Indian Institute of Technology Kharagpur

AlooParatha

Aashirwaad Mishra, Sayandeep Bhowmick, Sujan Jain

2024-12-19

Contest (1)

template.cpp

30 lines

```
#include <bits/stdc++.h>
// #pragma GCC optimize("O3,unroll-loops")
using namespace std;
#define fr(i, a, b) for(int i = a; i < (b);
    ++i)
#define rev(i, a, b) for (ll i = a; i >= b;
    i--)
#define push_back pb
#define all(x) begin(x), end(x)
#define sz(x) (int)(x).size()
using ll = long long;
#define int long long
typedef pair<int, int> pii;
typedef vector<int> vi;
const ll mod1 = 1e9+7, mod2 = 998244353;
const ll INF = 1e9, INF2 = 1e18;
#define ff first
#define ss second
void solve(){}
int32_t main()
{
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(NULL);
    #ifndef ONLINE_JUDGE
        freopen("input.txt","r",stdin);
        freopen("output.txt","w",stdout);
    #endif
    int t=1;cin >> t;
    while(t--) solve();
    return 0;
}
```

Mathematics (2)

2.1 Equations

$$ax^2 + bx + c = 0 \Rightarrow x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

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

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

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

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

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

2.2 Recurrences

If $a_n = c_1 a_{n-1} + \cdots + c_k a_{n-k}$, and $r_1, \ldots, r_k$ are distinct roots of $x^k - c_1 x^{k-1} - \cdots - c_k$, there are $d_1, \ldots, d_k$ s.t.

$$a_n = d_1 r_1^n + \cdots + d_k r_k^n.$$

Non-distinct roots r become polynomial factors, e.g. $a_n = (d_1 n + d_2) r^n$.

2.3 Trigonometry

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

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

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

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

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

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

2.4 Geometry

2.4.1 Triangles

Side lengths: a, b, c

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

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

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

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

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

Length of bisector (divides angles in two):

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

Law of sines: $\dfrac{\sin \alpha}{a} = \dfrac{\sin \beta}{b} = \dfrac{\sin \gamma}{c} = \dfrac{1}{2R}$

Law of cosines: $a^2 = b^2 + c^2 - 2bc\cos\alpha$

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

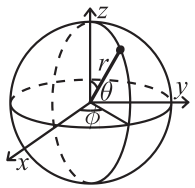
2.4.2 Quadrilaterals

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

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

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

2.4.3 Spherical coordinates



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

2.5 Derivatives/Integrals

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

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

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

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

2.7 Series

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

2.8 Probability theory

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

Expectation is linear:

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

For independent X and Y,

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

2.8.1 Discrete distributions

Binomial distribution

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

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

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

$\operatorname{Bin}(n, p)$ 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 $\operatorname{Fs}(p)$, $0 \leq p \leq 1$.

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

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

Poisson distribution

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

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

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

2.8.2 Continuous distributions

Uniform distribution

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

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

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

Exponential distribution

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

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

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

Normal distribution

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

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

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

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

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 $X_1, X_2, \ldots$ be a sequence of random variables generated by the Markov process. Then there is a transition matrix $\mathbf{P} = (p_{ij})$, with $p_{ij} = \Pr(X_n = i | X_{n-1} = j)$, and $\mathbf{p}^{(n)} = \mathbf{P}^n \mathbf{p}^{(0)}$ is the probability distribution for X_n (i.e., $p_i^{(n)} = \Pr(X_n = i)$), where $\mathbf{p}^{(0)}$ is the initial distribution.

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

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

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

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

2.9.1 Gambler's Ruin Problem

A gambler starts with an initial fortune of i dollars. On each game, the gambler wins \$1 with probability p or loses \$1 with probability $q = 1-p$, where $0 \leq p \leq 1$. The gambler will stop playing if either N dollars are accumulated or all money has been lost.

The natural question is: what is the probability that the gambler will end up with N dollars?

$$P_i = \begin{cases} \frac{1-\left(\frac{q}{p}\right)^i}{1-\left(\frac{q}{p}\right)^N} & \text{if } p \neq q, \\ \frac{i}{N} & \text{if } p = q = 0.5. \end{cases}$$

The expected number of moves to stop is given by:

$$E(\text{moves}) = i(N-i).$$

2.9.2 General Random Walk

Let $a>0$ and $b>0$ be integers, and let R_n denote a simple random walk with $R_0 = 0$. Let:

$$p(a) = P(R_n \text{ hits level } a \text{ before hitting level } -b).$$

By letting $a = N-i$ and $b = i$ (so that $N = a+b$), we can imagine a gambler who starts with $i = b$ and wishes to reach $N = a+b$ before going broke. So we can compute $p(a)$ by casting the problem into the framework of the gambler's ruin problem:

$$p(a) = P_i \quad \text{where } N = a+b,\ i = b.$$

The following equation holds:

$$p(a) = \begin{cases} \frac{1-\left(\frac{q}{p}\right)^b}{1-\left(\frac{q}{p}\right)^{a+b}} & \text{if } p \neq q, \\ \frac{b}{a+b} & \text{if } p = q = 0.5. \end{cases}$$

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. To get a map, change null_type.
Time: $\mathcal{O}(\log N)$

<ext/pb_ds/assoc_container.hpp>, <ext/pb_ds/tree_policy.hpp> 819d08, 15 lines

```
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
}
```

HashMap.h

Description: Hash map with mostly the same API as unordered_map, but ~3x faster. Uses 1.5x memory. Initial capacity must be a power of 2 (if provided).

d77092, 7 lines

```
#include <bits/extc++.h>
// To use most bits rather than just the
    lowest ones:
struct chash { // large odd number for C
  const uint64_t C = ll(4e18 * acos(0)) |
      71;
  ll operator()(ll x) const { return
      __builtin_bswap64(x*C); }
};
__gnu_pbds::gp_hash_table<ll,int,chash> h({}
    ,{},{},{},{1<<16});
```

SegmentTree.h

Description: Zero-indexed max-tree. Bounds are inclusive to the left and exclusive to the right. Can be changed by modifying T, f and unit.
Time: $\mathcal{O}(\log N)$

0f4bdb, 19 lines

```
struct Tree {
  typedef int T;
  static constexpr T unit = INT_MIN;
  T f(T a, T b) { return max(a, b); } // (
      any associative fn)
  vector<T> s; int n;
  Tree(int n = 0, T def = unit) : s(2*n, def
      ), n(n) {}
  void update(int pos, T val) {
    for (s[pos += n] = val; pos /= 2;)
      s[pos] = f(s[pos * 2], s[pos * 2 + 1])
          ;
  }
  T query(int b, int e) { // query [b, e)
    T ra = unit, rb = unit;
    for (b += n, e += n; b < e; b /= 2, e /=
        2) {
      if (b % 2) ra = f(ra, s[b++]);
      if (e % 2) rb = f(s[--e], rb);
    }
    return f(ra, rb);
  }
};
```

LazySegmentTree.h

Description: LazySegmentTree.h

0666ae, 70 lines

```
class LazySegmentTree {
private:
    vector<int> t, lazy;
    int n;

    void build(vector<int>& a, int v, int tl
        , int tr) {
        if (tl == tr) {
            t[v] = a[tl];
        } else {
            int tm = (tl + tr) / 2;
            build(a, v*2, tl, tm);
            build(a, v*2+1, tm+1, tr);
            t[v] = combine(t[v*2], t[v*2 +
                1]);
        }
    }

    void push(int v) {
        t[v*2] += lazy[v];
        lazy[v*2] += lazy[v];
        t[v*2+1] += lazy[v];
        lazy[v*2+1] += lazy[v];
        lazy[v] = 0;
    }

    void update(int v, int tl, int tr, int l
        , int r, int addend) {
        if (l > r)
            return;
        if (l == tl && tr == r) {
            t[v] += addend;
            lazy[v] += addend;
        } else {
            push(v);
            int tm = (tl + tr) / 2;
            update(v*2, tl, tm, l, min(r, tm
                ), addend);
            update(v*2+1, tm+1, tr, max(l,
                tm+1), r, addend);
            t[v] = combine(t[v*2], t[v*2+1])
                ;
        }
    }

    int query(int v, int tl, int tr, int l,
        int r) {
        if (l > r)
            return -INF;
        if (l == tl && tr == r)
            return t[v];
        push(v);
        int tm = (tl + tr) / 2;
        return combine(query(v*2, tl, tm, l,
             min(r, tm)),
                        query(v*2+1, tm+1, tr
                            , max(l, tm+1), r
                            ));
    }

    int combine(int a, int b) {
        return max(a, b); // Change this
            according to your requirement
    }
```

```
public:
    LazySegmentTree(vector<int>& a) {
        n = a.size();
        t.assign(4*n, 0);
        lazy.assign(4*n, 0);
        build(a, 1, 0, n-1);
    }

    void update(int l, int r, int addend) {
        update(1, 0, n-1, l, r, addend);
    }

    int query(int l, int r) {
        return query(1, 0, n-1, l, r);
    }
};
```

UnionFind.h

Description: UnionFind.h

3624b6, 17 lines

```
struct DSU
{
  vi par,size;
    DSU(int n) : par(n), size(n, 1) { iota(
        par.begin(), par.end(), 0); }
  int find(int x){return x == par[x] ? x :
      par[x] = find(par[x]);}
  void merge(int x, int y)
    {
        int nx = find(x);
        int ny = find(y);
        if(nx!=ny)
        {
            if(size[nx]<size[ny]) swap(nx,ny
                );
            par[ny] = nx;
            size[nx]+=size[ny];
    }
    }
};
```

UnionFindRollback.h

Description: UnionFindRollback.h

6c5dd9, 34 lines

```
class DSU {
  private:
  vector<int> p, sz;
  // stores previous unites
  vector<pair<int &, int>> history;

  public:
  DSU(int n) : p(n), sz(n, 1) { iota(p.begin
      (), p.end(), 0); }

  int get(int x) { return x == p[x] ? x :
     get(p[x]); }

  void unite(int a, int b) {
    a = get(a);
    b = get(b);
    if (a == b) { return; }
    if (sz[a] < sz[b]) { swap(a, b); }

    // save this unite operation
    history.push_back({sz[a], sz[a]});
    history.push_back({p[b], p[b]});

    p[b] = a;
    sz[a] += sz[b];
  }

  int snapshot() { return history.size(); }

  void rollback(int until) {
    while (snapshot() > until) {
      history.back().first = history.back().
          second;
      history.pop_back();
    }
  }
};
```

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: $\mathcal{O}\left(N^2+Q\right)$

c59ada, 13 lines

```
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: Matrix.h

5742e0, 32 lines

```
template<class T> struct Matrix {
  typedef Matrix M;
  vector<vector<T>> d;
    Matrix(int n){
        d.resize(n,vector<T>(n,0));
    };
  M operator*(const M& m) const {
    M a(m.d.size());
        int N = m.d.size();
    rep(i,0,N) rep(j,0,N)
      rep(k,0,N) {a.d[i][j] += (d[i][k]*m.d[
          k][j])%mod1;a.d[i][j]%=mod1;}
    return a;
  }
  vector<T> operator*(const vector<T>& vec)
      const {
        int N = this->d.size();
    vector<T> ret(N);
    rep(i,0,N) rep(j,0,N) {ret[i] += (d[i][j
        ] * vec[j])%mod1;ret[i]%=mod1;}
    return ret;
  }
  M operator^(ll p) const {
    assert(p >= 0);
    M a(this->d.size()), b(*this);
        int N = this->d.size();
    rep(i,0,N) a.d[i][i] = 1;
    while (p) {
      if (p&1) a = a*b;
      b = b*b;
      p >>= 1;
```

```
    }
    return a;
  }
};
```

LineContainer.h

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

8ec1c7, 30 lines

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

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

Treap.h

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

9556fc, 55 lines

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

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

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

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

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

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

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

FenwickTree.h

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

e62fac, 22 lines

```
struct FT {
  vector<ll> s;
  FT(int n) : s(n) {}
  void update(int pos, ll dif) { // a[pos] += dif
    for (; pos < sz(s); pos |= pos + 1) s[pos] += dif;
  }
  ll query(int pos) { // sum of values in [0, pos)
    ll res = 0;
```

```
    for (; pos > 0; pos &= pos - 1) res += s[pos-1];
    return res;
  }
  int lower_bound(ll sum) {// min pos st sum of [0, pos] >= sum
    // Returns n if no sum is >= sum, or -1 if empty sum is.
    if (sum <= 0) return -1;
    int pos = 0;
    for (int pw = 1 << 25; pw; pw >>= 1) {
      if (pos + pw <= sz(s) && s[pos + pw-1] < sum)
        pos += pw, sum -= s[pos-1];
    }
    return pos;
  }
};
```

FenwickTree2d.h

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

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

"FenwickTree.h" 157f07, 22 lines

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

RMQ.h

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

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

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

510c32, 16 lines

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

MoQueries.h

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

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

436b77, 49 lines

```
class mo_algorithm
{
public:
    int n, q, block_size;
    vector<int> a;
    vector<pair<int, pii>> queries;
    vector<int> answers;
    int answer, val;

    mo_algorithm(int n, int q, vector<int> a, vector<pair<int, int>> queries)
    {
        this->n = n;
        this->q = q;
        this->a = a;
        for (int i = 0; i < q; i++)
            this->queries.push_back({queries[i].first, {queries[i].second, i}});
        block_size = sqrt(n);
        answers.resize(q);
        val = 0;
    }

    inline void add(int x) {val--;} // Try your best to keep this O(1) since n*root(n)*log(n) is too slow
    inline void remove(int x) {val--;}

    void process()
    {
        sort(queries.begin(), queries.end(), [this](pair<int, pii> x, pair<int, pii> y) {
            int block_x = x.first / block_size;
            int block_y = y.first / block_size;
            if (block_x != block_y)
                return block_x < block_y;
            return x.second.first < y.second.first;
        });
        int l = 0, r = -1;
        for (auto z : queries)
        {
            int x = z.first, y = z.second.first;
            while (r < y)
                add(a[++r]);
```

```
            while (r > y)
                remove(a[r--]);
            while (l < x)
                remove(a[l++]);
            while (l > x)
                add(a[--l]);
            answers[z.second.second] = (val
                == 0);
        }
    }
};
```

SegTree.h

Description: Segment tree implementation for range minimum query with count

1e12fc, 56 lines

```
struct node {
    int mini;
    int ct;
    node(int m=1e9, int c=0) {
        mini = m;
        ct = c;
    }
};
const int range = 1e5;
int arr[range];
node segment[4*range];
node merge(node& a,node& b)
{
    if(a.mini==b.mini)
    {
        node c(a.mini,a.ct+b.ct);
        return c;
    }
    else if(a.mini<b.mini) return a;
    else return b;
}
void build(int idx,int low,int high)
{
    if(low==high)
    {
        segment[idx] = node(arr[low],1);
        return;
    }
    int mid = low + (high - low)/2;
    build(2*idx,low,mid);
    build(2*idx+1,mid+1,high);
    segment[idx] = merge(segment[2*idx],
        segment[2*idx+1]);
}
node query(int idx,int low,int high,int l,
    int r)
{
    if(l<=low&&high<=r) return segment[idx];
    if (high<l||low>r) return node();
    int mid = low + (high-low)/2;
    node left = query(2*idx,low,mid,l,r);
    node right = query(2*idx+1,mid+1,high,l,
        r);
    return merge(left,right);
}
void pointUpdate(int idx,int low,int high,
    int pos_in_arr,int val)
{
    if(pos_in_arr<low||pos_in_arr>high)
        return;
    if(low==high)
    {
        segment[idx]=node(val,1);
        arr[low] = val;
        return;
    }
    int mid = low + (high - low)/2;
    pointUpdate(2*idx,low,mid,pos_in_arr,val
        );
    pointUpdate(2*idx+1,mid+1,high,
        pos_in_arr,val);
    segment[idx] = merge(segment[2*idx],
        segment[2*idx+1]);
}
```

Numerical (4)

4.1 Polynomials and recurrences

Polynomial.h

c9b7b0, 17 lines

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

PolyRoots.h

Description: Finds the real roots to a polynomial.

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

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

"Polynomial.h" b00bfe, 23 lines

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

PolyInterpolate.h

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

08bf48, 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;
}
```

BerlekampMassey.h

Description: Recovers any n-order linear recurrence relation from the first $2n$ terms of the recurrence. Useful for guessing linear recurrences after brute-forcing the first terms. Should work on any field, but numerical stability for floats is not guaranteed. Output will have size $\leq n$.
Usage: `berlekampMassey({0, 1, 1, 3, 5, 11}) // {1, 2}`
Time: $\mathcal{O}\left(N^2\right)$

"../number-theory/ModPow.h" 96548b, 20 lines

```
vector<ll> berlekampMassey(vector<ll> s) {
  int n = sz(s), L = 0, m = 0;
  vector<ll> C(n), B(n), T;
  C[0] = B[0] = 1;

  ll b = 1;
  rep(i,0,n) { ++m;
    ll d = s[i] % mod;
    rep(j,1,L+1) d = (d + C[j] * s[i - j]) %
        mod;
    if (!d) continue;
    T = C; ll coef = d * modpow(b, mod-2) %
        mod;
    rep(j,m,n) C[j] = (C[j] - coef * B[j - m
        ]) % mod;
    if (2 * L > i) continue;
    L = i + 1 - L; B = T; b = d; m = 0;
  }

  C.resize(L + 1); C.erase(C.begin());
  for (ll& x : C) x = (mod - x) % mod;
  return C;
}
```

LinearRecurrence.h

Description: Generates the k'th term of an n-order linear recurrence $S[i] = \sum_j S[i-j-1]tr[j]$, given $S[0 \dots \geq n-1]$ and $tr[0 \dots n-1]$. Faster than matrix multiplication. Useful together with Berlekamp–Massey.
Usage: `linearRec({0, 1}, {1, 1}, k) // k'th Fibonacci number`
Time: $\mathcal{O}\left(n^2 \log k\right)$

f4e444, 26 lines

```
typedef vector<ll> Poly;
ll linearRec(Poly S, Poly tr, ll k) {
  int n = sz(tr);

  auto combine = [&](Poly a, Poly b) {
    Poly res(n * 2 + 1);
    rep(i,0,n+1) rep(j,0,n+1)
      res[i + j] = (res[i + j] + a[i] * b[j
          ]) % mod;
    for (int i = 2 * n; i > n; --i) rep(j,0,
        n)
      res[i - 1 - j] = (res[i - 1 - j] + res
          [i] * tr[j]) % mod;
    res.resize(n + 1);
    return res;
  };

  Poly pol(n + 1), e(pol);
  pol[0] = e[1] = 1;

  for (++k; k; k /= 2) {
    if (k % 2) pol = combine(pol, e);
    e = combine(e, e);
  }

  ll res = 0;
  rep(i,0,n) res = (res + pol[i + 1] * S[i])
      % mod;
  return res;
}
```

4.2 Optimization

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.

4756fc, 7 lines

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

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: $\mathcal{O}\left(NM * \#pivots\right)$, where a pivot may be e.g. an edge relaxation. $\mathcal{O}\left(2^n\right)$ in the general case.

aa8530, 68 lines

```
typedef double T; // long double, Rational,
    double + mod<P>...
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;
}
};
```

/

4.3 Matrices

Determinant.h

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

bd5cec, 15 lines

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

IntDeterminant.h

Description: Calculates determinant using modular arithmetics. Modulos can also be removed to get a pure-integer version.
Time: $\mathcal{O}\left(N^3\right)$

3313dc, 18 lines

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

SolveLinear.h

Description: Solves $A * x = b$. If there are multiple solutions, an arbitrary one is returned. Returns rank, or -1 if no solutions. Data in A and b is lost.
Time: $\mathcal{O}\left(n^2 m\right)$

44c9ab, 38 lines

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

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

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

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

SolveLinear2.h

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

"SolveLinear.h" 08e495, 7 lines

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

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 mod p, and k is doubled in each step.

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

ebfff6, 35 lines

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

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

4.4 Fourier transforms

FastFourierTransform.h

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

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

ccab8f, 113 lines

```
typedef long double ld;

#define mp make_pair
#define eprintf(...) fprintf(stderr,
    __VA_ARGS__)
#define sz(x) ((int)(x).size())
#define TASKNAME "text"

const ld pi = acos((ld)-1);

namespace FFT {
    struct com {
        ld x, y;

        com(ld _x = 0, ld _y = 0) : x(_x), y
           (_y) {}

        inline com operator+(const com &c)
            const {
             return com(x + c.x, y + c.y);
        }
        inline com operator-(const com &c)
            const {
             return com(x - c.x, y - c.y);
        }
        inline com operator*(const com &c)
            const {
             return com(x * c.x - y * c.y, x
                 * c.y + y * c.x);
        }
        inline com conj() const {
             return com(x, -y);
        }
    };
```

```
const static int maxk = 21, maxn = (1 <<
    maxk) + 1;
com ws[maxn];
int dp[maxn];
com rs[maxn];
int n, k;
int lastk = -1;

void fft(com *a, bool torev = 0) {
    if (lastk != k) {
        lastk = k;
        dp[0] = 0;

        for (int i = 1, g = -1; i < n;
            ++i) {
            if (!(i & (i - 1))) {
                ++g;
            }
            dp[i] = dp[i ^ (1 << g)] ^
                (1 << (k - 1 - g));
        }

        ws[1] = com(1, 0);
        for (int two = 0; two < k - 1;
            ++two) {
            ld alf = pi / n * (1 << (k -
                1 - two));
            com cur = com(cos(alf), sin(
                alf));

            int p2 = (1 << two), p3 = p2
                * 2;
            for (int j = p2; j < p3; ++j
                ) {
                ws[j * 2 + 1] = (ws[j *
                    2] = ws[j]) * cur;
            }
        }
    }
    for (int i = 0; i < n; ++i) {
        if (i < dp[i]) {
            swap(a[i], a[dp[i]]);
        }
    }
    if (torev) {
        for (int i = 0; i < n; ++i) {
            a[i].y = -a[i].y;
        }
    }
    for (int len = 1; len < n; len <<=
        1) {
        for (int i = 0; i < n; i += len)
            {
            int wit = len;
            for (int it = 0, j = i + len
                ; it < len; ++it, ++i,
                ++j) {
                com tmp = a[j] * ws[wit
                    ++];
                a[j] = a[i] - tmp;
                a[i] = a[i] + tmp;
            }
        }
    }
}

com a[maxn];
int mult(int na, int *_a, int nb, int *
    _b, long long *ans) {
    if (!na || !nb) {
        return 0;
    }
    for (k = 0, n = 1; n < na + nb - 1;
        n <<= 1, ++k);
    assert(n < maxn);
    for (int i = 0; i < n; ++i) {
        a[i] = com(i < na ? _a[i] : 0, i
            < nb ? _b[i] : 0);
    }
    fft(a);
    a[n] = a[0];
    for (int i = 0; i <= n - i; ++i) {
        a[i] = (a[i] * a[i] - (a[n - i]
            * a[n - i]).conj()) * com(0,
            (ld)-1 / n / 4);
        a[n - i] = a[i].conj();
    }
    fft(a, 1);
    int res = 0;
    for (int i = 0; i < n; ++i) {
        long long val = (long long)round
            (a[i].x);
        assert(abs(val - a[i].x) < 1e-1)
            ;
        if (val) {
            assert(i < na + nb - 1);
            while (res < i) {
                ans[res++] = 0;
            }
            ans[res++] = val;
        }
    }
    return res;
  }
};
```

FastSubsetTransform.h

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

464cf3, 16 lines

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

Number theory (5)

5.1 Modular arithmetic

ModularArithmetic.h

Description: ModularArithmetic.h

9f3a25, 43 lines

```
template <const int64_t MOD = mod1>
struct modint {
    int64_t value;
    modint() = default;
    modint(int64_t value_) : value(value_% MOD) {}
    modint<MOD> operator + (modint<MOD> other) const { int64_t c = this->value + other.value; return modint<MOD>(c >= MOD ? c - MOD : c); }
    modint<MOD> operator - (modint<MOD> other) const { int64_t c = this->value - other.value; return modint<MOD>(c <    0 ? c + MOD : c); }
    modint<MOD> operator * (modint<MOD> other) const { int64_t c = (int64_t)this->value * other.value % MOD; return modint<MOD>(c < 0 ? c + MOD : c); }
    modint<MOD> & operator += (modint<MOD> other) { this->value += other.value; if (this->value >= MOD) this->value -= MOD; return *this; }
    modint<MOD> & operator -= (modint<MOD> other) { this->value -= other.value; if (this->value <    0) this->value += MOD; return *this; }
    modint<MOD> & operator *= (modint<MOD> other) { this->value = (int64_t)this->value * other.value % MOD; if (this->value < 0) this->value += MOD; return *this; }
    modint<MOD> operator - () const { return modint<MOD>(this->value ? MOD - this->value : 0); }
    modint<MOD> pow(uint64_t k) const { modint<MOD> x = *this, y = 1; for (; k; k >>= 1) { if (k & 1) y *= x; x *= x; } return y; }
    modint<MOD> inv() const { return pow(MOD - 2); }  // MOD must be a prime*
    modint<MOD> operator /  (modint<MOD> other) const { return *this *  other.inv(); }
    modint<MOD> operator /= (modint<MOD> other)       { return *this *= other.inv(); }
    bool operator == (modint<MOD> other) const { return value == other.value; }
    bool operator != (modint<MOD> other) const { return value != other.value; }
    bool operator < (modint<MOD> other) const { return value < other.value; }
    bool operator > (modint<MOD> other) const { return value > other.value; }
    friend modint<MOD> operator * (int64_t value, modint<MOD> n) { return modint<MOD>(value % MOD) * n; }
    friend istream & operator >> (istream & in, modint<MOD> &n) { return in >> n.value; }
    friend ostream & operator << (ostream & out, modint<MOD> n) { return out << n.value; }
};
using mint = modint<>;

template<const int64_t mod = mod1>
struct combi{
  int n; vector<mint> facts, finvs, invs;
  combi(int _n): n(_n), facts(_n), finvs(_n), invs(_n){
    facts[0] = finvs[0] = 1;
    invs[1] = 1;
    for (int i = 2; i < n; i++) invs[i] = invs[mod % i] * (-mod / i);
    for(int i = 1; i < n; i++){
      facts[i] = facts[i - 1] * i;
      finvs[i] = finvs[i - 1] * invs[i];
    }
  }
  inline mint fact(int n) { return facts[n]; }
  inline mint finv(int n) { return finvs[n]; }
  inline mint inv(int n) { return invs[n]; }
  inline mint ncr(int n, int k) { return n < k or k < 0 ? 0 : facts[n] * finvs[k] * finvs[n-k]; }
};
```

ModSqrt.h

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

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

"ModPow.h" 19a793, 24 lines

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

5.2 Primality

FastEratosthenes.h

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

Time: LIM=1e9 $\approx$ 1.5s

ca7ef6, 140 lines

```cpp
int prime[1000000+1];
const int range = 1e6;
// Seive of Eratosthenes
void isPrime()
{
    for (int i = 2; i*i <= range; i++)
    {
        if(prime[i]==0)
        {
            // its fine to start from i*i
            // 2*i,4*i,.. will be already
                marked by some smaller prime
                numb
            for (int j = 1ll*i*i; j <= range
                ; j+=i)
            {
                prime[j]=1;
            }
        }
    }
    // if prime[i]==0 it means i is prime
}
// To find any n is prime or not
bool isPrime(ll n)
{
    if (n <= 1)
    {
        return false;
    }
    if (n <= 3)
    {
        return true;
    }
    if (n % 2 == 0 || n % 3 == 0)
    {
        return false;
    }
    for (ll i = 5; i * i <= n; i = i + 6)
    {
        if (n % i == 0 || n % (i + 2) == 0)
        {
            return false;
        }
    }
    return true;
}
// code for finding number of divisors for
    all numbers from 2 to N.
void divisors()
{
    // TC - O(NlogN)
    for (int i = 2; i < N; i++)
    {
        for (int j = i; j < N; j += i)
        {
            divis[j]++;
        }
    }
}
// Code for finding divisors of a number x
vi divisor(int x)
{
    vi ans;
    int i = 1;
    while (i * i <= x)
    {
        if (x % i == 0)
        {
            ans.pb(i);
            if (x / i != i)
            {
                ans.pb(x / i);
            }
        }
        i++;
    }
    return ans;
}

// Code for finding factors of number x;
vector<pi> factor(int x)
{
    vector<pi> ans;
    for (int i = 2; i*i<=x; i++)
    {
        if(x%i==0)
        {
            int cnt=0;
            while(x%i==0)
            {
                cnt++;
                x/=i;
            }
            ans.pb({i,cnt});
        }
    }
    if(x>1) ans.pb({x,1});
    return ans;
}
// Fast Factorisation
// Code for finding all numbers upto X --
    (10^6)
// Normal Approach TC-O(N*sqrt(N))
// Store Lowest prime for each number using
    SIEVE IDEA
// Recursively call N/p till it reaches 1
// How it will calculate in log(X) becoz max
    prime factors of X are log2(X)
void all_prime_factors(int X)
{
    // creating sp array
    int sp[1000000+1]; // initially zeroed
    int prime[1000000+1];  // initially
        zeroed
    const int range = 1e6;
    // Seive of Eratosthenes
    for (int i = 2; i <= range; i++)
    {
        if(prime[i]==0)
        {
            sp[i] = i;
            // its fine to start from i*i
            // 2*i,4*i,.. will be already
                marked by some smaller prime
                numb
            for (int j = i*i; j <= range; j
                +=i)
            {
                if(prime[j]==0)
                {
                    prime[j]=1;
                    sp[j] = i;
```

```
                }
            }
        }
    }
    // if prime[i]==0 it means i is prime
    vector<int> factors[range+1];
    for (int i = 2; i < X+1; i++)
    {
        int num = i;
        while (num!=1)
        {
            factors[i].push_back(sp[num]);
            num/=sp[num];
        }

    }
    // final ans is stored in factors array
}
```

MillerRabin.h

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

Time: 7 times the complexity of $a^b \mod c$.

"ModMulLL.h" 60dcd1, 12 lines

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

Factor.h

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

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

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

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

5.3 Divisibility

euclid.h

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

33ba8f, 5 lines

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

CRT.h

Description: Chinese Remainder Theorem.

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

Time: $\log(n)$

"euclid.h" 04d93a, 7 lines

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

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

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

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

cf7d6d, 8 lines

```
const int LIM = 5000000;
int phi[LIM];

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

5.4 Fractions

ContinuedFractions.h

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

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

dd6c5e, 21 lines

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

5.5 Pythagorean Triples

The Pythagorean triples are uniquely generated by

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

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

5.6 Primes

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

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

5.7 Estimates

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

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

5.8 Mobius Function

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

Mobius Inversion:

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

Other useful formulas/forms:

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

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

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

Combinatorial (6)

6.1 Permutations

6.1.1 Factorial

6.1.2 Cycles

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

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

6.1.3 Derangements

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

$$D(n) = (n-1)(D(n-1) + D(n-2)) = nD(n-1) + (-1)^n$$

6.1.4 Burnside's lemma

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

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

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

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

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

6.2 Partitions and subsets

6.2.1 Partition function

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

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

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

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

6.2.2 Lucas' Theorem

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

6.2.3 Binomials

multinomial.h

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

a0a312, 5 lines

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

6.3 General purpose numbers

6.3.1 Stirling numbers of the first kind

Number of permutations on n items with k cycles.

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

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

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

6.3.2 Eulerian numbers

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

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

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

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

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

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

6.3.5 Labeled unrooted trees

\# on n vertices: n^{n-2}
\# on k existing trees of size n_i: $n_1 n_2 \cdots n_k n^{k-2}$
\# with degrees d_i: $(n-2)!/((d_1-1)! \cdots (d_n-1)!)$

6.3.6 Catalan numbers

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

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

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

- sub-diagonal monotone paths in an $n \times n$ grid.
- strings with n pairs of parenthesis, correctly nested.
- binary trees with with $n+1$ leaves (0 or 2 children).
- ordered trees with $n+1$ vertices.
- ways a convex polygon with $n+2$ sides can be cut into triangles by connecting vertices with straight lines.
- permutations of $[n]$ with no 3-term increasing subseq.

Graph (7)

7.1 Fundamentals

BellmanFord.h

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

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

830a8f, 23 lines

```
const ll inf = LLONG_MAX;
struct Ed { int a, b, w, s() { return a < b
    ? a : -a; }};
struct Node { ll dist = inf; int prev = -1;
    };

void bellmanFord(vector<Node>& nodes, vector
    <Ed>& eds, int s) {
  nodes[s].dist = 0;
  sort(all(eds), [](Ed a, Ed b) { return a.s
      () < b.s(); });
```

```
  int lim = sz(nodes) / 2 + 2; // /3+100 with shuffled vertices
  rep(i,0,lim) for (Ed ed : eds) {
    Node cur = nodes[ed.a], &dest = nodes[ed.b];
    if (abs(cur.dist) == inf) continue;
    ll d = cur.dist + ed.w;
    if (d < dest.dist) {
      dest.prev = ed.a;
      dest.dist = (i < lim-1 ? d : -inf);
    }
  }
  rep(i,0,lim) for (Ed e : eds) {
    if (nodes[e.a].dist == -inf)
      nodes[e.b].dist = -inf;
  }
}
```

FloydWarshall.h

Description: Calculates all-pairs shortest path in a directed graph that might have negative edge weights. Input is an distance matrix m, where $m[i][j] =$ `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: $\mathcal{O}\left(N^3\right)$

531245, 12 lines

```
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], 0LL);
  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;
}
```

Dijkstra.h

Description: Dijkstra.h

38cd71, 31 lines

```
const int mx = 1e5+10;
const int INF = 1e9+10;
// taking input for graph(connection,wt)
vector<pair<int,int>> g[mx];
vector<int> d(mx,INF),par(mx); // array for storing d
void dijkstra(int source)
{
    vector<int> vis(mx,0); // visited array
    set<pair<int,int>> st;
    st.insert({0,source});
    d[source] = 0;
    while (st.size()>0)
    {
        auto node = *st.begin();
        int v = node.second;
        int dist = node.first;
        st.erase(st.begin());
        if(vis[v]) continue;
        vis[v]=1;
        for (auto child : g[v])
        {
            int child_v = child.first;
            int wt = child.second;
            if(d[v]+wt<d[child_v])
            {
                d[child_v] = d[v] + wt;
                st.insert({d[child_v], child_v});
            }
        }
    }
}
```

TopoSort.h

Description: Topological sorting. Given is an oriented graph. Output is an ordering of vertices, such that there are edges only from left to right. If there are cycles, the returned list will have size smaller than n – nodes reachable from cycles will not be returned.

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

d678d8, 8 lines

```
vi topoSort(const vector<vi>& gr) {
  vi indeg(sz(gr)), q;
  for (auto& li : gr) for (int x : li) indeg[x]++;
  rep(i,0,sz(gr)) if (indeg[i] == 0) q.push_back(i);
  rep(j,0,sz(q)) for (int x : gr[q[j]])
    if (--indeg[x] == 0) q.push_back(x);
  return q;
}
```

7.2 Network flow

PushRelabel.h

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

Time: $\mathcal{O}\left(V^2\sqrt{E}\right)$

0ae1d4, 48 lines

```
struct PushRelabel {
  struct Edge {
    int dest, back;
    ll f, c;
  };
  vector<vector<Edge>> g;
  vector<ll> ec;
  vector<Edge*> cur;
  vector<vi> hs; vi H;
  PushRelabel(int n) : g(n), ec(n), cur(n), hs(2*n), H(n) {}

  void addEdge(int s, int t, ll cap, ll rcap=0) {
    if (s == t) return;
    g[s].push_back({t, sz(g[t]), 0, cap});
    g[t].push_back({s, sz(g[s])-1, 0, rcap});
  }

  void addFlow(Edge& e, ll f) {
    Edge &back = g[e.dest][e.back];
    if (!ec[e.dest] && f) hs[H[e.dest]].push_back(e.dest);
    e.f += f; e.c -= f; ec[e.dest] += f;
    back.f -= f; back.c += f; ec[back.dest] -= f;
  }
  ll calc(int s, int t) {
    int v = sz(g); H[s] = v; ec[t] = 1;
```

```
    vi co(2*v); co[0] = v-1;
    rep(i,0,v) cur[i] = g[i].data();
    for (Edge& e : g[s]) addFlow(e, e.c);

    for (int hi = 0;;) {
      while (hs[hi].empty()) if (!hi--)
        return -ec[s];
      int u = hs[hi].back(); hs[hi].pop_back
        ();
      while (ec[u] > 0)  // discharge u
        if (cur[u] == g[u].data() + sz(g[u])
          ) {
          H[u] = 1e9;
          for (Edge& e : g[u]) if (e.c && H[
            u] > H[e.dest]+1)
            H[u] = H[e.dest]+1, cur[u] = &e;
          if (++co[H[u]], !--co[hi] && hi <
            v)
            rep(i,0,v) if (hi < H[i] && H[i]
              < v)
              --co[H[i]], H[i] = v + 1;
          hi = H[u];
        } else if (cur[u]->c && H[u] == H[
          cur[u]->dest]+1)
          addFlow(*cur[u], min(ec[u], cur[u
            ]->c));
        else ++cur[u];
    }
  }
  bool leftOfMinCut(int a) { return H[a] >=
    sz(g); }
};
```

MinCostMaxFlow.h

Description: Min-cost max-flow. If costs can be negative, call setpi before maxflow, but note that negative cost cycles are not supported. To obtain the actual flow, look at positive values only.
Time: $\mathcal{O}(FE\log(V))$ where F is max flow. $\mathcal{O}(VE)$ for setpi.
58385b, 79 lines

```
#include <bits/extc++.h>

const ll INF = numeric_limits<ll>::max() /
  4;

struct MCMF {
  struct edge {
    int from, to, rev;
    ll cap, cost, flow;
  };
  int N;
  vector<vector<edge>> ed;
  vi seen;
  vector<ll> dist, pi;
  vector<edge*> par;

  MCMF(int N) : N(N), ed(N), seen(N), dist(N
    ), pi(N), par(N) {}

  void addEdge(int from, int to, ll cap, ll
    cost) {
    if (from == to) return;
    ed[from].push_back(edge{ from,to,sz(ed[
      to]),cap,cost,0 });
    ed[to].push_back(edge{ to,from,sz(ed[
      from])-1,0,-cost,0 });
  }

  void path(int s) {
    fill(all(seen), 0);
    fill(all(dist), INF);
    dist[s] = 0; ll di;

    __gnu_pbds::priority_queue<pair<ll, int
      >> q;
    vector<decltype(q)::point_iterator> its(
      N);
    q.push({ 0, s });

    while (!q.empty()) {
      s = q.top().second; q.pop();
      seen[s] = 1; di = dist[s] + pi[s];
      for (edge& e : ed[s]) if (!seen[e.to])
        {
        ll val = di - pi[e.to] + e.cost;
        if (e.cap - e.flow > 0 && val < dist
          [e.to]) {
          dist[e.to] = val;
          par[e.to] = &e;
          if (its[e.to] == q.end())
            its[e.to] = q.push({ -dist[e.to
              ], e.to });
          else
            q.modify(its[e.to], { -dist[e.to
              ], e.to });
        }
      }
    }
    rep(i,0,N) pi[i] = min(pi[i] + dist[i],
      INF);
  }

  pair<ll, ll> maxflow(int s, int t) {
    ll totflow = 0, totcost = 0;
    while (path(s), seen[t]) {
      ll fl = INF;
      for (edge* x = par[t]; x; x = par[x->
        from])
        fl = min(fl, x->cap - x->flow);

      totflow += fl;
      for (edge* x = par[t]; x; x = par[x->
        from]) {
        x->flow += fl;
        ed[x->to][x->rev].flow -= fl;
      }
    }
    rep(i,0,N) for(edge& e : ed[i]) totcost
      += e.cost * e.flow;
    return {totflow, totcost/2};
  }

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

EdmondsKarp.h

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

482fe0, 36 lines

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

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

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

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

MinCut.h

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

GlobalMinCut.h

Description: Find a global minimum cut in an undirected graph, as represented by an adjacency matrix.
Time: $\mathcal{O}\left(V^3\right)$

8b0e19, 21 lines

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

7.3 Matching

hopcroftKarp.h

Description: Fast bipartite matching algorithm. Graph g should be a list of neighbors of the left partition, and *btoa* should be a vector full of -1's of the same size as the right partition. Returns the size of the matching. $btoa[i]$ will be the match for vertex i on the right side, or -1 if it's not matched.
Usage: `vi btoa(m, -1); hopcroftKarp(g, btoa);`
Time: $\mathcal{O}\left(\sqrt{V}E\right)$

f612e4, 42 lines

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

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

```
}
```

DFSMatching.h

Description: Simple bipartite matching algorithm. Graph g should be a list of neighbors of the left partition, and $btoa$ should be a vector full of -1's of the same size as the right partition. Returns the size of the matching. $btoa[i]$ will be the match for vertex i on the right side, or -1 if it's not matched.
Usage: `vi btoa(m, -1); dfsMatching(g, btoa);`
Time: $\mathcal{O}(VE)$

522b98, 22 lines

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

BipartiteMatching.h

Description: bipartite matching

da1d4b, 70 lines

```
struct bipartite {
    int n, m;
    vector<vector<int>> g;
    vector<bool> paired;
    vector<int> match;

    bipartite(int n, int m): n(n), m(m), g(n
        ), paired(n), match(m, -1) {}

    void add(int a, int b) {
        g[a].push_back(b);
    }

    vector<size_t> ptr;
    bool kuhn(int v) {
        for(size_t &i = ptr[v]; i < g[v].
            size(); i++) {
            int &u = match[g[v][i]];
            if(u == -1 || (dist[u] == dist[v
                ] + 1 && kuhn(u))) {
                u = v;
                paired[v] = true;
                return true;
            }

        }
        return false;
    }

    vector<int> dist;
    bool bfs() {
        dist.assign(n, n);
        int que[n];
        int st = 0, fi = 0;
        for(int v = 0; v < n; v++) {
            if(!paired[v]) {
                dist[v] = 0;
                que[fi++] = v;
            }
        }
        bool rep = false;
        while(st < fi) {
            int v = que[st++];
            for(auto e: g[v]) {
                int u = match[e];
                rep |= u == -1;
                if(u != -1 && dist[v] + 1 <
                    dist[u]) {
                    dist[u] = dist[v] + 1;
                    que[fi++] = u;
                }
            }
        }
        return rep;
    }

    auto matching() {
        while(bfs()) {
            ptr.assign(n, 0);
            for(int v = 0; v < n; v++) {
                if(!paired[v]) {
                    kuhn(v);
                }
            }
        }
        vector<pair<int, int>> ans;
        for(int u = 0; u < m; u++) {
            if(match[u] != -1) {
                ans.emplace_back(match[u], u
                    );
            }
        }
        return ans;
    }
};
```

MinimumVertexCover.h

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

"DFSMatching.h" da4196, 20 lines

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

```
  rep(i,0,m) if (seen[i]) cover.push_back(n+i);
  assert(sz(cover) == res);
  return cover;
}
```

WeightedMatching.h

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

a8683a, 76 lines

```
typedef long double ld;

vector<int> hungarian(const vector<vector<ld>>& A, int n) {
    // Labels for workers (u) and jobs (v)
    vector<ld> u(n + 1, 0.0), v(n + 1, 0.0);

    // p[j] - the worker assigned to job j
    vector<int> p(n + 1, 0);

    // way[j] - the previous job in the augmenting path for job j
    vector<int> way(n + 1, 0);

    for(int i = 1; i <= n; ++i){
        p[0] = i;
        int j0 = 0;
        // minv[j] - minimum reduced cost for job j
        vector<ld> minv(n + 1, inf);
        // used[j] - whether job j is used in the current augmenting path
        vector<bool> used(n + 1, false);

        int j1;
        while(true){
            used[j0] = true;
            int i0 = p[j0];
            ld delta = inf;
            j1 = 0;

            // Iterate over all jobs to find the minimum delta
            for(int j = 1; j <= n; ++j){
                if(!used[j]){
                    ld cur = A[i0 - 1][j - 1] - u[i0] - v[j];
                    if(cur < minv[j]){
                        minv[j] = cur;
                        way[j] = j0;
                    }
                    if(minv[j] < delta){
                        delta = minv[j];
                        j1 = j;
                    }
                }
            }

            // Update labels
            for(int j = 0; j <= n; ++j){
                if(used[j]){
                    u[p[j]] += delta;
                    v[j] -= delta;
                }
                else{
                    minv[j] -= delta;
                }
            }

            j0 = j1;
            if(p[j0] == 0)
                break;
        }

        // Augmenting path: update the matching
        do{
            int j1 = way[j0];
            p[j0] = p[j1];
            j0 = j1;
        } while(j0 != 0);
    }

    // Construct the result: ans[i] = j means worker i is assigned to job j
    vector<int> ans(n, -1);
    for(int j = 1; j <= n; ++j){
        if(p[j] != 0){
            ans[p[j] - 1] = j - 1;
        }
    }

    return ans;
}
```

GeneralMatching.h

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

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

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

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

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

  vi has(M, 1); vector<pii> ret;
  rep(it,0,M/2) {
    rep(i,0,M) if (has[i])
      rep(j,i+1,M) if (A[i][j] && mat[i][j]) {
```

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

7.4 DFS algorithms

SCC.h

Description: SCC.h

5a2d60, 59 lines

```
vector<bool> visited; // keeps track of
    which vertices are already visited

// runs depth first search starting at
    vertex v.
// each visited vertex is appended to the
    output vector when dfs leaves it.
void dfs(int v, vector<vector<int>> const&
    adj, vector<int> &output) {
    visited[v] = true;
    for (auto u : adj[v])
        if (!visited[u])
            dfs(u, adj, output);
    output.push_back(v);
}

// input: adj -- adjacency list of G
// output: components -- the strongy
    connected components in G
// output: adj_cond -- adjacency list of G^
    SCC (by root vertices)
void strongly_connected_components(vector<
    vector<int>> const& adj,
                                   vector<
                                     vector
                                     <int>>
                                      &
                                     components
                                     ,
                                   vector<
                                     vector
                                     <int>>
                                      &
                                     adj_cond
                                     ) {
    int n = adj.size();
    components.clear(), adj_cond.clear();

    vector<int> order; // will be a sorted
        list of G's vertices by exit time

    visited.assign(n, false);

    // first series of depth first searches
    for (int i = 0; i < n; i++)
        if (!visited[i])
            dfs(i, adj, order);

    // create adjacency list of G^T
    vector<vector<int>> adj_rev(n);
    for (int v = 0; v < n; v++)
        for (int u : adj[v])
            adj_rev[u].push_back(v);

    visited.assign(n, false);
    reverse(order.begin(), order.end());

    vector<int> roots(n, 0); // gives the
        root vertex of a vertex's SCC

    // second series of depth first searches
    for (auto v : order)
        if (!visited[v]) {
            std::vector<int> component;
            dfs(v, adj_rev, component);
            components.push_back(component);
            int root = *min_element(begin(
                component), end(component));
            for (auto u : component)
                roots[u] = root;
        }

    // add edges to condensation graph
    adj_cond.assign(n, {});
    for (int v = 0; v < n; v++)
        for (auto u : adj[v])
            if (roots[v] != roots[u])
                adj_cond[roots[v]].push_back
                    (roots[u]);
}
```

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

c6b7c7, 32 lines

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

```
      }
      else if (up < me) st.push_back(e);
      else { /* e is a bridge */ }
    }
  }
  return top;
}

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

bridges.h

Description: Bridges and Articulation Points in a graph calculate low[v] for every vertex low[v] = min(tin[v], tin[to] such that (v,to) is a backedge, note that to is not parent of v, low[to] such that (v,to) is a tree edge, calculate after dfs call) if(low[to] > tin[v]) then (v,to) is a bridge if(low[to] >= tin[v]) then v is a articulation point
add online bridges implementation

e7c1a5, 99 lines

```
vector<int> par, dsu_2ecc, dsu_cc,
   dsu_cc_size;
int bridges;
int lca_iteration;
vector<int> last_visit;
void init(int n) {
    par.resize(n);
    dsu_2ecc.resize(n);
    dsu_cc.resize(n);
    dsu_cc_size.resize(n);
    lca_iteration = 0;
    last_visit.assign(n, 0);
    for (int i=0; i<n; ++i) {
        dsu_2ecc[i] = i;
        dsu_cc[i] = i;
        dsu_cc_size[i] = 1;
        par[i] = -1;
    }
    bridges = 0;
}
int find_2ecc(int v) {
    if (v == -1)
        return -1;
    return dsu_2ecc[v] == v ? v : dsu_2ecc[v
       ] = find_2ecc(dsu_2ecc[v]);
}
int find_cc(int v) {
    v = find_2ecc(v);
    return dsu_cc[v] == v ? v : dsu_cc[v] =
       find_cc(dsu_cc[v]);
}
void make_root(int v) {
    int root = v;
    int child = -1;
    while (v != -1) {
        int p = find_2ecc(par[v]);
        par[v] = child;
        dsu_cc[v] = root;
        child = v;
        v = p;
    }
    dsu_cc_size[root] = dsu_cc_size[child];
}
void merge_path (int a, int b) {
    ++lca_iteration;
    vector<int> path_a, path_b;
    int lca = -1;
    while (lca == -1) {
        if (a != -1) {
            a = find_2ecc(a);
            path_a.push_back(a);
            if (last_visit[a] ==
               lca_iteration){
                lca = a;
                break;
                }
            last_visit[a] = lca_iteration;
            a = par[a];
        }
        if (b != -1) {
            b = find_2ecc(b);
            path_b.push_back(b);
            if (last_visit[b] ==
               lca_iteration){
                lca = b;
                break;
                }
            last_visit[b] = lca_iteration;
            b = par[b];
        }
    }
    for (int v : path_a) {
        dsu_2ecc[v] = lca;
        if (v == lca)
            break;
        --bridges;
    }
    for (int v : path_b) {
        dsu_2ecc[v] = lca;
        if (v == lca)
            break;
        --bridges;
    }
}
void add_edge(int a, int b) {
    a = find_2ecc(a);
    b = find_2ecc(b);
    if (a == b)
        return;
    int ca = find_cc(a);
    int cb = find_cc(b);
    if (ca != cb) {
        ++bridges;
        if (dsu_cc_size[ca] > dsu_cc_size[cb
           ]) {
            swap(a, b);
            swap(ca, cb);
        }
        make_root(a);
        par[a] = dsu_cc[a] = b;
        dsu_cc_size[cb] += dsu_cc_size[a];
    } else {
        merge_path(a, b);
    }
}
```

2sat.h

Description: 2sat.h

3524b8, 68 lines

```
class TwoSAT {
private:
    int n;
    std::vector<std::vector<int>> adj, adj_t
       ;
```

```
    std::vector<bool> used;
    std::vector<int> order, comp;
    std::vector<bool> assignment;

    void dfs1(int v) {
        used[v] = true;
        for (int u : adj[v]) {
            if (!used[u])
                dfs1(u);
        }
        order.push_back(v);
    }

    void dfs2(int v, int cl) {
        comp[v] = cl;
        for (int u : adj_t[v]) {
            if (comp[u] == -1)
                dfs2(u, cl);
        }
    }

public:
    TwoSAT(int size) : n(size), adj(2 * n),
        adj_t(2 * n), used(2 * n), comp(2 *
        n), assignment(n) {}

    bool solve() {
        order.clear();
        used.assign(2 * n, false);
        for (int i = 0; i < 2 * n; ++i) {
            if (!used[i])
                dfs1(i);
        }

        comp.assign(2 * n, -1);
        for (int i = 0, j = 0; i < 2 * n; ++
            i) {
            int v = order[2 * n - i - 1];
            if (comp[v] == -1)
                dfs2(v, j++);
        }

        assignment.assign(n, false);
        for (int i = 0; i < 2 * n; i += 2) {
            if (comp[i] == comp[i + 1])
                return false;
            assignment[i / 2] = comp[i] >
                comp[i + 1];
        }
        return true;
    }

    void add_disjunction(int a, bool na, int
        b, bool nb) {
        // na and nb signify whether a and b
            are to be negated
        a = 2 * a ^ na;
        b = 2 * b ^ nb;
        int neg_a = a ^ 1;
        int neg_b = b ^ 1;
        adj[neg_a].push_back(b);
        adj[neg_b].push_back(a);
        adj_t[b].push_back(neg_a);
        adj_t[a].push_back(neg_b);
    }

    std::vector<bool> get_assignment() {
        return assignment;
    }
};
```

EulerWalk.h

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

780b64, 15 lines

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

7.5 Coloring

EdgeColoring.h

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

e210e2, 31 lines

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

```
      int left = fan[i], right = fan[++i], e
          = cc[i];
      adj[u][e] = left;
      adj[left][e] = u;
      adj[right][e] = -1;
      free[right] = e;
    }
    adj[u][d] = fan[i];
    adj[fan[i]][d] = u;
    for (int y : {fan[0], u, end})
      for (int& z = free[y] = 0; adj[y][z]
          != -1; z++);
  }
  rep(i,0,sz(eds))
    for (tie(u, v) = eds[i]; adj[u][ret[i]]
        != v;) ++ret[i];
  return ret;
}
```

7.6 Heuristics

MaximalCliques.h

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

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

b0d5b1, 12 lines

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

MaximumClique.h

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

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

f7c0bc, 49 lines

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

MaximumIndependentSet.h

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

7.7 Trees

BinaryLifting.h

Description: BinaryLifting.h

5c04ac, 142 lines

```
class Binary_lift{
    public:
        int n,l,timer;
        vector<vector<int>> adj;
        vector<vector<int>> up;
        vector<vector<int>> min_v;
        vector<int> depth;
        vector<int> tin;
        vector<int> tout;
        Binary_lift(int n){
            this->n = n;
            this->l = log2(n)+1;
            adj.resize(n);
            up.resize(n, vector<int>(l, -1))
                ;
```

```
        min_v.resize(n, vector<int>(l,
            inf));
        depth.resize(n);tin.resize(n);
            tout.resize(n);
        timer = 0;
    }

    void set_min_v(vi& a){
        fr(i,0,n){
            min_v[i][0] = a[i];}
    }
    void add_edge(int u, int v){
        adj[u].push_back(v);adj[v].
            push_back(u);}
    void dfs(int u, int p, vi& a, int d
        =0){
        up[u][0] = p;
        depth[u] = d;
        tin[u] = timer++;
        for(int i=1;i<l;i++){
            if(up[u][i-1] != -1){
                up[u][i] = up[up[u][i
                    -1]][i-1];
                min_v[u][i] = min(min_v[
                    u][i-1], min_v[up[u
                    ][i-1]][i-1]);
            }
        }
        for(int v: adj[u]){
            if(v != p){
                dfs(v, u,a,d+1);}
        }
        tout[u] = timer;
    }

    int lift(int u, int k){
        for(int i=l-1;i>=0;i--){
            if(k >= (1<<i)){
                u = up[u][i];
                k -= (1<<i);
            }
        }
        return u;
    }

    int lca(int u, int v){
        if(depth[u] < depth[v]){
            swap(u,v);
        }
        u = lift(u, depth[u]-depth[v]);
        if(u == v){
            return u;
        }

        for(int i=l-1;i>=0;i--){
            if(depth[u]<(1<<i))
                continue;
            if(up[u][i] != up[v][i]){
                u = up[u][i];
                v = up[v][i];
            }
        }
        return up[u][0];
    }
    int get_kth_node_on_path(int u, int
        v, int k){
        int lca = this->lca(u,v);
        int dist = this->depth[u] + this
            ->depth[v] - 2*this->depth[
            lca];
        if(k > dist){
            return -1;
        }
        if(k == 0){
            return u;
        }
        if(k == dist){
            return v;
        }
        if(this->depth[u] - this->depth[
            lca] >= k){
            return this->lift(u, k);
        }
        return this->lift(v, dist-k);
    }
    int get_min_on_path(int u, int v){
        int lca = this->lca(u,v);
        int ans = inf;
        for(int i=l-1;i>=0;i--){
            if(this->depth[u] - (1<<i)
                >= this->depth[lca]){
                ans = min(ans, this->
                    min_v[u][i]);
                u = this->up[u][i];
            }
        }
        for(int i=l-1;i>=0;i--){
            if(this->depth[v] - (1<<i)
                >= this->depth[lca]){
                ans = min(ans, this->
                    min_v[v][i]);
                v = this->up[v][i];
            }
        }
        ans = min(ans, this->min_v[u
            ][0]);
        ans = min(ans, this->min_v[v
            ][0]);
        return ans;
    }
    int first_node_less_equal_k_on_path(
        int u, int v, int k, vi& a){
        if(a[u] <= k) return u;
        int lca = this->lca(u,v);
        for(int i=l-1;i>=0;i--){
            if(this->depth[u] - (1<<i)
                >= this->depth[lca]){
                if(this->min_v[u][i] <=
                    k) continue;
                u = this->up[u][i];
            }
        }
        int j = -1;
        if(u!=lca) return u;
        if(a[u] <= k) return u;
        for(int i=l-1;i>=0;i--){
            if(this->depth[v] - (1<<i)
                >= this->depth[lca]){
                int height = this->depth
                    [v] - this->depth[
                    lca] - (1<<i);
                int node = this->lift(v,
                     height);
                j=i;
                if(this->min_v[node][i]
                    <= k) v = node;
                else lca = up[v][i];
```

```
                    break;
                }
            }
            for(int i=j;i>0;i--){
                if(this->depth[v] - (1<<i)
                    >= this->depth[lca]){
                    int node = this->up[v][i
                        -1];
                    if(this->min_v[node][i
                        -1] <= k) v = node;
                    else lca = node;
                }
            }
            return v;
        }
        int get_dist(int u, int v){
            return this->depth[u] + this->
                depth[v] - 2*this->depth[
                this->lca(u,v)];
        }
};
```

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" 9775a0, 21 lines

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

CentroidDecomposition.h

Description: Centroid Decomposition of a tree

2e2603, 59 lines

```
class CentroidDecomposition
{
    // 1 - based indexing
private:
    int n;
    vector<bool> vis;
    vector<int> sz;
    const vector<vector<int>> &tree;

    int find_size(int v, int p = -1)
    {
        if (vis[v])
            return 0;
        sz[v] = 1;
        for (const int &x : tree[v])
            if (x != p)
                sz[v] += find_size(x, v);
        return sz[v];
    }

    int find_centroid(int v, int p, int
        cur_sz)
    {
        for (const int &x : tree[v])
            if (x != p)
                if (!vis[x] && sz[x] > (
                    cur_sz / 2))
                    return find_centroid(x,
                        v, cur_sz);
        return v;
    }

    void init_centroid(int v, int p)
    {
        find_size(v);
        int c = find_centroid(v, -1, sz[v]);
        vis[c] = true;
        centroid_par[c] = p;
        if (p == -1)
            root = c;
        else
            centorid_tree[p].push_back(c);
        for (const int &x : tree[c])
            if (!vis[x])
                init_centroid(x, c);
    }

public:
    vector<vector<int>> centorid_tree;
    vector<int> centroid_par;
    int root;
    CentroidDecomposition(vector<vector<int
        >> &_tree) : tree(_tree)
    {
        root = 1;
        n = tree.size();
        centorid_tree.resize(n);
        vis.resize(n, false);
        sz.resize(n, 0);
        centroid_par.resize(n, -1);
        init_centroid(1, -1);
    }
};
```

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. fr(i, 0, n) b[pos[i]] = a[i];

85f97b, 67 lines

```
class HLD{
public:
    vector<int> parent, depth, heavy, head,
        pos;
    int cur_pos;
    vector<vector<int>> adj;

    int dfs(int v) {
        int size = 1;
        int max_c_size = 0;
        for (int c : adj[v]) {
```

```
        if (c != parent[v]) {
            parent[c] = v, depth[c] =
                depth[v] + 1;
            int c_size = dfs(c);
            size += c_size;
            if (c_size > max_c_size)
                max_c_size = c_size,
                    heavy[v] = c;
        }
    }
    return size;
}

void decompose(int v, int h) {
    head[v] = h, pos[v] = cur_pos++;
    if (heavy[v] != -1)
        decompose(heavy[v], h);
    for (int c : adj[v]) {
        if (c != parent[v] && c != heavy
            [v])
            decompose(c, c);
    }
}

void build()
{
    dfs(0);
    decompose(0, 0);
}

HLD(int n) {
    parent = vector<int>(n);
    depth = vector<int>(n);
    heavy = vector<int>(n, -1);
    head = vector<int>(n);
    pos = vector<int>(n);
    adj = vector<vector<int>>(n);
    cur_pos = 0;
}

void add_edge(int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

```
  vi query(int a, int b, int x,
      SegmentTree& st) {
    vi res;
    for (; head[a] != head[b]; b =
        parent[head[b]]) {
      if (depth[head[a]] > depth[head[
          b]])
        swap(a, b);
      vi cur_heavy_path_max = st.query
          (pos[head[b]], pos[b], x);
      for(auto i: cur_heavy_path_max)
          res.pb(i);
    }
    if (depth[a] > depth[b])
      swap(a, b);
    vi last_heavy_path_max = st.query(
        pos[a], pos[b], x);
    for(auto i: last_heavy_path_max) res
        .pb(i);
    return res;
  }
};
```

DirectedMST.h

Description: DirectedMST.h

f4c895, 31 lines

```
class Solution
{
public:
  int spanningTree(int V, vector<vector<int
      >> adj[])
  {
    priority_queue<pair<int, int>,
                   vector<pair<int, int> >,
                     greater<pair<int, int
                     >>> pq;

    vector<int> vis(V, 0);
    pq.push({0, 0});
    int sum = 0;
    while (!pq.empty()) {
      auto it = pq.top();
      pq.pop();
      int node = it.second;
      int wt = it.first;
      if (vis[node] == 1) continue;
      vis[node] = 1;
      sum += wt;
      for (auto it : adj[node]) {
        int adjNode = it[0];
        int edW = it[1];
        if (!vis[adjNode]) {
          pq.push({edW, adjNode});
        }
      }
    }
    return sum;
  }
};
```

7.8 Math

7.8.1 Number of Spanning Trees

Create an $N \times N$ matrix `mat`, and for each edge $a \to b \in G$, do `mat[a][b]--, mat[b][b]++` (and `mat[b][a]--, mat[a][a]++` if G is undirected). Remove the ith row and column and take the determinant; this yields the number of directed spanning trees rooted at i (if G is undirected, remove any row/column).

7.8.2 Erdős–Gallai theorem

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

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

Geometry (8)

8.1 Geometric primitives

Point.h

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

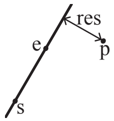
47ec0a, 28 lines

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

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

lineDistance.h

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

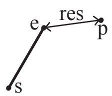


"Point.h" f6bf6b, 4 lines

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

SegmentDistance.h

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



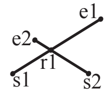
Usage: `Point<double> a, b(2,2), p(1,1);`
`bool onSegment = segDist(a,b,p) < 1e-10;`

"Point.h" 5c88f4, 6 lines

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

SegmentIntersection.h

Description:
If a unique intersection point between the line segments going from s1 to e1 and from s2 to e2 exists then it is returned. If no intersection point exists an empty vector is returned. If infinitely many exist a vector with 2 elements is returned, containing the endpoints of the common line segment. The wrong position will be returned if P is Point<ll> and the intersection point does not have integer coordinates. Products of three coordinates are used in intermediate steps so watch out for overflow if using int or long long.



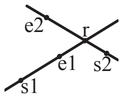
Usage: `vector<P> inter =`
```
segInter(s1,e1,s2,e2);
if (sz(inter)==1)
cout << "segments intersect at " <<
inter[0] << endl;
```

"Point.h", "OnSegment.h" 9d57f2, 13 lines

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

lineIntersection.h

Description:
If a unique intersection point of the lines going through s1,e1 and s2,e2 exists {1, point} is returned. If no intersection point exists {0, (0,0)} is returned and if infinitely many exists {-1, (0,0)} is returned. The wrong position will be returned if P is Point<ll> and the intersection point does not have integer coordinates. Products of three coordinates are used in intermediate steps so watch out for overflow if using int or ll.



Usage: `auto res = lineInter(s1,e1,s2,e2);`
```
if (res.first == 1)
cout << "intersection point at " <<
res.second << endl;
```
"Point.h" a01f81, 8 lines
```
template<class P>
pair<int, P> lineInter(P s1, P e1, P s2, P
    e2) {
  auto d = (e1 - s1).cross(e2 - s2);
  if (d == 0) // if parallel
    return {-(s1.cross(e1, s2) == 0), P(0,
        0)};
  auto p = s2.cross(e1, e2), q = s2.cross(e2
      , s1);
  return {1, (s1 * p + e1 * q) / d};
}
```

sideOf.h

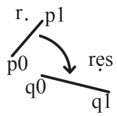
Description: Returns where p is as seen from s towards e. 1/0/-1 ⇔ left/on line/right. If the optional argument eps is given 0 is returned if p is within distance eps from the line. P is supposed to be Point<T> where T is e.g. double or long long. It uses products in intermediate steps so watch out for overflow if using int or long long.
Usage: `bool left = sideOf(p1,p2,q)==1;`
"Point.h" 3af81c, 9 lines
```
template<class P>
int sideOf(P s, P e, P p) { return sgn(s.
    cross(e, p)); }

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

OnSegment.h

Description: Returns true iff p lies on the line segment from s to e. Use `(segDist(s,e,p)<=epsilon)` instead when using Point<double>.
"Point.h" c597e8, 3 lines
```
template<class P> bool onSegment(P s, P e, P
    p) {
  return p.cross(s, e) == 0 && (s - p).dot(e
      - p) <= 0;
}
```

linearTransformation.h

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



"Point.h" 03a306, 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. Sometimes also represents points or vectors.
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
```
0f0602, 35 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 b) const { return {x
      -b.x, y-b.y, t}; }
  int half() const {
    assert(x || y);
    return y < 0 || (y == 0 && x < 0);
  }
  Angle t90() const { return {-y, x, t + (
      half() && x >= 0)}; }
  Angle t180() const { return {-x, -y, t +
      half()}; }
  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.half(), a.y * (ll
      )b.x) <
         make_tuple(b.t, b.half(), a.x * (ll
             )b.y);
}

// 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) { // point
    a + vector b
  Angle r(a.x + b.x, a.y + b.y, a.t);
  if (a.t180() < r) r.t--;
  return r.t180() < a ? r.t360() : r;
}
Angle angleDiff(Angle a, Angle b) { // angle
    b - angle a
  int tu = b.t - a.t; a.t = b.t;
  return {a.x*b.x + a.y*b.y, a.x*b.y - a.y*b
      .x, tu - (b < a)};
}
```

8.2 Circles

CircleIntersection.h

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

"Point.h" 84d6d3, 11 lines

```
typedef Point<double> P;
bool circleInter(P a,P b,double r1,double r2
    ,pair<P, P>* out) {
  if (a == b) { assert(r1 != r2); return
      false; }
  P vec = b - a;
  double d2 = vec.dist2(), sum = r1+r2, dif
      = r1-r2,
         p = (d2 + r1*r1 - r2*r2)/(d2*2), h2
             = r1*r1 - p*p*d2;
  if (sum*sum < d2 || dif*dif > d2) return
      false;
  P mid = a + vec*p, per = vec.perp() * sqrt
      (fmax(0, h2) / d2);
  *out = {mid + per, mid - per};
  return true;
}
```

CircleTangents.h

Description: Finds the external tangents of two circles, or internal if r2 is negated. Can return 0, 1, or 2 tangents – 0 if one circle contains the other (or overlaps it, in the internal case, or if the circles are the same); 1 if the circles are tangent to each other (in which case .first = .second and the tangent line is perpendicular to the line between the centers). .first and .second give the tangency points at circle 1 and 2 respectively. To find the tangents of a circle with a point set r2 to 0.

"Point.h" b0153d, 13 lines

```
template<class P>
vector<pair<P, P>> tangents(P c1, double r1,
    P c2, double r2) {
  P d = c2 - c1;
  double dr = r1 - r2, d2 = d.dist2(), h2 =
      d2 - dr * dr;
  if (d2 == 0 || h2 < 0)  return {};
  vector<pair<P, P>> out;
  for (double sign : {-1, 1}) {
    P v = (d * dr + d.perp() * sqrt(h2) *
        sign) / d2;
    out.push_back({c1 + v * r1, c2 + v * r2}
        );
  }
  if (h2 == 0) out.pop_back();
  return out;
}
```

CirclePolygonIntersection.h

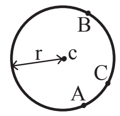
Description: Returns the area of the intersection of a circle with a ccw polygon.
Time: $\mathcal{O}(n)$

"../../content/geometry/Point.h" a1ee63, 19 lines

```
typedef Point<double> P;
#define arg(p, q) atan2(p.cross(q), p.dot(q)
    )
double circlePoly(P c, double r, vector<P>
    ps) {
  auto tri = [&](P p, P q) {
    auto r2 = r * r / 2;
    P d = q - p;
    auto a = d.dot(p)/d.dist2(), b = (p.
        dist2()-r*r)/d.dist2();
    auto det = a * a - b;
    if (det <= 0) return arg(p, q) * r2;
    auto s = max(0., -a-sqrt(det)), t = min
        (1., -a+sqrt(det));
    if (t < 0 || 1 <= s) return arg(p, q) *
        r2;
    P u = p + d * s, v = p + d * t;
    return arg(p,u) * r2 + u.cross(v)/2 +
        arg(v,q) * r2;
  };
  auto sum = 0.0;
  rep(i,0,sz(ps))
    sum += tri(ps[i] - c, ps[(i + 1) % sz(ps
        )] - c);
  return sum;
}
```

circumcircle.h

Description:
The circumcirle 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" 1caa3a, 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" 09dd0a, 17 lines

```
pair<P, double> mec(vector<P> ps) {
  shuffle(all(ps), mt19937(time(0)));
  P o = ps[0];
  double r = 0, EPS = 1 + 1e-8;
  rep(i,0,sz(ps)) if ((o - ps[i]).dist() > r
      * EPS) {
    o = ps[i], r = 0;
    rep(j,0,i) if ((o - ps[j]).dist() > r *
        EPS) {
      o = (ps[i] + ps[j]) / 2;
      r = (o - ps[i]).dist();
      rep(k,0,j) if ((o - ps[k]).dist() > r
          * EPS) {
        o = ccCenter(ps[i], ps[j], ps[k]);
        r = (o - ps[i]).dist();
      }
    }
  }
  return {o, r};
}
```

8.3 Polygons

InsidePolygon.h

Description: Returns true if p lies within the polygon. If strict is true, it returns false for points on the boundary. The algorithm uses products in intermediate steps so watch out for overflow.

Usage: `vector<P> v = {P{4,4}, P{1,2}, P{2,1}};`
`bool in = inPolygon(v, P{3, 3}, false);`

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

"Point.h", "OnSegment.h", "SegmentDistance.h" 2bf504, 11 lines

```
template<class P>
bool inPolygon(vector<P> &p, P a, bool
    strict = true) {
  int cnt = 0, n = sz(p);
  rep(i,0,n) {
    P q = p[(i + 1) % n];
    if (onSegment(p[i], q, a)) return !
        strict;
    //or: if (segDist(p[i], q, a) <= eps)
        return !strict;
    cnt ^= ((a.y<p[i].y) - (a.y<q.y)) * a.
        cross(p[i], q) > 0;
  }
  return cnt;
}
```

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" f12300, 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.

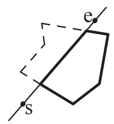
Time: $\mathcal{O}(n)$

"Point.h" 9706dc, 9 lines

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

PolygonCut.h

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



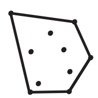
Usage: `vector<P> p = ...;`
`p = polygonCut(p, P(0,0), P(1,0));`

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

```
typedef Point<double> P;
vector<P> polygonCut(const vector<P>& poly,
    P s, P e) {
  vector<P> res;
  rep(i,0,sz(poly)) {
    P cur = poly[i], prev = i ? poly[i-1] :
        poly.back();
    bool side = s.cross(e, cur) < 0;
    if (side != (s.cross(e, prev) < 0))
      res.push_back(lineInter(s, e, cur,
          prev).second);
    if (side)
      res.push_back(cur);
  }
  return res;
}
```

ConvexHull.h

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



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

f5a3ef, 28 lines

```
template <class T> int sgn(T x) { return (x
    > 0) - (x < 0); }
template<class T>
struct Point {
  typedef Point P;
  T x, y;
  explicit Point(T x=0, T y=0) : x(x), y(y)
      {}
  bool operator<(P p) const { return tie(x,y
      ) < tie(p.x,p.y); }
  bool operator==(P p) const { return tie(x,
      y)==tie(p.x,p.y); }
  P operator-(P p) const { return P(x-p.x, y
      -p.y); }
  T cross(P a, P b) const { return (a-*this)
      .cross(b-*this); }
  T cross(P p) const { return x*p.y - y*p.x;
      }
  friend ostream& operator<<(ostream& os, P
      p) {
    return os << "(" << p.x << "," << p.y <<
        ")"; }
};

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

HullDiameter.h

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

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

"Point.h" c571b8, 12 lines

```
typedef Point<ll> P;
array<P, 2> hullDiameter(vector<P> S) {
  int n = sz(S), j = n < 2 ? 0 : 1;
```

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

PointInsideHull.h

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

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

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

```
typedef Point<ll> P;

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

LineHullIntersection.h

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

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

"Point.h" 7cf45b, 39 lines

```
#define cmp(i,j) sgn(dir.perp().cross(poly[(
    i)%n]-poly[(j)%n]))
#define extr(i) cmp(i + 1, i) >= 0 && cmp(i,
     i - 1 + n) < 0
template <class P> int extrVertex(vector<P>&
     poly, P dir) {
  int n = sz(poly), lo = 0, hi = n;
  if (extr(0)) return 0;
  while (lo + 1 < hi) {
    int m = (lo + hi) / 2;
    if (extr(m)) return m;
    int ls = cmp(lo + 1, lo), ms = cmp(m +
        1, m);
    (ls < ms || (ls == ms && ls == cmp(lo, m
        )) ? hi : lo) = m;
  }
  return lo;
}

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

8.4 Misc. Point Set Problems

ClosestPair.h

Description: Finds the closest pair of points.

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

"Point.h" ac41a6, 17 lines

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

kdTree.h

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

"Point.h" bac5b0, 63 lines

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

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

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

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

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

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

  pair<T, P> search(Node *node, const P& p) {
    if (!node->first) {
      // uncomment if we should not find the point itself:
      // if (p == node->pt) return {INF, P()};
      return make_pair((p - node->pt).dist2(), node->pt);
    }

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

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

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

8.5 3D

PolyhedronVolume.h

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

3058c3, 6 lines

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

Point3D.h

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

8058ae, 32 lines

```
template<class T> struct Point3D {
  typedef Point3D P;
  typedef const P& R;
  T x, y, z;
  explicit Point3D(T x=0, T y=0, T z=0) : x(x), y(y), z(z) {}
  bool operator<(R p) const {
    return tie(x, y, z) < tie(p.x, p.y, p.z); }
  bool operator==(R p) const {
    return tie(x, y, z) == tie(p.x, p.y, p.z); }
  P operator+(R p) const { return P(x+p.x, y+p.y, z+p.z); }
  P operator-(R p) const { return P(x-p.x, y-p.y, z-p.z); }
  P operator*(T d) const { return P(x*d, y*d, z*d); }
  P operator/(T d) const { return P(x/d, y/d, z/d); }
  T dot(R p) const { return x*p.x + y*p.y + z*p.z; }
  P cross(R p) const {
    return P(y*p.z - z*p.y, z*p.x - x*p.z, x*p.y - y*p.x);
  }
  T dist2() const { return x*x + y*y + z*z; }
```

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

3dHull.h

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

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

"Point3D.h" 5b45fc, 49 lines

```
typedef Point3D<double> P3;

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

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

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

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

sphericalDistance.h

Description: Returns the shortest distance on the sphere with radius radius between the points with azimuthal angles (longitude) f1 (ϕ_1) and f2 (ϕ_2) from x axis and zenith angles (latitude) t1 (θ_1) and t2 (θ_2) from z axis (0 = north pole). All angles measured in radians. The algorithm starts by converting the spherical coordinates to cartesian coordinates so if that is what you have you can use only the two last rows. dx*radius is then the difference between the two points in the x direction and d*radius is the total distance between the points.

611f07, 8 lines

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

Strings (9)

KMP.h

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

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

d4375c, 16 lines

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

vi match(const string& s, const string& pat)
    {
  vi p = pi(pat + '\0' + s), res;
  rep(i,sz(p)-sz(s),sz(p))
```

```
    if (p[i] == sz(pat)) res.push_back(i - 2
        * sz(pat));
  return res;
}
```

Zfunc.h

Description: z[i] computes the length of the longest common prefix of s[i:] and s, except z[0] = 0. (abacaba -> 0010301)
Time: $\mathcal{O}(n)$

ee09e2, 12 lines

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

Manacher.h

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

e7ad79, 13 lines

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

MinRotation.h

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

d07a42, 8 lines

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

SuffixArray.h

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

148d7d, 36 lines

```
struct SuffixArray {
  vi sa, lcp;
  SuffixArray(string& s, int lim=256) { //
      or basic_string<int>
    int n = sz(s) + 1, k = 0, a, b;
    vi x(all(s)), y(n), ws(max(n, lim));
    x.push_back(0), sa = lcp = y, iota(all(
        sa), 0);
    for (int j = 0, p = 0; p < n; j = max(1,
        j * 2), lim = p) {
      p = j, iota(all(y), n - j);
      fr(i,0,n) if (sa[i] >= j) y[p++] = sa[
          i] - j;
      fill(all(ws), 0);
      fr(i,0,n) ws[x[i]]++;
      fr(i,1,lim) ws[i] += ws[i - 1];
      for (int i = n; i--;) sa[--ws[x[y[i
          ]]]] = y[i];
      swap(x, y), p = 1, x[sa[0]] = 0;
      fr(i,1,n) a = sa[i - 1], b = sa[i], x[
          b] =
          (y[a] == y[b] && y[a + j] == y[b + j
            ]) ? p - 1 : p++;
    }
    for (int i = 0, j; i < n - 1; lcp[x[i
        ++]] = k)
      for (k && k--, j = sa[x[i] - 1];
          s[i + k] == s[j + k]; k++);
  }
};
int lower_bound(string& t,vector<int> &a,
    string &s){
  int l = 1,r=sz(a);
  while(l<r){
    int m = (l+r)/2;
    if(s.substr(a[m],min(sz(s)-a[m],sz(t)+1)
        ) >= t) r = m;
    else l = m+1;}
  return l;}
int upper_bound(string& t,vector<int> &a,
    string &s){
  int l = 1,r=sz(a);
  while(l<r){
    int m = (l+r)/2;
    if(s.substr(a[m],min(sz(a)-a[m],sz(t)))
        > t) r = m;
    else l = m+1;}
  return l;}
```

Hashing.h

Description: Self-explanatory methods for string hashing.

2d2a67, 44 lines

```
// Arithmetic mod 2^64-1. 2x slower than mod
    2^64 and more
// code, but works on evil test data (e.g.
    Thue-Morse, where
// ABBA... and BAAB... of length 2^10 hash
    the same mod 2^64).
// "typedef ull H;" instead if you think
    test data is random,
// or work mod 10^9+7 if the Birthday
    paradox is not a problem.
typedef uint64_t ull;
struct H {
  ull x; H(ull x=0) : x(x) {}
```

```
  H operator+(H o) { return x + o.x + (x + o
      .x < x); }
  H operator-(H o) { return *this + ~o.x; }
  H operator*(H o) { auto m = (__uint128_t)x
       * o.x;
    return H((ull)m) + (ull)(m >> 64); }
  ull get() const { return x + !~x; }
  bool operator==(H o) const { return get()
      == o.get(); }
  bool operator<(H o) const { return get() <
       o.get(); }
};
static const H C = (ll)1e11+3; // (order ~ 3
   e9; random also ok)

struct HashInterval {
  vector<H> ha, pw;
  HashInterval(string& 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;
  }
  H hashInterval(int a, int b) { // hash [a,
       b)
    return ha[b] - ha[a] * pw[b - a];
  }
};

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

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

Trie.h

Description: Trie.h

2ae50f, 199 lines

```
class Trie {
public:

  //N is number of possible characters in a
      string
  const static int N = 26;

  //baseChar defines the base character for
      possible characters
  //like '0' for '0','1','2'... as possible
      characters in string
    const static char baseChar = 'a';

  struct TrieNode
  {
    int next[N];
    //if isEnd is set to true , a string
        ended here
    bool isEnd;
    //freq is how many times this prefix
        occurs
      int freq;

    TrieNode()
    {
      for(int i=0;i<N;i++)
        next[i] = -1;
      isEnd = false;
      freq = 0;
    }
  };

  //the implementation is via vector and
      each position in this vector
  //is similar as new pointer in pointer
      type implementation
  vector <TrieNode> tree;

  //Base Constructor
  Trie ()
  {
    tree.push_back(TrieNode());
  }

  //inserting a string in trie
  void insert(const string &s)
  {
      int p = 0;
      tree[p].freq++;
      for(int i=0;i<s.size();i++)
      {
        // tree[]
          if(tree[p].next[s[i]-baseChar]
              == -1)
          {
              tree.push_back(TrieNode());
              tree[p].next[s[i]-baseChar]
                  = tree.size()-1;
          }

          p = tree[p].next[s[i]-baseChar];
          tree[p].freq++;
      }
      tree[p].isEnd = true;
  }

  //check if a string exists as prefix
  bool checkPrefix(const string &s)
  {
    int p = 0;
    for(int i=0;i<s.size();i++)
    {
      if(tree[p].next[s[i]-baseChar] ==
          -1)
        return false;

      p = tree[p].next[s[i]-baseChar];
    }
    return true;
  }

  //check is string exists
  bool checkString(const string &s)
  {
    int p = 0;
    for(int i=0;i<s.size();i++)
    {
      if(tree[p].next[s[i]-baseChar] ==
          -1)
```

```cpp
        return false;

      p = tree[p].next[s[i]-baseChar];
    }

    return tree[p].isEnd;
  }

  //persistent insert
  //returns location of new head
  int persistentInsert(int head , const string &s)
  {
    int old = head;

    tree.push_back(TrieNode());
    int now = tree.size()-1;
    int newHead = now;

    int i,j;

    for(i=0;i<s.size();i++)
    {
      if(old == -1)
      {
        tree.push_back(TrieNode());
        tree[now].next[s[i]-baseChar] = tree.size() - 1;
        tree[now].freq++;
        now = tree[now].next[s[i]-baseChar];
        continue;
      }
      for(j=0;j<N;j++)
        tree[now].next[j] = tree[old].next[j];
      tree[now].freq = tree[old].freq;
      tree[now].isEnd = tree[old].isEnd;

      tree[now].freq++;

      tree.push_back(TrieNode());
      tree[now].next[s[i]-baseChar] = tree.size()-1;

      old = tree[old].next[s[i]-baseChar];
      now = tree[now].next[s[i]-baseChar];
    }

    tree[now].freq++;
    tree[now].isEnd = true;

    return newHead;
  }

  //persistent check prefix
  bool persistentCheckPrefix(int head, const string &s)
  {
    int p = head;
    for(int i=0;i<s.size();i++)
    {
      if(tree[p].next[s[i]-baseChar] == -1)
        return false;

      p = tree[p].next[s[i]-baseChar];
    }
    return true;
  }

  //persistent check string
  bool persistentCheckString(int head, const string &s)
  {
    int p = head;
    for(int i=0;i<s.size();i++)
    {
      if(tree[p].next[s[i]-baseChar] == -1)
        return false;

      p = tree[p].next[s[i]-baseChar];
    }
    return tree[p].isEnd;
  }
};

string s,temp;
int main()
{
    Trie trie = Trie();

    cout << trie.checkString("hello") << endl;  //output : 0

    trie.insert("hello");

    cout << trie.checkPrefix("hell") << endl; //output : 1

    cout << trie.checkString("hell") << endl; //output : 0

    cout << trie.checkString("hello") << endl;  //output : 1

    //Example for persistent trie
    Trie persistentTrie = Trie();
    vector <int> heads;

    //insert words
    heads.push_back(0);
    heads.push_back(persistentTrie.persistentInsert(heads[heads.size()-1] , "hello"));
    heads.push_back(persistentTrie.persistentInsert(heads[heads.size()-1] , "world"));
    heads.push_back(persistentTrie.persistentInsert(heads[heads.size()-1] , "persistent"));
    heads.push_back(persistentTrie.persistentInsert(heads[heads.size()-1] , "trie"));

    cout << persistentTrie.persistentCheckString(heads[0] , "hello") << endl; //output : 0
    cout << persistentTrie.persistentCheckString(heads[1] , "hello") << endl; //output : 1

    cout << persistentTrie.persistentCheckString(heads[1] , "world") << endl; //output : 0
```

```
    cout << persistentTrie.
       persistentCheckString(heads[2] , "
       world") << endl; //output : 1

    cout << persistentTrie.
       persistentCheckString(heads[2] , "
       persistent") << endl;  //output : 0
    cout << persistentTrie.
       persistentCheckString(heads[3] , "
       persistent") << endl;  //output : 1

    cout << persistentTrie.
       persistentCheckString(heads[3] , "
       trie") << endl;  //output : 0
    cout << persistentTrie.
       persistentCheckString(heads[4] , "
       trie") << endl;  //output : 1

    return 0;
}
```

Various (10)

10.1 Intervals

IntervalContainer.h

Description: IntervalContainer.h

2b074b, 35 lines

```
struct non_overlapping_segment{
    set<pair<int,int>> seg;
    non_overlapping_segment()
    {
        seg.clear();
    }
    int insert(int lo, int hi)
    {
        auto it = seg.upper_bound({lo,0});
        int added = 0;
        if(it != seg.begin())
        {
            --it;
            if((*it).ss >= lo)
            {
                added -= (*it).ss - (*it).ff
                    + 1;
                lo = (*it).ff;
                hi = max(hi,(*it).ss);
                seg.erase(it);
            }
        }
        while(true)
        {
            auto it = seg.lower_bound({lo,0}
               );
            if(it == seg.end()) break;
            if((*it).ff > hi) break;
            hi = max(hi,(*it).ss);
            added -= (*it).ss - (*it).ff +
               1;
            seg.erase(it);
        }
        added += hi - lo + 1;
        seg.insert({lo,hi});
        return added;
    }
};
```

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

9e9d8d, 19 lines

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

ConstantIntervals.h

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

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

Time: $\mathcal{O}\left(k \log \frac{n}{k}\right)$

753a4c, 19 lines

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

10.2 Misc. algorithms

TernarySearch.h

Description: Find the smallest i in $[a, b]$ that maximizes $f(i)$, assuming that $f(a) < \dots < f(i) \geq \dots \geq f(b)$. To reverse which of the sides allows non-strict inequalities, change the < marked with (A) to <=, and reverse the loop at (B). To minimize f, change it to >, also at (B).

Usage: `int ind = ternSearch(0,n-1,[&](int i){return a[i];});`

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

9155b4, 11 lines

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

LIS.h

Description: Compute indices for the longest increasing subsequence.

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

2932a0, 17 lines

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

FastKnapsack.h

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

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

b20ccc, 16 lines

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

10.3 Dynamic programming

KnuthDP.h

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

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

DivideAndConquerDP.h

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

Time: $\mathcal{O}((N + (hi - lo)) \log N)$

d38d2b, 18 lines

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

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

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 theory
  Dynamic graphs (extra book-keeping)
  Breadth first search
  Depth first search
  * Normal trees / DFS trees
  Dijkstra's algorithm
  MST: Prim's algorithm
  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
  Euler cycles
  Flow networks
  * Augmenting paths
  * Edmonds-Karp
  Bipartite matching
  Min. path cover
  Topological sorting
  Strongly connected components
  2-SAT
  Cut vertices, cut-edges and 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
  Euclidean algorithm
  Modular arithmetic
  * Modular multiplication
  * Modular inverses
  * Modular exponentiation by squaring
  Chinese remainder theorem
  Fermat's little 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 polynomial 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
```