

## Contest (1)

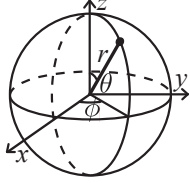
template.cpp	15 lines
<pre>#include &lt;bits/stdc++.h&gt; using namespace std;  #define rep(i, a, b) for(int i = a; i &lt; (b); ++i) #define trav(a, x) for(auto&amp; a : x) #define all(x) x.begin(), x.end() #define sz(x) (int)(x).size() typedef long long ll; typedef pair&lt;int, int&gt; pii; typedef vector&lt;int&gt; vi;  int main() {     cin.sync_with_stdio(0); cin.tie(0);     cin.exceptions(cin.failbit); }</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' #caps =◇</pre>	
.vimrc	2 lines
<pre>set cin aw ai is ts=4 sw=4 tm=50 nu noeb bg=dark ru cul sy on   im jk &lt;esc&gt;   im kj &lt;esc&gt;   no ; :</pre>	
troubleshoot.txt	52 lines
<p>Pre-submit:</p> <p>Write a few simple test cases, if sample is not enough.</p> <p>Are time limits close? If so, generate max cases.</p> <p>Is the memory usage fine?</p> <p>Could anything overflow?</p> <p>Make sure to submit the right file.</p> <p>Wrong answer:</p> <p>Print your solution! Print debug output, as well.</p> <p>Are you clearing all datastructures between test cases?</p> <p>Can your algorithm handle the whole range of input?</p> <p>Read the full problem statement again.</p> <p>Do you handle all corner cases correctly?</p> <p>Have you understood the problem correctly?</p> <p>Any uninitialized variables?</p> <p>Any overflows?</p> <p>Confusing N and M, i and j, etc.?</p> <p>Are you sure your algorithm works?</p> <p>What special cases have you not thought of?</p> <p>Are you sure the STL functions you use work as you think?</p> <p>Add some assertions, maybe resubmit.</p> <p>Create some testcases to run your algorithm on.</p> <p>Go through the algorithm for a simple case.</p> <p>Go through this list again.</p> <p>Explain your algorithm to a team mate.</p> <p>Ask the team mate to look at your code.</p> <p>Go for a small walk, e.g. to the toilet.</p> <p>Is your output format correct? (including whitespace)</p> <p>Rewrite your solution from the start or let a team mate do it.</p> <p>Runtime error:</p> <p>Have you tested all corner cases locally?</p> <p>Any uninitialized variables?</p> <p>Are you reading or writing outside the range of any vector?</p> <p>Any assertions that might fail?</p> <p>Any possible division by 0? (mod 0 for example)</p>	

<p>Any possible infinite recursion?</p> <p>Invalidated pointers or iterators?</p> <p>Are you using too much memory?</p> <p>Debug with resubmits (e.g. remapped signals, see Various).</p> <p>Time limit exceeded:</p> <p>Do you have any possible infinite loops?</p> <p>What is the complexity of your algorithm?</p> <p>Are you copying a lot of unnecessary data? (References)</p> <p>How big is the input and output? (consider scanf)</p> <p>Avoid vector, map. (use arrays/unordered_map)</p> <p>What do your team mates think about your algorithm?</p> <p>Memory limit exceeded:</p> <p>What is the max amount of memory your algorithm should need?</p> <p>Are you clearing all datastructures between test cases?</p>	
<h2>Mathematics (2)</h2>	
<h3>2.1 Equations</h3>	
$ax^2 + bx + c = 0 \Rightarrow x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$	
<p>The extremum is given by <math>x = -b/2a</math>.</p>	
$\begin{matrix} ax + by = e \\ cx + dy = f \end{matrix} \Rightarrow \begin{matrix} x = \frac{ed - bf}{ad - bc} \\ y = \frac{af - ec}{ad - bc} \end{matrix}$	
<p>In general, given an equation <math>Ax = b</math>, the solution to a variable <math>x_i</math> is given by</p> $x_i = \frac{\det A'_i}{\det A}$	
<p>where <math>A'_i</math> is <math>A</math> with the <math>i</math>'th column replaced by <math>b</math>.</p>	
<h3>2.2 Recurrences</h3>	
<p>If <math>a_n = c_1a_{n-1} + \dots + c_ka_{n-k}</math>, and <math>r_1, \dots, r_k</math> are distinct roots of <math>x^k + c_1x^{k-1} + \dots + c_k</math>, there are <math>d_1, \dots, d_k</math> s.t.</p> $a_n = d_1r_1^n + \dots + d_kr_k^n.$	
<p>Non-distinct roots <math>r</math> become polynomial factors, e.g.</p> $a_n = (d_1n + d_2)r^n.$	
<h3>2.3 Trigonometry</h3>	
$\sin(v + w) = \sin v \cos w + \cos v \sin w$ $\cos(v + w) = \cos v \cos w - \sin v \sin w$	

$\tan(v + w) = \frac{\tan v + \tan w}{1 - \tan v \tan w}$ $\sin v + \sin w = 2 \sin \frac{v + w}{2} \cos \frac{v - w}{2}$ $\cos v + \cos w = 2 \cos \frac{v + w}{2} \cos \frac{v - w}{2}$	
$(V + W) \tan(v - w)/2 = (V - W) \tan(v + w)/2$	
<p>where <math>V, W</math> are lengths of sides opposite angles <math>v, w</math>.</p>	
$a \cos x + b \sin x = r \cos(x - \phi)$ $a \sin x + b \cos x = r \sin(x + \phi)$	
<p>where <math>r = \sqrt{a^2 + b^2}, \phi = \operatorname{atan2}(b, a)</math>.</p>	
$\theta = 2 \operatorname{atan2}((v_1 \times v_2) \cdot \hat{z}, \ v_1\  \ v_2\  + v_1 \cdot v_2).$ This gives a signed angle!!!	
<h3>2.4 Geometry</h3>	
<h4>2.4.1 Triangles</h4>	
<p>Side lengths: <math>a, b, c</math></p> <p>Semiperimeter: <math>p = \frac{a + b + c}{2}</math></p> <p>Area: <math>A = \sqrt{p(p - a)(p - b)(p - c)}</math></p> <p>Circumradius: <math>R = \frac{abc}{4A}</math></p> <p>Inradius: <math>r = \frac{A}{p}</math></p> <p>Length of median (divides triangle into two equal-area triangles): <math>m_a = \frac{1}{2}\sqrt{2b^2 + 2c^2 - a^2}</math></p> <p>Length of bisector (divides angles in two):</p> $s_a = \sqrt{bc \left[ 1 - \left( \frac{a}{b + c} \right)^2 \right]}$	
<p>Law of sines: <math>\frac{\sin \alpha}{a} = \frac{\sin \beta}{b} = \frac{\sin \gamma}{c} = \frac{1}{2R}</math></p> <p>Law of cosines: <math>a^2 = b^2 + c^2 - 2bc \cos \alpha</math></p> <p>Law of tangents: <math>\frac{a + b}{a - b} = \frac{\tan \frac{\alpha + \beta}{2}}{\tan \frac{\alpha - \beta}{2}}</math></p>	
<h4>2.4.2 Quadrilaterals</h4>	
<p>With side lengths <math>a, b, c, d</math>, diagonals <math>e, f</math>, diagonals angle <math>\theta</math>, area <math>A</math> and magic flux <math>F = b^2 + d^2 - a^2 - c^2</math>:</p> $4A = 2ef \cdot \sin \theta = F \tan \theta = \sqrt{4e^2f^2 - F^2}$	

For cyclic quadrilaterals the sum of opposite angles is  $180^\circ$ ,  $ef = ac + bd$ , 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

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

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

## 2.7 Series

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

## 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  $\sigma$  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 which 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  $\kappa$  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  $\mu$  and variance  $\sigma^2$  are well described by  $\mathcal{N}(\mu, \sigma^2)$ ,  $\sigma > 0$ .

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

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

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

2.9 Markov chains

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

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

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

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

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

Data structures (3)

OrderStatisticTree.h

**Description:** A set (not multiset!) with support for finding the n'th element, and finding the index of an element.

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

```
#include <bits/extc++.h>
using namespace __gnu_pbds;

template <class T>
using Tree = tree<T, null_type, less<T>, rb_tree_tag,
tree_order_statistics_node_update>;

void example() {
    Tree<int> t, t2; t.insert(8);
    auto it = t.insert(10).first;
    assert(it == t.lower_bound(9));
    assert(t.order_of_key(10) == 1);
    assert(t.order_of_key(11) == 2);
    assert(*t.find_by_order(0) == 8);
    t.join(t2); // assuming T< T2 or T> T2, merge t2 into t
}
```

SegmentTree.h

**Description:** Zero-indexed max-tree. Bounds are inclusive to the left and exclusive to the right. Can be changed by modifying T, LOW and f.

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

```
struct Tree {
    typedef int T;
    const T LOW = -1234567890;
    T f(T a, T b) { return max(a, b); }

    int n;
    vi s;
    Tree() {}
    Tree(int m, T def=0) { init(m, def); }
    void init(int m, T def) {
        n = 1; while (n < m) n *= 2;
        s.assign(n + m, def);
        s.resize(2 * n, LOW);
        for (int i = n; i --> 1; )
            s[i] = f(s[i * 2], s[i * 2 + 1]);
    }
    void update(int pos, T val) {
        pos += n;
        s[pos] = val;
        for (pos /= 2; pos >= 1; pos /= 2)
            s[pos] = f(s[pos * 2], s[pos * 2 + 1]);
    }
    T query(int l, int r) { return que(l, l, r, 0, n); }
    T que(int pos, int l, int r, int lo, int hi) {
        if (r <= lo || hi <= l) return LOW;
        if (l <= lo && hi <= r) return s[pos];
        int m = (lo + hi) / 2;
        return f(que(2 * pos, l, r, lo, m),
            que(2 * pos + 1, l, r, m, hi));
    }
};
```

LazySegmentTree.h

**Description:** Segment tree with ability to add or set values of large intervals, and compute max of intervals. Can be changed to other things. Use with a bump allocator for better performance, and SmallPtr or implicit indices to save memory.

**Usage:** Node\* tr = new Node(v, 0, sz(v));

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

```
"../various/BumpAllocator.h"
const int inf = 1e9;
```

struct Node {

```
Node *l = 0, *r = 0;
int lo, hi, mset = inf, madd = 0, val = -inf;
Node(int lo, int hi):lo(lo),hi(hi){} // Large interval of -inf
Node(vi& v, int lo, int hi) : lo(lo), hi(hi) {
    if (lo + 1 < hi) {
        int mid = lo + (hi - lo)/2;
        l = new Node(v, lo, mid); r = new Node(v, mid, hi);
        val = max(l->val, r->val);
    }
    else val = v[lo];
}

int query(int L, int R) {
    if (R <= lo || hi <= L) return -inf;
    if (L <= lo && hi <= R) return val;
    push();
    return max(l->query(L, R), r->query(L, R));
}

void set(int L, int R, int x) {
    if (R <= lo || hi <= L) return;
    if (L <= lo && hi <= R) mset = val = x, madd = 0;
    else {
        push(), l->set(L, R, x), r->set(L, R, x);
        val = max(l->val, r->val);
    }
}

void add(int L, int R, int x) {
    if (R <= lo || hi <= L) return;
    if (L <= lo && hi <= R) {
        if (mset != inf) mset += x;
        else madd += x;
        val += x;
    }
    else {
        push(), l->add(L, R, x), r->add(L, R, x);
        val = max(l->val, r->val);
    }
}

void push() {
    if (!l) {
        int mid = lo + (hi - lo)/2;
        l = new Node(lo, mid); r = new Node(mid, hi);
    }
    if (mset != inf)
        l->set(lo,hi,mset), r->set(lo,hi,mset), mset = inf;
    else if (madd)
        l->add(lo,hi,madd), r->add(lo,hi,madd), madd = 0;
}
};
```

UnionFind.h

**Description:** Disjoint-set data structure.

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

```
struct UF {
    vi e;
    UF(int n) : e(n, -1) {}
    bool same_set(int a, int b) { return find(a) == find(b); }
    int size(int x) { return -e[find(x)]; }
    int find(int x) { return e[x] < 0 ? x : e[x] = find(e[x]); }
    void join(int a, int b) {
        a = find(a), b = find(b);
        if (a == b) return;
        if (e[a] > e[b]) swap(a, b);
        e[a] += e[b]; e[b] = a;
    }
};
```

SubMatrix.h

**Description:** Calculate submatrix sums quickly, given upper-left and lower-right corners (half-open).

**Usage:** SubMatrix<int> m(matrix);

m.sum(0, 0, 2, 2); // top left 4 elements

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

```
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;

```
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.

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

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

struct LineContainer : multiset<Line> {
```

```
// (for doubles, use inf = 1/.0, div(a,b) = a/b)
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()) { x->p = inf; return false; }
    if (x->k == y->k) x->p = x->m > y->m ? inf : -inf;
    else x->p = div(y->m - x->m, x->k - y->k);
    return x->p >= y->p;
}

void add(ll k, ll m) {
    auto z = insert({k, m, 0}), y = z++, x = y;
    while (isect(y, z)) z = erase(z);
    if (x != begin() && isect(--x, y)) isect(x, y = erase(y));
    while ((y = x) != begin() && (--x)->p >= y->p)
        isect(x, erase(y));
}

ll query(ll x) {
    assert(!empty());
    Q = 1; auto l = *lower_bound({0,0,x}); Q = 0;
    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:**  $\mathcal{O}(\log N)$

```
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 >= v" for lower_bound(v)
        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);
        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));
}
```

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  $\mathcal{O}(\log N)$ .

```
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"
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() {
        trav(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], ... V[b - 1]) in constant time. Set inf to something reasonable before use.

**Usage:** RMQ rmq(values);  
rmq.query(inclusive, exclusive);

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

21 lines

```
const int inf = numeric_limits<int>::max();

template <class T>
struct RMQ {
    vector<vector<T>> jmp;

    RMQ(const vector<T>& V) {
        int N = sz(V), on = 1, depth = 1;
        while (on < sz(V)) on *= 2, depth++;
        jmp.assign(depth, V);
        rep(i,0,depth-1) rep(j,0,N)
            jmp[i+1][j] = min(jmp[i][j],
                               jmp[i][min(N - 1, j + (1 << i))]);
    }

    T query(int a, int b) {
        if (b <= a) return inf;
        int dep = 31 - __builtin_clz(b - a);
        return min(jmp[dep][a], jmp[dep][b - (1 << dep)]);
    }
};
```

## Numerical (4)

Polynomial.h

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

18 lines

BinarySearch.h

**Description:** Finds a zero point of f on the interval [a b]. f(a) must be less than 0 and f(b) greater than 0. Useful for solving equations like kx=sin(x) as in the example below.

**Usage:** double func(double x) { return .23\*x-sin(x); }

double x0 = bs(0,4,func);

**Time:**  $\mathcal{O}(\log((b-a)/\epsilon))$

9 lines

```
double bs(double a, double b, double (*f)(double)) {
```

```
    //for(int i = 0; i < 60; ++i){
    while (b-a > 1e-6) {
        double m = (a+b)/2;
        if (f(m) > 0) b = m;
        else a = m;
    }
    return a;
}
```

GoldenSectionSearch.h

**Description:** Finds the argument minimizing the function  $f$  in the interval  $[a, b]$  assuming  $f$  is unimodal on the interval, i.e. has only one local minimum. The maximum error in the result is  $\epsilon$ . Works equally well for maximization with a small change in the code. See TernarySearch.h in the Various chapter for a discrete version.

**Usage:** double func(double x) { return 4+x+.3\*x\*x; }

double xmin = gss(-1000,1000,func);

**Time:**  $\mathcal{O}(\log((b-a)/\epsilon))$

14 lines

```
double gss(double a, double b, double (*f)(double)) {
    double r = (sqrt(5)-1)/2, eps = 1e-7;
    double x1 = b - r*(b-a), x2 = a + r*(b-a);
    double f1 = f(x1), f2 = f(x2);
    while (b-a > eps)
        if (f1 < f2) { //change to > to find maximum
            b = x2; x2 = x1; f2 = f1;
            x1 = b - r*(b-a); f1 = f(x1);
        } else {
            a = x1; x1 = x2; f1 = f2;
            x2 = a + r*(b-a); f2 = f(x2);
        }
    return a;
}
```

PolyRoots.h

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

**Usage:** vector<double> roots; Polynomial p(2);  
p.a[0] = 2; p.a[1] = -3; p.a[2] = 1;  
poly\_roots(p,-1e10,1e10,roots); // x^2-3x+2=0

25 lines

```
"Polynomial.h"

void poly_roots(const Polynomial& p, double xmin, double xmax,
    vector<double>& roots) {
    if (p.n == 1) { roots.push_back(-p.a.front()/p.a.back()); }
    else {
        Polynomial d = p;
        d.diff();
        vector<double> dr;
        poly_roots(d, xmin, xmax, dr);
        dr.push_back(xmin-1);
        dr.push_back(xmax+1);
        sort(all(dr));
        for (auto i = dr.begin(), j = i++; i != dr.end(); j = i++){
            double l = *j, h = *i, m, f;
            bool sign = p(l) > 0;
            if (sign ^ (p(h) > 0)) {
                //for(int i = 0; i < 60; ++i){
                while(h - l > 1e-8) {
                    m = (l + h) / 2, f = p(m);
                    if ((f <= 0) ^ sign) l = m;
                    else h = m;
                }
                roots.push_back((l + h) / 2);
            }
        }
    }
}
```

PolyInterpolate.h

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

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

13 lines

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

HillClimbing.h

**Description:** Poor man's optimization for unimodal functions.

16 lines

```
typedef array<double, 2> P;

double func(P p);

pair<double, P> hillClimb(P start) {
    pair<double, P> cur(func(start), start);
    for (double jmp = 1e9; jmp > 1e-20; jmp /= 2) {
        rep(j,0,100) rep(dx,-1,2) rep(dy,-1,2) {
            P p = cur.second;
            p[0] += dx*jmp;
            p[1] += dy*jmp;
            cur = min(cur, make_pair(func(p), p));
        }
    }
    return cur;
}
```

Integrate.h

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

8 lines

```
double quad(double (*f)(double), double a, double b) {
    const int n = 1000;
    double h = (b - a) / 2 / n;
    double v = f(a) + f(b);
    rep(i,1,n*2)
        v += f(a + i*h) * (i&1 ? 4 : 2);
    return v * h / 3;
}
```

IntegrateAdaptive.h

**Description:** Fast integration using an adaptive Simpson's rule.

**Usage:** double z, y;  
double h(double x) { return x\*x + y\*y + z\*z <= 1; }  
double g(double y) { ::y = y; return quad(h, -1, 1); }  
double f(double z) { ::z = z; return quad(g, -1, 1); }  
double sphereVol = quad(f, -1, 1), pi = sphereVol\*3/4;

16 lines

```
typedef double d;
d simpson(d (*f)(d), d a, d b) {
    d c = (a+b) / 2;
    return (f(a) + 4*f(c) + f(b)) * (b-a) / 6;
}
d rec(d (*f)(d), d a, d b, d eps, d S) {
```

```
d c = (a+b) / 2;
d S1 = simpson(f, a, c);
d S2 = simpson(f, c, b), T = S1 + S2;
if (abs (T - S) <= 15*eps || b-a < 1e-10)
    return T + (T - S) / 15;
return rec(f, a, c, eps/2, S1) + rec(f, c, b, eps/2, S2);
}

d quad(d (*f)(d), d a, d b, d eps = 1e-8) {
    return rec(f, a, b, eps, simpson(f, a, b));
}
```

Determinant.h

**Description:** Calculates determinant of a matrix. Destroys the matrix.  
**Time:**  $\mathcal{O}(N^3)$

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

```
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;
}
```

Simplex.h

**Description:** Solves a general linear maximization problem: maximize  $c^T x$  subject to  $Ax \leq b, x \geq 0$ . Returns -inf if there is no solution, inf if there are arbitrarily good solutions, or the maximum value of  $c^T x$  otherwise. The input vector is set to an optimal  $x$  (or in the unbounded case, an arbitrary solution fulfilling the constraints). Numerical stability is not guaranteed. For better performance, define variables such that  $x = 0$  is viable.  
**Usage:** vvd A = {{1,-1}, {-1,1}, {-1,-2}};  
vd b = {1,1,-4}, c = {-1,-1}, x;  
T val = LPSolver(A, b, c).solve(x);  
**Time:**  $\mathcal{O}(NM * \#pivots)$ , where a pivot may be e.g. an edge relaxation.  
 $\mathcal{O}(2^n)$  in the general case.

```
typedef double T; // long double, Rational, double + modP>...
typedef vector<T> vd;
typedef vector<vd> vvd;

const T eps = 1e-8, inf = 1/.0;
#define MP make_pair
#define ltj(X) if(s == -1 || MP(X[j],N[j]) < MP(X[s],N[s])) s=j

struct LPSolver {
    int m, n;
    vi N, B;
    vvd D;

    LPSolver(const vvd& A, const vd& b, const vd& c) :
        m(sz(b)), n(sz(c)), N(n+1), B(m), D(m+2, vd(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;
    }
};
```

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

```
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"
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:**  $\mathcal{O}(n^2 m)$

```
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  $A \pmod{p}$ , and  $k$  is doubled in each step.  
**Time:**  $\mathcal{O}(n^3)$

36 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;
        if (fabs(A[r][c]) < 1e-12) return i;
        A[i].swap(A[r]); tmp[i].swap(tmp[r]);
        rep(j,0,n)
            swap(A[j][i], A[j][c]), swap(tmp[j][i], tmp[j][c]);
        swap(col[i], col[c]);
        double v = A[i][i];
        rep(j,i+1,n) {
            double f = A[j][i] / v;
            A[j][i] = 0;
            rep(k,i+1,n) A[j][k] -= f*A[i][k];
            rep(k,0,n) tmp[j][k] -= f*tmp[i][k];
        }
        rep(j,i+1,n) A[i][j] /= v;
        rep(j,0,n) tmp[i][j] /= v;
        A[i][i] = 1;
    }

    for (int i = n-1; i > 0; --i) rep(j,0,i) {
        double v = A[j][i];
        rep(k,0,n) tmp[j][k] -= v*tmp[i][k];
    }

    rep(i,0,n) rep(j,0,n) A[col[i]][col[j]] = tmp[i][j];
    return n;
}
```

Tridiagonal.h

**Description:** Solves a linear equation system with a tridiagonal matrix with diagonal diag, subdiagonal sub and superdiagonal super, i.e.,  $x = \text{tridiagonal}(d,p,q,b)$  solves the equation system

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

The size of diag and b should be the same and super and sub should be one element shorter. T is intended to be double. This is useful for solving problems on the type

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

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

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

**Usage:** int n = 1000000;  
vector<double> diag(n,-1), sup(n-1,.5), sub(n-1,.5), b(n,1);  
vector<double> x = tridiagonal(diag, super, sub, b);  
**Time:**  $\mathcal{O}(N)$

14 lines

```
template <class T>
vector<T> tridiagonal(vector<T> diag, const vector<T>& super,
    const vector<T>& sub, vector<T> b) {
    rep(i,0,sz(b)-1) {
        diag[i+1] -= super[i]*sub[i]/diag[i];
        b[i+1] -= b[i]*sub[i]/diag[i];
    }
    for (int i = sz(b); --i > 0; ) {
        b[i] /= diag[i];
        b[i-1] -= b[i]*super[i-1];
    }
    b[0] /= diag[0];
    return b;
}
```

FFT.h

**Description:** Fast Fourier transform. Also includes a function for convolution: conv(a, b) = c, where  $c[x] = \sum a[i]b[x-i]$ .  $a$  and  $b$  should be of roughly equal size. For convolutions of integers, rounding the results of conv works if  $(|a| + |b|) \max(a,b) < \sim 10^9$  (in theory maybe  $10^6$ ); you may want to use an NTT from the Number Theory chapter instead.  
**Time:**  $\mathcal{O}(N \log N)$

<valarray>29 lines

```
typedef valarray<complex<double> > carray;
void fft(carray& x, carray& roots) {
    int N = sz(x);
    if (N <= 1) return;
    carray even = x[slice(0, N/2, 2)];
    carray odd = x[slice(1, N/2, 2)];
    carray rs = roots[slice(0, N/2, 2)];
    fft(even, rs);
    fft(odd, rs);
    rep(k,0,N/2) {
        auto t = roots[k] * odd[k];
        x[k] = even[k] + t;
        x[k+N/2] = even[k] - t;
    }
}
```

```
}
}

typedef vector<double> vd;
vd conv(const vd& a, const vd& b) {
    int s = sz(a) + sz(b) - 1, L = 32-__builtin_clz(s), n = 1<<L;
    if (s <= 0) return {};
    carray av(n), bv(n), roots(n);
    rep(i,0,n) roots[i] = polar(1.0, -2 * M_PI * i / n);
    copy(all(a), begin(av)); fft(av, roots);
    copy(all(b), begin(bv)); fft(bv, roots);
    roots = roots.apply(conj);
    carray cv = av * bv; fft(cv, roots);
    vd c(s); rep(i,0,s) c[i] = cv[i].real() / n;
    return c;
}
}
```

Divide.h

**Description:** Division  
**Time:**  $\mathcal{O}(1)$

9 lines

```
ll div_floor(ll a, ll b) {
    if (b < 0) a = -a, b = -b;
    return a/b - (a%b < 0);
}

ll div_ceil(ll a, ll b) {
    if (b < 0) a = -a, b = -b;
    return a/b + (a%b > 0);
}
```

## Number theory (5)

### 5.1 Modular arithmetic

ModularArithmetic.h

**Description:** Operators for modular arithmetic. You need to set mod to some number first and then you can use the structure.

"euclid.h"18 lines

```
const ll mod = 17; // change to something else
struct Mod {
    ll x;
    Mod(ll xx) : x(xx) {}
    Mod operator+(Mod b) { return Mod((x + b.x) % mod); }
    Mod operator-(Mod b) { return Mod((x - b.x + mod) % mod); }
    Mod operator*(Mod b) { return Mod((x * b.x) % mod); }
    Mod operator/(Mod b) { return *this * invert(b); }
    Mod invert(Mod a) {
        ll x, y, g = euclid(a.x, mod, x, y);
        assert(g == 1); return Mod((x + mod) % mod);
    }
    Mod operator^(ll e) {
        if (!e) return Mod(1);
        Mod r = *this ^ (e / 2); r = r * r;
        return e&1 ? *this * r : r;
    }
};
```

ModInverse.h

**Description:** Pre-computation of modular inverses. Assumes  $\text{LIM} \leq \text{mod}$  and that mod is a prime.

3 lines

**const** ll mod = 1000000007, LIM = 200000;  
ll\* inv = **new** ll[LIM] - 1; inv[1] = 1;  
rep(i,2,LIM) inv[i] = mod - (mod / i) \* inv[mod % i] % mod;

ModPow.h

6 lines

```
const ll mod = 1000000007; // faster if const
ll modpow(ll a, ll e) {
    if (e == 0) return 1;
    ll x = modpow(a * a % mod, e >> 1);
    return e & 1 ? x * a % mod : x;
}
```



ModSum.h

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

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

**Time:**  $\log(m)$ , with a large constant.

21 lines

```
typedef unsigned long long ull;
ull sumsq(ull to) { return to / 2 * ((to-1) | 1); }
```

```
ull divsum(ull to, ull c, ull k, ull m) {
    ull res = k / m * sumsq(to) + c / m * to;
    k %= m; c %= m;
    if (k) {
        ull to2 = (to * k + c) / m;
        res += to * to2;
        res -= divsum(to2, m-1 - c, m, k) + to2;
    }
    return res;
}
```

```
ll modsum(ull to, ll c, ll k, ll m) {
    c %= m;
    k %= m;
    if (c < 0) c += m;
    if (k < 0) k += m;
    return to * c + k * sumsq(to) - m * divsum(to, c, k, m);
}
```

ModMulLL.h

**Description:** Calculate  $a \cdot b \bmod c$  (or  $a^b \bmod c$ ) for large  $c$ .

**Time:**  $\mathcal{O}(64/bits \cdot \log b)$ , where  $bits = 64 - k$ , if we want to deal with  $k$ -bit numbers.

19 lines

```
typedef unsigned long long ull;
const int bits = 10;
// if all numbers are less than 2^k, set bits = 64-k
const ull po = 1 << bits;
ull mod_mul(ull a, ull b, ull &c) {
    ull x = a * (b & (po - 1)) % c;
    while ((b >= bits) > 0) {
        a = (a << bits) % c;
        x += (a * (b & (po - 1))) % c;
    }
    return x % c;
}
ull mod_pow(ull a, ull b, ull mod) {
    if (b == 0) return 1;
    ull res = mod_pow(a, b / 2, mod);
    res = mod_mul(res, res, mod);
    if (b & 1) return mod_mul(res, a, mod);
    return res;
}
```

ModSqrt.h

**Description:** Tonelli-Shanks algorithm for modular square roots.

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

30 lines

```
while (modpow(n, (p - 1) / 2, p) != p - 1) ++n;
ll x = modpow(a, (s + 1) / 2, p);
ll b = modpow(a, s, p);
ll g = modpow(n, s, p);
for (;;) {
    ll t = b;
    int m = 0;
    for (; m < r; ++m) {
        if (t == 1) break;
        t = t * t % p;
    }
    if (m == 0) return x;
    ll gs = modpow(g, 1 << (r - m - 1), p);
    g = gs * gs % p;
    x = x * gs % p;
    b = b * g % p;
    r = m;
}
}
```

## 5.2 Number theoretic transform

NTT.h

**Description:** Number theoretic transform. Can be used for convolutions modulo specific nice primes of the form  $2^a b + 1$ , where the convolution result has size at most  $2^a$ . For other primes/integers, use two different primes and combine with CRT. May return negative values.

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

38 lines

```
const ll mod = (119 << 23) + 1, root = 3; // = 998244353
// For p < 2^30 there is also e.g. (5 << 25, 3), (7 << 26, 3),
// (479 << 21, 3) and (483 << 21, 5). The last two are > 10^9.

typedef vector<ll> vl;
void ntt(ll* x, ll* temp, ll* roots, int N, int skip) {
    if (N == 1) return;
    int n2 = N/2;
    ntt(x, temp, roots, n2, skip*2);
    ntt(x+skip, temp, roots, n2, skip*2);
    rep(i,0,N) temp[i] = x[i*skip];
    rep(i,0,n2) {
        ll s = temp[2*i], t = temp[2*i+1] * roots[skip*i];
        x[skip*i] = (s + t) % mod; x[skip*(i+n2)] = (s - t) % mod;
    }
}
void ntt(vl& x, bool inv = false) {
    ll e = modpow(root, (mod-1) / sz(x));
    if (inv) e = modpow(e, mod-2);
    vl roots(sz(x), 1), temp = roots;
    rep(i,1,sz(x)) roots[i] = roots[i-1] * e % mod;
    ntt(&x[0], &temp[0], &roots[0], sz(x), 1);
}
vl conv(vl a, vl b) {
    int s = sz(a) + sz(b) - 1; if (s <= 0) return {};
    int L = s > 1 ? 32 - __builtin_clz(s - 1) : 0, n = 1 << L;
    if (s <= 200) { // (factor 10 optimization for |a|,|b| = 10)
        vl c(s);
        rep(i,0,sz(a)) rep(j,0,sz(b))
            c[i + j] = (c[i + j] + a[i] * b[j]) % mod;
        return c;
    }
    a.resize(n); ntt(a);
    b.resize(n); ntt(b);
    vl c(n); ll d = modpow(n, mod-2);
    rep(i,0,n) c[i] = a[i] * b[i] % mod * d % mod;
    ntt(c, true); c.resize(s); return c;
}
```

## 5.3 Primality

eratosthenes.h

**Description:** Prime sieve for generating all primes up to a certain limit. isprime[i] is true iff i is a prime.

**Time:**  $\lim=100'000'000 \approx 0.8$  s. Runs 30% faster if only odd indices are stored.

11 lines

```
const int MAX_PR = 5000000;
bitset<MAX_PR> isprime;
vi eratosthenes_sieve(int lim) {
    isprime.set(); isprime[0] = isprime[1] = 0;
    for (int i = 4; i < lim; i += 2) isprime[i] = 0;
    for (int i = 3; i*i < lim; i += 2) if (isprime[i])
        for (int j = i*i; j < lim; j += i*2) isprime[j] = 0;
    vi pr;
    rep(i,2,lim) if (isprime[i]) pr.push_back(i);
    return pr;
}
```

MillerRabin.h

**Description:** Miller-Rabin primality probabilistic test. Probability of failing one iteration is at most  $1/4$ . 15 iterations should be enough for 50-bit numbers.

**Time:** 15 times the complexity of  $a^b \bmod c$ .

16 lines

```
bool prime(ull p) {
    if (p == 2) return true;
    if (p == 1 || p % 2 == 0) return false;
    ull s = p - 1;
    while (s % 2 == 0) s /= 2;
    rep(i,0,15) {
        ull a = rand() % (p - 1) + 1, tmp = s;
        ull mod = mod_pow(a, tmp, p);
        while (tmp != p - 1 && mod != 1 && mod != p - 1) {
            mod = mod_mul(mod, mod, p);
            tmp *= 2;
        }
        if (mod != p - 1 && tmp % 2 == 0) return false;
    }
    return true;
}
```

factor.h

**Description:** Pollard's rho algorithm. It is a probabilistic factorisation algorithm, whose expected time complexity is good. Before you start using it, run init(bits), where bits is the length of the numbers you use. Returns factors of the input without duplicates.

**Time:** Expected running time should be good enough for 50-bit numbers.

35 lines

```
vector<ull> pr;
ull f(ull a, ull n, ull &has) {
    return (mod_mul(a, a, n) + has) % n;
}
vector<ull> factor(ull d) {
    vector<ull> res;
    for (int i = 0; i < sz(pr) && pr[i]*pr[i] <= d; i++)
        if (d % pr[i] == 0) {
            while (d % pr[i] == 0) d /= pr[i];
            res.push_back(pr[i]);
        }
    //d is now a product of at most 2 primes.
    if (d > 1) {
        if (prime(d))
            res.push_back(d);
        else while (true) {
            ull has = rand() % 2321 + 47;
```

```
ull x = 2, y = 2, c = 1;
for (; c==1; c = __gcd((y > x ? y - x : x - y), d)) {
    x = f(x, d, has);
    y = f(f(y, d, has), d, has);
}
if (c != d) {
    res.push_back(c); d /= c;
    if (d != c) res.push_back(d);
    break;
}
}
}
return res;
}

void init(int bits) { //how many bits do we use?
    vi p = eratosthenes_sieve(1 << ((bits + 2) / 3));
    pr.assign(all(p));
}
```

5.4 Divisibility

euclid.h  
**Description:** Finds the Greatest Common Divisor to the integers  $a$  and  $b$ . Euclid also finds two integers  $x$  and  $y$ , such that  $ax + by = \gcd(a, b)$ . If  $a$  and  $b$  are coprime, then  $x$  is the inverse of  $a \pmod b$ .

```
11 gcd(11 a, 11 b) { return __gcd(a, b); }

11 euclid(11 a, 11 b, 11 &x, 11 &y) {
    if (b) { 11 d = euclid(b, a % b, y, x);
        return y -= a/b * x, d; }
    return x = 1, y = 0, a;
}
```

```
Euclid.java
Description: Finds {x, y, d} s.t. ax + by = d = gcd(a, b).
11 lines

static BigInteger[] euclid(BigInteger a, BigInteger b) {
    BigInteger x = BigInteger.ONE, yy = x;
    BigInteger y = BigInteger.ZERO, xx = y;
    while (b.signum() != 0) {
        BigInteger q = a.divide(b), t = b;
        b = a.mod(b); a = t;
        t = xx; xx = x.subtract(q.multiply(xx)); x = t;
        t = yy; yy = y.subtract(q.multiply(yy)); y = t;
    }
    return new BigInteger[]{x, y, a};
}
```

5.4.1 Bézout’s identity

For  $a \neq, b \neq 0$ , 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 totient or Euler’s phi function is defined as  $\phi(n) := \#$  of positive integers  $\leq n$  that are coprime with  $n$ . The cototient is  $n - \phi(n)$ .  $\phi(1) = 1, p$  prime  $\Rightarrow \phi(p^k) = (p - 1)p^{k-1}, m, n$  coprime  $\Rightarrow \phi(mn) = \phi(m)\phi(n)$ . If  $n = p_1^{k_1} p_2^{k_2} \dots p_r^{k_r}$  then  $\phi(n) = (p_1 - 1)p_1^{k_1-1} \dots (p_r - 1)p_r^{k_r-1}$ .  $\phi(n) = n \cdot \prod_{p|n} (1 - 1/p)$ .  
 $\sum_{d|n} \phi(d) = n, \sum_{1 \leq k \leq n, \gcd(k, n)=1} k = n\phi(n)/2, n > 1$   
**Euler’s thm:**  $a, n$  coprime  $\Rightarrow a^{\phi(n)} \equiv 1 \pmod n$ .  
**Fermat’s little thm:**  $p$  prime  $\Rightarrow a^{p-1} \equiv 1 \pmod p \ \forall a$ .

```
10 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] /= i) *= i-1;
}
```

5.5 Chinese remainder theorem

chinese.h  
**Description:** Chinese Remainder Theorem. `chinese(a, m, b, n)` returns a number  $x$ , such that  $x \equiv a \pmod m$  and  $x \equiv b \pmod n$ . For not coprime  $n, m$ , use `chinese.common`. Note that all numbers must be less than  $2^{31}$  if you have `Z = unsigned long long`.  
**Time:**  $\log(m + n)$

```
13 lines
"euclid.h"

template <class Z> Z chinese(Z a, Z m, Z b, Z n) {
    Z x, y; euclid(m, n, x, y);
    Z ret = a * (y + m) % m * n + b * (x + n) % n * m;
    if (ret >= m * n) ret -= m * n;
    return ret;
}

template <class Z> Z chinese_common(Z a, Z m, Z b, Z n) {
    Z d = gcd(m, n);
    if ((b -= a) % n < 0) b += n;
    if (b % d) return -1; // No solution
    return d * chinese(Z(0), m/d, b/d, n/d) + a;
}
```

5.6 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.

5.7 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  $\mathbb{Z}_2 \times \mathbb{Z}_{2^{a-2}}$ .

5.8 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  $n < 1e19$ .

Combinatorial (6)

6.1 The Twelfold Way

Counts the  $\#$  of functions  $f : N \rightarrow K, |N| = n, |K| = k$ . The elements in  $N$  and  $K$  can be distinguishable or indistinguishable, while  $f$  can be injective (one-to-one) of surjective (onto).

$N$	$K$	none	injective	surjective
dist	dist	$k^n$	$\frac{k!}{(k-n)!}$	$k!S(n, k)$
indist	dist	$\binom{n+k-1}{n}$	$\binom{k}{n}$	$\binom{n-1}{n-k}$
dist	indist	$\sum_{t=0}^k S(n, t)$	$[n \leq k]$	$S(n, k)$
indist	indist	$\sum_{t=1}^k p(n, t)$	$[n \leq k]$	$p(n, k)$

Here,  $S(n, k)$  is the Stirling number of the second kind, and  $p(n, k)$  is the partition number.

6.2 Permutations

6.2.1 Factorial

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

```
intperm.h
Description: Permutations to/from integers. The bijection is order pre-serving.
Time: O(n^2)
20 lines

int factorial[] = {1, 1, 2, 6, 24, 120, 720, 5040}; // etc.
template <class Z, class It>
void perm_to_int(Z& val, It begin, It end) {
    int x = 0, n = 0;
    for (It i = begin; i != end; ++i, ++n)
        if (*i < *begin) ++x;
    if (n > 2) perm_to_int<Z>(val, ++begin, end);
    else val = 0;
}
```

```
    val += factorial[n-1]*x;
}
/* range [begin, end) does not have to be sorted. */
template <class Z, class It>
void int_to_perm(Z val, It begin, It end) {
    Z fac = factorial[end - begin - 1];
    // Note that the division result will fit in an integer!
    int x = val / fac;
    nth_element(begin, begin + x, end);
    swap(*begin, *(begin + x));
    if (end - begin > 2) int_to_perm(val % fac, ++begin, end);
}
```

6.2.2 Cycles

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

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

6.2.3 Derangements

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

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

derangements.h  
**Description:** Generates the  $i$ :th derangement of  $S_n$  (in lexicographical order).

```
template <class T, int N>
struct derangements {
    T dgen[N][N], choose[N][N], fac[N];
    derangements() {
        fac[0] = choose[0][0] = 1;
        memset(dgen, 0, sizeof(dgen));
        rep(m,1,N) {
            fac[m] = fac[m-1] * m;
            choose[m][0] = choose[m][m] = 1;
            rep(k,1,m)
                choose[m][k] = choose[m-1][k-1] + choose[m-1][k];
        }
    }
    T DGen(int n, int k) {
        T ans = 0;
        if (dgen[n][k]) return dgen[n][k];
        rep(i,0,k+1)
            ans += (i&1?-1:1) * choose[k][i] * fac[n-i];
        return dgen[n][k] = ans;
    }
    void generate(int n, T idx, int *res) {
        int vals[N];
        rep(i,0,n) vals[i] = i;
        rep(i,0,n) {
            int j, k = 0, m = n - i;
            rep(j,0,m) if (vals[j] > i) ++k;
            rep(j,0,m) {
                T p = 0;
```

derangements

```
        if (vals[j] > i) p = DGen(m-1, k-1);
        else if (vals[j] < i) p = DGen(m-1, k);
        if (idx <= p) break;
        idx -= p;
    }
    res[i] = vals[j];
    memmove(vals + j, vals + j + 1, sizeof(int)*(m-j-1));
}
};
```

6.2.4 Involutions

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

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

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

6.2.5 Stirling numbers of the first kind

$$s(n, k) = (-1)^{n-k} c(n, k)$$

$c(n, k)$  is the unsigned Stirling numbers of the first kind, and they count the number of permutations on  $n$  items with  $k$  cycles.

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

6.2.6 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.  $\pi(j) > j$ .

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

6.2.7 Burnside’s lemma

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

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

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

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

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

6.3 Partitions and subsets

6.3.1 Partition function

Partitions of  $n$  with exactly  $k$  parts,  $p(n, k)$ , i.e., writing  $n$  as a sum of  $k$  positive integers, disregarding the order of the summands.

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

For partitions with any number of parts,  $p(n)$  obeys

$$p(0) = 1, \quad p(n) = \sum_{k \in \mathbb{Z} \setminus \{0\}} (-1)^{k+1} p(n-k(3k-1)/2)$$
$$p(n) \sim 0.145/n \cdot \exp(2.56\sqrt{n})$$

$n$	0	1	2	3	4	5	6	7	8	9	20	50	100
$p(n)$	1	1	2	3	5	7	11	15	22	30	627	$\sim 2\text{e}5$	$\sim 2\text{e}8$

### 6.3.2   Binomials

binomial.h  
**Description:**   The number of  $k$ -element subsets of an  $n$ -element set,  $\binom{n}{k} = \frac{n!}{k!(n-k)!}$   
**Time:**    $\mathcal{O}(\min(k, n-k))$

```
6 lines
11 choose(int n, int k) {
    ll c = 1, to = min(k, n-k);
    if (to < 0) return 0;
    rep(i,0,to) c = c * (n - i) / (i + 1);
    return c;
}
```

binomialModPrime.h  
**Description:**   Lucas' thm: Let  $n, m$  be non-negative integers and  $p$  a prime. Write  $n = n_kp^k + \dots + n_1p + n_0$  and  $m = m_kp^k + \dots + m_1p + m_0$ . Then  $\binom{n}{m} \equiv \prod_{i=0}^k \binom{n_i}{m_i} \pmod p$ . fact and invfact must hold pre-computed factorials / inverse factorials, e.g. from ModInverse.h.  
**Time:**    $\mathcal{O}(\log_p n)$

```
10 lines
11 chooseModP(ll n, ll m, int p, vi& fact, vi& invfact) {
    ll c = 1;
    while (n || m) {
        ll a = n % p, b = m % p;
        if (a < b) return 0;
        c = c * fact[a] % p * invfact[b] % p * invfact[a - b] % p;
        n /= p; m /= p;
    }
    return c;
}
```

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

```
14 lines
const ll mod = 1000000007;
vector<ll> invs; // precomputed up to max n, inclusively
struct Bin {
    int N = 0, K = 0; ll r = 1;
    void m(ll a, ll b) { r = r * a % mod * invs[b] % mod; }
    ll choose(int n, int k) {
        if (k > n || k < 0) return 0;
        while (N < n) ++N, m(N, N-K);
        while (K < k) ++K, m(N-K+1, K);
        while (K > k) m(K, N-K+1), --K;
        while (N > n) m(N-K, N), --N;
        return r;
    }
};
```

multinomial.h  
**Description:**    $\binom{\sum k_i}{k_1, k_2, \dots, k_n} = \frac{(\sum k_i)!}{k_1!k_2! \dots k_n!}$   
**Time:**    $\mathcal{O}((\sum k_i) - k_1)$

```
6 lines
11 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;
}
```

### 6.3.3   Stirling numbers of the second kind

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

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

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

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

### 6.3.4   Bell numbers

Total number of partitions of  $n$  distinct elements.

$$B(n) = \sum_{k=1}^n \binom{n-1}{k-1} B(n-k) = \sum_{k=1}^n S(n, k)$$

$$B(0) = B(1) = 1$$

The first are 1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975, 678570, 4213597. For a prime  $p$

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

### 6.3.5   Triangles

Given rods of length  $1, \dots, n$ ,

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

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

## 6.4   General purpose numbers

### 6.4.1   Catalan numbers

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

$$C_{n+1} = \frac{2(2n+1)}{n+2} C_n$$

$$C_0 = 1, C_{n+1} = \sum C_i C_{n-i}$$

First few are 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900.

- # of monotonic lattice paths of a  $n \times n$ -grid which do not pass above the diagonal.

- # of expressions containing  $n$  pairs of parenthesis which are correctly matched.

- # of full binary trees with with  $n+1$  leaves (0 or 2 children).

- # of non-isomorphic ordered trees with  $n+1$  vertices.

- # of ways a convex polygon with  $n+2$  sides can be cut into triangles by connecting vertices with straight lines.

- # of permutations of  $[n]$  with no three-term increasing subsequence.

### 6.4.2   Super Catalan numbers

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

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

$$S(1) = S(2) = 1$$

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

### 6.4.3   Motzkin numbers

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

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

$$M(0) = M(1) = 1$$

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

### 6.4.4   Narayana numbers

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

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

$$N(n, 1) = N(n, n) = 1$$

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

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

### 6.4.5 Schröder numbers

Number of lattice paths from (0,0) to (n,n) using only steps N,NE,E, never going above the diagonal. Number of lattice paths from (0,0) to (2n,0) using only steps NE, SE and double east EE, never going below the x-axis. Twice the Super Catalan number, except for the first term. 1, 2, 6, 22, 90, 394, 1806, 8558, 41586, 206098

## Graph (7)

### 7.1 Fundamentals

bellmanFord.h  
**Description:** Calculates shortest path in a graph that might have negative edge distances. Propagates negative infinity distances (sets dist = -inf), and returns true if there is some negative cycle. Unreachable nodes get dist = inf.  
**Time:**  $\mathcal{O}(EV)$

```
typedef ll T; // or whatever
struct Edge { int src, dest; T weight; };
struct Node { T dist; int prev; };
struct Graph { vector<Node> nodes; vector<Edge> edges; };

const T inf = numeric_limits<T>::max();
bool bellmanFord2(Graph& g, int start_node) {
    trav(n, g.nodes) { n.dist = inf; n.prev = -1; }
    g.nodes[start_node].dist = 0;

    rep(i,0,sz(g.nodes)) trav(e, g.edges) {
        Node& cur = g.nodes[e.src];
        Node& dest = g.nodes[e.dest];
        if (cur.dist == inf) continue;
        T ndist = cur.dist + (cur.dist == -inf ? 0 : e.weight);
        if (ndist < dest.dist) {
            dest.prev = e.src;
            dest.dist = (i >= sz(g.nodes)-1 ? -inf : ndist);
        }
    }
    bool ret = 0;
    rep(i,0,sz(g.nodes)) trav(e, g.edges) {
        if (g.nodes[e.src].dist == -inf)
            g.nodes[e.dest].dist = -inf, ret = 1;
    }
    return ret;
}
```

FloydWarshall.h  
**Description:** Calculates all-pairs shortest path in a directed graph that might have negative edge distances. Input is an distance matrix m, where m[i][j] = inf if i and j are not adjacent. As output, m[i][j] is set to the shortest distance between i and j, inf if no path, or -inf if the path goes through a negative-weight cycle.  
**Time:**  $\mathcal{O}(N^3)$

```
const ll inf = 1LL << 62;
void floydWarshall(vector<vector<ll>>& m) {
```

```
int n = sz(m);
rep(i,0,n) m[i][i] = min(m[i][i], {});
rep(k,0,n) rep(i,0,n) rep(j,0,n)
    if (m[i][k] != inf && m[k][j] != inf) {
        auto newDist = max(m[i][k] + m[k][j], -inf);
        m[i][j] = min(m[i][j], newDist);
    }
rep(k,0,n) if (m[k][k] < 0) rep(i,0,n) rep(j,0,n)
    if (m[i][k] != inf && m[k][j] != inf) m[i][j] = -inf;
}
```

DirectedMST.h  
**Description:** Given a directed weighted graph and root node, computes the minimum spanning directed tree (arborescence) on it.  
**Time:**  $\mathcal{O}(N \cdot M)$

```
struct Edge { int x, y, w; };
int dmst(int N, vector<Edge> E, int root) {
    const int oo = 1e9;

    vi cost(N), back(N), label(N), bio(N);
    int ret = 0;

    for (;;) {
        REP(i, N) cost[i] = oo;
        for (auto e : E) {
            if (e.x == e.y) continue;
            if (e.w < cost[e.y]) cost[e.y] = e.w, back[e.y] = e.x;
        }
        cost[root] = 0;
        REP(i, N) if (cost[i] == oo) return -1;
        REP(i, N) ret += cost[i];

        int K = 0;
        REP(i, N) label[i] = -1;
        REP(i, N) bio[i] = -1;

        REP(i, N) {
            int x = i;
            for (; x != root && bio[x] == -1; x = back[x]) bio[x] = i;

            if (x != root && bio[x] == i) {
                for (; label[x] == -1; x = back[x]) label[x] = K;
                ++K;
            }
        }
        if (K == 0) break;

        REP(i, N) if (label[i] == -1) label[i] = K++;

        for (auto &e : E) {
            int xx = label[e.x];
            int yy = label[e.y];
            if (xx != yy) e.w -= cost[e.y];
            e.x = xx;
            e.y = yy;
        }

        root = label[root];
        N = K;
    }

    return ret;
}
```

### 7.2 Euler walk

EulerWalk.h  
**Description:** Eulerian undirected/directed path/cycle algorithm. 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, also put it->second in s (and then ret).  
**Time:**  $\mathcal{O}(E)$  where E is the number of edges.

```
struct V {
    vector<pii> outs; // (dest, edge index)
    int nins = 0;
};

vi euler_walk(vector<V>& nodes, int nedges, int src=0) {
    int c = 0;
    trav(n, nodes) c += abs(n.nins - sz(n.outs));
    if (c > 2) return {};
    vector<vector<pii>::iterator> its;
    trav(n, nodes)
        its.push_back(n.outs.begin());
    vector<bool> eu(nedges);
    vi ret, s = {src};
    while(!s.empty()) {
        int x = s.back();
        auto& it = its[x], end = nodes[x].outs.end();
        while(it != end && eu[it->second]) ++it;
        if(it == end) { ret.push_back(x); s.pop_back(); }
        else { s.push_back(it->first); eu[it->second] = true; }
    }
    if(sz(ret) != nedges+1)
        ret.clear(); // No Eulerian cycles/paths.
    // else, non-cycle if ret.front() != ret.back()
    reverse(all(ret));
    return ret;
}
```

### 7.3 Network flow

SDinic.h  
**Description:** To use, call init(n), then add edges using edge(x, y, c1, c2), and finally call run(src, sink).  
**Time:**  $\mathcal{O}(V^2E)$

```
namespace dinic {
    const int MAXV = 1000100;
    const int MAXE = 1000100;
    const ll oo = 1e18;

    int V, E;
    int last[MAXV], dist[MAXV], curr[MAXV];
    int next[MAXE], adj[MAXE]; ll cap[MAXE];

    void init(int n) {
        V = n;
        E = 0;
        REP(i, V) last[i] = -1;
    }

    void edge(int x, int y, ll c1, ll c2) {
        adj[E] = y; cap[E] = c1; next[E] = last[x]; last[x] = E++;
        adj[E] = x; cap[E] = c2; next[E] = last[y]; last[y] = E++;
    }

    ll push(int x, int sink, ll flow) {
        if (x == sink) return flow;

        for (int &e = curr[x]; e != -1; e = next[e]) {
```

```
int y = adj[e];

if (cap[e] && dist[x] + 1 == dist[y])
    if (ll f = push(y, sink, min(flow, cap[e])))
        return cap[e] -= f, cap[e^1] += f, f;
}
return 0;
}

ll run(int src, int sink) {
    ll ret = 0;
    for (;;) {
        REP(i, V) curr[i] = last[i];
        REP(i, V) dist[i] = -1;

        queue<int> Q;
        Q.push(src), dist[src] = 0;

        while (!Q.empty()) {
            int x = Q.front(); Q.pop();

            for (int e = last[x]; e != -1; e = next[e]) {
                int y = adj[e];
                if (cap[e] && dist[y] == -1) Q.push(y), dist[y] =
                    dist[x] + 1;
            }
        }
        if (dist[sink] == -1) break;

        while (ll f = push(src, sink, oo)) ret += f;
    }
    return ret;
}
}
```

SMCMF.h

**Description:** To use, call init(n), then add edges using edge(x, y, c, w), and finally call run(src, sink). Uses DFS, good for shallow graphs  
**Time:**  $\mathcal{O}(VE^3)$

74 lines

```
namespace mcmf {
    const int MAXV = 1000100;
    const int MAXE = 1000100;
    const ll oo = 1e18;

    int V, E;
    int last[MAXV], curr[MAXV], bio[MAXV]; ll pi[MAXV];
    int next[MAXE], adj[MAXE]; ll cap[MAXE], cost[MAXE];

    void init(int n) {
        V = n;
        E = 0;
        REP(i, V) last[i] = -1;
    }

    void edge(int x, int y, ll c, ll w) {
        adj[E] = y; cap[E] = c; cost[E] = +w; next[E] = last[x]
        ]; last[x] = E++;
        adj[E] = x; cap[E] = 0; cost[E] = -w; next[E] = last[y]
        ]; last[y] = E++;
    }

    ll push(int x, int sink, ll flow) {
        if (x == sink) return flow;
        if (bio[x]) return 0;
        bio[x] = true;

        for (int &e = curr[x]; e != -1; e = next[e]) {
            int y = adj[e];
```

```
        if (cap[e] && pi[x] == pi[y] + cost[e])
            if (ll f = push(y, sink, min(flow, cap[e])))
                return cap[e] -= f, cap[e^1] += f, f;
        }
        return 0;
    }

    pair<ll, ll> run(int src, int sink) {
        ll total = 0;
        ll flow = 0;

        REP(i, V) pi[i] = oo;
        pi[sink] = 0;

        for (;;) {
            bool done = true;
            REP(x, V) for (int e = last[x]; e != -1; e = next[e]
                )
                if (cap[e] && pi[x] > pi[adj[e]] + cost[e])
                    pi[x] = pi[adj[e]] + cost[e], done = false;
            if (done) break;
        }

        for (;;) {
            REP(i, V) bio[i] = false;
            REP(i, V) curr[i] = last[i];

            while (ll f = push(src, sink, oo)) {
                total += pi[src] * f;
                flow += f;
                REP(i, V) bio[i] = false;
            }

            ll inc = oo;
            REP(x, V) if (bio[x]) {
                for (int e = last[x]; e != -1; e = next[e]) {
                    int y = adj[e];
                    if (cap[e] && !bio[y]) inc = min(inc, pi[y]
                        + cost[e] - pi[x]);
                }
            }
            if (inc == oo) break;

            REP(i, V) if (bio[i]) pi[i] += inc;
        }
        return {total, flow};
    }
}
```

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.

1 lines

GlobalMinCut.h

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

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

31 lines

```
pair<int, vi> GetMinCut(vector<vi>& weights) {
    int N = sz(weights);
    vi used(N), cut, best_cut;
    int best_weight = -1;

    for (int phase = N-1; phase >= 0; phase--) {
        vi w = weights[0], added = used;
```

```
int prev, k = 0;
rep(i,0,phase){
    prev = k;
    k = -1;
    rep(j,1,N)
        if (!added[j] && (k == -1 || w[j] > w[k])) k = j;
    if (i == phase-1) {
        rep(j,0,N) weights[prev][j] += weights[k][j];
        rep(j,0,N) weights[j][prev] = weights[prev][j];
        used[k] = true;
        cut.push_back(k);
        if (best_weight == -1 || w[k] < best_weight) {
            best_cut = cut;
            best_weight = w[k];
        }
    } else {
        rep(j,0,N)
            w[j] += weights[k][j];
        added[k] = true;
    }
}
return {best_weight, best_cut};
}
```

7.4 Matching

hopcroftKarp.h

**Description:** Find a maximum matching in a bipartite graph.  
**Usage:** vi ba(m, -1); hopcroftKarp(g, ba);  
**Time:**  $\mathcal{O}(\sqrt{VE})$

48 lines

```
bool dfs(int a, int layer, const vector<vi>& g, vi& btoa,
    vi& A, vi& B) {
    vi& A, vi& B) {
        if (A[a] != layer) return 0;
        A[a] = -1;
        trav(b, g[a]) if (B[b] == layer + 1) {
            B[b] = -1;
            if (btoa[b] == -1 || dfs(btoa[b], layer+2, g, btoa, A, B))
                return btoa[b] = a, 1;
        }
        return 0;
    }

    int hopcroftKarp(const 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), -1);

            cur.clear();
            trav(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 += 2) {
                bool islast = 0;
                next.clear();
                trav(a, cur) trav(b, g[a]) {
                    if (btoa[b] == -1) {
                        B[b] = lay;
                        islast = 1;
                    }
                    else if (btoa[b] != a && B[b] == -1) {
                        B[b] = lay;
                        next.push_back(btoa[b]);
                    }
                }
            }
        }
    }
}
```

```
        if (islast) break;
        if (next.empty()) return res;
        trav(a, next) A[a] = lay+1;
        cur.swap(next);
    }

    rep(a,0,sz(g)) {
        if(dfs(a, 0, g, btoa, A, B))
            ++res;
    }
}

}
}
```

DFSMatching.h  
**Description:** This is a simple matching algorithm but should be just fine in most cases. Graph  $g$  should be a list of neighbours of the left partition.  $n$  is the size of the left partition and  $m$  is the size of the right partition. If you want to get the matched pairs,  $match[i]$  contains match for vertex  $i$  on the right side or  $-1$  if it's not matched.  
**Time:**  $\mathcal{O}(EV)$  where  $E$  is the number of edges and  $V$  is the number of vertices.

24 lines

```
vi match;
vector<bool> seen;
bool find(int j, const vector<vi>& g) {
    if (match[j] == -1) return 1;
    seen[j] = 1; int di = match[j];
    trav(e, g[di])
        if (!seen[e] && find(e, g)) {
            match[e] = di;
            return 1;
        }
    return 0;
}
int dfs_matching(const vector<vi>& g, int n, int m) {
    match.assign(m, -1);
    rep(i,0,n) {
        seen.assign(m, 0);
        trav(j,g[i])
            if (find(j, g)) {
                match[j] = i;
                break;
            }
    }
    return m - (int)count(all(match), -1);
}
```

WeightedMatching.h  
**Description:** Min cost bipartite matching. Negate costs for max cost.  
**Time:**  $\mathcal{O}(N^3)$

79 lines

```
typedef vector<double> vd;
bool zero(double x) { return fabs(x) < 1e-10; }
double MinCostMatching(const vector<vd>& cost, vi& L, vi& R) {
    int n = sz(cost), mated = 0;
    vd dist(n), u(n), v(n);
    vi dad(n), seen(n);

    rep(i,0,n) {
        u[i] = cost[i][0];
        rep(j,1,n) u[i] = min(u[i], cost[i][j]);
    }
    rep(j,0,n) {
        v[j] = cost[0][j] - u[0];
        rep(i,1,n) v[j] = min(v[j], cost[i][j] - u[i]);
    }

    L = R = vi(n, -1);
    rep(i,0,n) rep(j,0,n) {
```

```
        if (R[j] != -1) continue;
        if (zero(cost[i][j] - u[i] - v[j])) {
            L[i] = j;
            R[j] = i;
            mated++;
            break;
        }
    }

    for (; mated < n; mated++) { // until solution is feasible
        int s = 0;
        while (L[s] != -1) s++;
        fill(all(dad), -1);
        fill(all(seen), 0);
        rep(k,0,n)
            dist[k] = cost[s][k] - u[s] - v[k];

        int j = 0;
        for (;;) {
            j = -1;
            rep(k,0,n){
                if (seen[k]) continue;
                if (j == -1 || dist[k] < dist[j]) j = k;
            }
            seen[j] = 1;
            int i = R[j];
            if (i == -1) break;
            rep(k,0,n) {
                if (seen[k]) continue;
                auto new_dist = dist[j] + cost[i][k] - u[i] - v[k];
                if (dist[k] > new_dist) {
                    dist[k] = new_dist;
                    dad[k] = j;
                }
            }
        }

        rep(k,0,n) {
            if (k == j || !seen[k]) continue;
            auto w = dist[k] - dist[j];
            v[k] += w, u[R[k]] -= w;
        }
        u[s] += dist[j];

        while (dad[j] >= 0) {
            int d = dad[j];
            R[j] = R[d];
            L[R[j]] = j;
            j = d;
        }
        R[j] = s;
        L[s] = j;
    }

    auto value = vd(1)[0];
    rep(i,0,n) value += cost[i][L[i]];
    return value;
}
```

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

40 lines

```
"../numerical/MatrixInverse-mod.h"
vector<pii> generalMatching(int N, vector<pii>& ed) {
    vector<vector<ll>> mat(N, vector<ll>(N)), A;
    trav(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;
}
```

## 7.5 Minimum vertex cover

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 an independent set.

20 lines

```
"DFSMatching.h"
vi cover(vector<vi>& g, int n, int m) {
    int res = dfs_matching(g, n, m);
    seen.assign(m, false);
    vector<bool> lfound(n, true);
    trav(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;
        trav(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;
}
```

## 7.6 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:**  $\mathcal{O}(E + V)$

```
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);
    trav(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);
}
```

24 lines

BiconnectedComponents.h

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

```
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;
    trav(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)
```

34 lines

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

**Usage:** `TwoSat ts(number of boolean variables);` `ts.either(0, ~3);` // Var 0 is true or var 3 is false `ts.set.value(2);` // Var 2 is true `ts.at.most.one({0,~1,2});` //  $\leq 1$  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.

57 lines

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

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

    int add_var() { // (optional)
        gr.emplace_back();
        gr.emplace_back();
        return N++;
    }

    void either(int f, int j) {
        f = (f >= 0 ? 2*f : -1-2*f);
        j = (j >= 0 ? 2*j : -1-2*j);
        gr[f^1].push_back(j);
        gr[j^1].push_back(f);
    }

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

    void at_most_one(const vi& li) { // (optional)
        if (sz(li) <= 1) return;
        int cur = ~li[0];
        rep(i,2,sz(li)) {
            int next = add_var();
            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);
        trav(e, gr[i]) if (!comp[e])
            low = min(low, val[e] ?: dfs(e));
        ++time;
        if (low == val[i]) do {
            x = z.back(); z.pop_back();
```

```
            comp[x] = time;
            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;
    }
};
```

7.7 Trees

TreePower.h

**Description:** Calculate power of two jumps in a tree. Assumes the root node points to itself.

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

14 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;
}
```

LCA.h

**Description:** Lowest common ancestor. Finds the lowest common ancestor in a tree (with 0 as root). C should be an adjacency list of the tree, either directed or undirected. Can also find the distance between two nodes.

**Usage:** `LCA lca(undirGraph);` `lca.query(firstNode, secondNode);` `lca.distance(firstNode, secondNode);`

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

"../data-structures/RMQ.h"

38 lines

```
typedef vector<pii> vpi;
typedef vector<vpi> graph;
const pii inf(1 << 29, -1);

struct LCA {
    vi time;
    vector<ll> dist;
    RMQ<pii> rmq;

    LCA(graph& C) : time(sz(C), -99), dist(sz(C)), rmq(dfs(C)) {}

    vpi dfs(graph& C) {
        vector<tuple<int, int, int, ll> > q(1);
        vpi ret;
        int T = 0, v, p, d; ll di;
        while (!q.empty()) {
            tie(v, p, d, di) = q.back();
            q.pop_back();
            if (d) ret.emplace_back(d, p);
            time[v] = T++;
```



```

    dist[v] = di;
    trav(e, C[v]) if (e.first != p)
        q.emplace_back(e.first, v, d+1, di + e.second);
}
return ret;
}

int query(int a, int b) {
    if (a == b) return a;
    a = time[a], b = time[b];
    return rmq.query(min(a, b), max(a, b)).second;
}

ll distance(int a, int b) {
    int lca = query(a, b);
    return dist[a] + dist[b] - 2 * dist[lca];
}
};

```

## CompressTree.h

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

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

```

"LCa.h" 20 lines

vpi compressTree(LCA& lca, const vi& subset) {
    static vi rev; rev.resize(sz(lca.dist));
    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.query(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.query(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. The function of the HLD can be changed by modifying  $T$ ,  $LOW$  and  $f$ .  $f$  is assumed to be associative and commutative.

**Usage:** HLD hld(G);  
hld.update(index, value);  
tie(value, lca) = hld.query(n1, n2);

```

"../data-structures/SegmentTree.h" 93 lines

```

```
typedef vector<pii> vpi;
```

```

struct Node {
    int d, par, val, chain = -1, pos = -1;
};

```

```

struct Chain {
    int par, val;
    vector<int> nodes;
    Tree tree;
};

```

```

struct HLD {
    typedef int T;
    const T LOW = -(1<<29);
    void f(T& a, T b) { a = max(a, b); }

    vector<Node> V;
    vector<Chain> C;

    HLD(vector<vpi>& g) : V(sz(g)) {
        dfs(0, -1, g, 0);
        trav(c, C) {
            c.tree.init(sz(c.nodes), 0);
            for (int ni : c.nodes)
                c.tree.update(V[ni].pos, V[ni].val);
        }
    }
}

```

```

void update(int node, T val) {
    Node& n = V[node]; n.val = val;
    if (n.chain != -1) C[n.chain].tree.update(n.pos, val);
}

```

```

int pard(Node& nod) {
    if (nod.par == -1) return -1;
    return V[nod.chain == -1 ? nod.par : C[nod.chain].par].d;
}

```

```

// query all *edges* between n1, n2
pair<T, int> query(int i1, int i2) {
    T ans = LOW;
    while(i1 != i2) {
        Node n1 = V[i1], n2 = V[i2];
        if (n1.chain != -1 && n1.chain == n2.chain) {
            int lo = n1.pos, hi = n2.pos;
            if (lo > hi) swap(lo, hi);
            f(ans, C[n1.chain].tree.query(lo, hi));
            i1 = i2 = C[n1.chain].nodes[hi];
        } else {
            if (pard(n1) < pard(n2))
                n1 = n2, swap(i1, i2);
            if (n1.chain == -1)
                f(ans, n1.val), i1 = n1.par;
            else {
                Chain& c = C[n1.chain];
                f(ans, n1.pos ? c.tree.query(n1.pos, sz(c.nodes))
                    : c.tree.s[1]);
                i1 = c.par;
            }
        }
    }
    return make_pair(ans, i1);
}

```

```

// query all *nodes* between n1, n2
pair<T, int> query2(int i1, int i2) {
    pair<T, int> ans = query(i1, i2);
    f(ans.first, V[ans.second].val);
    return ans;
}

```

```

pii dfs(int at, int par, vector<vpi>& g, int d) {
    V[at].d = d; V[at].par = par;
    int sum = 1, ch, nod, sz;
    tuple<int,int,int> mx(-1,-1,-1);
    trav(e, g[at]){
        if (e.first == par) continue;
        tie(sz, ch) = dfs(e.first, at, g, d+1);
        V[e.first].val = e.second;
        sum += sz;
    }
}

```

```

    mx = max(mx, make_tuple(sz, e.first, ch));
}
tie(sz, nod, ch) = mx;
if (2*sz < sum) return pii(sum, -1);
if (ch == -1) { ch = sz(C); C.emplace_back(); }
V[nod].pos = sz(C[ch].nodes);
V[nod].chain = ch;
C[ch].par = at;
C[ch].nodes.push_back(nod);
return pii(sum, ch);
}
};

```

## 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  $\mathcal{O}(\log N)$ .

```

96 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 push_flip() {
        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 (push_flip(); p; ) {
            if (p->p) p->p->push_flip();
            p->push_flip(); push_flip();
            int c1 = up(), c2 = p->up();
            if (c2 == -1) p->rot(c1, 2);
            else p->p->rot(c2, c1 != c2);
        }
    }
    Node* first() {
        push_flip();
        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));
make_root(&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];
    make_root(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 make_root(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;
}
};
```

## 7.8 Matrix tree theorem

MatrixTree.h

Description: To count the number of spanning trees in an undirected graph  $G$ : create an  $N \times N$  matrix  $\text{mat}$ , and for each edge  $(a,b) \in G$ , do  $\text{mat}[a][a]++$ ,  $\text{mat}[b][b]++$ ,  $\text{mat}[a][b]--$ ,  $\text{mat}[b][a]--$ . Remove the last row and column, and take the determinant.

1 lines

## 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.)

25 lines

```
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)); }
};
```

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.

4 lines

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

SegmentDistance.h

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

Usage: Point<double> a, b(2,2), p(1,1);

bool onSegment = segDist(a,b,p) < 1e-10;

6 lines

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

SegmentIntersection.h

Description: If a unique intersestion point between the line segments going from s1 to e1 and from s2 to e2 exists r1 is set to this point and l is returned. If no intersection point exists 0 is returned and if infinitely many exists 2 is returned and r1 and r2 are set to the two ends of the common line. The wrong position will be returned if P is Point<int> 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. Use segmentIntersectionQ to get just a true/false answer.

Usage: Point<double> intersection, dummy;

if (segmentIntersection(s1,e1,s2,e2,intersection,dummy)==1) cout << "segments intersect at " << intersection << endl;

27 lines

```
template <class P>
int segmentIntersection(const P& s1, const P& e1,
    const P& s2, const P& e2, P& r1, P& r2) {
    if (e1==s1) {
        if (e2==s2) {
            if (e1==e2) { r1 = e1; return 1; } //all equal
            else return 0; //different point segments
        } else return segmentIntersection(s2,e2,s1,e1,r1,r2); //swap
    }
    //segment directions and separation
    P v1 = e1-s1, v2 = e2-s2, d = s2-s1;
    auto a = v1.cross(v2), al = v1.cross(d), a2 = v2.cross(d);
    if (a == 0) { //if parallel
        auto b1=s1.dot(v1), c1=e1.dot(v1),
            b2=s2.dot(v1), c2=e2.dot(v1);
        if (al || a2 || max(b1,min(b2,c2))>min(c1,max(b2,c2)))
            return 0;
        r1 = min(b2,c2)<b1 ? s1 : (b2<c2 ? s2 : e2);
        r2 = max(b2,c2)>c1 ? e1 : (b2>c2 ? s2 : e2);
        return 2-(r1==r2);
    }
    if (a < 0) { a = -a; al = -al; a2 = -a2; }
    if (0<al || a<-al || 0<a2 || a<-a2)
        return 0;
    r1 = s1-v1*a2/a;
    return 1;
}
```

SegmentIntersectionQ.h

Description: Like segmentIntersection, but only returns true/false. Products of three coordinates are used in intermediate steps so watch out for overflow if using int or long long.

16 lines

```
template <class P>
bool segmentIntersectionQ(P s1, P e1, P s2, P e2) {
    if (e1 == s1) {
        if (e2 == s2) return e1 == e2;
        swap(s1,s2); swap(e1,e2);
    }
    P v1 = e1-s1, v2 = e2-s2, d = s2-s1;
    auto a = v1.cross(v2), al = d.cross(v1), a2 = d.cross(v2);
    if (a == 0) { // parallel
        auto b1 = s1.dot(v1), c1 = e1.dot(v1),
            b2 = s2.dot(v1), c2 = e2.dot(v1);
        return !al && max(b1,min(b2,c2)) <= min(c1,max(b2,c2));
    }
    if (a < 0) { a = -a; al = -al; a2 = -a2; }
    return (0 <= al && al <= a && 0 <= a2 && a2 <= a);
}
```

### lineIntersection.h

**Description:**  
If a unique intersetion point of the lines going through s1,e1 and s2,e2 exists r is set to this point and 1 is returned. If no intersection point exists 0 is returned and if infinitely many exists -1 is returned. If s1==e1 or s2==e2 -1 is returned. The wrong position will be returned if P is Point<int> 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:** point<double> intersection;  
if (1 == LineIntersection(s1,e1,s2,e2,intersection))  
cout << "intersection point at " << intersection << endl;

"Point.h"

9 lines

```
template <class P>
int lineIntersection(const P& s1, const P& e1, const P& s2,
    const P& e2, P& r) {
    if ((e1-s1).cross(e2-s2)) { //if not parallel
        r = s2-(e2-s2)*(e1-s1).cross(s2-s1)/(e1-s1).cross(e2-s2);
        return 1;
    } else
        return -((e1-s1).cross(s2-s1)==0 || s2==e2);
}
```

### sideOf.h

**Description:** Returns where *p* is as seen from *s* towards *e*. 1/0/-1 ⇔ 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;

"Point.h"

11 lines

```
template <class P>
int sideOf(const P& s, const P& e, const P& p) {
    auto a = (e-s).cross(p-s);
    return (a > 0) - (a < 0);
}

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. Intended for use with e.g. Point<long long> where overflow is an issue. Use (segDist(s,e,p)<=epsilon) instead when using Point<double>.

"Point.h"

5 lines

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

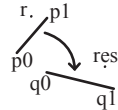
### linearTransformation.h

**Description:**  
Apply the linear transformation (translation, rotation and scaling) which takes line p0-p1 to line q0-q1 to point r.

"Point.h"

6 lines

```
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();
}
```



### Angle.h

**Description:** A class for ordering angles (as represented by int points and a number of rotations around the origin). Useful for rotational sweeping.  
**Usage:** vector<Angle> v = {w[0], w[0].t360() ...}; *// sorted*  
int j = 0; rep(i,0,n) {  
 while (v[j] < v[i].t180()) ++j;  
} *// sweeps j such that (j-i) represents the number of positively oriented triangles with vertices at 0 and i*

"Point.h"

37 lines

```
struct Angle {
    int x, y;
    int t;
    Angle(int x, int y, int t=0) : x(x), y(y), t(t) {}
    Angle operator-(Angle a) const { return {x-a.x, y-a.y, t}; }
    int quad() const {
        assert(x || y);
        if (y < 0) return (x >= 0) + 2;
        if (y > 0) return (x <= 0);
        return (x <= 0) * 2;
    }
    Angle t90() const { return {-y, x, t + (quad() == 3)}; }
    Angle t180() const { return {-x, -y, t + (quad() >= 2)}; }
    Angle t360() const { return {x, y, t + 1}; }
};

bool operator<(Angle a, Angle b) {
    // add a.dist2() and b.dist2() to also compare distances
    return make_tuple(a.t, a.quad(), a.y * (1l)b.x) <
        make_tuple(b.t, b.quad(), a.x * (1l)b.y);
}

bool operator>=(Angle a, Angle b) { return !(a < b); }
bool operator>(Angle a, Angle b) { return b < a; }
bool operator<=(Angle a, Angle b) { return !(b < a); }

// Given two points, this calculates the smallest angle between
// them, i.e., the angle that covers the defined line segment.
pair<Angle, Angle> segmentAngles(Angle a, Angle b) {
    if (b < a) swap(a, b);
    return (b < a.t180() ?
        make_pair(a, b) : make_pair(b, a.t360()));
}

Angle operator+(Angle a, Angle b) { // where b is a vector
    Angle r(a.x + b.x, a.y + b.y, a.t);
    if (r > a.t180()) r.t--;
    return r.t180() < a ? r.t360() : r;
}
```

## 8.2 Circles

### CircleIntersection.h

**Description:** Computes a pair of points at which two circles intersect. Returns false in case of no intersection.

"Point.h"

14 lines

```
typedef Point<double> P;
bool circleIntersection(P a, P b, double r1, double r2,
    pair<P, P>* out) {
    P delta = b - a;
    assert(delta.x || delta.y || r1 != r2);
    if (!delta.x && !delta.y) return false;
    double r = r1 + r2, d2 = delta.dist2();
    double p = (d2 + r1*r1 - r2*r2) / (2.0 * d2);
    double h2 = r1*r1 - p*p*d2;
    if (d2 > r*r || h2 < 0) return false;
    P mid = a + delta*p, per = delta.perp() * sqrt(h2 / d2);
    *out = {mid + per, mid - per};
    return true;
}
```

### circleTangents.h

**Description:**  
Returns a pair of the two points on the circle with radius r centered around c whos tangent lines intersect p. If p lies within the circle NaN-points are returned. P is intended to be Point<double>. The first point is the one to the right as seen from the p towards c.  
**Usage:** typedef Point<double> P;  
pair<P,P> p = circleTangents(P(100,2),P(0,0),2);

"Point.h"

6 lines

```
template <class P>
pair<P,P> circleTangents(const P &p, const P &c, double r) {
    P a = p-c;
    double x = r*r/a.dist2(), y = sqrt(x-x*x);
    return make_pair(c+a*x+a.perp()*y, c+a*x-a.perp()*y);
}

circumcircle.h
Description:
```

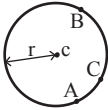
The circumcircle of a triangle is the circle intersecting all three vertices. ccRadius returns the radius of the circle going through points A, B and C and ccCenter returns the center of the same circle.

"Point.h"

9 lines

```
typedef Point<double> P;
double ccRadius(const P& A, const P& B, const P& C) {
    return (B-A).dist()*(C-B).dist()*(A-C).dist()/
        abs((B-A).cross(C-A))/2;
}

P ccCenter(const P& A, const P& B, const P& C) {
    P b = C-A, c = B-A;
    return A + (b*c.dist2()-c*b.dist2()).perp()/b.cross(c)/2;
}
```



### MinimumEnclosingCircle.h

**Description:** Computes the minimum circle that encloses a set of points.  
**Time:** expected  $\mathcal{O}(n)$

"circumcircle.h"

28 lines

```
pair<double, P> mec2(vector<P>& S, P a, P b, int n) {
    double hi = INFINITY, lo = -hi;
    rep(i,0,n) {
        auto si = (b-a).cross(S[i]-a);
        if (si == 0) continue;
        P m = ccCenter(a, b, S[i]);
        auto cr = (b-a).cross(m-a);
        if (si < 0) hi = min(hi, cr);
        else lo = max(lo, cr);
    }
    double v = (0 < lo ? lo : hi < 0 ? hi : 0);
    P c = (a + b) / 2 + (b - a).perp() * v / (b - a).dist2();
    return {(a - c).dist2(), c};
}

pair<double, P> mec(vector<P>& S, P a, int n) {
    random_shuffle(S.begin(), S.begin() + n);
    P b = S[0], c = (a + b) / 2;
    double r = (a - c).dist2();
    rep(i,1,n) if ((S[i] - c).dist2() > r * (1 + 1e-8)) {
        tie(r,c) = (n == sz(S) ?
            mec(S, S[i], i) : mec2(S, a, S[i], i));
    }
    return {r, c};
}

pair<double, P> enclosingCircle(vector<P> S) {
    assert(!S.empty()); auto r = mec(S, S[0], sz(S));
    return {sqrt(r.first), r.second};
}
```

### 8.3 Polygons

#### insidePolygon.h

**Description:** Returns true if p lies within the polygon described by the points between iterators begin and end. If strict false is returned when p is on the edge of the polygon. Answer is calculated by counting the number of intersections between the polygon and a line going from p to infinity in the positive x-direction. The algorithm uses products in intermediate steps so watch out for overflow. If points within epsilon from an edge should be considered as on the edge replace the line "if (onSegment..." with the comment bellow it (this will cause overflow for int and long long).

**Usage:** typedef Point<int> pi; vector<pi> v; v.push\_back(pi(4,4)); v.push\_back(pi(1,2)); v.push\_back(pi(2,1)); bool in = insidePolygon(v.begin(),v.end(), pi(3,4), false); **Time:**  $\mathcal{O}(n)$

```
"Point.h", "onSegment.h", "SegmentDistance.h" 14 lines

template <class It, class P>
bool insidePolygon(It begin, It end, const P& p,
    bool strict = true) {
    int n = 0; //number of isects with line from p to (inf,p.y)
    for (It i = begin, j = end-1; i != end; j = i++) {
        //if p is on edge of polygon
        if (onSegment(*i, *j, p)) return !strict;
        //or: if (segDist(*i, *j, p) <= epsilon) return !strict;
        //increment n if segment intersects line from p
        n += (max(i->y,j->y) > p.y && min(i->y,j->y) <= p.y &&
            ((*j-*i).cross(p-*i) > 0) == (i->y <= p.y));
    }
    return n&1; //inside if odd number of intersections
}
```

#### PolygonArea.h

**Description:** Returns twice the signed area of a polygon. Clockwise enumeration gives negative area. Watch out for overflow if using int as T!

```
"Point.h" 6 lines

template <class T>
T polygonArea2(vector<Point<T>>& v) {
    T a = v.back().cross(v[0]);
    rep(i,0,sz(v)-1) a += v[i].cross(v[i+1]);
    return a;
}
```

#### PolygonCenter.h

**Description:** Returns the center of mass for a polygon.

```
"Point.h" 10 lines

typedef Point<double> P;
Point<double> polygonCenter(vector<P>& v) {
    auto i = v.begin(), end = v.end(), j = end-1;
    Point<double> res(0,0); double A = 0;
    for (; i != end; j=i++) {
        res = res + (*i + *j) * j->cross(*i);
        A += j->cross(*i);
    }
    return res / A / 3;
}
```

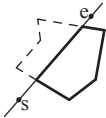
#### PolygonCut.h

**Description:** Returns a vector with the vertices of a polygon with everything 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" 15 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.emplace_back();
        lineIntersection(s, e, cur, prev, res.back());
    }
    if (side)
        res.push_back(cur);
}
return res;
}
```

#### ConvexHull.h

**Description:** Returns a vector of indices 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.

**Usage:** vector<P> ps, hull; trav(i, convexHull(ps)) hull.push\_back(ps[i]); **Time:**  $\mathcal{O}(n \log n)$

```
"Point.h" 20 lines

typedef Point<ll> P;
pair<vi, vi> ulHull(const vector<P>& S) {
    vi Q(sz(S)), U, L;
    iota(all(Q), 0);
    sort(all(Q), [&S](int a, int b){ return S[a] < S[b]; });
    trav(it, Q) {
#define ADDP(C, cmp) while (sz(C) > 1 && S[C[sz(C)-2]].cross(\
        S[it], S[C.back()]) cmp 0) C.pop_back(); C.push_back(it);
        ADDP(U, <=); ADDP(L, >=);
    }
    return {U, L};
}
```

```
vi convexHull(const vector<P>& S) {
    vi u, l; tie(u, l) = ulHull(S);
    if (sz(S) <= 1) return u;
    if (S[u[0]] == S[u[1]]) return {0};
    l.insert(l.end(), u.rbegin()+1, u.rend()-1);
    return l;
}
```

#### PolygonDiameter.h

**Description:** Calculates the max squared distance of a set of points.

```
"ConvexHull.h" 19 lines

vector<pii> antipodal(const vector<P>& S, vi& U, vi& L) {
    vector<pii> ret;
    int i = 0, j = sz(L) - 1;
    while (i < sz(U) - 1 || j > 0) {
        ret.emplace_back(U[i], L[j]);
        if (j == 0 || (i != sz(U)-1 && (S[L[j]] - S[L[j-1]])
            .cross(S[U[i+1]] - S[U[i]]) > 0)) ++i;
        else --j;
    }
    return ret;
}
```

```
pii polygonDiameter(const vector<P>& S) {
    vi U, L; tie(U, L) = ulHull(S);
    pair<ll, pii> ans;
    trav(x, antipodal(S, U, L))
        ans = max(ans, {S[x.first] - S[x.second].dist2(), x});
    return ans.second;
}
```



#### PointInsideHull.h

**Description:** Determine whether a point t lies inside a given polygon (counter-clockwise order). The polygon must be such that every point on the circumference is visible from the first point in the vector. It returns 0 for points outside, 1 for points on the circumference, and 2 for points inside.

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

"Point.h", "sideOf.h", "onSegment.h" 22 lines

typedef Point<ll> P;
int insideHull2(const vector<P>& H, int L, int R, const P& p) {
    int len = R - L;
    if (len == 2) {
        int sa = sideOf(H[0], H[L], p);
        int sb = sideOf(H[L], H[L+1], p);
        int sc = sideOf(H[L+1], H[0], p);
        if (sa < 0 || sb < 0 || sc < 0) return 0;
        if (sb==0 || (sa==0 && L == 1) || (sc == 0 && R == sz(H)))
            return 1;
        return 2;
    }
    int mid = L + len / 2;
    if (sideOf(H[0], H[mid], p) >= 0)
        return insideHull2(H, mid, R, p);
    return insideHull2(H, L, mid+1, p);
}
```

```
int insideHull(const vector<P>& hull, const P& p) {
    if (sz(hull) < 3) return onSegment(hull[0], hull.back(), p);
    else return insideHull2(hull, 1, sz(hull), p);
}
```

#### LineHullIntersection.h

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

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

```
"Point.h" 63 lines

ll sgn(ll a) { return (a > 0) - (a < 0); }
typedef Point<ll> P;
struct HullIntersection {
    int N;
    vector<P> p;
    vector<pair<P, int>> a;

    HullIntersection(const vector<P>& ps) : N(sz(ps)), p(ps) {
        p.insert(p.end(), all(ps));
        int b = 0;
        rep(i,1,N) if (P{p[i].y,p[i].x} < P{p[b].y, p[b].x}) b = i;
        rep(i,0,N) {
            int f = (i + b) % N;
            a.emplace_back(p[f+1] - p[f], f);
        }

        int qd(P p) {
            return (p.y < 0) ? (p.x >= 0) + 2
                : (p.x <= 0) * (1 + (p.y <= 0));
        }
}
```

```
int bs(P dir) {
    int lo = -1, hi = N;
    while (hi - lo > 1) {
        int mid = (lo + hi) / 2;
        if (make_pair(qd(dir), dir.y * a[mid].first.x) <
            make_pair(qd(a[mid].first), dir.x * a[mid].first.y))
```

```
        hi = mid;
    } else lo = mid;
}
return a[hi%N].second;
}

bool isign(P a, P b, int x, int y, int s) {
    return sgn(a.cross(p[x], b)) * sgn(a.cross(p[y], b)) == s;
}

int bs2(int lo, int hi, P a, P b) {
    int L = lo;
    if (hi < lo) hi += N;
    while (hi - lo > 1) {
        int mid = (lo + hi) / 2;
        if (isign(a, b, mid, L, -1)) hi = mid;
        else lo = mid;
    }
    return lo;
}

pii isct(P a, P b) {
    int f = bs(a - b), j = bs(b - a);
    if (isign(a, b, f, j, 1)) return {-1, -1};
    int x = bs2(f, j, a, b)%N,
        y = bs2(j, f, a, b)%N;
    if (a.cross(p[x], b) == 0 &&
        a.cross(p[x+1], b) == 0) return {x, x};
    if (a.cross(p[y], b) == 0 &&
        a.cross(p[y+1], b) == 0) return {y, y};
    if (a.cross(p[f], b) == 0) return {f, -1};
    if (a.cross(p[j], b) == 0) return {j, -1};
    return {x, y};
}
};
```

## 8.4 Misc. Point Set Problems

### closestPair.h

**Description:**  $i1, i2$  are the indices to the closest pair of points in the point vector  $p$  after the call. The distance is returned.  
**Time:**  $\mathcal{O}(n \log n)$

```
"Point.h" 58 lines

template <class It>
bool it_less(const It& i, const It& j) { return *i < *j; }
template <class It>
bool y_it_less(const It& i, const It& j) {return i->y < j->y;}

template<class It, class IIt> /* IIt = vector<It>::iterator */
double cp_sub(IIt ya, IIt yaend, IIt xa, It &i1, It &i2) {
    typedef typename iterator_traits<It>::value_type P;
    int n = yaend-ya, split = n/2;
    if(n <= 3) { // base case
        double a = (*xa[1]-*xa[0]).dist(), b = 1e50, c = 1e50;
        if(n==3) b=(*xa[2]-*xa[0]).dist(), c=(*xa[2]-*xa[1]).dist()
            ;
        if(a <= b) { i1 = xa[1];
            if(a <= c) return i2 = xa[0], a;
            else return i2 = xa[2], c;
        } else { i1 = xa[2];
            if(b <= c) return i2 = xa[0], b;
            else return i2 = xa[1], c;
        }
    }
    vector<It> ly, ry, stripy;
    P splitp = *xa[split];
    double splitx = splitp.x;
    for(IIt i = ya; i != yaend; ++i) { // Divide
```

```
        if(*i != xa[split] && (**i-splitp).dist2() < 1e-12)
            return i1 = *i, i2 = xa[split], 0; // nasty special case!
        if (**i < splitp) ly.push_back(*i);
        else ry.push_back(*i);
    } // assert((signed)lefty.size() == split)
    It j1, j2; // Conquer
    double a = cp_sub(ly.begin(), ly.end(), xa, i1, i2);
    double b = cp_sub(ry.begin(), ry.end(), xa+split, j1, j2);
    if(b < a) a = b, i1 = j1, i2 = j2;
    double a2 = a*a;
    for(IIt i = ya; i != yaend; ++i) { // Create strip (y-sorted)
        double x = (*i)->x;
        if(x >= splitx-a && x <= splitx+a) stripy.push_back(*i);
    }
    for(IIt i = stripy.begin(); i != stripy.end(); ++i) {
        const P &p1 = **i;
        for(IIt j = i+1; j != stripy.end(); ++j) {
            const P &p2 = **j;
            if(p2.y-p1.y > a) break;
            double d2 = (p2-p1).dist2();
            if(d2 < a2) i1 = *i, i2 = *j, a2 = d2;
        }
    }
    return sqrt(a2);
}

template<class It> // It is random access iterators of point<T>
double closestpair(It begin, It end, It &i1, It &i2 ) {
    vector<It> xa, ya;
    assert(end-begin >= 2);
    for (It i = begin; i != end; ++i)
        xa.push_back(i), ya.push_back(i);
    sort(xa.begin(), xa.end(), it_less<It>);
    sort(ya.begin(), ya.end(), y_it_less<It>);
    return cp_sub(ya.begin(), ya.end(), xa.begin(), i1, i2);
}
```

### kdTree.h

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

```
"Point.h" 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 the box is wider than high (not best
            heuristic...)
            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);
    }
};
```

### DelaunayTriangulation.h

**Description:** Computes the Delaunay triangulation of a set of points. Each circumcircle contains none of the input points. If any three points are colinear or any four are on the same circle, behavior is undefined.

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

```
"Point.h", "3dHull.h" 10 lines

template<class P, class F>
void delaunay(vector<P>& ps, F trifun) {
    if (sz(ps) == 3) { int d = (ps[0].cross(ps[1], ps[2]) < 0);
        trifun(0,1+d,2-d); }
    vector<P> p3;
    trav(p, ps) p3.emplace_back(p.x, p.y, p.dist2());
    if (sz(ps) > 3) trav(t, hull3d(p3)) if ((p3[t.b]-p3[t.a]).
        cross(p3[t.c]-p3[t.a]).dot(P3(0,0,1)) < 0)
        trifun(t.a, t.c, t.b);
}
```

## 8.5 3D

### PolyhedronVolume.h

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

```
"Point.h", "3dHull.h" 6 lines

template <class V, class L>
double signed_poly_volume(const V& p, const L& trilst) {
    double v = 0;
    trav(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.

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

3dHull.h

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

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

```
49 lines
"Point3D.h"
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);
    }
    }
    trav(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. 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.

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

## 8.6 Java

JavaGeometry.java

**Description:** Java Geometry

```
66 lines
import java.util.*;
import java.awt.geom.*;
import java.io.*;

public class JavaGeometry {
    // make an Area object from the coordinates of a polygon
    static Area makeArea(double[] pts) {
        Path2D.Double p = new Path2D.Double();
        p.moveTo(pts[0], pts[1]);
        for (int i = 2; i < pts.length; i += 2) p.lineTo(pts[i]
            , pts[i+1]);
        p.closePath();
        return new Area(p);
    }

    // compute the area of an Area object containing several
    disjoint polygons
    static double computeArea(Area area) {
        double totArea = 0;
        PathIterator iter = area.getPathIterator(null);
```

```
ArrayList<Point2D.Double> points = new ArrayList<
    Point2D.Double>();

while (!iter.isDone()) {
    double[] buffer = new double[6];
    switch (iter.currentSegment(buffer)) {
        case PathIterator.SEG_MOVETO:
        case PathIterator.SEG_LINETO:
            points.add(new Point2D.Double(buffer[0], buffer
                [1]));
            break;
        case PathIterator.SEG_CLOSE:
            totArea += computePolygonArea(points);
            points.clear();
            break;
    }
    iter.next();
}
return totArea;

public static void main(String args[]) throws Exception {
    // areaB.subtract(areaA);
    // areaB.exclusiveOr (areaA);
    // areaB.add (areaA);
    // areaB.intersect (areaA);

    // (1) determine whether B - A is a single closed shape
    // (as
    //     opposed to multiple shapes)
    boolean isSingle = areaB.isSingular();
    // also,
    // areaB.isEmpty();
    // areaB.contains(x, y); // doubles
    // Finally, some useful things we didn't use in this
    // example:

    // Ellipse2D.Double ellipse = new Ellipse2D.Double (
    //     double x, double y,
    //     double w, double h);

    //     creates an ellipse inscribed in box with bottom-
    //     left corner (x,y)
    //     and upper-right corner (x+y,w+h)

    // Rectangle2D.Double rect = new Rectangle2D.Double (
    //     double x, double y,
    //     double w, double h);

    //     creates a box with bottom-left corner (x,y) and
    //     upper-right
    //     corner (x+y,w+h)

    // Each of these can be embedded in an Area object (e.g
    // ., new Area (rect)).
}
```

## 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 This is used by find to find all occurrences of a string.

**Usage:** vi p = pi(pattern); vi occ = find(word, p);

<b>Time:</b> $\mathcal{O}(\textit{pattern})$ for pi, $\mathcal{O}(\textit{word} + \textit{pattern})$ for find	16 lines
<pre>vi pi(const string&amp; s) {     vi p(sz(s));     rep(i,1,sz(s)) {         int g = p[i-1];         while (g &amp;&amp; s[i] != s[g]) g = p[g-1];         p[i] = g + (s[i] == s[g]);     }     return p; }  vi match(const string&amp; s, const string&amp; 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; }</pre>	

### 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).

<b>Time:</b> $\mathcal{O}(N)$	11 lines
<pre>void manacher(const string&amp; s) {     int n = sz(s);     vi p[2] = {vi(n+1), vi(n)};     rep(z,0,2) for (int i=0,l=0,r=0; i &lt; n; i++) {         int t = r-i+!z;         if (i&lt;r) p[z][i] = min(t, p[z][l+t]);         int L = i-p[z][i], R = i+p[z][i]-!z;         while (L&gt;=1 &amp;&amp; R+1&lt;n &amp;&amp; s[L-1] == s[R+1])             p[z][i]++, L--, R++;         if (R&gt;r) l=L, r=R;     } }</pre>	

### MinRotation.h

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

	8 lines
<pre>int min_rotation(string s) {     int a=0, N=sz(s); s += s;     rep(b,0,N) rep(i,0,N) {         if (a+i == b    s[a+i] &lt; s[b+i]) {b += max(0, i-1); break;}         if (s[a+i] &gt; s[b+i]) { a = b; break; }     }     return a; }</pre>	

### SuffixArray.h

**Description:** Builds suffix array for a string.  $a[z]$  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  $a[0] = n$ . The lcp function calculates longest common prefixes for neighbouring strings in suffix array. The returned vector is of size  $n+1$ , and  $ret[0] = 0$ .

**Memory:**  $\mathcal{O}(N)$   
**Time:**  $\mathcal{O}(N \log^2 N)$  where  $N$  is the length of the string for creation of the SA.  $\mathcal{O}(N)$  for longest common prefixes.

	61 lines
<pre>typedef pair&lt;ll, int&gt; pli; void count_sort(vector&lt;pli&gt; &amp;b, int bits) { // (optional)     //this is just 3 times faster than stl sort for N=10^6     int mask = (1 &lt;&lt; bits) - 1;     rep(it,0,2) {         int move = it * bits;         vi q(1 &lt;&lt; bits), w(sz(q) + 1);         rep(i,0,sz(b))</pre>	

<pre>        q[(b[i].first &gt;&gt; move) &amp; mask]++;         partial_sum(q.begin(), q.end(), w.begin() + 1);         vector&lt;pli&gt; res(b.size());         rep(i,0,sz(b))             res[w[(b[i].first &gt;&gt; move) &amp; mask]++] = b[i];         swap(b, res);     } }  struct SuffixArray {     vi a;     string s;     SuffixArray(const string&amp; _s) : s(_s + '\0') {         int N = sz(s);         vector&lt;pli&gt; b(N);         a.resize(N);         rep(i,0,N) {             b[i].first = s[i];             b[i].second = i;         }          int q = 8;         while ((1 &lt;&lt; q) &lt; N) q++;         for (int moc = 0;; moc++) {             count_sort(b, q); // sort(all(b)) can be used as well             a[b[0].second] = 0;             rep(i,1,N)                 a[b[i].second] = a[b[i - 1].second] +                     (b[i - 1].first != b[i].first);              if ((1 &lt;&lt; moc) &gt;= N) break;             rep(i,0,N) {                 b[i].first = (ll)a[i] &lt;&lt; q;                 if (i + (1 &lt;&lt; moc) &lt; N)                     b[i].first += a[i + (1 &lt;&lt; moc)];                 b[i].second = i;             }         }         rep(i,0,sz(a)) a[i] = b[i].second;     } }  vi lcp() {     // longest common prefixes: res[i] = lcp(a[i], a[i-1])     int n = sz(a), h = 0;     vi inv(n), res(n);     rep(i,0,n) inv[a[i]] = i;     rep(i,0,n) if (inv[i] &gt; 0) {         int p0 = a[inv[i] - 1];         while (s[i + h] == s[p0 + h]) h++;         res[inv[i]] = h;         if (h &gt; 0) h--;     }     return res; } };</pre>	
---	--

### 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).

<b>Time:</b> $\mathcal{O}(26N)$	50 lines
<pre>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;</pre>	

<pre>void ukkadd(int i, int c) { suff:     if (r[v]&lt;=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&lt;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;     } }</pre>	
<pre>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])); }</pre>	
<pre>// example: find longest common substring (uses ALPHA = 28) pii best; int lcs(int node, int i1, int i2, int olen) {     if (l[node] &lt;= i1 &amp;&amp; i1 &lt; r[node]) return 1;     if (l[node] &lt;= i2 &amp;&amp; i2 &lt; 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; } };</pre>	

### Hashing.h

**Description:** Various self-explanatory methods for string hashing. 45 lines

<pre>typedef unsigned long long H; static const H C = 123891739; // arbitrary</pre>	
<pre>// Arithmetic mod 2^64-1. 5x slower than mod 2^64 and more // code, but works on evil test data (e.g. Thue-Morse). // "typedef H K;" instead if you think test data is random. struct K {     typedef __uint128_t H2;     H x; K(H x=0) : x(x) {}     K operator+(K o){ return x + o.x + H(((H2)x + o.x)&gt;&gt;64); }     K operator*(K o){ return K(x*o.x) + H(((H2)x * o.x)&gt;&gt;64); }     H operator-(K o) { K a = *this + ~o.x; return a.x + !~a.x; } };</pre>	
<pre>struct HashInterval {     vector&lt;K&gt; ha, pw;     HashInterval(string&amp; str) : ha(sz(str)+1), pw(ha) {         pw[0] = 1;         rep(i,0,sz(str))             ha[i+1] = ha[i] * C + str[i],             pw[i+1] = pw[i] * C;</pre>	



```
    }
    H hashInterval(int a, int b) { // hash [a, b)
        return ha[b] - ha[a] * pw[b - a];
    }
};

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

H hashString(string& s) {
    K h = 0;
    trav(c, s) h = h * C + c;
    return h - 0;
}
```

**AhoCorasick.h**  
**Description:** Aho-Corasick tree is used for multiple pattern matching. Initialize the tree with create(patterns). find(word) returns for each position the index of the longest word that ends there, or -1 if none. findAll(., word) finds all words (up to  $N\sqrt{N}$  many if no duplicate patterns) that start at each position (shortest first). Duplicate patterns are allowed; empty patterns are not. To find the longest words that start at each position, reverse all input. **Time:** Function create is  $\mathcal{O}(26N)$  where  $N$  is the sum of length of patterns. find is  $\mathcal{O}(M)$  where  $M$  is the length of the word. findAll is  $\mathcal{O}(NM)$ ..67 lines

```
struct AhoCorasick {
    enum {alpha = 26, first = 'A'};
    struct Node {
        // (nmatches is optional)
        int back, next[alpha], start = -1, end = -1, nmatches = 0;
        Node(int v) { memset(next, v, sizeof(next)); }
    };
    vector<Node> N;
    vector<int> backp;
    void insert(string& s, int j) {
        assert(!s.empty());
        int n = 0;
        trav(c, s) {
            int& m = N[n].next[c - first];
            if (m == -1) { n = m = sz(N); N.emplace_back(-1); }
            else n = m;
        }
        if (N[n].end == -1) N[n].start = j;
        backp.push_back(N[n].end);
        N[n].end = j;
        N[n].nmatches++;
    }
    AhoCorasick(vector<string>& pat) {
        N.emplace_back(-1);
        rep(i, 0, sz(pat)) insert(pat[i], i);
        N[0].back = sz(N);
        N.emplace_back(0);

        queue<int> q;
        for (q.push(0); !q.empty(); q.pop()) {
            int n = q.front(), prev = N[n].back;
            rep(i, 0, alpha) {
                int &ed = N[n].next[i], y = N[prev].next[i];
                if (ed == -1) ed = y;
            }
        }
    }
};
```

```
        else {
            N[ed].back = y;
            (N[ed].end == -1 ? N[ed].end : backp[N[ed].start])
                = N[y].end;
            N[ed].nmatches += N[y].nmatches;
            q.push(ed);
        }
    }
}

vi find(string word) {
    int n = 0;
    vi res; // ll count = 0;
    trav(c, word) {
        n = N[n].next[c - first];
        res.push_back(N[n].end);
        // count += N[n].nmatches;
    }
    return res;
}

vector<vi> findAll(vector<string>& pat, string word) {
    vi r = find(word);
    vector<vi> res(sz(word));
    rep(i, 0, sz(word)) {
        int ind = r[i];
        while (ind != -1) {
            res[i - sz(pat[ind]) + 1].push_back(ind);
            ind = backp[ind];
        }
    }
    return res;
}

};
```

## Various (10)

### 10.1 Intervals

**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:**  $\mathcal{O}(\log N)$ 25 lines

```
template <class T>
auto addInterval(set<pair<T, T>>& is, T L, T R) {
    if (L == R) return is.end();
    auto it = is.lower_bound({L, R}), before = it;
    while (it != is.end() && it->first <= R) {
        R = max(R, it->second);
        before = it = is.erase(it);
    }
    if (it != is.begin() && (--it)->second >= L) {
        L = min(L, it->first);
        R = max(R, it->second);
        is.erase(it);
    }
    return is.insert(before, {L, R});
}

template <class T>
void removeInterval(set<pair<T, T>>& is, T L, T R) {
    if (L == R) return;
    auto it = addInterval(is, L, R);
    T r2 = it->second;
    if (it->first == L) is.erase(it);
    else (T&)it->second = L;
}
```

**IntervalCover.h**  
**Description:** Compute indices of smallest set of intervals covering another interval. Intervals should be [inclusive, exclusive). To support [inclusive, inclusive], change (A) to add || R.empty(). Returns empty set on failure (or if G is empty). **Time:**  $\mathcal{O}(N \log N)$ 19 lines

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

**ConstantIntervals.h**  
**Description:** Split a monotone function on [from, to) into a minimal set of half-open intervals on which it has the same value. Runs a callback g for each such interval. **Usage:** constantIntervals(0, sz(v), [&](int x){return v[x];}, [&](int lo, int hi, T val){...}); **Time:**  $\mathcal{O}(k \log \frac{n}{k})$ 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) < ... < f(i) ≥ ... ≥ 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:**  $\mathcal{O}(\log(b - a))$ 13 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)
            a = mid;
        else
            b = mid+1;
    }
    rep(i,a+1,b+1) if (f(a) < f(i)) a = i; // (B)
    return a;
}
```

Karatsuba.h

**Description:** Faster-than-naive convolution of two sequences:  $c[x] = \sum a[i]b[x-i]$ . Uses the identity  $(aX + b)(cX + d) = acX^2 + bd + ((a + c)(b + d) - ac - bd)X$ . Doesn't handle sequences of very different length well. See also FFT, under the Numerical chapter.  
**Time:**  $\mathcal{O}(N^{1.6})$

1 lines

LIS.h

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

17 lines

```
template<class I> vi lis(vector<I> S) {
    vi prev(sz(S));
    typedef pair<I, int> p;
    vector<p> res;
    rep(i,0,sz(S)) {
        p el { S[i], i };
        //S[i]+1 for non-decreasing
        auto it = lower_bound(all(res), p { S[i], 0 });
        if (it == res.end()) res.push_back(el), it = --res.end();
        *it = el;
        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;
}
```

LCS.h

**Description:** Finds the longest common subsequence.  
**Memory:**  $\mathcal{O}(nm)$ .  
**Time:**  $\mathcal{O}(nm)$  where n and m are the lengths of the sequences.

14 lines

```
template <class T> T lcs(const T &X, const T &Y) {
    int a = sz(X), b = sz(Y);
    vector<vi> dp(a+1, vi(b+1));
    rep(i,1,a+1) rep(j,1,b+1)
        dp[i][j] = X[i-1]==Y[j-1] ? dp[i-1][j-1]+1 :
            max(dp[i][j-1],dp[i-1][j]);
    int len = dp[a][b];
    T ans(len,0);
    while(a && b)
        if(X[a-1]==Y[b-1]) ans[--len] = X[--a], --b;
        else if(dp[a][b-1]>dp[a-1][b]) --b;
        else --a;
    return ans;
}
```

## 10.3 Dynamic programming

### DivideAndConquerDP.h

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

```
struct DP { // 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); }
};
```

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

## 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` violations 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

### 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 for loops and optimizes floating points better (assumes associativity and turns off denormals).
- `#pragma GCC target ("avx,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).

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

### BumpAllocatorSTL.h

**Description:** BumpAllocator for STL containers.  
**Usage:** `vector<vector<int, small<int>>> ed(N);`

### Unrolling.h

```
#define F {...; ++i;}
int i = from;
while (i&3 && i < to) F // for alignment, if needed
while (i + 4 <= to) { F F F F }
while (i < to) F
```

### 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 "immintrin.h"

typedef _mm256i 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 mmm

// 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_epi16(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;
}
```

Techniques (A)

techniques.txt	159 lines
Recursion	
Divide and conquer	
Finding interesting points in N log N	
Algorithm analysis	
Master theorem	
Amortized time complexity	
Greedy algorithm	
Scheduling	
Max contiguous subvector sum	
Invariants	
Huffman encoding	
Graph teory	
Dynamic graphs (extra book-keeping)	
Breadth first search	
Depth first search	
* Normal trees / DFS trees	
Dijkstra's algoritm	
MST: Prim's algoritm	
Bellman-Ford	
Konig's theorem and vertex cover	
Min-cost max flow	
Lovasz toggle	
Matrix tree theorem	
Maximal matching, general graphs	
Hopcroft-Karp	
Hall's marriage theorem	
Graphical sequences	
Floyd-Warshall	
Eulercykler	
Flow networks	
* Augumenting paths	
* Edmonds-Karp	
Bipartite matching	
Min. path cover	
Topological sorting	
Strongly connected components	
2-SAT	
Cutvertices, cutedges och biconnected components	
Edge coloring	
* Trees	
Vertex coloring	
* Bipartite graphs (=> trees)	
* 3^n (special case of set cover)	
Diameter and centroid	
K'th shortest path	
Shortest cycle	
Dynamic programming	
Knapsack	
Coin change	
Longest common subsequence	
Longest increasing subsequence	
Number of paths in a dag	
Shortest path in a dag	
Dynprog over intervals	
Dynprog over subsets	
Dynprog over probabilities	
Dynprog over trees	
3^n set cover	
Divide and conquer	
Knuth optimization	
Convex hull optimizations	
RMQ (sparse table a.k.a 2^k-jumps)	
Bitonic cycle	
Log partitioning (loop over most restricted)	
Combinatorics	

Computation of binomial coefficients
Pigeon-hole principle
Inclusion/exclusion
Catalan number
Pick's theorem
Number theory
Integer parts
Divisibility
Euklidean algorithm
Modular arithmetic
* Modular multiplication
* Modular inverses
* Modular exponentiation by squaring
Chinese remainder theorem
Fermat's small theorem
Euler's theorem
Phi function
Frobenius number
Quadratic reciprocity
Pollard-Rho
Miller-Rabin
Hensel lifting
Vieta root jumping
Game theory
Combinatorial games
Game trees
Mini-max
Nim
Games on graphs
Games on graphs with loops
Grundy numbers
Bipartite games without repetition
General games without repetition
Alpha-beta pruning
Probability theory
Optimization
Binary search
Ternary search
Unimodality and convex functions
Binary search on derivative
Numerical methods
Numeric integration
Newton's method
Root-finding with binary/ternary search
Golden section search
Matrices
Gaussian elimination
Exponentiation by squaring
Sorting
Radix sort
Geometry
Coordinates and vectors
* Cross product
* Scalar product
Convex hull
Polygon cut
Closest pair
Coordinate-compression
Quadtrees
KD-trees
All segment-segment intersection
Sweeping
Discretization (convert to events and sweep)
Angle sweeping
Line sweeping
Discrete second derivatives
Strings
Longest common substring
Palindrome subsequences

Knuth-Morris-Pratt
Tries
Rolling polynom hashes
Suffix array
Suffix tree
Aho-Corasick
Manacher's algorithm
Letter position lists
Combinatorial search
Meet in the middle
Brute-force with pruning
Best-first (A*)
Bidirectional search
Iterative deepening DFS / A*
Data structures
LCA (2^k-jumps in trees in general)
Pull/push-technique on trees
Heavy-light decomposition
Centroid decomposition
Lazy propagation
Self-balancing trees
Convex hull trick (wcipeg.com/wiki/Convex_hull_trick)
Monotone queues / monotone stacks / sliding queues
Sliding queue using 2 stacks
Persistent segment tree