fi, fj;
 assert(r % 2 == 0);

 if (M != N) do {
 mat.resize(M, vector<ll>(M));
```

```
rep(i,0,N) {
 mat[i].resize(M);
 rep(j,N,M) {
 int r = rand() % mod;
 mat[i][j] = r, mat[j][i] = (mod - r) % mod;
 }
} while (matInv(A = mat) != M);

vi has(M, 1); vector<pii> ret;
rep(it,0,M/2) {
 rep(i,0,M) if (has[i])
 rep(j,i+1,M) if (A[i][j] && mat[i][j]) {
 fi = i; fj = j; goto done;
 }
 assert(0); done:
 if (fj < N) ret.emplace_back(fi, fj);
 has[fi] = has[fj] = 0;
 rep(sw,0,2) {
 ll a = modpow(A[fi][fj], mod-2);
 rep(i,0,M) if (has[i] && A[i][fj]) {
 ll b = A[i][fj] * a % mod;
 rep(j,0,M) A[i][j] = (A[i][j] - A[fi][j] * b) % mod;
 }
 swap(fi,fj);
 }
}
return ret;
}
```

7.4 DFS algorithms

**SCC.h**  
**Description:** Finds strongly connected components in a directed graph. If vertices  $u, v$  belong to the same component, we can reach  $u$  from  $v$  and vice versa.  
**Usage:** `scc(graph, [&](vi& v) { ... })` visits all components in reverse topological order. `comp[i]` holds the component index of a node (a component only has edges to components with lower index). `ncomps` will contain the number of components.  
**Time:**  $\mathcal{O}(E + V)$

76b5c9, 24 lines

```
vi val, comp, z, cont;
int Time, ncomps;
template<class G, class F> int dfs(int j, G& g, F& f) {
 int low = val[j] = ++Time, x; z.push_back(j);
 for (auto e : g[j]) if (comp[e] < 0)
 low = min(low, val[e] ?: dfs(e,g,f));

 if (low == val[j]) {
 do {
 x = z.back(); z.pop_back();
 comp[x] = ncomps;
 cont.push_back(x);
 } while (x != j);
 f(cont); cont.clear();
 ncomps++;
 }
 return val[j] = low;
}
template<class G, class F> void scc(G& g, F f) {
 int n = sz(g);
 val.assign(n, 0); comp.assign(n, -1);
 Time = ncomps = 0;
 rep(i,0,n) if (comp[i] < 0) dfs(i, g, f);
}
```

BiconnectedComponents.h

SCC BiconnectedComponents 2sat EulerWalk

**Description:** Finds all biconnected components in an undirected graph, and runs a callback for the edges in each. In a biconnected component there are at least two distinct paths between any two nodes. Note that a node can be in several components. An edge which is not in a component is a bridge, i.e., not part of any cycle.  
**Usage:** `int eid = 0; ed.resize(N);` for each edge `(a,b)` `ed[a].emplace_back(b, eid);` `ed[b].emplace_back(a, eid++);` `bicomps([&](const vi& edgelist) {...});`  
**Time:**  $\mathcal{O}(E + V)$

2965e5, 33 lines

```
vi num, st;
vector<vector<pii>> ed;
int Time;
template<class F>
int dfs(int at, int par, F& f) {
 int me = num[at] = ++Time, e, y, top = me;
 for (auto pa : ed[at]) if (pa.second != par) {
 tie(y, e) = pa;
 if (num[y]) {
 top = min(top, num[y]);
 if (num[y] < me)
 st.push_back(e);
 } else {
 int si = sz(st);
 int up = dfs(y, e, f);
 top = min(top, up);
 if (up == me) {
 st.push_back(e);
 f(vi(st.begin() + si, st.end()));
 st.resize(si);
 }
 else if (up < me) st.push_back(e);
 else { /* e is a bridge */ }
 }
 }
 return top;
}

template<class F>
void bicomps(F f) {
 num.assign(sz(ed), 0);
 rep(i,0,sz(ed)) if (!num[i]) dfs(i, -1, f);
}
```

2sat.h

**Description:** Calculates a valid assignment to boolean variables  $a, b, c, \dots$  to a 2-SAT problem, so that an expression of the type  $(a|||b)&\&(!a|||c)&\&(d|||!b)&\&\dots$  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.setValue(2);` // Var 2 is true  
`ts.atMostOne({0,~1,2});` //  $\leq 1$  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.

5f9706, 56 lines

```
struct TwoSat {
 int N;
 vector<vi> gr;
 vi values; // 0 = false, 1 = true

 TwoSat(int n = 0) : N(n), gr(2*n) {}

 int addVar() { // (optional)
 gr.emplace_back();
 gr.emplace_back();
 }
}
```

```
return N++;
}

void either(int f, int j) {
 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);
}

void setValue(int x) { either(x, x); }

void atMostOne(const vi& li) { // (optional)
 if (sz(li) <= 1) return;
 int cur = ~li[0];
 rep(i,2,sz(li)) {
 int next = addVar();
 either(cur, ~li[i]);
 either(cur, next);
 either(~li[i], next);
 cur = ~next;
 }
 either(cur, ~li[1]);
}

vi val, comp, z; int time = 0;
int dfs(int i) {
 int low = val[i] = ++time, x; z.push_back(i);
 for(int e : gr[i]) if (!comp[e])
 low = min(low, val[e] ?: dfs(e));
 if (low == val[i]) do {
 x = z.back(); z.pop_back();
 comp[x] = low;
 if (values[x>>1] == -1)
 values[x>>1] = x&1;
 } while (x != i);
 return val[i] = low;
}

bool solve() {
 values.assign(N, -1);
 val.assign(2*N, 0); comp = val;
 rep(i,0,2*N) if (!comp[i]) dfs(i);
 rep(i,0,N) if (comp[2*i] == comp[2*i+1]) return 0;
 return 1;
}
};
```

EulerWalk.h

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

780b64, 15 lines

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

```
}
```

## 7.5 Coloring

### EdgeColoring.h

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

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

```
e210e2, 31 lines
vi edgeColoring(int N, vector<pii> eds) {
 vi cc(N + 1), ret(sz(eds)), fan(N), free(N), loc;
 for (pii e : eds) ++cc[e.first], ++cc[e.second];
 int u, v, ncols = *max_element(all(cc)) + 1;
 vector<vi> adj(N, vi(ncols, -1));
 for (pii e : eds) {
 tie(u, v) = e;
 fan[0] = v;
 loc.assign(ncols, 0);
 int at = u, end = u, d, c = free[u], ind = 0, i = 0;
 while (d = free[v], !loc[d] && (v = adj[u][d]) != -1)
 loc[d] = ++ind, cc[ind] = d, fan[ind] = v;
 cc[loc[d]] = c;
 for (int cd = d; at != -1; cd ^= c ^ d, at = adj[at][cd])
 swap(adj[at][cd], adj[end = at][cd ^ c ^ d]);
 while (adj[fan[i]][d] != -1) {
 int left = fan[i], right = fan[++i], e = cc[i];
 adj[u][e] = left;
 adj[left][e] = u;
 adj[right][e] = -1;
 free[right] = e;
 }
 adj[u][d] = fan[i];
 adj[fan[i]][d] = u;
 for (int y : {fan[0], u, end})
 for (int& z = free[y] = 0; adj[y][z] != -1; z++);
 }
 rep(i, 0, sz(eds))
 for (tie(u, v) = eds[i]; adj[u][ret[i]] != v;) ++ret[i];
 return ret;
}
```

## 7.6 Heuristics

### MaximalCliques.h

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

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

```
b0d5b1, 12 lines
typedef bitset<128> B;
template<class F>
void cliques(vector& 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];
 rep(i, 0, sz(eds)) if (cands[i]) {
 R[i] = 1;
 cliques(eds, f, P & eds[i], X & eds[i], R);
 R[i] = P[i] = 0; X[i] = 1;
 }
}
```

### MaximumClique.h

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

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

```
f7c0bc, 49 lines
typedef vector<bitset<200>> vb;
struct Maxclique {
 double limit=0.025, pk=0;
 struct Vertex { int i, d=0; };
 typedef vector<Vertex> vv;
 vb e;
 vv V;
 vector<vi> C;
 vi qmax, q, S, old;
 void init(vv& r) {
 for (auto& v : r) v.d = 0;
 for (auto& v : r) for (auto j : r) v.d += e[v.i][j.i];
 sort(all(r), [](auto a, auto b) { return a.d > b.d; });
 int mxD = r[0].d;
 rep(i, 0, sz(r)) r[i].d = min(i, mxD) + 1;
 }
 void expand(vv& R, int lev = 1) {
 S[lev] += S[lev - 1] - old[lev];
 old[lev] = S[lev - 1];
 while (sz(R)) {
 if (sz(q) + R.back().d <= sz(qmax)) return;
 q.push_back(R.back().i);
 vv T;
 for(auto v:R) if (e[R.back().i][v.i]) T.push_back({v.i});
 if (sz(T)) {
 if (S[lev]++ / ++pk < limit) init(T);
 int j = 0, mxk = 1, mnk = max(sz(qmax) - sz(q) + 1, 1);
 C[1].clear(), C[2].clear();
 for (auto v : T) {
 int k = 1;
 auto f = [&](int i) { return e[v.i][i]; };
 while (any_of(all(C[k]), f)) k++;
 if (k > mxk) mxk = k, C[mxk + 1].clear();
 if (k < mnk) T[j++].i = v.i;
 C[k].push_back(v.i);
 }
 if (j > 0) T[j - 1].d = 0;
 rep(k, mnk, mxk + 1) for (int i : C[k])
 T[j].i = i, T[j++].d = k;
 expand(T, lev + 1);
 } else if (sz(q) > sz(qmax)) qmax = q;
 q.pop_back(), R.pop_back();
 }
 }
 vi maxClique() { init(V), expand(V); return qmax; }
 Maxclique(vb conn) : e(conn), C(sz(e)+1), S(sz(C)), old(S) {
 rep(i, 0, sz(e)) V.push_back({i});
 }
};
```

### MaximumIndependentSet.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.

## 7.7 Trees

### BinaryLifting.h

**Description:** Calculate power of two jumps in a tree, to support fast upward jumps and LCAs. Assumes the root node points to itself.

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

```
bfce85, 25 lines
vector<vi> treeJump(vi& P) {
 int on = 1, d = 1;
 while (on < sz(P)) on *= 2, d++;
 vector<vi> jmp(d, P);
```

```
 rep(i, 1, d) rep(j, 0, sz(P))
 jmp[i][j] = jmp[i-1][jmp[i-1][j]];
 return jmp;
}

int jmp(vector<vi>& tbl, int nod, int steps) {
 rep(i, 0, sz(tbl))
 if (steps & (1 << i)) nod = tbl[i][nod];
 return nod;
}

int lca(vector<vi>& tbl, vi& depth, int a, int b) {
 if (depth[a] < depth[b]) swap(a, b);
 a = jmp(tbl, a, depth[a] - depth[b]);
 if (a == b) return a;
 for (int i = sz(tbl); i--;) {
 int c = tbl[i][a], d = tbl[i][b];
 if (c != d) a = c, b = d;
 }
 return tbl[0][a];
}
```

### LCA.h

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

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

```
0f62fb, 21 lines
".../data-structures/RMQ.h"
struct LCA {
 int T = 0;
 vi time, path, ret;
 RMQ<int> rmq;

 LCA(vector<vi>& C) : time(sz(C)), rmq((dfs(C, 0, -1), ret)) {}
 void dfs(vector<vi>& C, int v, int par) {
 time[v] = T++;
 for (int y : C[v]) if (y != par) {
 path.push_back(v), ret.push_back(time[v]);
 dfs(C, y, v);
 }
 }

 int lca(int a, int b) {
 if (a == b) return a;
 tie(a, b) = minmax(time[a], time[b]);
 return path[rmq.query(a, b)];
 }

 //dist(a,b){return depth[a] + depth[b] - 2*depth[lca(a,b)];}
};
```

### CompressTree.h

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

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

```
9775a0, 21 lines
"LCA.h"
typedef vector<pair<int, int>> vpi;
vpi compressTree(LCA& lca, const vi& subset) {
 static vi rev; rev.resize(sz(lca.time));
 vi li = subset, &T = lca.time;
 auto cmp = [&](int a, int b) { return T[a] < T[b]; };
 sort(all(li), cmp);
 int m = sz(li)-1;
 rep(i, 0, m) {
 int a = li[i], b = li[i+1];
 li.push_back(lca.lca(a, b));
 }
}
```



```

sort(all(li), cmp);
li.erase(unique(all(li)), li.end());
rep(i,0,sz(li)) rev[li[i]] = i;
vpi ret = {pii(0, li[0])};
rep(i,0,sz(li)-1) {
 int a = li[i], b = li[i+1];
 ret.emplace_back(rev[lca.lca(a, b)], b);
}
return ret;
}

```

## HLD.h

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

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

../data-structures/LazySegmentTree.h 6f34db, 46 lines

```

template <bool VALS_EDGES> struct HLD {
 int N, tim = 0;
 vector<vi> adj;
 vi par, siz, depth, rt, pos;
 Node *tree;
 HLD(vector<vi> adj_)
 : N(sz(adj_)), adj(adj_), par(N, -1), siz(N, 1), depth(N),
 rt(N), pos(N), tree(new Node(0, N)){ dfsSz(0); dfsHld(0); }
 void dfsSz(int v) {
 if (par[v] != -1) adj[v].erase(find(all(adj[v]), par[v]));
 for (int& u : adj[v]) {
 par[u] = v, depth[u] = depth[v] + 1;
 dfsSz(u);
 siz[v] += siz[u];
 if (siz[u] > siz[adj[v][0]]) swap(u, adj[v][0]);
 }
 }
 void dfsHld(int v) {
 pos[v] = tim++;
 for (int u : adj[v]) {
 rt[u] = (u == adj[v][0] ? rt[v] : u);
 dfsHld(u);
 }
 }
 template <class B> void process(int u, int v, B op) {
 for (; rt[u] != rt[v]; v = par[rt[v]]) {
 if (depth[rt[u]] > depth[rt[v]]) swap(u, v);
 op(pos[rt[v]], pos[v] + 1);
 }
 if (depth[u] > depth[v]) swap(u, v);
 op(pos[u] + VALS_EDGES, pos[v] + 1);
 }
 void modifyPath(int u, int v, int val) {
 process(u, v, [&](int l, int r) { tree->add(l, r, val); });
 }
 int queryPath(int u, int v) { // Modify depending on problem
 int res = -1e9;
 process(u, v, [&](int l, int r) {
 res = max(res, tree->query(l, r));
 });
 return res;
 }
 int querySubtree(int v) { // modifySubtree is similar
 return tree->query(pos[v] + VALS_EDGES, pos[v] + siz[v]);
 }
};

```

## LinkCutTree.h

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

**Time:** All operations take amortized  $\mathcal{O}(\log N)$ .

5909e2, 90 lines

```

struct Node { // Splay tree. Root's pp contains tree's parent.
 Node *p = 0, *pp = 0, *c[2];
 bool flip = 0;
 Node() { c[0] = c[1] = 0; fix(); }
 void fix() {
 if (c[0]) c[0]->p = this;
 if (c[1]) c[1]->p = this;
 // (+ update sum of subtree elements etc. if wanted)
 }
 void pushFlip() {
 if (!flip) return;
 flip = 0; swap(c[0], c[1]);
 if (c[0]) c[0]->flip ^= 1;
 if (c[1]) c[1]->flip ^= 1;
 }
 int up() { return p ? p->c[1] == this : -1; }
 void rot(int i, int b) {
 int h = i ^ b;
 Node *x = c[i], *y = b == 2 ? x : x->c[h], *z = b ? y : x;
 if ((y->p = p)) p->c[up()] = y;
 c[i] = z->c[i ^ 1];
 if (b < 2) {
 x->c[h] = y->c[h ^ 1];
 z->c[h ^ 1] = b ? x : this;
 }
 y->c[i ^ 1] = b ? this : x;
 fix(); x->fix(); y->fix();
 if (p) p->fix();
 swap(pp, y->pp);
 }
 void splay() {
 for (pushFlip(); p;) {
 if (p->p) p->p->pushFlip();
 p->pushFlip(); pushFlip();
 int c1 = up(), c2 = p->up();
 if (c2 == -1) p->rot(c1, 2);
 else p->p->rot(c2, c1 != c2);
 }
 }
 Node* first() {
 pushFlip();
 return c[0] ? c[0]->first() : (splay(), this);
 }
};

struct LinkCut {
 vector<Node> node;
 LinkCut(int N) : node(N) {}

 void link(int u, int v) { // add an edge (u, v)
 assert(!connected(u, v));
 makeRoot(&node[u]);
 node[u].pp = &node[v];
 }
 void cut(int u, int v) { // remove an edge (u, v)
 Node *x = &node[u], *top = &node[v];
 makeRoot(top); x->splay();
 assert(top == (x->pp ? x->c[0]));
 if (x->pp) x->pp = 0;
 else {
 x->c[0] = top->p = 0;
 x->fix();
 }
 }
};

```

```

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

```

## DirectedMST.h

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

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

../data-structures/UnionFindRollback.h 39e620, 60 lines

```

struct Edge { int a, b; ll w; };
struct Node {
 Edge key;
 Node *l, *r;
 ll delta;
 void prop() {
 key.w += delta;
 if (l) l->delta += delta;
 if (r) r->delta += delta;
 delta = 0;
 }
 Edge top() { prop(); return key; }
};
Node *merge(Node *a, Node *b) {
 if (!a || !b) return a ? b : a->prop(), b->prop();
 if (a->key.w > b->key.w) swap(a, b);
 swap(a->l, (a->r = merge(b, a->r)));
 return a;
}
void pop(Node&& a) { a->prop(); a = merge(a->l, a->r); }

```

```

pair<ll, vi> dmst(int n, int r, vector<Edge>& g) {
 RollbackUF uf(n);
 vector<Node*> heap(n);
 for (Edge e : g) heap[e.b] = merge(heap[e.b], new Node(e));
 ll res = 0;
 vi seen(n, -1), path(n), par(n);
 seen[r] = r;
 vector<Edge> Q(n), in(n, {-1, -1}), comp;
 deque<tuple<int, int, vector<Edge>>> cycs;
 rep(s, 0, n) {
 int u = s, qi = 0, w;
 while (seen[u] < 0) {
 if (!heap[u]) return {-1, {}};
 Edge e = heap[u]->top();

```

```
heap[u]->delta -= e.w, pop(heap[u]);
Q[qi] = e, path[qi++] = u, seen[u] = s;
res += e.w, u = uf.find(e.a);
if (seen[u] == s) {
 Node* cyc = 0;
 int end = qi, time = uf.time();
 do cyc = merge(cyc, heap[w = path[--qi]]);
 while (uf.join(u, w));
 u = uf.find(u), heap[u] = cyc, seen[u] = -1;
 cycs.push_front({u, time, {&Q[qi], &Q[end]}});
}
}
rep(i,0,qi) 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;
}
rep(i,0,n) par[i] = in[i].a;
return {res, par};
}
```

7.8 Math

7.8.1 Number of Spanning Trees

Create an  $N \times N$  matrix  $mat$ , 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).

7.8.2 Erdős–Gallai theorem

A simple graph with node degrees  $d_1 \geq \dots \geq d_n$  exists iff  $d_1 + \dots + d_n$  is even and for every  $k = 1 \dots n$ ,

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

Geometry (8)

8.1 Geometric primitives

Point.h

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

47ec0a, 28 lines

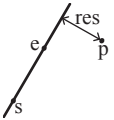
```
template <class T> int sgn(T x) { return (x > 0) - (x < 0); }
template<class T>
struct Point {
 typedef Point P;
 T x, y;
 explicit Point(T x=0, T y=0) : x(x), y(y) {}
 bool operator<(P p) const { return tie(x,y) < tie(p.x,p.y); }
 bool operator==(P p) const { return tie(x,y)==tie(p.x,p.y); }
 P operator+(P p) const { return P(x+p.x, y+p.y); }
 P operator-(P p) const { return P(x-p.x, y-p.y); }
 P operator*(T d) const { return P(x*d, y*d); }
 P operator/(T d) const { return P(x/d, y/d); }
 T dot(P p) const { return x*p.x + y*p.y; }
```

```
T cross(P p) const { return x*p.y - y*p.x; }
T cross(P a, P b) const { return (a-*this).cross(b-*this); }
T dist2() const { return x*x + y*y; }
double dist() const { return sqrt((double)dist2()); }
// angle to x-axis in interval [-pi, pi]
double angle() const { return atan2(y, x); }
P unit() const { return *this/dist(); } // makes dist()==1
P perp() const { return P(-y, x); } // rotates +90 degrees
P normal() const { return perp().unit(); }
// returns point rotated 'a' radians ccw around the origin
P rotate(double a) const {
 return P(x*cos(a)-y*sin(a),x*sin(a)+y*cos(a)); }
friend ostream& operator<<(ostream& os, P p) {
 return os << "(" << p.x << "," << p.y << ")"; }
};
```

lineDistance.h

Description:

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



f6bf6b, 4 lines

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

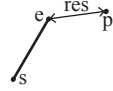
"Point.h"
```

SegmentDistance.h

Description:

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

Usage: Point<double> a, b(2,2), p(1,1);  
bool onSegment = segDist(a,b,p) < 1e-10;



5c88f4, 6 lines

```
"Point.h"

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

SegmentIntersection.h

Description:

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

Usage: vector<P> inter = segInter(s1,e1,s2,e2);

if (sz(inter)==1)  
cout << "segments intersect at " << inter[0] << endl;

9d57f2, 13 lines

```
"Point.h", "OnSegment.h"

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

```
if (onSegment(c, d, a)) s.insert(a);
if (onSegment(c, d, b)) s.insert(b);
if (onSegment(a, b, c)) s.insert(c);
if (onSegment(a, b, d)) s.insert(d);
return {all(s)};
}
```

lineIntersection.h

Description:

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

Usage: auto res = lineInter(s1,e1,s2,e2);  
if (res.first == 1)  
cout << "intersection point at " << res.second << endl;

a01f81, 8 lines

```
"Point.h"

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

sideOf.h

Description:

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

Usage: bool left = sideOf(p1,p2,q)==1;

3af81c, 9 lines

```
"Point.h"

template<class P>
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); }
}
```

OnSegment.h

Description:

Returns true iff  $p$  lies on the line segment from  $s$  to  $e$ . Use (segDist( $s,e,p$ )<=epsilon) instead when using Point<double>.

c597e8, 3 lines

```
"Point.h"

template<class P> bool onSegment(P s, P e, P p) {
 return p.cross(s, e) == 0 && (s - p).dot(e - p) <= 0; }
}
```

linearTransformation.h

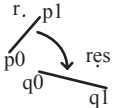
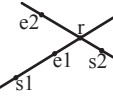
Description:

Apply the linear transformation (translation, rotation and scaling) which takes line  $p_0$ - $p_1$  to line  $q_0$ - $q_1$  to point  $r$ .

03a306, 6 lines

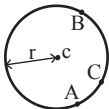
```
"Point.h"

typedef Point<double> P;
P linearTransformation(const P& p0, const P& p1,
 const P& q0, const P& q1, const P& r) {
 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(); }
}
```



## CircleTangents.h

**Description:** Computes the minimum circle that encloses a set of points.



```

"Point.h" 9706dc, 9 lines
typedef Point<double> P;
P polygonCenter(const vector<P>& v) {
 P res(0, 0); double A = 0;
 for (int i = 0, j = sz(v) - 1; i < sz(v); j = i++) {
 res = res + (v[i] + v[j]) * v[j].cross(v[i]);
 A += v[j].cross(v[i]);
 }
 return res / A / 3;
}

```

### PolygonCut.h

**Description:**  
Returns a vector with the vertices of a polygon with every-thing to the left of the line going from s to e cut away.

**Usage:** vector<P> p = ...;  
p = polygonCut(p, P(0,0), P(1,0));

"Point.h", "lineIntersection.h" f2b7d4, 13 lines

```
typedef Point<double> P;
vector<P> polygonCut(const vector<P>& poly, P s, P e) {
 vector<P> res;
 rep(i,0,sz(poly)) {
 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;
}
```

### ConvexHull.h

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

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

"Point.h" 310954, 13 lines

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

### HullDiameter.h

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

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

"Point.h" c571b8, 12 lines

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

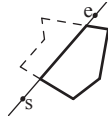
### PointInsideHull.h

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

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

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

```
typedef Point<ll> P;
```



f2b7d4, 13 lines



310954, 13 lines

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

### LineHullIntersection.h

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

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

"Point.h" 7cf45b, 39 lines

```
#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
template <class P> int extrVertex(vector<P>& poly, P dir) {
 int n = sz(poly), lo = 0, hi = n;
 if (extr(0)) return 0;
 while (lo + 1 < hi) {
 int m = (lo + hi) / 2;
 if (extr(m)) return m;
 int ls = cmp(lo + 1, lo), ms = cmp(m + 1, m);
 (ls < ms || (ls == ms && ls == cmp(lo, m)) ? hi : lo) = m;
 }
 return lo;
}
```

```
#define cmpL(i) sgn(a.cross(poly[i], b))
template <class P>
array<int, 2> lineHull(P a, P b, vector<P>& poly) {
 int endA = extrVertex(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;
 rep(i,0,2) {
 int lo = endB, hi = endA, n = sz(poly);
 while ((lo + 1) % n != hi) {
 int m = ((lo + hi + (lo < hi ? 0 : n)) / 2) % n;
 (cmpL(m) == cmpL(endB) ? lo : hi) = m;
 }
 res[i] = (lo + !cmpL(hi)) % n;
 swap(endA, endB);
 }
 if (res[0] == res[1]) return {res[0], -1};
 if (!cmpL(res[0]) && !cmpL(res[1]))
 switch ((res[0] - res[1] + sz(poly) + 1) % sz(poly)) {
 case 0: return {res[0], res[0]};
 case 2: return {res[1], res[1]};
 }
 return res;
}
```

## 8.4 Misc. Point Set Problems

### ClosestPair.h

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

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

"Point.h" ac41a6, 17 lines

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

### kdTree.h

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

"Point.h" bac5b0, 63 lines

```
typedef long long T;
typedef Point<T> P;
const T INF = numeric_limits<T>::max();
```

```
bool on_x(const P& a, const P& b) { return a.x < b.x; }
bool on_y(const P& a, const P& b) { return a.y < b.y; }
```

```
struct Node {
 P pt; // if this is a leaf, the single point in it
 T x0 = INF, x1 = -INF, y0 = INF, y1 = -INF; // bounds
 Node *first = 0, *second = 0;

 T distance(const P& p) { // min squared distance to a point
 T x = (p.x < x0 ? x0 : p.x > x1 ? x1 : p.x);
 T y = (p.y < y0 ? y0 : p.y > y1 ? y1 : p.y);
 return (P(x,y) - p).dist2();
 }
}
```

```
Node(vector<P>&& vp) : pt(vp[0]) {
 for (P p : vp) {
 x0 = min(x0, p.x); x1 = max(x1, p.x);
 y0 = min(y0, p.y); y1 = max(y1, p.y);
 }
 if (vp.size() > 1) {
 // split on x if width >= height (not ideal...)
 sort(all(vp), x1 - x0 >= y1 - y0 ? on_x : on_y);
 // divide by taking half the array for each child (not
 // best performance with many duplicates in the middle)
 int half = sz(vp)/2;
 first = new Node({vp.begin(), vp.begin() + half});
 second = new Node({vp.begin() + half, vp.end()});
 }
}
```

```
struct KDTree {
 Node* root;
 KDTree(const vector<P>& vp) : root(new Node({all(vp)})) {}

 pair<T, P> search(Node *node, const P& p) {
 if (!node->first) {
 // uncomment if we should not find the point itself:
```

```
// if (p == node->pt) return {INF, P()};
return make_pair((p - node->pt).dist2(), node->pt);
}

Node *f = node->first, *s = node->second;
T bfirst = f->distance(p), bsec = s->distance(p);
if (bfirst > bsec) swap(bsec, bfirst), swap(f, s);

// search closest side first, other side if needed
auto best = search(f, p);
if (bsec < best.first)
 best = min(best, search(s, p));
return best;
}

// find nearest point to a point, and its squared distance
// (requires an arbitrary operator< for Point)
pair<T, P> nearest(const P& p) {
 return search(root, p);
}

};
```

FastDelaunay.h

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

```
"Point.h" eefdf5, 88 lines

typedef Point<ll> P;
typedef struct Quad* Q;
typedef __int128_t ll1; // (can be ll if coords are < 2e4)
P arb(LLONG_MAX,LLONG_MAX); // not equal to any other point
```

```
struct Quad {
 Q rot, o; P p = arb; bool mark;
 P& F() { return r()->p; }
 Q& r() { return rot->rot; }
 Q prev() { return rot->o->rot; }
 Q next() { return r()->prev(); }
} *H;

bool circ(P p, P a, P b, P c) { // is p in the circumcircle?
 ll1 p2 = p.dist2(), A = a.dist2()-p2,
 B = b.dist2()-p2, C = c.dist2()-p2;
 return p.cross(a,b)*C + p.cross(b,c)*A + p.cross(c,a)*B > 0;
}

Q makeEdge(P orig, P dest) {
 Q r = H ? H : new Quad{new Quad{new Quad{new Quad{0}}}};
 H = r->o; r->r()->r() = r;
 rep(i,0,4) r = r->rot, r->p = arb, r->o = i & 1 ? r : r->r();
 r->p = orig; r->F() = dest;
 return r;
}

void splice(Q a, Q b) {
 swap(a->o->rot->o, b->o->rot->o); swap(a->o, b->o);
}

Q connect(Q a, Q b) {
 Q q = makeEdge(a->F(), b->p);
 splice(q, a->next());
 splice(q->r(), b);
 return q;
}

pair<Q,Q> rec(const vector<P>& s) {
 if (sz(s) <= 3) {
 Q a = makeEdge(s[0], s[1]), b = makeEdge(s[1], s.back());
 if (sz(s) == 2) return { a, a->r() };
 }
```

FastDelaunay PolyhedronVolume Point3D 3dHull

```
splice(a->r(), b);
auto side = s[0].cross(s[1], s[2]);
Q c = side ? connect(b, a) : 0;
return {side < 0 ? c->r() : a, side < 0 ? c : b->r() };
}

#define H(e) e->F(), e->p
#define valid(e) (e->F().cross(H(base)) > 0)
Q A, B, ra, rb;
int half = sz(s) / 2;
tie(ra, A) = rec({all(s) - half});
tie(B, rb) = rec({sz(s) - half + all(s)});
while ((B->p.cross(H(A)) < 0 && (A = A->next()) ||
 (A->p.cross(H(B)) > 0 && (B = B->r()->o)));
Q base = connect(B->r(), A);
if (A->p == ra->p) ra = base->r();
if (B->p == rb->p) rb = base;

#define DEL(e, init, dir) Q e = init->dir; if (valid(e)) \
 while (circ(e->dir->F(), H(base), e->F())) { \
 Q t = e->dir; \
 splice(e, e->prev()); \
 splice(e->r(), e->r()->prev()); \
 e->o = H; H = e; e = t; \
 }
for (;;) {
 DEL(LC, base->r(), o); DEL(RC, base, prev());
 if (!valid(LC) && !valid(RC)) break;
 if (!valid(LC) || (valid(RC) && circ(H(RC), H(LC))))
 base = connect(RC, base->r());
 else
 base = connect(base->r(), LC->r());
}
return { ra, rb };
}
```

```
vector<P> triangulate(vector<P> pts) {
 sort(all(pts)); assert(unique(all(pts)) == pts.end());
 if (sz(pts) < 2) return {};
 Q e = rec(pts).first;
 vector<Q> q = {e};
 int qi = 0;
 while (e->o->F().cross(e->F(), e->p) < 0) e = e->o;
 #define ADD { Q c = e; do { c->mark = 1; pts.push_back(c->p); \
 q.push_back(c->r()); c = c->next(); } while (c != e); }
 ADD; pts.clear();
 while (qi < sz(q)) if (!(e = q[qi++])->mark) ADD;
 return pts;
}
```

8.5 3D

PolyhedronVolume.h

**Description:** Magic formula for the volume of a polyhedron. Faces should point outwards.

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

Point3D.h

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

```
template<class T> struct Point3D {
 typedef Point3D P;
 typedef const P& R;
```

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

3dHull.h

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

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

```
"Point3D.h" 5b45fc, 49 lines

typedef Point3D<double> P3;
```

```
struct PR {
 void ins(int x) { (a == -1 ? a : b) = x; }
 void rem(int x) { (a == x ? a : b) = -1; }
 int cnt() { return (a != -1) + (b != -1); }
 int a, b;
};
```

```
struct F { P3 q; int a, b, c; };
```

```
vector<F> hull3d(const vector<P3>& A) {
 assert(sz(A) >= 4);
 vector<vector<PR>> E(sz(A), vector<PR>(sz(A), {-1, -1}));
 #define E(x,y) E[f.x][f.y]
 vector<F> FS;
 auto mf = [&](int i, int j, int k, int l) {
 P3 q = (A[j] - A[i]).cross((A[k] - A[i]));
 if (q.dot(A[l]) > q.dot(A[i]))
 q = q * -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);
 };
 rep(i,0,4) rep(j,i+1,4) rep(k,j+1,4)
 mf(i, j, k, 6 - i - j - k);

 rep(i,4,sz(A)) {
 rep(j,0,sz(FS)) {
 F f = FS[j];
 if (f.q.dot(A[i]) > f.q.dot(A[f.a])) {
 E(a,b).rem(f.c);
 E(a,c).rem(f.b);
```

```

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

```

### sphericalDistance.h

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

```

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

```

## Strings (9)

#### KMP.h

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

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

```

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

```

```

vi match(const string& s, const string& pat) {
 vi p = pi(pat + '\0' + s), res;
 rep(i,sz(p)-sz(s),sz(p))
 if (p[i] == sz(pat)) res.push_back(i - 2 * sz(pat));
 return res;
}

```

#### Zfunc.h

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

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

```

vi Z(const string& S) {
 vi z(sz(S));

```

```

 int l = -1, r = -1;
 rep(i,1,sz(S)) {
 z[i] = i >= r ? 0 : min(r - i, z[i - 1]);
 while (i + z[i] < sz(S) && S[i + z[i]] == S[z[i]])
 z[i]++;
 if (i + z[i] > r)
 l = i, r = i + z[i];
 }
 return z;
}

```

#### Manacher.h

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

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

```

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

```

#### MinRotation.h

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

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

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

```

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

```

#### SuffixArray.h

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

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

```

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

```

```

 }
 rep(i,1,n) rank[sa[i]] = i;
 for (int i = 0, j; i < n - 1; lcp[rank[i++]] = k)
 for (k && k--, j = sa[rank[i] - 1];
 s[i + k] == s[j + k]; k++);
 }
};

```

#### SuffixTree.h

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

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

```

struct SuffixTree {
 enum { N = 200010, ALPHA = 26 }; // N ~ 2*maxlen+10
 int toi(char c) { return c - 'a'; }
 string a; // v = cur node, q = cur position
 int t[N][ALPHA], l[N], r[N], p[N], s[N], v=0, q=0, m=2;

 void ukkadd(int i, int c) { suff:
 if (r[v]<=q) {
 if (t[v][c]==-1) { t[v][c]=m; l[m]=i;
 p[m++]=v; v=s[v]; q=r[v]; goto suff; }
 v=t[v][c]; q=l[v];
 }
 if (q==-1 || c==toi(a[q])) q++; else {
 l[m+1]=i; p[m+1]=m; l[m]=l[v]; r[m]=q;
 p[m]=p[v]; t[m][c]=m+1; t[m][toi(a[q])]=v;
 l[v]=q; p[v]=m; t[p[m]][toi(a[l[m]])]=m;
 v=s[p[m]]; q=l[m];
 while (q<r[m]) { v=t[v][toi(a[q])]; q+=r[v]-l[v]; }
 if (q==r[m]) s[m]=v; else s[m]=m+2;
 q=r[v]-(q-r[m]); m+=2; goto suff;
 }
 }
}

```

```

SuffixTree(string a) : a(a) {
 fill(r,r+N,sz(a));
 memset(s, 0, sizeof s);
 memset(t, -1, sizeof t);
 fill(t[1],t[1]+ALPHA,0);
 s[0] = 1; l[0] = l[1] = -1; r[0] = r[1] = p[0] = p[1] = 0;
 rep(i,0,sz(a)) ukkadd(i, toi(a[i]));
}

```

```

// example: find longest common substring (uses ALPHA = 28)
pii best;
int lcs(int node, int i1, int i2, int olen) {
 if (l[node] <= i1 && i1 < r[node]) return 1;
 if (l[node] <= i2 && i2 < r[node]) return 2;
 int mask = 0, len = node ? olen + (r[node] - l[node]) : 0;
 rep(c,0,ALPHA) if (t[node][c] != -1)
 mask |= lcs(t[node][c], i1, i2, len);
 if (mask == 3)
 best = max(best, {len, r[node] - len});
 return mask;
}
static pii LCS(string s, string t) {
 SuffixTree st(s + (char)('z' + 1) + t + (char)('z' + 2));
 st.lcs(0, sz(s), sz(s) + 1 + sz(t), 0);
 return st.best;
}
};

```

2d2a67, 44 lines

```
vector<H> getHashes(string& str, int length) {
 if (sz(str) < length) return {};
 H h = 0, pw = 1;
 rep(i, 0, length)
 h = h * C + str[i], pw = pw * C;
 vector<H> ret = {h};
 rep(i, length, sz(str)) {
 ret.push_back(h = h * C + str[i] - pw * str[i-length]);
 }
 return ret;
}

H hashString(string& s){H h{}; for(char c:s) h=h*C+c;return h;}
```

## f35677, 66 lines

edce47, 23 lines

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

9e9d8d, 19 lines

753a4c, 19 lines

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



```

 }
}

template<class F, class G>
void constantIntervals(int from, int to, F f, G g) {
 if (to <= from) return;
 int i = from; auto p = f(i), q = f(to-1);
 rec(from, to-1, f, g, i, p, q);
 g(i, to, q);
}

```

10.2 Misc. algorithms

TernarySearch.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).  
**Usage:** `int ind = ternSearch(0, n-1, [&](int i){return a[i];});`  
**Time:**  $\mathcal{O}(\log(b-a))$

9155b4, 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;
 }
 rep(i, a+1, b+1) if (f(a) < f(i)) a = i; // (B)
 return a;
}

```

LIS.h

**Description:** Compute indices for the longest increasing subsequence.  
**Time:**  $\mathcal{O}(N \log N)$

2932a0, 17 lines

```

template<class I> vi lis(const vector<I>& S) {
 if (S.empty()) return {};
 vi prev(sz(S));
 typedef pair<I, int> p;
 vector<p> res;
 rep(i, 0, sz(S)) {
 // change 0 -> i for longest non-decreasing subsequence
 auto it = lower_bound(all(res), p{S[i], 0});
 if (it == res.end()) res.emplace_back(), it = res.end()-1;
 *it = {S[i], i};
 prev[i] = it == res.begin() ? 0 : (it-1)->second;
 }
 int L = sz(res), cur = res.back().second;
 vi ans(L);
 while (L--) ans[L] = cur, cur = prev[cur];
 return ans;
}

```

FastKnapsack.h

**Description:** Given  $N$  non-negative integer weights  $w$  and a non-negative target  $t$ , computes the maximum  $S \leq t$  such that  $S$  is the sum of some subset of the weights.  
**Time:**  $\mathcal{O}(N \max(w_i))$

b20ccc, 16 lines

```

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

```

```

 for (x = 2*m; --x > m;) rep(j, max(0, u[x]), v[x])
 v[x-w[j]] = max(v[x-w[j]], j);
 }
 for (a = t; v[a+m-t] < 0; a--) ;
 return a;
}

```

10.3 Dynamic programming

KnuthDP.h

**Description:** When doing DP on intervals:  $a[i][j] = \min_{i < k < j} (a[i][k] + a[k][j]) + f(i, j)$ , where the (minimal) optimal  $k$  increases with both  $i$  and  $j$ , one can solve intervals in increasing order of length, and search  $k = p[i][j]$  for  $a[i][j]$  only between  $p[i][j-1]$  and  $p[i+1][j]$ . This is known as Knuth DP. Sufficient criteria for this are if  $f(b, c) \leq f(a, d)$  and  $f(a, c) + f(b, d) \leq f(a, d) + f(b, c)$  for all  $a \leq b \leq c \leq d$ . Consider also: LineContainer (ch. Data structures), monotone queues, ternary search.  
**Time:**  $\mathcal{O}(N^2)$

d38d2b, 18 lines

```

struct DP { // Modify at will:
 int lo(int ind) { return 0; }
 int hi(int ind) { return ind; }
 ll f(int ind, int k) { return dp[ind][k]; }
 void store(int ind, int k, ll v) { res[ind] = pii(k, v); }

 void rec(int L, int R, int LO, int HI) {
 if (L >= R) return;
 int mid = (L + R) >> 1;
 pair<ll, int> best (LLONG_MAX, LO);
 rep(k, max(LO, lo(mid)), min(HI, hi(mid)))
 best = min(best, make_pair(f(mid, k), k));
 store(mid, best.second, best.first);
 rec(L, mid, LO, best.second+1);
 rec(mid+1, R, best.second, HI);
 }
 void solve(int L, int R) { rec(L, R, INT_MIN, INT_MAX); }
};

```

ConvexHullDP.h

**Description:** When doing DP, consider cases where  $DP[i]$  is a function of  $i$  and  $\min_{j < i} (f_j(q_i))$ , where the  $f_j$ 's belong to a family of functions where any two intersect at most once (i.e. lines). For any two functions  $f$  and  $g$  in this family, call  $f$  steeper than  $g$  if for some  $x_0$ ,  $f(x) > g(x)$  for all  $x > x_0$ . Instead of computing  $f_j(i)$  for all  $j$  and all  $i$ , add the  $f_j$ 's to a data structure and use that structure to evaluate  $\min_{j < i} (f_j(q_i))$ . If the  $f_j$ 's are sorted by slope and the  $q_i$ 's are sorted, this can be done in  $\mathcal{O}(N)$ . Otherwise, this becomes  $\mathcal{O}(N \log N)$  using a LineContainer or LiChaoTree. Naively, it would be  $\mathcal{O}(N^2)$ .  
**Time:**  $\mathcal{O}(N)$  or  $\mathcal{O}(N \log N)$

AliensTrick.h

**Description:** When minimizing some cost where  $k$  operations are allowed, let  $f(k)$  be the optimal answer. If  $f(k)$  is convex, instead minimize the cost where  $k$  can vary, but there is some possibly negative added cost lambda per operation. Find the minimum lambda such that the smallest value of  $k$  that achieves the minimum is at most the actual value of  $k$  from the problem. (maybe binary search on lambda?) The answer will be the cost this lambda value achieves minus  $k$  times lambda. If  $f$  is strictly convex or the problem lets you use fewer than  $k$  operations, this minimum will be achieved at a correct  $k$  value, so you can reconstruct the minimizing solution (it's the same as the modified optimization problem's solution).  
**Time:**  $\mathcal{O}(\log N)$

2445de, 8 lines

```

ll k; // real operation count
ll lo, hi; // bounds on lambda (maybe double)
while (lo < hi) {
 ll lam = (lo + hi) / 2;
 auto [cost, k2] = solve(lam);
 if (k2 > k) lo = lam + 1;
 else ans = cost - (hi = lam) * k;
}

```

10.4 Debugging tricks

- `signal(SIGSEGV, [](int) { _Exit(0); });` converts segfaults into Wrong Answers. Similarly one can catch SIGABRT (assertion failures) and SIGFPE (zero divisions). `_GLIBCXX_DEBUG` failures generate SIGABRT (or SIGSEGV on gcc 5.4.0 apparently).
- `feenableexcept(29);` kills the program on NaNs (1), 0-divs (4), infinities (8) and denormals (16).

10.5 Optimization tricks

`__builtin_ia32_ldmxcsr(40896);` disables denormals (which make floats 20x slower near their minimum value).

10.5.1 Bit hacks

- `x & -x` is the least bit in  $x$ .
- `for (int x = m; x; ) { --x &= m; ... }` loops over all subset masks of  $m$  (except  $m$  itself).
- `c = x&-x, r = x+c; ((r^x) >> 2)/c | r` is the next number after  $x$  with the same number of bits set.
- `rep(b, 0, K) rep(i, 0, (1 << K))` if `(i & 1 << b) D[i] += D[i^(1 << b)];` computes all sums of subsets.

10.5.2 Pragmas

- `#pragma GCC optimize ("Ofast")` will make GCC auto-vectorize loops and optimizes floating points better.
- `#pragma GCC target ("avx2")` can double performance of vectorized code, but causes crashes on old machines.
- `#pragma GCC optimize ("trapv")` kills the program on integer overflows (but is really slow).

FastMod.h

**Description:** Compute  $a \% b$  about 5 times faster than usual, where  $b$  is constant but not known at compile time. Returns a value congruent to  $a \pmod b$  in the range  $[0, 2b)$ .

751a02, 8 lines

```

typedef unsigned long long ull;
struct FastMod {
 ull b, m;
 FastMod(ull b) : b(b), m(-1ULL / b) {}
 ull reduce(ull a) { // a % b + (0 or b)
 return a - (ull)((__uint128_t(m) * a) >> 64) * b;
 }
};

```



FastInput.h

**Description:** Read an integer from stdin. Usage requires your program to pipe in input from file.

**Usage:** ./a.out < input.txt

**Time:** About 5x as fast as cin/scanf.

---

```
inline char gc() { // like getchar()
 static char buf[1 << 16];
 static size_t bc, be;
 if (bc >= be) {
 buf[0] = 0, bc = 0;
 be = fread(buf, 1, sizeof(buf), stdin);
 }
 return buf[bc++]; // returns 0 on EOF
}
```

```
int readInt() {
 int a, c;
 while ((a = gc()) < 40);
 if (a == '-') return -readInt();
 while ((c = gc()) >= 48) a = a * 10 + c - 480;
 return a - 48;
}
```

BumpAllocator.h

**Description:** When you need to dynamically allocate many objects and don't care about freeing them. "new X" otherwise has an overhead of something like 0.05us + 16 bytes per allocation.

---

```
// Either globally or in a single class:
static char buf[450 << 20];
void* operator new(size_t s) {
 static size_t i = sizeof(buf);
 assert(s < i);
 return (void*)&buf[i -= s];
}
void operator delete(void*) {}
```

SmallPtr.h

**Description:** A 32-bit pointer that points into BumpAllocator memory.

---

```
template<class T> struct ptr {
 unsigned ind;
 ptr(T* p = 0) : ind(p ? unsigned((char*)p - buf) : 0) {
 assert(ind < sizeof(buf));
 }
 T& operator*() const { return *(T*)(buf + ind); }
 T* operator->() const { return &*this; }
 T& operator[](int a) const { return (&*this)[a]; }
 explicit operator bool() const { return ind; }
};
```

BumpAllocatorSTL.h

**Description:** BumpAllocator for STL containers.

**Usage:** vector<vector<int, small<int>>>> ed(N);

---

```
char buf[450 << 20] alignas(16);
size_t buf_ind = sizeof(buf);

template<class T> struct small {
 typedef T value_type;
 small() {}
 template<class U> small(const U&) {}
 T* allocate(size_t n) {
 buf_ind -= n * sizeof(T);
 buf_ind &= 0 - alignof(T);
 return (T*)(buf + buf_ind);
 }
 void deallocate(T*, size_t) {}
};
```

```
};

SIMD.h
Description: Cheat sheet of SSE/AVX intrinsics, for doing arithmetic on several numbers at once. Can provide a constant factor improvement of about 4, orthogonal to loop unrolling. Operations follow the pattern "_mm(256)?_name_(si(128|256)|epi(8|16|32|64)|pd|ps)". Not all are described here; grep for _mm_ in /usr/lib/gcc/*/4.9/include/ for more. If AVX is unsupported, try 128-bit operations, "emmintrin.h" and #define __SSE__ and __MMX__ before including it. For aligned memory use _mm_malloc(size, 32) or int buf[N] alignas(32), but prefer loadu/storeu.

#pragma GCC target ("avx2") // or sse4.1
#include "emmintrin.h"
```

```
typedef __m256i mi;
#define L(x) _mm256_loadu_si256((mi*)&(x))

// High-level/specific methods:
// load(u)?_si256, store(u)?_si256, setzero_si256, _mm_malloc
// blendv(epi8|ps|pd) (z?y:x), movemask_epi8 (hibits of bytes)
// i32gather_epi32(addr, x, 4): map addr[] over 32-b parts of x
// sad_epu8: sum of absolute differences of u8, outputs 4xi64
// maddubs_epi16: dot product of unsigned i7's, outputs 16xi15
// madd_epi16: dot product of signed i16's, outputs 8xi32
// extractf128_si256(, i) (256->128), cvtsi128_si32 (128->lo32)
// permute2f128_si256(x,x,1) swaps 128-bit lanes
// shuffle_epi32(x, 3*64+2*16+1*4+0) == x for each lane
// shuffle_epi8(x, y) takes a vector instead of an imm

// Methods that work with most data types (append e.g. _epi32):
// set1, blend (i8?x:y), add, adds (sat.), mullo, sub, and/or,
// andnot, abs, min, max, sign(1,x), cmp(gt|eq), unpack(lo|hi)
```

```
int sumi32(mi m) { union {int v[8]; mi m;} u; u.m = m;
 int ret = 0; rep(i,0,8) ret += u.v[i]; return ret; }
mi zero() { return _mm256_setzero_si256(); }
mi one() { return _mm256_set1_epi32(-1); }
bool all_zero(mi m) { return _mm256_testz_si256(m, m); }
bool all_one(mi m) { return _mm256_testc_si256(m, one()); }
```

```
ll example_filteredDotProduct(int n, short* a, short* b) {
 int i = 0; ll r = 0;
 mi zero = _mm256_setzero_si256(), acc = zero;
 while (i + 16 <= n) {
 mi va = L(a[i]), vb = L(b[i]); i += 16;
 va = _mm256_and_si256(_mm256_cmpgt_epi16(vb, va), va);
 mi vp = _mm256_madd_epil6(va, vb);
 acc = _mm256_add_epi64(_mm256_unpacklo_epi32(vp, zero),
 _mm256_add_epi64(acc, _mm256_unpackhi_epi32(vp, zero)));
 }
 union {ll v[4]; mi m;} u; u.m = acc; rep(i,0,4) r += u.v[i];
 for (;i<n;++i) if (a[i] < b[i]) r += a[i]*b[i]; //<- equiv
 return r;
}
```