Jagiellonian University in Krakow

Jagiellonian teapots

Antoni Długosz, Kacper Paciorek, Kacper Topolski

2024-08-16

Contest (1)

sol.cpp

Description: Tempelka

<bits/stdc++.h> c23d80, 29 lines

```
using namespace std;
#define fwd(i, a, n) for (int i = (a); i < (n); i++)
#define rep(i, n) fwd(i, 0, n)
#define all(X) X.begin(), X.end()
#define sz(X) int(size(X))
#define pb push_back
#define eb emplace_back
#define st first
#define nd second
using pii = pair<int, int>; using vi = vector<int>;
using ll = long long; using ld = long double;
#ifdef LOC
auto SS = signal(6, [](int) { *(int *)0 = 0; });
#define DTP(x, y) auto operator << (auto &o, auto a) -> decltype(y, o) { o << "("; x; return o << ")"; }
DTP(o << a.st << ", " << a.nd, a.nd);
DTP(for (auto i : a) o << i << ", ", all(a));
void dump(auto... x) { (( cerr << x << ", " ), ...) << '\n'; }
#define deb(x...) cerr << setw(4) << __LINE__ << ":[" #x "]: ", dump(x)
#else
#define deb(...) 0
#endif
int32_t main() {
  cin.tie(0)->sync_with_stdio(0);
  cout << fixed << setprecision(10);
  #ifdef LOCF
  cout.flush(); cerr << "- - - - - - - - -\n";
  (void)!system("grep VmPeak /proc/$PPID/status | sed s/....kB/\' MB\'/1 >&2"); // 4x.kB ....kB
  #endif
}
```

.bashrc

8 lines

```
c() {
 g++ -std=c++20 -Wall -Wextra -Wshadow -Wconversion \
 -Wno-sign-conversion -Wfloat-equal -D_GLIBCXX_DEBUG \
 -D_GLIBCXX_DEBUG_PEDANTIC -fsanitize=address,undefined \
 -ggdb3 -DLOC $1.cpp -o$1; }
cf() { g++ -std=c++20 -static -O3 -DLOCF $1.cpp -o$1; }
libhash() { cat $1.cpp | cpp -dD -P -fpreprocessed | \
tr -d '[:space:]'| md5sum |cut -c-6; }
```

Mathematics (2)

2.1 Equations

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

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 Trigonometry

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

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

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

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

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

2.3 Geometry

2.3.1 Triangles

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

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

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

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

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

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

2.3.2 Quadrilaterals

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

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

For cyclic quadrilaterals $ef = ac + bd$, and:

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

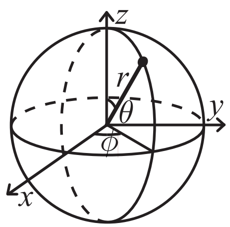
2.4 Derivatives/Integrals

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

2.5 Series

$$\ln(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots, (-1 < x \le 1)$$
$$\sqrt{1+x} = 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{2x^3}{32} - \frac{5x^4}{128} + \ldots, (-1 \le x \le 1)$$

2.5.1 Spherical coordinates



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

2.6 Sums

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

2.6.1 Discrete distributions

Binomial distribution

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

$$p(k) = \binom{n}{k} p^k (1-p)^{n-k}$$
$$\mu = np, \ \sigma^2 = np(1-p)$$

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

First success distribution

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

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

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 $\operatorname{Po}(\lambda)$, $\lambda = t\kappa$.

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

2.6.2 Continuous distributions

Uniform distribution

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

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

Exponential distribution

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

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

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

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

2.7 Markov chains

A *Markov chain* is a discrete random process with the property that the next state depends only on the current state. Let $X_1, X_2, \ldots$ 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 $t_i = 1 + \sum_{k\in\mathbf{G}} p_{ki}t_k$.

Data structures (3)

PBDS.h

Description: Policy Based Data Structures 460200, 17 lines

```
// Order Statistics Tree: Caution: Not a multiset!
#include <bits/extc++.h>
using namespace __gnu_pbds;
template <class T> using Tree = tree<T, null_type, less
      <T>, rb_tree_tag,
      tree_order_statistics_node_update>;
Tree<int> t, t2;
auto it = t.insert(10).first; // it == t.upper_bound(9)
    ;
t.order_of_key(10); // # of entries strictly smaller
    than key
t.join(t2); // fast only if max(T) < min(T2) or min(T)
    > max(T2)
// Hash Table: faster but can lead to MLE (1.5x worse
    performance), initial capacity must = 2^k
struct chash { // large odd number for C
  const uint64_t C = ll(4e18 * acos(0)) | 71;
  ll operator()(ll x) const { return __builtin_bswap64(
      x * C); }
};
gp_hash_table<ll, int, chash> h({}, {}, {}, {}, {1 <<
    16}); // cc_hash_table also exists if needed
```

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);
  }
};
```

PersistentSegmentTreePointUpdate.h

Description: sparse (N can be up to 1e18) persistent segment tree supporting point updates and range queries. Ranges are inclusive
Time: $\mathcal{O}(\log N)$ 138ef8, 48 lines

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

PersistentSegmentTreeLazy.h

Description: sparse (N can be up to 1e18) persistent segment tree supporting lazy propagation. Ranges are inclusive
Time: $\mathcal{O}(\log N)$ 7427f5, 73 lines

```
struct LazyPSegmentTree { // default: update +, query
    max
  typedef int val;
  val idntV = 0; // identity value
  val fV(val l, val r) {
    return max(l, r); // implement combining values
  }
  typedef int lazy;
  lazy idntL = 0;
  lazy fL(lazy prv, lazy nxt) {
    return prv + nxt; // implement combining lazy
  }
  val apl(val x, lazy lz) {
    return x + lz; // implement applying lazy
  }
  struct node {
    int l = 0, r = 0;
    val x;
    lazy lz;
    node(val x, lazy lz) : x(x), lz(lz) {
    }
  };
  int N;
  vector<node> t;
  LazyPSegmentTree(int N) : N(N) {
    t.pb(
      node(idntV, idntL)); // 0th node is the root of
          an empty tree
              // t.reserve() in case of memory
                  issues
  }
  int cpy(int v) {
    t.pb(t[v]);
    return sz(t) - 1;
  }
  void aplV(int v, lazy lz) {
    t[v].lz = fL(t[v].lz, lz);
    t[v].x = apl(t[v].x, lz);
  }
  // creates 4 * lgN +- eps new nodes
  int upd(int v, int l, int r, lazy lz, int a = 0, int
      b = -1, int u = -1) {
    if (u == -1) {
      u = cpy(v);
      b = N - 1;
    }
    if (l > b || r < a)
      return u;
    if (a >= l && b <= r) {
      aplV(u, lz);
      return u;
    }
    int c = (a + b) / 2;
    t[u].l = cpy(t[v].l);
    t[u].r = cpy(t[v].r);
    aplV(t[u].l, t[u].lz);
    aplV(t[u].r, t[u].lz);
    upd(t[v].l, l, r, lz, a, c, t[u].l);
    upd(t[v].r, l, r, lz, c + 1, b, t[u].r);
    t[u].lz = idntL;
    t[u].x = fV(t[t[u].l].x, t[t[u].r].x);
    return u;
  }
  // doesn't create new nodes
  val get(int v, int l, int r, int cl = 0, int cr = -1)
        {
    if (cr == -1)
      cr = N - 1;
    if (!v || l > cr || r < cl)
      return idntV;
    if (cl >= l && cr <= r)
      return t[v].x;
    int m = (cl + cr) / 2;
    return apl(
      fV(get(t[v].l, l, r, cl, m), get(t[v].r, l, r, m
          + 1, cr)),
      t[v].lz);
  }
};
```

LazySegmentTree.h

Description: Segment tree with ability to add or set values of large intervals, and compute max of intervals. Can be changed to other things. Use with a bump allocator for better performance, and SmallPtr or implicit indices to save memory.
Usage: Node* tr = new Node(v, 0, sz(v));
Time: $\mathcal{O}(\log N)$.
"../various/BumpAllocator.h" 34ecf5, 50 lines

```
const int inf = 1e9;
struct Node {
  Node *l = 0, *r = 0;
  int lo, hi, mset = inf, madd = 0, val = -inf;
  Node(int lo,int hi):lo(lo),hi(hi){} // Large interval
      of -inf
  Node(vi& v, int lo, int hi) : lo(lo), hi(hi) {
    if (lo + 1 < hi) {
      int mid = lo + (hi - lo)/2;
      l = new Node(v, lo, mid); r = new Node(v, mid, hi
          );
      val = max(l->val, r->val);
    }
    else val = v[lo];
  }
  int query(int L, int R) {
    if (R <= lo || hi <= L) return -inf;
    if (L <= lo && hi <= R) return val;
    push();
    return max(l->query(L, R), r->query(L, R));
  }
  void set(int L, int R, int x) {
    if (R <= lo || hi <= L) return;
    if (L <= lo && hi <= R) mset = val = x, madd = 0;
    else {
      push(), l->set(L, R, x), r->set(L, R, x);
      val = max(l->val, r->val);
    }
  }
  void add(int L, int R, int x) {
    if (R <= lo || hi <= L) return;
    if (L <= lo && hi <= R) {
      if (mset != inf) mset += x;
      else madd += x;
      val += x;
    }
    else {
      push(), l->add(L, R, x), r->add(L, R, x);
      val = max(l->val, r->val);
    }
  }
  void push() {
    if (!l) {
      int mid = lo + (hi - lo)/2;
      l = new Node(lo, mid); r = new Node(mid, hi);
    }
    if (mset != inf)
      l->set(lo,hi,mset), r->set(lo,hi,mset), mset =
          inf;
    else if (madd)
      l->add(lo,hi,madd), r->add(lo,hi,madd), madd = 0;
  }
};
```

UnionFind.h

Description: Disjoint-set data structure.
Time: $\mathcal{O}(\alpha(N))$ 7aa27c, 14 lines

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

UnionFindRollback.h

Description: Disjoint-set data structure with undo. If undo is not needed, skip st, time() and rollback().
Usage: int t = uf.time(); ...; uf.rollback(t);
Time: $\mathcal{O}(\log(N))$ 84e98b, 21 lines

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

DequeRollback.h

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

```
struct DequeUndo {
  // Argument for insert(...) method of DS.
```

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

LineContainer.h

Description: Container where you can add lines of the form kx+m, and query maximum values at points x. Useful for dynamic programming ("convex hull trick").
Time: $\mathcal{O}(\log N)$ 8ec1c7, 30 lines

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

Treap.h

Description: Self balancing tree with different operations in logn expected. 0c61d4, 101 lines

```
struct Treap {
  struct Node {
    // E[0] = left child, E[1] = right child
    // weight = node random weight (for treap)
    // size = subtree size, par = parent node
    int E[2] = {-1, -1}, weight = rand();
    int size = 1, par = -1;
    bool flip = 0; // Is interval reversed?
  };
  vector<Node> G;
  Treap(int n = 0) : G(n) {} // makes n disjoint nodes
  int make() { G.pb({}); return sz(G)-1; }
  int size(int x) { // subtree size of node x
    return (x >= 0 ? G[x].size : 0);
  }
  void push(int x) { // x can be -1 !!!
    if (x >= 0 && G[x].flip) {
      G[x].flip = 0;
      swap(G[x].E[0], G[x].E[1]);
      for(auto e : G[x].E) if (e>=0) G[e].flip ^= 1;
    } // + any other lazy operations
  }
  void update(int x) { // pull data up (reverse of push
        )
    if (x >= 0) {
      int& s = G[x].size = 1;
      G[x].par = -1;
      for (auto e : G[x].E) if (e >= 0) {
        s += G[e].size;
        G[e].par = x;
      }
    } // + any other aggregates
  }
  // Split treap x into treaps l and r
  // such that l contains first i elements
  // and r the remaining ones.
  // x, l, r can be -1; time: ~O(lg n)
  void split(int x, int& l, int& r, int i) {
    push(x); l = r = -1;
    if (x < 0) return;
    int key = size(G[x].E[0]);
    if (i <= key) {
      split(G[x].E[0], l, G[x].E[0], i);
      r = x;
    } else {
      split(G[x].E[1], G[x].E[1], r, i-key-1);
      l = x;
    }
    update(x);
  }
  // Join treaps l and r into one treap left to right
  // l, r and returned value can be -1.
  int join(int l, int r) { // time: ~O(lg n)
    push(l); push(r);
    if (l < 0 || r < 0) return max(l, r);
    if (G[l].weight < G[r].weight) {
      G[l].E[1] = join(G[l].E[1], r);
      update(l);
      return l;
    }
    G[r].E[0] = join(l, G[r].E[0]);
    update(r);
    return r;
  }
  // Find i-th node in treap x.
  // Returns its key or -1 if not found.
  // x can be -1; time: ~O(lg n)
  int find(int x, int i) {
    while (x >= 0) {
      push(x);
      int key = size(G[x].E[0]);
      if (key == i) return x;
      x = G[x].E[key < i];
      if (key < i) i -= key+1;
    }
    return -1;
  }
  // Get key of treap containing node x
  // (key of treap root). x can be -1.
  int root(int x) { // time: ~O(lg n)
    while (G[x].par >= 0) x = G[x].par;
    return x;
  }
  // Get position of node x in its treap.
  // x is assumed to NOT be -1; time: ~O(lg n)
  int index(int x) {
    int p, i = size(G[x].E[G[x].flip]);
    while ((p = G[x].par) >= 0) {
      if (G[p].E[1] == x) i+=size(G[p].E[0])+1;
      if (G[p].flip) i = G[p].size-i-1;
      x = p;
    }
    return i;
  }
  // Reverse interval [l;r) in treap x.
  // Returns new key of treap; time: ~O(lg n)
  int reverse(int x, int l, int r) {
    int a, b, c;
    split(x, b, c, r);
    split(b, a, b, l);
    if (b >= 0) G[b].flip ^= 1;
    return join(join(a, b), c); } };
```

LiChao.h

Description: Extended Li Chao tree (segment tree for functions). Let F be a family of functions closed under function addition, such that for every $f \neq g$ from the family F there exists x such that $f(z) \leq g(z)$ for $z \leq x$ else $f(z) \geq g(z)$ or the other way around (intersect at one point). Typically F is the family of linear functions. DS maintains a sequence $c_0, c_1 \ldots c_{n-1}$ under operations max, add. b88a40, 74 lines

```
struct LiChao {
  struct Func {
    ll a, b; // a*x + b
    // Evaluate function in point x
    ll operator()(ll x) const { return a*x+b; }
    Func operator+(Func r) const {
      return {a+r.a, b+r.b};
    } // Sum of two functions
  }; // ID_ADD/MAX neutral elements for add/max
  static constexpr Func ID_ADD{0, 0};
  static constexpr Func ID_MAX{0, ll(-1e9)};
  vector<Func> val, lazy;
  int len;
  // Initialize tree for n elements; time: O(n)
  LiChao(int n = 0) {
    for (len = 1; len < n; len *= 2);
    val.resize(len*2, ID_MAX);
    lazy.resize(len*2, ID_ADD);
  }
  void push(int i) {
    if (i < len) rep(j, 2) {
      lazy[i*2+j] = lazy[i*2+j] + lazy[i];
      val[i*2+j] = val[i*2+j] + lazy[i];
    }
    lazy[i] = ID_ADD;
  } // For each x in [vb;ve)
    // set c[x] = max(c[x], f(x));
    // time: O(log^2 n) in general case,
    //       O(log n) if [vb;ve) = [0;len)
  void max(int vb, int ve, Func f,
           int i = 1, int b = 0, int e = -1) {
    if (e < 0) e = len;
    if (vb >= e || b >= ve || i >= len*2)
      return;
    int m = (b+e) / 2;
    push(i);
    if (b >= vb && e <= ve) {
      auto& g = val[i];
      if (g(m) < f(m)) swap(g, f);
      if (g(b) < f(b))
        max(vb, ve, f, i*2, b, m);
      else
        max(vb, ve, f, i*2+1, m, e);
    } else {
      max(vb, ve, f, i*2, b, m);
      max(vb, ve, f, i*2+1, m, e);
    }
  } // For each x in [vb;ve)
    // set c[x] = c[x] + f(x);
    // time: O(log^2 n) in general case,
    //       O(1) if [vb;ve) = [0;len)
  void add(int vb, int ve, Func f,
           int i = 1, int b = 0, int e = -1) {
    if (e < 0) e = len;
    if (vb >= e || b >= ve) return;
    if (b >= vb && e <= ve) {
      lazy[i] = lazy[i] + f;
      val[i] = val[i] + f;
    } else {
      int m = (b+e) / 2;
      push(i);
      max(b, m, val[i], i*2, b, m);
      max(m, e, val[i], i*2+1, m, e);
      val[i] = ID_MAX;
      add(vb, ve, f, i*2, b, m);
      add(vb, ve, f, i*2+1, m, e);
    }
  } // Get value of c[x]; time: O(log n)
  auto query(int x) {
    int i = x+len;
    auto ret = val[i](x);
    while (i /= 2)
      ret = ::max(ret+lazy[i](x), val[i](x));
    return ret; } };
```

IntSet.h

Description: bitset with fast predecessor and successor queries. Assumes x86 shift overflows. Extremely fast (50-200mln operations in 1 second). 85cd6f, 32 lines

```
template<int N>
struct IntSet {
  static constexpr int B = 64;
  uint64_t V[N / B + 1] = {};
  IntSet<(N < B + 1 ? 0 : N / B + 1)> up;
  bool has(int i) { return (V[i / B] >> i) & 1; }
  void add(int i) {
    if (!V[i / B]) up.add(i / B);
    V[i / B] |= 1ull << i;
  }
  void del(int i) {
    if (!(V[i / B] &= ~(1ull << i))) up.del(i / B);
  }
  int next(int i) { // j > i such that j inside or -1
    auto x = V[i / B] >> i;
    if (x &= ~1) return i + __builtin_ctzll(x);
    return (i = up.next(i / B)) < 0 ? i :
      i * B + __builtin_ctzll(V[i]);
  }
  int prev(int i) { // j < i such that j inside or -1
    auto x = V[i / B] << (B - i - 1);
    if (x &= INT64_MAX)
      return i-__builtin_clzll(x);
    return (i = up.prev(i / B)) < 0 ? i :
      i * B + B - 1 - __builtin_clzll(V[i]);
  }
};
template<>
struct IntSet<0> {
  void add(int) {} void del(int) {}
  int next(int) { return -1; }
  int prev(int) { return -1; } };
```

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 $\mathcal{O}(\log N)$.)
"FenwickTree.h" fa9f0d, 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].pb(y);
  }
  void init() {
    for (vi& v : ys) sort(all(v)), ft.eb(sz(v));
  }
  int ind(int x, int y) {
```

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

WaveletTree.h

Description: Wavelet tree. Supports fast kth order statistics on ranges (no updates).
Time: $\mathcal{O}(\log N)$ 587095, 35 lines

```
struct WaveletTree {
  vector<vi> seq, left;
  int len;
  WaveletTree() {}
  // time and space: O((n+maxVal) log maxVal)
  // Values are expected to be in [0;maxVal).
  WaveletTree(const vi& elems, int maxVal) {
    for (len = 1; len < maxVal; len *= 2);
    seq.resize(len*2); left.resize(len*2);
    seq[1] = elems; build(1, 0, len);
  }
  void build(int i, int b, int e) {
    if (i >= len) return;
    int m = (b+e) / 2;
    left[i].pb(0);
    for(auto &x : seq[i]) {
      left[i].pb(left[i].back() + (x < m));
      seq[i*2 + (x >= m)].pb(x);
    }
    build(i*2, b, m); build(i*2+1, m, e);
  } // Find k-th (0 indexed) smallest element in [begin;end)
  int kth(int begin, int end, int k, int i=1) {
    if (i >= len) return seq[i][0];
    int x = left[i][begin], y = left[i][end];
    if (k < y-x) return kth(x, y, k, i*2);
    return kth(begin-x, end-y, k-y+x, i*2+1);
  } // Count number of elements >= vb and < ve
  int count(int begin, int end, int vb, int ve, int i = 1, int b = 0, int e = -1) {
    if (e < 0) e = len;
    if (b >= ve || vb >= e) return 0;
    if (b >= vb && e <= ve) return end-begin;
    int m = (b+e) / 2; int x = left[i][begin], y = left[i][end];
    return count(x, y, vb, ve, i*2, b, m) + count(begin-x, end-y, vb, ve, i*2+1,m,e);
  }
};
```

RMQ.h

Description: RMQ on intervals [l, r]. Second one has 2-3x less memory and is 2-3x faster Size of array CANNOT be zero. e87fd2, 41 lines

```
template<class T>
struct RMQ { // #define / undef min func for different merging
  vector<vector<T> > s; RMQ(){} // only if needed
  RMQ(vector<T>& a) : s(1, a) {
    if (!sz(a)) return;
    rep(d, __lg(sz(a))) {
      s.eb(sz(a) - (1 << d) * 2 + 1);
      rep(j, sz(s[d + 1]))
        s[d + 1][j] = min(s[d][j], s[d][j + (1 << d)]);
    }
  }
  T get(int l, int r) {
    int d = __lg(r - l + 1);
    return min(s[d][l], s[d][r - (1 << d) + 1]);
  }
};
template<class T>
struct RMQF {
  static constexpr int B = 32; // not larger!
  RMQ<T> s;
  vector<uint32_t> m;
  vector<T> a, c; RMQF(){} // only if needed
  RMQF(vector<T>& A) : m(sz(A)), a(A), c(sz(A)) {
    vector<T> b(sz(a) / B + 1);
    uint32_t mi = 0;
    rep(i, sz(a)) {
      b[i / B] = (i % B ? min(b[i / B], a[i]) : a[i]);
      mi <<= 1;
      while (mi && a[i] < a[i - __builtin_ctz(mi)])
        mi ^= (1u << __builtin_ctz(mi));
      m[i] = mi ^= 1; c[i] = a[i - __lg(m[i])];
    }
    s = RMQ(b);
  }
  T get(int l, int r) {
    if (r - l + 1 < B)
      return a[r - __lg(m[r] & ((1u << (r - l + 1)) - 1))];
    T k = min(c[r], c[l + B - 1]);
    l = (l + B - 1) / B, r = r / B - 1;
    if (l <= r) k = min(k, s.get(l, r));
    return k; } };
```

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: $\mathcal{O}\left(N\sqrt{Q}\right)$ 1957f4, 49 lines

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

SubsetSumMod.h

Description: Modular subset sum in nlogn. 408e76, 134 lines

```
// Shift-tree with splitmix64 hashing.
struct ShiftTree {
  vector<uint64_t> H;
  int len, delta;
  // Init tree of size n = 2^d.
  ShiftTree(int n) : H(n*2), len(n), delta(0) {
    assert(n && !(n & (n-1)));
  }
  // Set a[i] := 1; time: O(log n)
  void set(int i) {
    H[i = (i+len-delta) % len + len] = 1;
    for (int d = delta; i > 1; d /= 2)
      update(i = parent(i, d%2), d%2);
  }
  // Cyclically shift by k to the right;
  // time: O(n / 2^j), where j max s.t. 2^j | k
  void shift(int k) {
    if (k %= len) {
      delta = (delta+len+k) % len;
      int div = k & ~(k-1), d = delta / div;
      for (int t = len/div/2; t >= 1; t /= 2) {
        fwd(i, t, t*2) update(i, d%2);
        d /= 2;
      }
    }
  }
  // Find mismatches between T[a:b) and Q[a:b);
  // time: O((|D|+1) log n)
  void diff(vi& out, const ShiftTree& T,
            int vb, int ve, int lvl = -1,
            int b = 0, int e = -1,
            int i = 1, int j = 1) {
    if (e < 0) lvl = __lg(e=len)-1;
    if (b >= ve || vb >= e || H[i] == T.H[j])
      return;
    if (e-b == 1) return out.push_back(b);
    int m = (b+e) / 2;
    int s1 = (delta >> lvl) & 1;
    int s2 = (T.delta >> lvl) & 1;
    diff(out, T, vb, ve, lvl-1, b, m,
      left(i, s1), left(j, s2));
    diff(out, T, vb, ve, lvl-1, m, e,
      right(i, s1), right(j, s2));
  }
  void update(int i, int s) {
    auto x = H[left(i, s)] +
      H[right(i, s)] * 0x9E37'79B9'7F4A'7C15;
    x = (x ^ (x>>30)) * 0xBF58'476D'1CE4'E5B9;
    x = (x ^ (x>>27)) * 0x94D0'49BB'1331'11EB;
    H[i] = x ^ (x >> 31);
  }
  int parent(int i, int s) {
    int k = i + s;
    return k&i ? k/2 : k/4;
  }
  int left(int i, int s) {
    int k = i*2, j = k - s;
    return k&j ? j : k|j;
  }
  int right(int i, int s) {
    return i*2 + !s;
  }
};
int bitrev(int n, int bits) {
  int ret = 0;
  rep(i, bits)
    ret |= ((n >> i) & 1) << (bits-i-1);
  return ret;
}
// Find all attainable subset sums modulo m;
// time: O(m log m)
// Input elements are given by frequency array,
// i.e. counts[x] = how many times element x
// is contained in the input multiset.
// Size of `counts` is the modulus m.
// The returned array encodes solutions,
// which can be recovered using `recover`.
// ans[x] != -1 <=> subset with sum x exists
vi subsetSumMod(const vi& counts) {
  int mod = sz(counts), len = 1, k = 0;
  while (len < mod*2) len *= 2, k++;
  vi tmp, ans(mod, -1);
  ShiftTree T(len), Q(len);
  ans[0] = 0;
  T.set(0);
  Q.set(0);
  Q.set(-mod);
  fwd(i, 1, len) {
    int x = bitrev(i, k);
    if (x >= mod || !counts[x]) continue;
    Q.shift(x - Q.delta);
    rep(j, counts[x]) {
      tmp.clear();
      T.diff(tmp, Q, 0, mod);
      if (tmp.empty()) break;
      for (auto &d : tmp) if (ans[d] == -1) {
        ans[d] = x;
        T.set(d);
        Q.set(d+x);
        Q.set(d+x-mod);
      }
    }
  }
  return ans;
}
vi recoverSubset(const vi& dp, int s) {
  assert(dp[s] != -1);
  vi ret;
  while (s) {
    ret.pb(dp[s]);
    s = (s - dp[s] + sz(dp)) % sz(dp);
  }
  return ret;
}
```

3.1 Potepa Drzewka

SGTPotepaConfig.h

Description: Segment tree config. 6ef40e, 150 lines

```
// Segment tree configurations to be used
// in general_fixed and general_persistent.
// See comments in TREE_PLUS version
// to understand how to create custom ones.
// Capabilities notation: (update; query)
#if TREE_PLUS // (+; sum, max, max count)
  // time: O(lg n)
  using T = int; // Data type for update
                 // operations (lazy tag)
  static constexpr T ID = 0; // Neutral value
                 // for updates and lazy tags
  // This structure keeps aggregated data
  struct Agg {
    // Aggregated data: sum, max, max count
    // Default values should be neutral
    // values, i.e. "aggregate over empty set"
    T sum = 0, vMax = INT_MIN, nMax = 0;
    int cnt = 0; // And node count.
    // Initialize as leaf (single value)
    void leaf() { sum=vMax=0; nMax=cnt=1; }
    // Combine data with aggregated data
    // from node to the right
    void merge(const Agg& r) {
      if (vMax < r.vMax) nMax = r.nMax;
      else if (vMax == r.vMax) nMax += r.nMax;
      vMax = max(vMax, r.vMax);
      sum += r.sum;
      cnt += r.cnt;
    }
    // Apply update provided in `x`:
    // - update aggregated data and `lazy` tag
    // - return 0 if update should branch
    //   (can be used in "segment tree beats")
    // - if you change value of `x` it will be
    //   passed to next node to the right
    //   during updates
    bool apply(T& lazy, T& x) {
      lazy += x;
      sum += x*cnt;
      vMax += x;
      return 1;
    }
  };
#elif TREE_MAX // (max; max, max count)
  // time: O(lg n)
  using T = int;
  static constexpr T ID = INT_MIN;
  struct Agg {
    // Aggregated data: max value, max count
    T vMax = INT_MIN, nMax = 0, cnt = 0;
    void leaf() { vMax = 0; nMax = cnt = 1; }
    void merge(const Agg& r) {
      if (vMax < r.vMax) nMax = r.nMax;
      else if (vMax == r.vMax) nMax += r.nMax;
      vMax = max(vMax, r.vMax);
      cnt += r.cnt;
    }
    bool apply(T& lazy, T& x) {
      if (vMax <= x) nMax = cnt;
      lazy = max(lazy, x);
      vMax = max(vMax, x);
      return 1;
    }
  };
#elif TREE_SET // (=; sum, max, max count)
  // time: O(lg n)
  // Set ID to some unused value.
  using T = int;
  static constexpr T ID = INT_MIN;
```

```
  struct Agg {
    // Aggregated data: sum, max, max count
    T sum = 0, vMax = INT_MIN, nMax = 0, cnt=0;
    void leaf() { sum=vMax=0; nMax=cnt=1; }
    void merge(const Agg& r) {
      if (vMax < r.vMax) nMax = r.nMax;
      else if (vMax == r.vMax) nMax += r.nMax;
      vMax = max(vMax, r.vMax);
      sum += r.sum;
      cnt += r.cnt;
    }
    bool apply(T& lazy, T& x) {
      if (x != ID) {
        lazy = x;
        sum = x*cnt;
        vMax = x;
        nMax = cnt;
      }
      return 1;
    }
  };
#elif TREE_BEATS // (+, min; sum, max)
  // time: amortized O(lg n) if not using +
  //       amortized O(lg^2 n) if using +
  // Lazy tag is pair (add, min).
  // To add x: run update with {x, INT_MAX},
  // to min x: run update with {0, x}.
  // If both parts are provided, addition
  // is applied first, then minimum.
  using T = pii;
  static constexpr T ID = {0, INT_MAX};
  struct Agg {
    // Aggregated data: max value, max count,
    //                  second max value, sum
    int vMax = INT_MIN, nMax = 0;
    int max2 = INT_MIN, sum = 0, cnt = 0;
    void leaf() { sum=vMax=0; nMax=cnt=1; }
    void merge(const Agg& r) {
      if (r.vMax > vMax) {
        max2 = vMax;
        vMax = r.vMax;
        nMax = r.nMax;
      } else if (r.vMax == vMax) {
        nMax += r.nMax;
      } else if (r.vMax > max2) {
        max2 = r.vMax;
      }
      max2 = max(max2, r.max2);
      sum += r.sum;
      cnt += r.cnt;
    }
    bool apply(T& lazy, T& x) {
      if (max2 != INT_MIN && max2+x.x >= x.y)
        return 0;
      lazy.x += x.x;
      sum += x.x*cnt;
      vMax += x.x;
      if (max2 != INT_MIN) max2 += x.x;
      if (x.y < vMax) {
        sum -= (vMax-x.y) * nMax;
        vMax = x.y;
      }
      lazy.y = vMax;
      return 1;
    }
  };
#endif
```

SGTPotepaGeneral.h

Description: Segment tree general lazy. 83feaa, 71 lines

```
// Highly configurable statically allocated
// interval-interval segment tree; space: O(n)
struct SegTree {
  // Choose/write configuration
  //#include "general_config.h"
  // Root node is 1, left is i*2, right i*2+1
  vector<Agg> agg; // Aggregated data for nodes
  vector<T> lazy;  // Lazy tags for nodes
  int len = 1;     // Number of leaves
  // Initialize tree for n elements; time: O(n)
  SegTree(int n = 0) {
    while (len < n) len *= 2;
    agg.resize(len*2);
    lazy.resize(len*2, ID);
    rep(i, n) agg[len+i].leaf();
    for (int i = len; --i;) pull(i);
  }
  void pull(int i) {
    (agg[i] = agg[i*2]).merge(agg[i*2+1]);
  }
  void push(int i) {
    rep(c, 2)
      agg[i*2+c].apply(lazy[i*2+c], lazy[i]);
    lazy[i] = ID;
  }
  template<bool U>
  void go(int vb, int ve, int i, int b, int e,
          auto fn) {
    if (vb < e && b < ve)
      if (b < vb || ve < e || !fn(i)) {
        int m = (b+e) / 2;
        push(i);
        go<U>(vb, ve, i*2, b, m, fn);
        go<U>(vb, ve, i*2+1, m, e, fn);
        if (U) pull(i);
      }
  }
  // Modify interval [b;e) with val; O(lg n)
  void update(int b, int e, T val) {
    go<1>(b, e, 1, 0, len, [&](int i) {
      return agg[i].apply(lazy[i], val);
    });
  }
  // Query interval [b;e); time: O(lg n)
  Agg query(int b, int e) {
    Agg t; go<0>(b, e, 1, 0, len, [&](int i) {
      return t.merge(agg[i]), 1;
    });
    return t;
  }
  // Find smallest `j` such that
  // g(aggregate of [0,j)) is true; O(lg n)
  // The predicate `g` must be monotonic.
  // Returns -1 if no such prefix exists.
  int lowerBound(auto g) {
    if (!g(agg[1])) return -1;
    Agg x, s;
    int i = 1;
    for (; i < len; g(s) || (x = s, i++))
      push(i), (s = x).merge(agg[i *= 2]);
    return i - len + !g(x);
  }
};
```

SGTPotepaGeneralPersistent.h

Description: Segment tree lazy persistent. 8c161b, 99 lines

```
// Highly configurable interval-interval
// persistent segment tree; space: O(q lg n)
// First tree version number is 0.
struct SegTree {
  // Choose/write configuration
  //#include "general_config.h"
  vector<Agg> agg{{}}; // Aggregated data
  vector<T> lazy{ID};  // Lazy tags
  vector<bool> cow{0}; // Copy children on push
  vi L{0}, R{0};       // Children links
  int len{1};          // Number of leaves
  // Initialize tree for n elements; O(lg n)
  SegTree(int n = 0) {
    int k = 3;
    while (len < n) len *= 2, k += 3;
    fwd(i, 1, k) fork(0);
    iota(all(R)-3, 3);
    L = R;
    if (n--) {
      agg[k -= 3].leaf();
      agg[k+1].leaf();
      for (int i = k-3; i >= 0; i -= 3, n /= 2)
        (n%2 ? L[i] : ++R[i])++;
      while (k--) pull(k);
    }
  }
  // New version from version `i`; time: O(1)
  int fork(int i) {
    L.pb(L[i]); R.pb(R[i]); cow.pb(cow[i] = 1);
    agg.pb(agg[i]); lazy.pb(lazy[i]);
    return sz(L)-1;
  }
  void pull(int i) {
    (agg[i] = agg[L[i]]).merge(agg[R[i]]);
  }
  void push(int i, bool w) {
    if (w || lazy[i] != ID) {
      if (cow[i]) {
        int x = fork(L[i]), y = fork(R[i]);
        L[i] = x; R[i] = y; cow[i] = 0;
      }
      agg[L[i]].apply(lazy[L[i]], lazy[i]);
      agg[R[i]].apply(lazy[R[i]], lazy[i]);
      lazy[i] = ID;
    }
  }
  template<bool U>
  void go(int vb, int ve, int i, int b, int e,
          auto fn) {
    if (vb < e && b < ve)
      if (b < vb || ve < e || !fn(i)) {
        int m = (b+e) / 2;
        push(i, U);
        go<U>(vb, ve, L[i], b, m, fn);
        go<U>(vb, ve, R[i], m, e, fn);
        if (U) pull(i);
      }
  }
  // Modify interval [b;e) with val
  // in tree version `j`; time: O(lg n)
  void update(int j, int b, int e, T val) {
    go<1>(b, e, j, 0, len, [&](int i) {
      return agg[i].apply(lazy[i], val);
    });
  }
  // Query interval [b;e) in tree version `j`;
  Agg query(int j, int b, int e) { // O(lg n)
    Agg t; go<0>(b, e, j, 0, len, [&](int i) {
      return t.merge(agg[i]), 1;
    });
    return t;
  }
  // Find smallest `j` such that
  // g(aggregate of [0,j)) is true
  // in tree version `i`; time: O(lg n)
  // The predicate `g` must be monotonic.
  // Returns -1 if no such prefix exists.
  int lowerBound(int i, auto g) {
    if (!g(agg[i])) return -1;
    Agg x, s;
    int p = 0, k = len;
    while (L[i]) {
      push(i, 0);
      (s = x).merge(agg[L[i]]);
      k /= 2;
      i = g(s) ? L[i] : (x = s, p += k, R[i]);
    }
    return p + !g(x);
  }
};
```

SGTPotepaPoint.h

Description: Segment tree point. 4b61a1, 52 lines

```
// Point-interval segment tree
// - T - stored data type
// - ID - neutral element for query operation
// - f(a, b) - associative aggregate function
struct SegTree {
  using T = int;
  static constexpr T ID = INT_MIN;
  T f(T a, T b) { return max(a, b); }
  #endif //!HIDE
  vector<T> V;
  int len = 1;
  // Initialize tree for n elements; time: O(n)
  SegTree(int n = 0, T def = 0) {
    while (len < n) len *= 2;
    V.resize(len+n, def);
    V.resize(len*2, ID);
    for (int i = len; --i;)
      V[i] = f(V[i*2], V[i*2+1]);
  }
  // Set element `i` to `val`; time: O(lg n)
  void set(int i, T val) {
    V[i += len] = val;
    while (i /= 2) V[i] = f(V[i*2], V[i*2+1]);
  }
  // Query interval [b;e); time: O(lg n)
  T query(int b, int e) {
    T x = ID, y = ID;
    b += len;
    for (e += len; b < e; b /= 2, e /= 2) {
      if (b % 2) x = f(x, V[b++]);
      if (e % 2) y = f(V[--e], y);
    }
    return f(x, y);
  }
  // Find smallest `j` such that
  // g(aggregate of [0,j)) is true; O(lg n)
  // The predicate `g` must be monotonic.
  // Returns -1 if no such prefix exists.
  int lowerBound(auto g) {
    if (!g(V[1])) return -1;
    T s, x = ID;
    int j = 1;
    while (j < len)
      if (!g(s = f(x, V[j *= 2]))) x = s, j++;
    return j - len + !g(x);
  }
};
```

SGTPotepaPointPersistent.h

Description: Segment tree point persistent. f3296c, 88 lines

```
// Point-interval persistent segment tree
// - T - stored data type
// - ID - neutral element for query operation
// - f(a, b) - associative aggregate function
// First tree version number is 0.
struct SegTree {
  using T = int;
  static constexpr T ID = INT_MIN;
  T f(T a, T b) { return max(a, b); }
  #endif //!HIDE
  vector<T> agg{ID};   // Aggregated data
  vector<bool> cow{1}; // Copy children on push
  vi L{0}, R{0};       // Children links
  int len{1};          // Number of leaves
  // Initialize tree for n elements; O(lg n)
  SegTree(int n = 0, T def = 0) {
    int k = 3;
    while (len < n) len *= 2, k += 3;
    fwd(i, 1, k) fork(0);
    iota(all(R)-3, 3);
    L = R;
    if (n--) {
      k -= 3;
      agg[k] = agg[k+1] = def;
      for (int i = k-3; i >= 0; i -= 3, n /= 2)
        (n%2 ? L[i] : ++R[i])++;
      while (k--)
        agg[k] = f(agg[L[k]], agg[R[k]]);
    }
  }
  // New version from version `i`; time: O(1)
  int fork(int i) {
    L.pb(L[i]); R.pb(R[i]);
    agg.pb(agg[i]); cow.pb(cow[i] = 1);
    return sz(L)-1;
  }
  // Set element `pos` to `val` in version `i`;
  // time: O(lg n)
  void set(int i, int pos, T val,
           int b = 0, int e = 0) {
    if (L[i]) {
      if (!e) e = len;
      if (cow[i]) {
        int x = fork(L[i]), y = fork(R[i]);
        L[i] = x; R[i] = y; cow[i] = 0;
      }
      int m = (b+e) / 2;
      if (pos < m) set(L[i], pos, val, b, m);
      else set(R[i], pos, val, m, e);
      agg[i] = f(agg[L[i]], agg[R[i]]);
    } else {
      agg[i] = val;
    }
  }
  // Query interval [b;e) in tree version `i`;
  // time: O(lg n)
  T query(int i, int vb, int ve,
          int b = 0, int e = 0) {
    if (!e) e = len;
    if (vb >= e || b >= ve) return ID;
    if (b >= vb && e <= ve) return agg[i];
    int m = (b+e) / 2;
    return f(query(L[i], vb, ve, b, m),
             query(R[i], vb, ve, m, e));
  }
  // Find smallest `j` such that
  // g(aggregate of [0,j)) is true
  // in tree version `i`; time: O(lg n)
  // The predicate `g` must be monotonic.
  // Returns -1 if no such prefix exists.
  int lowerBound(int i, auto g) {
    if (!g(agg[i])) return -1;
    T x = ID;
    int p = 0, k = len;
    while (L[i]) {
      T s = f(x, agg[L[i]]);
      k /= 2;
      i = g(s) ? L[i] : (x = s, p += k, R[i]);
    }
    return p + !g(x);
```

```
  }
};
```

Numerical (4)

4.1 Polynomials and recurrences

Polynomial.h

5307ee, 17 lines

```
struct Poly {
  vector<double> a;
  double operator()(double x) const {
    double val = 0;
    for (int i = sz(a); i--;) (val *= x) += a[i];
    return val;
  }
  void diff() {
    fwd(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" f2255f, 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.pb(xmin-1);
  dr.pb(xmax+1);
  sort(all(dr));
  rep(i,sz(dr)-1) {
    double l = dr[i], h = dr[i+1];
    bool sign = p(l) > 0;
    if (sign ^ (p(h) > 0)) {
      fwd(it,0,60) { // while (h - l > 1e-8)
        double m = (l + h) / 2, f = p(m);
        if ((f <= 0) ^ sign) l = m;
        else h = m;
      }
      ret.pb((l + h) / 2);
    }
  }
  return ret;
}
```

PolyInterpolate.h

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

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

8dba48, 33 lines

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

BerlekampMassey.h

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

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

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

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

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

LinearRecurrence.h

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

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

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

1868dd, 26 lines

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

PolynomialPotepa.h

Description: Poynomials. Implement Zp, or modify to use ll modulo mod.

Time: see below

7ee1c7, 271 lines

```
using Poly = vector<Zp>;
// Cut off trailing zeroes; time: O(n)
void norm(Poly &P) {
  while (!P.empty() && !P.back().x)
    P.pop_back();
}
// Evaluate polynomial at x; time: O(n)
Zp eval(const Poly &P, Zp x) {
  Zp n = 0, y = 1;
  each(a, P) n += a * y, y *= x;
  return n;
}
// Add polynomial; time: O(n)
Poly &operator+=(Poly &l, const Poly &r) {
  l.resize(max(sz(l), sz(r)));
  rep(i, sz(r)) l[i] += r[i];
  norm(l);
  return l;
}
Poly operator+(Poly l, const Poly &r) { return l += r;
     }
// Subtract polynomial; time: O(n)
Poly &operator-=(Poly &l, const Poly &r) {
  l.resize(max(sz(l), sz(r)));
  rep(i, sz(r)) l[i] -= r[i];
  norm(l);
  return l;
}
Poly operator-(Poly l, const Poly &r) { return l -= r;
     }
// Multiply by polynomial; time: O(n lg n)
Poly &operator*=(Poly &l, const Poly &r) {
  if (min(sz(l), sz(r)) < 50) {
    // Naive multiplication
    Poly p(sz(l) + sz(r));
    rep(i, sz(l)) rep(j, sz(r)) p[i + j] += l[i] * r[j
         ];
    l.swap(p);
  } else {
    // FFT multiplication
  }
  norm(l);
  return l;
}
Poly operator*(Poly l, const Poly &r) { return l *= r;
     }
// Compute inverse series mod x^n; O(n lg n) Requires P
     (0) != 0.
Poly invert(const Poly &P, int n) {
  assert(!P.empty() && P[0].x);
  Poly tmp{P[0]}, ret = {P[0].inv()};
  for (int i = 1; i < n; i *= 2) {
    fwd(j, i, min(i * 2, sz(P))) tmp.pb(P[j]);
    (ret *= Poly{2} - tmp * ret).resize(i * 2);
  }
  ret.resize(n);
  return ret;
}
// Floor division by polynomial; O(n lg n)
Poly &operator/=(Poly &l, Poly r) {
  norm(l);
  norm(r);
  int d = sz(l) - sz(r) + 1;
  if (d <= 0)
    return l.clear(), l;
  reverse(all(l));
  reverse(all(r));
  l.resize(d);
  l *= invert(r, d);
  l.resize(d);
  reverse(all(l));
  return l;
}
Poly operator/(Poly l, const Poly &r) { return l /= r;
     }
// Remainder modulo a polynomial; O(n lg n)
Poly operator%(const Poly &l, const Poly &r) { return l
     - r * (l / r); }
Poly &operator%=(Poly &l, const Poly &r) { return l -=
     r * (l / r); }
// Compute a^e mod x^n, where a is polynomial;
// time: O(n log n log e)
Poly pow(Poly a, ll e, int n) {
  Poly t = {1};
  while (e) {
    if (e % 2)
      (t *= a).resize(n);
    e /= 2;
    (a *= a).resize(n);
  }
  norm(t);
  return t;
}
// Compute a^e mod m, where a and m are
// polynomials; time: O(|m| log |m| log e)
Poly pow(Poly a, ll e, const Poly &m) {
  Poly t = {1};
  while (e) {
    if (e % 2)
      t = t * a % m;
    e /= 2;
    a = a * a % m;
  }
  return t;
}
// Derivate polynomial; time: O(n)
Poly derivate(Poly P) {
  if (!P.empty()) {
    fwd(i, 1, sz(P)) P[i - 1] = P[i] * i;
    P.pop_back();
  }
  return P;
}
// Integrate polynomial; time: O(n)
Poly integrate(Poly P) {
  if (!P.empty()) {
    P.pb(0);
    for (int i = sz(P); --i;)
      P[i] = P[i - 1] / i;
    P[0] = 0;
  }
  return P;
}
// Compute ln(P) mod x^n; time: O(n log n)
Poly log(const Poly &P, int n) {
  Poly a = integrate(derivate(P) * invert(P, n));
  a.resize(n);
  return a;
}
// Compute exp(P) mod x^n; time: O(n lg n)  Requires P
     (0) = 0.
Poly exp(Poly P, int n) {
  assert(P.empty() || !P[0].x);
  Poly tmp{P[0] + 1}, ret = {1};
  for (int i = 1; i < n; i *= 2) {
    fwd(j, i, min(i * 2, sz(P))) tmp.pb(P[j]);
    (ret *= (tmp - log(ret, i * 2))).resize(i * 2);
  }
  ret.resize(n);
  return ret;
}
// Compute sqrt(P) mod x^n; Requiers ModSqrt.h time: O(
     n log n)
bool sqrt(Poly &P, int n) {
  norm(P);
  if (P.empty())
    return P.resize(n), 1;
  int tail = 0;
  while (!P[tail].x)
    tail++;
  if (tail % 2)
    return 0;
  ll sq = modSqrt(P[tail].x, MOD);
  if (sq == -1)
    return 0;
  Poly tmp{P[tail]}, ret = {sq};
  for (int i = 1; i < n - tail / 2; i *= 2) {
    fwd(j, i, min(i * 2, sz(P) - tail)) tmp.pb(P[tail +
         j]);
    (ret += tmp * invert(ret, i * 2)).resize(i * 2);
    each(e, ret) e /= 2;
  }
  P.resize(tail / 2);
  P.insert(P.end(), all(ret));
  P.resize(n);
  return 1;
}
// Compute polynomial P(x+c); time: O(n lg n)
Poly shift(Poly P, Zp c) {
  int n = sz(P);
  Poly Q(n, 1);
  Zp fac = 1;
  fwd(i, 1, n) {
    P[i] *= (fac *= i);
    Q[n - i - 1] = Q[n - i] * c / i;
  }
  P *= Q;
  if (sz(P) < n)
    return {};
  P.erase(P.begin(), P.begin() + n - 1);
  fac = 1;
  fwd(i, 1, n) P[i] /= (fac *= i);
  return P;
}
// Compute values P(x^0), ..., P(x^{n-1}); time: O(n lg
     n)
Poly chirpz(Poly P, Zp x, int n) {
  int k = sz(P);
  Poly Q(n + k);
  rep(i, n + k) Q[i] = x.pow(i * (i - 1) / 2);
  rep(i, k) P[i] /= Q[i];
  reverse(all(P));
  P *= Q;
  rep(i, n) P[i] = P[k + i - 1] / Q[i];
  P.resize(n);
  return P;
}
```

```
// Evaluate polynomial P in given points; time: O(n lg^2 n)
Poly eval(const Poly &P, Poly points) {
  int len = 1;
  while (len < sz(points))
    len *= 2;
  vector<Poly> tree(len * 2, {1});
  rep(i, sz(points)) tree[len + i] = {-points[i], 1};
  for (int i = len; --i;)
    tree[i] = tree[i * 2] * tree[i * 2 + 1];
  tree[0] = P;
  fwd(i, 1, len * 2) tree[i] = tree[i / 2] % tree[i];
  rep(i, sz(points)) {
    auto &vec = tree[len + i];
    points[i] = vec.empty() ? 0 : vec[0];
  }
  return points;
}
// Given n points (x, f(x)) compute n-1-degree polynomial f that
// passes through them; time: O(n lg^2 n)
Poly interpolate(const vector<pair<Zp, Zp>> &P) {
  int len = 1;
  while (len < sz(P))
    len *= 2;
  vector<Poly> mult(len * 2, {1}), tree(len * 2);
  rep(i, sz(P)) mult[len + i] = {-P[i].x, 1};
  for (int i = len; --i;)
    mult[i] = mult[i * 2] * mult[i * 2 + 1];
  tree[0] = derivate(mult[1]);
  fwd(i, 1, len * 2) tree[i] = tree[i / 2] % mult[i];
  rep(i, sz(P)) tree[len + i][0] = P[i].y / tree[len + i][0];
  for (int i = len; --i;)
    tree[i] = tree[i * 2] * mult[i * 2 + 1] + mult[i * 2] * tree[i * 2 + 1];
  return tree[1];
}
// Count number of possible subsets that sum
// to t for each t = 1, ..., n; O(n log n)
// Input elements are given by frequency array,
// i.e. counts[x] = how many times elements x
// is contained in the multiset.
// Requires counts[0] == 0.
//! Source: https://arxiv.org/pdf/1807.11597.pdf
Poly subsetSum(Poly counts, int n) {
  assert(counts[0].x == 0);
  Poly mul(n);
  rep(i, n)
    mul[i] = Zp(i).inv() * (i%2 ? 1 : -1);
  counts.resize(n);
  for (int i = n-2; i > 0; i--)
    for (int j = 2; i*j < n; j++)
      counts[i*j] += mul[j] * counts[i];
  return exp(counts, n);
}
```

PolyInterpolateFast.h

Description: Compute k-th term of an n-order linear recurrence C[i] = sum C[i-j-1]*D[j], given C[0..n-1] and D[0..n-1];
Time: $\mathcal{O}(n \log n \log k)$

"PolynomialPotepa.h" 2c1a5a, 8 lines

```
Zp linearRec(const Poly &C, const Poly &D, ll k) {
  Poly f(sz(D) + 1, 1);
  rep(i, sz(D)) f[i] = -D[sz(D) - i - 1];
  f = pow({0, 1}, k, f);
  Zp ret = 0;
  rep(i, sz(f)) ret += f[i] * C[i];
  return ret;
}
```

4.2 Optimization

GoldenSectionSearch.h

Description: Finds the argument minimizing the function f in the interval $[a, b]$ assuming f is unimodal on the interval, i.e. has only one local minimum. The maximum error in the result is eps. Works equally well for maximization with a small change in the code. See TernarySearch.h in the Various chapter for a discrete version.
Usage: `double func(double x) { return 4+x+.3*x*x; }`
`double xmin = gss(-1000,1000,func);`
Time: $\mathcal{O}(\log((b-a)/\epsilon))$ 31d45b, 14 lines

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

HillClimbing.h

Description: Poor man's optimization for unimodal functions. a6260e, 14 lines

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

Integrate.h

Description: Simple integration of a function over an interval using Simpson's rule. The error should be proportional to h^4, although in practice you will want to verify that the result is stable to desired precision when epsilon changes.
Time: $\mathcal{O}(n * \text{eval}(f))$ 0b1353, 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);
  fwd(i,1,n*2)
    v += f(a + i*h) * (i&1 ? 4 : 2);
  return v * h / 3;
}
```

IntegrateAdaptive.h

Description: Fast integration using an adaptive Simpson's rule.
Usage: `double sphereVolume = quad(-1, 1, [](double x) {`
`return quad(-1, 1, [&](double y) {`
`return quad(-1, 1, [&](double z) {`
`return x*x + y*y + z*z < 1; });});});` 92dd79, 15 lines

```
typedef double d;
#define S(a,b) (f(a) + 4*f((a+b) / 2) + f(b)) * (b-a) / 6
template <class F>
d rec(F& f, d a, d b, d eps, d S) {
  d c = (a + b) / 2;
  d S1 = S(a, c), S2 = S(c, b), T = S1 + S2;
  if (abs(T - S) <= 15 * eps || b - a < 1e-10)
    return T + (T - S) / 15;
  return rec(f, a, c, eps / 2, S1) + rec(f, c, b, eps / 2, S2);
}
template<class F>
d quad(d a, d b, F f, d eps = 1e-8) {
  return rec(f, a, b, eps, S(a, b));
}
```

Simplex.h

Description: Solves a general linear maximization problem: maximize $c^T x$ subject to $Ax \le b$, $x \ge 0$. Returns -inf if there is no solution, inf if there are arbitrarily good solutions, or the maximum value of $c^T x$ otherwise. The input vector is set to an optimal x (or in the unbounded case, an arbitrary solution fulfilling the constraints). Numerical stability is not guaranteed. For better performance, define variables such that $x = 0$ is viable.
Usage: `vvd A = {{1,-1}, {-1,1}, {-1,-2}};`
`vd b = {1,1,-4}, c = {-1,-1}, x;`
`T val = LPSolver(A, b, c).solve(x);`
Time: $\mathcal{O}(NM * \#pivots)$, where a pivot may be e.g. an edge relaxation. $\mathcal{O}(2^n)$ in the general case. 6210e7, 70 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,m) rep(j,n) D[i][j] = A[i][j];
      rep(i,m) { B[i] = n+i; D[i][n] = -1; D[i][n+1] = b[i];}
      rep(j,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,m+2) if (i != r && abs(D[i][s]) > eps) {
      T *b = D[i].data(), inv2 = b[s] * inv;
      rep(j,n+2) b[j] -= a[j] * inv2;
      b[s] = a[s] * inv2;
    }
    rep(j,n+2) if (j != s) D[r][j] *= inv;
    rep(i,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,n+1) if (N[j] != -phase) ltj(D[x]);
      if (D[x][s] >= -eps) return true;
      int r = -1;
      rep(i,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;
    fwd(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,m) if (B[i] == -1) {
        int s = 0;
        fwd(j,1,n+1) ltj(D[i]);
        pivot(i, s);
      }
    }
    bool ok = simplex(1); x = vd(n);
    rep(i,m) if (B[i] < n) x[B[i]] = D[i][n+1];
    return ok ? D[m][n+1] : inf;
  }
};
```

We want to minimize/maximize $f(\vec{x})$ subject to $g_i(\vec{x}) = 0$ for $i = 1, \ldots, k$. Form $f_\lambda(\vec{x}, \vec{\lambda}) = f(\vec{x}) - \sum_i \lambda_i g_i(\vec{x})$. Conditional extremums of f are extremal points of f_λ - points where its gradient is zero.

4.3 Matrices

Determinant.h

Description: Calculates determinant of a matrix. Destroys the matrix.
Time: $\mathcal{O}\left(N^3\right)$ 2e57b8, 15 lines

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

IntDeterminant.h

Description: Calculates determinant using modular arithmetics. Modulos can also be removed to get a pure-integer version.
Time: $\mathcal{O}\left(N^3\right)$ 139eec, 18 lines

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

SolveLinear.h

Description: Solves $A * x = b$. If there are multiple solutions, an arbitrary one is returned. Returns rank, or -1 if no solutions. Data in A and b is lost.
Time: $\mathcal{O}\left(n^2 m\right)$ 11d015, 38 lines

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

SolveLinear2.h

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

"SolveLinear.h" acb9c0, 7 lines

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

SolveLinearBinary.h

Description: Solves $Ax = b$ over $\mathbb{F}_2$. If there are multiple solutions, one is returned arbitrarily. Returns rank, or -1 if no solutions. Destroys A and b.
Time: $\mathcal{O}\left(n^2 m\right)$ d99ddb, 34 lines

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

MatrixInverse.h

Description: Invert matrix A. Returns rank; result is stored in A unless singular (rank < n). Can easily be extended to prime moduli; for prime powers, repeatedly set $A^{-1} = A^{-1}(2I - AA^{-1}) \pmod{p^k}$ where A^{-1} starts as the inverse of A mod p, and k is doubled in each step.
Time: $\mathcal{O}\left(n^3\right)$ 731fcb, 35 lines

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

Tridiagonal.h

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

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

This is useful for solving problems on the type

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

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

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

Fails if the solution is not unique.
If $|d_i| > |p_i| + |q_{i-1}|$ for all i, or $|d_i| > |p_{i-1}| + |q_i|$, or the matrix is positive definite, the algorithm is numerically stable and neither `tr` nor the check for `diag[i] == 0` is needed.
Time: $\mathcal{O}(N)$ 059430, 26 lines

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

4.4 Fourier transforms

FastFourierTransform.h

Description: 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/FFT-Mod.
Time: $\mathcal{O}(N \log N)$ with $N = |A| + |B|$ ($\sim$1s for $N = 2^{22}$) 28ed35, 35 lines

```
typedef complex<double> C;
typedef vector<double> vd;
void fft(vector<C>& a) {
  int n = sz(a), L = 31 - __builtin_clz(n);
  static vector<complex<long double>> R(2, 1);
  static vector<C> rt(2, 1);  // (^ 10% faster if
      double)
  for (static int k = 2; k < n; k *= 2) {
    R.resize(n); rt.resize(n);
    auto x = polar(1.0L, acos(-1.0L) / k);
    fwd(i,k,2*k) rt[i] = R[i] = i&1 ? R[i/2] * x : R[i
        /2];
  }
  vi rev(n);
  rep(i,n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
  rep(i,n) if (i < rev[i]) swap(a[i], a[rev[i]]);
  for (int k = 1; k < n; k *= 2)
    for (int i = 0; i < n; i += 2 * k) rep(j,k) {
      C z = rt[j+k] * a[i+j+k]; // (25% faster if hand-
          rolled)
      a[i + j + k] = a[i + j] - z;
      a[i + j] += z;
    }
}
vd conv(const vd& a, const vd& b) {
  if (a.empty() || b.empty()) return {};
  vd res(sz(a) + sz(b) - 1);
  int L = 32 - __builtin_clz(sz(res)), n = 1 << L;
  vector<C> in(n), out(n);
  copy(all(a), begin(in));
  rep(i,sz(b)) in[i].imag(b[i]);
  fft(in);
  for (C& x : in) x *= x;
  rep(i,n) out[i] = in[-i & (n - 1)] - conj(in[i]);
  fft(out);
  rep(i,sz(res)) res[i] = imag(out[i]) / (4 * n);
  return res;
}
```

FastFourierTransformMod.h

Description: Higher precision FFT, can be used for convolutions modulo arbitrary integers as long as $N \log_2 N \cdot \text{mod} < 8.6 \cdot 10^{14}$ (in practice 10^{16} or higher). Inputs must be in $[0, \text{mod})$.
Time: $\mathcal{O}(N \log N)$, where $N = |A| + |B|$ (twice as slow as NTT or FFT)
"FastFourierTransform.h" ff1e1b, 22 lines

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

NumberTheoreticTransform.h

Description: ntt(a) computes $\hat f(k) = \sum_x a[x] g^{xk}$ for all k, where $g = \text{root}^{(mod-1)/N}$. N must be a power of 2. Useful for convolution modulo specific nice primes of the form $2^a b + 1$, where the convolution result has size at most 2^a. For arbitrary modulo, see FFTMod. `conv(a, b) = c`, where $c[x] = \sum a[i]b[x-i]$. For manual convolution: NTT the inputs, multiply pointwise, divide by n, reverse(start+1, end), NTT back. Inputs must be in [0, mod).
Time: $\mathcal{O}(N \log N)$
"../number-theory/ModPow.h" 2e0a0e, 34 lines

```
const ll mod = (119 << 23) + 1, root = 62; // =
    998244353
// For p < 2^30 there is also e.g. 5 << 25, 7 << 26,
    479 << 21
// and 483 << 21 (same root). The last two are > 10^9.
// int128: (2147483641LL<<32) - but 2xll & crt is
    faster.
typedef vector<ll> vl;
void ntt(vl &a) {
  int n = sz(a), L = 31 - __builtin_clz(n);
  static vl rt(2, 1);
  for (static int k = 2, s = 2; k < n; k *= 2, s++) {
    rt.resize(n);
    ll z[] = {1, modpow(root, mod >> s)};
    fwd(i,k,2*k) rt[i] = rt[i / 2] * z[i & 1] % mod;
  }
  vi rev(n);
  rep(i,n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
  rep(i,n) if (i < rev[i]) swap(a[i], a[rev[i]]);
  for (int k = 1; k < n; k *= 2)
    for (int i = 0; i < n; i += 2 * k) rep(j,k) {
      ll z = rt[j + k] * a[i + j + k] % mod, &ai = a[i
          + j];
      a[i + j + k] = ai - z + (z > ai ? mod : 0);
      ai += (ai + z >= mod ? z - mod : z);
    }
}
vl conv(const vl &a, const vl &b) {
  if (a.empty() || b.empty()) return {};
  int s = sz(a) + sz(b) - 1, B = 32 - __builtin_clz(s),
       n = 1 << B;
  ll inv = modpow(n, mod - 2);
  vl L(a), R(b), out(n);
  L.resize(n), R.resize(n);
  ntt(L), ntt(R);
  rep(i,n) out[-i & (n - 1)] = (ll)L[i] * R[i] % mod *
      inv % mod;
  ntt(out);
  return {out.begin(), out.begin() + s};
}
```

FastSubsetTransform.h

Description: Transform to a basis with fast convolutions of the form $c[z] = \sum_{z=x\oplus y} a[x] \cdot b[y]$, where $\oplus$ is one of AND, OR, XOR. The size of a must be a power of two.
Time: $\mathcal{O}(N \log N)$ 790905, 16 lines

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

Number theory (5)

5.1 Modular arithmetic

ModInverse.h

Description: Pre-computation of modular inverses. Assumes LIM $\leq$ mod and that mod is a prime. 279cb5, 3 lines

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

ModPow.h

b83e45, 8 lines

```
const ll mod = 1000000007; // faster if const
ll modpow(ll b, ll e) {
  ll ans = 1;
  for (; e; b = b * b % mod, e /= 2)
    if (e & 1) ans = ans * b % mod;
  return ans;
}
```

ModLog.h

Description: Returns the smallest $x > 0$ s.t. $a^x = b \pmod{m}$, or -1 if no such x exists. modLog(a,1,m) can be used to calculate the order of a.
Time: $\mathcal{O}\left(\sqrt{m}\right)$ e593f3, 11 lines

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

ModSum.h

Description: Sums of mod'ed arithmetic progressions.

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

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

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

ModMulLL.h

Description: Calculate $a \cdot b \bmod c$ (or $a^b \bmod c$) for $0 \leq a, b \leq c \leq 7.2 \cdot 10^{18}$.

Time: $\mathcal{O}(1)$ for `modmul`, $\mathcal{O}(\log b)$ for `modpow` bbbd8f, 11 lines

```
typedef unsigned long long ull;
ull modmul(ull a, ull b, ull M) {
  ll ret = a * b - M * ull(1.L / M * a * b);
  return ret + M * (ret < 0) - M * (ret >= (ll)M);
}
ull modpow(ull b, ull e, ull mod) {
  ull ans = 1;
  for (; e; b = modmul(b, b, mod), e /= 2)
    if (e & 1) ans = modmul(ans, b, mod);
  return ans;
}
```

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, $\mathcal{O}(\log p)$ for most p

"ModPow.h" 19a793, 24 lines

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

5.2 Primality

FastEratosthenes.h

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

Time: LIM=1e9 $\approx$ 1.5s 4ea2fb, 20 lines

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

MillerRabin.h

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

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

"ModMulLL.h" 60dcd1, 11 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 $0 \leq x < \operatorname{lcm}(m, n)$. Assumes $mn < 2^{62}$.

Time: $\log(n)$

"euclid.h" 04d93a, 7 lines

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

phiFunction.h

Description: Inclusive prefix sums of *Euler's* ϕ. For MOD $> 4 \cdot 10^9$, answer will overflow.

Time: $\mathcal{O}\left(n^{2/3}\right)$ cd3370, 27 lines

```
constexpr int MOD = 998244353;
vector<ll> pSum; // [k] = phi sum from 0 to k
void calcPhiSum() {
  pSum.resize(1e7 + 7);
  iota(all(pSum), 0);
  fwd(i, 2, sz(pSum)) {
    if (pSum[i] == i) for (int j = i; j < sz(pSum); j += i) pSum[j] = pSum[j] / i * (i - 1);
    pSum[i] = (pSum[i] + pSum[i-1]) % MOD;
  }
}
ll getPhiSum(ll n) { // phi(0) + ... + phi(n)
  static unordered_map<ll, ll> big;
  if (!sz(pSum))
    calcPhiSum();
  if (n < sz(pSum))
    return pSum[n];
  if (big.count(n))
    return big[n];
  ll ret = (n % 2 ? n % MOD * ((n + 1) / 2 % MOD) : n / 2 % MOD * (n % MOD + 1)) % MOD;
  for (ll s, i = 2; i <= n; i = s + 1) {
    s = n / (n / i);
    ret -= (s - i + 1) % MOD * getPhiSum(n / i) % MOD;
  }
  return big[n] = (ret % MOD + MOD) % MOD;
}
```

Min25.h

Description: Calculates prefsums of multiplicative function at each floor(N/i). keys[id(N/i)]=N/i. Remember about overflows. See example below.

Time: $\mathcal{O}\left(\frac{n^{3/4}}{\log n}\right)$ f4fd1a, 50 lines

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

5.4 Pisano period

$\pi(n)$ is a period of Fibbonacci sequence modulo n.
$\pi(nm) = \pi(n)\pi(m)$ for $n \perp m$, $\pi(p^k) = p^{k-1}\pi(p)$.

$$\pi(p) \begin{cases} = 3 & p = 2 \\ = 20 & p = 5 \\ \mid p - 1 & p \equiv_{10} \pm 1 \\ \mid 2(p+1) & p \equiv_{10} \pm 3 \end{cases}$$

$F_i \equiv_p -F_{i+p+1}$ for $p \equiv_{10} \pm 3$. $\pi(n) \leq 4n$ for $n \neq 2 \cdot 5^r$.

5.5 Fractions

ContinuedFractions.h

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

Time: $\mathcal{O}(\log N)$ d64d49, 17 lines

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

FracBinarySearch.h

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

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

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

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

```
    A = B; B = !!adv;
  }
  return dir ? hi : lo;
}
```

5.6 Pythagorean Triples

The Pythagorean triples are uniquely generated by

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

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

5.7 Primes & primitive roots

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

5.8 Estimates

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

5.9 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)

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

Define Dirichlet convolution as $f * g(n) = \sum_{d|n} f(d) g(n/d)$. Let $s_f(n) = \sum_{i=1}^{n} f(i)$. Then $s_f(n) g(1) = s_{f*g}(n) - \sum_{d=2}^{n} s_f(\lfloor \frac{n}{d} \rfloor) g(d)$.

Combinatorial (6)

6.1 Permutations

6.1.1 Factorial

IntPerm.h

Description: Permutation -> integer conversion. (Not order preserving.) Integer -> permutation can use a lookup table.
Time: $\mathcal{O}(n)$ 044568, 6 lines

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

6.1.2 Cycles

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

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

6.1.3 Derangements

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

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

6.1.4 Burnside's lemma

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

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

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

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

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

6.2 Partitions and subsets

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

6.2.1 Partition function

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

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

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

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 $n = n_k p^k + \ldots + n_1 p + n_0$ and $m = m_k p^k + \ldots + m_1 p + m_0$. Then $\binom{n}{m} \equiv \prod_{i=0}^{k} \binom{n_i}{m_i} \pmod{p}$.

6.2.3 Binomials

multinomial.h

Description: Computes $\binom{k_1 + \cdots + k_n}{k_1, k_2, \ldots, k_n} = \frac{(\sum k_i)!}{k_1! k_2! \ldots k_n!}$. 16290f, 6 lines

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

6.3 General purpose numbers

6.3.1 Bernoulli numbers

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

Sums of powers:

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

Euler-Maclaurin formula for infinite sums:

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

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

6.3.2 Stirling numbers of the first kind

Number of permutations on n items with k cycles.

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

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

6.3.3 Eulerian numbers

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

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

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

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

6.3.4 Stirling numbers of the second kind

Partitions of n distinct elements into exactly k groups.

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

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

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

6.3.5 Bell numbers

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

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

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

6.3.6 Labeled unrooted trees

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

6.3.7 Catalan numbers

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

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

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

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

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

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

6.3.8 LGV Lemma

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

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

Particularly useful when only identity permutation is possible.

6.4 Other

DeBruijn.h

Description: Recursive FKM, given alphabet $[0, k)$ constructs cyclic string of length k^n that contains every length n string as substr. a7faa5, 13 lines

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

```
  };
  gen(1,1); return res;
}
```

NimProduct.h

Description: Nim Product. 9bba25, 17 lines

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

PermGroup.h

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

Time: $\mathcal{O}\left(n^6\right)$ d6edf4, 54 lines

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

```
  }
};
```

GrayCode.h

Description: Gray code: $\mathrm{gray}(0), \ldots, \mathrm{gray}(2^n - 1)$ - permutation in which each two consecutive (cyclically) numbers. differ in exactly one bit. b4cc82, 6 lines

```
using ull = unsigned long long;
ull gray(ull i) { return i^i>>1; }
ull invg(ull i) { // i=invg(gray(i))=gray(invg(i))
  i^=i>>1; i^=i>>2; i^=i>>4;
  i^=i>>8; i^=i>>16; i^=i>>32; return i;
}
```

Graph (7)

7.1 Fundamentals

BellmanFord.h

Description: BF dist(E). unreachable <=> dist[v] = INF. reachable by negative cycle dist[v] = -INF. par[v] = parent in shortest path tree cyc = negative vertices in order. set s=-1 to find any negative cycle set INF and T according to specific needs. optimize the middle loop if you dont need to reconstruct the negative cycle / dont need shortest path tree. set its = n / 3 + 100 if you random shuffle vertex ids beforehand

Time: $\mathcal{O}(VE)$ afe52b, 29 lines

```
using T = ll; const T INF = LLONG_MAX;
struct Edge {
  int v, u; T w; // var_u <= var_v + w
  int f() { return v < u ? v + 1 : -v; }
};
struct BF : vector<T> {
  vi par, cyc;
  BF(vector<Edge> E, int n, int s = -1)
  : vector<T>(n, ~s ? INF : 0), par(n, -1) {
    if (~s) { at(s) = 0; } int k, its = n / 2 + 2;
    sort(all(E), [&] (Edge a, Edge b) {
      return a.f() < b.f();
    });
    rep(i, its) { k = -1;
      for (auto [v, u, w] : E) {
        T d = (abs(at(v)) == INF ? INF : at(v) + w);
        if (d < at(u)) {
          at(u) = (i < its - 1 ? d : -INF);
          par[k = u] = v;
        }
      }
    } // optimize above if no need for negative cycle
    if (k >= 0) { // finds negative simple cycle
      rep(i, its) for (auto [v, u, w] : E)
      if (at(v) == -INF) at(u) = -INF;
      rep(i, n) { k = par[k]; } cyc = {k};
      for (int v = par[k]; v != k; v = par[v])
        cyc.pb(v);
      reverse(all(cyc)); } } };
```

SPFA.h

Description: SPFA with subtree erasure heuristic. Returns array of distances or empty array if negative cycle is reachable from source. par[v] = parent in shortest path tree

Time: $\mathcal{O}(VE)$ but fast on random bf4a3f, 33 lines

```
using Edge = pair<int, ll>;
vector<ll> spfa(vector<vector<Edge>>& G,
                vi& par, int src) {
  int n = sz(G); vi que, prv(n+1);
  iota(all(prv), 0); vi nxt = prv;
  vector<ll> dist(n, INT64_MAX);
  par.assign(n, -1);
  auto add = [&](int v, int p, ll d) {
    par[v] = p; dist[v] = d;
    prv[n] = nxt[prv[v] = prv[nxt[v] = n]] = v;
  };
  auto del = [&](int v) {
    nxt[prv[nxt[v]] = prv[v]] = nxt[v];
    prv[v] = nxt[v] = v;
  };
  for (add(src, -2, 0); nxt[n] != n;) {
    int v = nxt[n]; del(v);
    for (auto e : G[v]) {
      ll alt = dist[v] + e.y;
      if (alt < dist[e.x]) {
        que = {e.x};
        rep(i, sz(que)) {
          int w = que[i]; par[w] = -1;
          del(w);
          for (auto f : G[w])
            if (par[f.x] == w) que.pb(f.x);
        }
        if (par[v] == -1) return {};
        add(e.x, v, alt);
      }
    }
  }
  return dist; }
```

Shapes.h

Description: Counts all subgraph shapes with at most 4 edges. No multiedges / loops allowed;

Time: $\mathcal{O}\left(m\sqrt{m}\right)$ f62c2b, 39 lines

```
struct Shapes {
  ll tri = 0, rect = 0, path3 = 0, path4 = 0, star3 =
       0, p = 0;
  __int128_t y = 0, star4 = 0;
  Shapes(vector<vi> &g) {
    int n = sz(g);
    vector<vi> h(n);
    vector<ll> f(n), c(n), s(n);
    rep(v, n) f[v] = (s[v] = sz(g[v])) * n + v;
    rep(v, n) {
      ll x = 0;
      star3 += s[v] * (s[v] - 1) * (s[v] - 2);
      star4 += __int128_t(s[v] - 1) * s[v] * (s[v] - 2)
           * (s[v] - 3);
      for (auto u : g[v]) {
        path4 += s[u] * x - x; x += s[u] - 1;
        y += (s[v] - 1) * (s[u] - 1) * (s[u] - 2) / 2;
        if (f[u] < f[v]) h[v].pb(u);
      }
    }
    rep(v, n) {
      for (int u : h[v])
        for (int w : g[u]) if (f[v] > f[w])
          rect += c[w] ++;
      for(int u : h[v]) {
        tri += c[u]; c[u] *= -1;
        path3 += (s[v] - 1) * (s[u] - 1);
        for(int w : g[u])
          if (c[w] < 0)
            p += s[v] + s[u] + s[w] - 6, c[w] ++;
          else if (c[w] > 0)
            c[w] --;
      }
    }
    path3 -= 3 * tri;
    y -= 2 * p;
    path4 -= 4 * rect + 2 * p + 3 * tri;
    star3 /= 6;
    star4 /= 24;
  }
};
```

7.2 Network flow

EdmondsKarp.h

Description: Use if too tired of life to rewrite push relabel for the 100th time. 1575a3, 76 lines

```
using flow_t = int;
constexpr flow_t INF = 1e9+10;
// Edmonds-Karp algorithm for finding
// maximum flow in graph; time: O(V*E^2)
struct MaxFlow {
  struct Edge {
    int dst, inv;
    flow_t flow, cap;
  };
  vector<vector<Edge>> G;
  vector<flow_t> add;
  vi prev;
  // Initialize for n vertices
  MaxFlow(int n = 0) : G(n) {}
  // Add new vertex
  int addVert() { G.pb({}); return sz(G)-1; }
  // Add edge from u to v with capacity cap
  // and reverse capacity rcap.
  // Returns edge index in adjacency list of u.
  int addEdge(int u, int v,
              flow_t cap, flow_t rcap = 0) {
    G[u].pb({ v, sz(G[v]), 0, cap });
    G[v].pb({ u, sz(G[u])-1, 0, rcap });
    return sz(G[u])-1;
  }
  // Compute maximum flow from src to dst.
  flow_t maxFlow(int src, int dst) {
    flow_t i, m, f = 0;
    for(auto &v: G) for(auto &e: v) e.flow = 0;
  nxt:
    queue<int> Q;
    Q.push(src);
    prev.assign(sz(G), -1);
    add.assign(sz(G), -1);
    add[src] = INF;
    while (!Q.empty()) {
      m = add[i = Q.front()];
      Q.pop();
      if (i == dst) {
        while (i != src) {
          auto& e = G[i][prev[i]];
          e.flow -= m;
          G[i = e.dst][e.inv].flow += m;
        }
        f += m;
        goto nxt;
      }
      for(auto &e : G[i])
        if (add[e.dst] < 0 && e.flow < e.cap) {
          Q.push(e.dst);
          prev[e.dst] = e.inv;
          add[e.dst] = min(m, e.cap-e.flow);
        }
    }
    return f;
  }
  // Get flow through e-th edge of vertex v
  flow_t getFlow(int v, int e) {
    return G[v][e].flow;
  }
  // Get if v belongs to cut component with src
  bool cutSide(int v) { return add[v] >= 0; }
};
```

PushRelabel.h

Description: Fast.

Time: $\mathcal{O}\left(V^2\sqrt{E}\right)$ a76246, 115 lines

```
using flow_t = int;
// Push-relabel algorithm for maximum flow;
// O(V^2*sqrt(E)), but very fast in practice.
struct MaxFlow {
  struct Edge {
    int to, inv;
    flow_t rem, cap;
  };
  vector<basic_string<Edge>> G;
  vector<flow_t> extra;
  vi hei, arc, prv, nxt, act, bot;
  queue<int> Q;
  int n, high, cut, work;
  // Initialize for k vertices
  MaxFlow(int k = 0) : G(k) {}
  // Add new vertex
  int addVert() { G.pb({}); return sz(G)-1; }
  // Add edge from u to v with capacity cap
  // and reverse capacity rcap.
  // Returns edge index in adjacency list of u.
  int addEdge(int u, int v,
              flow_t cap, flow_t rcap = 0) {
    G[u].pb({ v, sz(G[v]), 0, cap });
    G[v].pb({ u, sz(G[u])-1, 0, rcap });
    return sz(G[u])-1;
  }
  void raise(int v, int h) {
    prv[nxt[prv[v]] = nxt[v]] = prv[v];
    hei[v] = h;
    if (extra[v] > 0) {
      bot[v] = act[h]; act[h] = v;
      high = max(high, h);
    }
    if (h < n) cut = max(cut, h+1);
    nxt[v] = nxt[prv[v] = h += n];
    prv[nxt[nxt[h] = v]] = v;
  }
  void global(int s, int t) {
    hei.assign(n, n*2);
    act.assign(n*2, -1);
    iota(all(prv), 0);
    iota(all(nxt), 0);
    hei[t] = high = cut = work = 0;
    hei[s] = n;
    for (int x : {t, s})
      for (Q.push(x); !Q.empty(); Q.pop()) {
        int v = Q.front();
        for (auto &e : G[v])
          if (hei[e.to] == n*2 &&
              G[e.to][e.inv].rem)
            Q.push(e.to), raise(e.to,hei[v]+1);
```

```
      }
  }
  void push(int v, Edge& e, bool z) {
    auto f = min(extra[v], e.rem);
    if (f > 0) {
      if (z && !extra[e.to]) {
        bot[e.to] = act[hei[e.to]];
        act[hei[e.to]] = e.to;
      }
      e.rem -= f; G[e.to][e.inv].rem += f;
      extra[v] -= f; extra[e.to] += f;
    }
  }
  void discharge(int v) {
    int h = n*2, k = hei[v];
    rep(j, sz(G[v])) {
      auto& e = G[v][arc[v]];
      if (e.rem) {
        if (k == hei[e.to]+1) {
          push(v, e, 1);
          if (extra[v] <= 0) return;
        } else h = min(h, hei[e.to]+1);
      }
      if (++arc[v] >= sz(G[v])) arc[v] = 0;
    }
    if (k < n && nxt[k+n] == prv[k+n]) {
      fwd(j, k, cut) while (nxt[j+n] < n)
        raise(nxt[j+n], n);
      cut = k;
    } else raise(v, h), work++;
  }
  // Compute maximum flow from src to dst
  flow_t maxFlow(int src, int dst) {
    extra.assign(n = sz(G), 0);
    arc.assign(n, 0);
    prv.resize(n*3);
    nxt.resize(n*3);
    bot.resize(n);
    for (auto &v : G) for(auto e : v) e.rem = e.cap;
    for (auto &e : G[src])
      extra[src] = e.cap, push(src, e, 0);
    global(src, dst);
    for (; high; high--)
      while (act[high] != -1) {
        int v = act[high];
        act[high] = bot[v];
        if (v != src && hei[v] == high) {
          discharge(v);
          if (work > 4*n) global(src, dst);
        }
      }
    return extra[dst];
  }
  // Get flow through e-th edge of vertex v
  flow_t getFlow(int v, int e) {
    return G[v][e].cap - G[v][e].rem;
  }
  // Get if v belongs to cut component with src
  bool cutSide(int v) { return hei[v] >= n; }
};
```

FlowDemands.h

Description: Flows with demands. e1c0d0, 52 lines

```
//#include "flow_edmonds_karp.h"
//#include "flow_push_relabel.h" // if you need
// Flow with demands; time: O(maxflow)
struct FlowDemands {
  MaxFlow net;
  vector<vector<flow_t>> demands;
  flow_t total = 0;
  // Initialize for k vertices
  FlowDemands(int k = 0) : net(2) {
    while (k--) addVert();
  }
  // Add new vertex
  int addVert() {
    int v = net.addVert();
    demands.pb({});
    net.addEdge(0, v, 0);
    net.addEdge(v, 1, 0);
    return v-2;
  }
  // Add edge from u to v with demand dem
  // and capacity cap (dem <= flow <= cap).
  // Returns edge index in adjacency list of u.
  int addEdge(int u, int v,
              flow_t dem, flow_t cap) {
    demands[u].pb(dem);
    demands[v].pb(0);
    total += dem;
    net.G[0][v].cap += dem;
    net.G[u+2][1].cap += dem;
    return net.addEdge(u+2, v+2, cap-dem) - 2;
  }
  // Check if there exists a flow with value f
  // for source src and destination dst.
  // For circulation, you can set args to 0.
  bool canFlow(int src, int dst, flow_t f) {
    net.addEdge(dst += 2, src += 2, f);
    f = net.maxFlow(0, 1);
    net.G[src].pop_back();
    net.G[dst].pop_back();
    return f == total;
  }
  // Get flow through e-th edge of vertex v
  flow_t getFlow(int v, int e) {
    return net.getFlow(v+2,e+2)+demands[v][e];
  }
};
```

MinCostKFlowFast.h

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

Time: $\mathcal{O}(INIT + Fn\log n)$ $INIT \leq VE$ and depends on first dist computation 3a395e, 70 lines

```
#include <bits/extc++.h>
struct MCMF {
  const ll INF = 2e18;
  struct edge {
    int from, to, rev; ll cap, cost, flow;
  };
  int N;
  vector<vector<edge>> ed;
  vi seen;
  vector<ll> dist, pi;
  vector<edge*> par;
  MCMF(int N) : N(N), ed(N), seen(N), dist(N), pi(N),
       par(N) {}
  void addEdge(int from, int to, ll cap, ll cost) {
    if (from == to) return;
    ed[from].pb(edge{ from,to,sz(ed[to]),cap,cost,0 });
    ed[to].pb(edge{ to,from,sz(ed[from])-1,0,-cost,0 })
         ;
  }
  void path(int s) {
    fill(all(seen), 0); fill(all(dist), INF);
    dist[s] = 0; ll di;
    __gnu_pbds::priority_queue<pair<ll, int>> q;
    vector<decltype(q)::point_iterator> its(N);
    q.push({0, s});
    while (!q.empty()) {
      s = q.top().second; q.pop();
      seen[s] = 1; di = dist[s] + pi[s];
      for (edge& e : ed[s]) if (!seen[e.to]) {
        ll val = di - pi[e.to] + e.cost;
        if (e.cap - e.flow > 0 && val < dist[e.to]) {
          dist[e.to] = val;
          par[e.to] = &e;
          if (its[e.to] == q.end())
            its[e.to] = q.push({ -dist[e.to], e.to });
          else
            q.modify(its[e.to], { -dist[e.to], e.to });
        }
      }
    }
    rep(i,N) pi[i] = min(pi[i] + dist[i], INF);
  }
  pair<ll, ll> maxflow(int s, int t, ll k = -1) {
    if (k == -1) k = INF;
    ll totflow = 0, totcost = 0;
    while (path(s), seen[t]) {
      ll fl = k - totflow;
      for (edge* x = par[t]; x; x = par[x->from])
        fl = min(fl, x->cap - x->flow);
      totflow += fl;
      for (edge* x = par[t]; x; x = par[x->from]) {
        x->flow += fl;
        ed[x->to][x->rev].flow -= fl;
      }
      if (totflow == k) break;
    }
    rep(i,N) for(edge& e : ed[i]) totcost += e.cost * e
         .flow;
    return {totflow, totcost/2};
  }
  // If some costs can be negative, call this before
       maxflow:
  void setpi(int s) { // (otherwise, leave this out)
    fill(all(pi), INF); pi[s] = 0;
    int it = N, ch = 1; ll v;
    while (ch-- && it--)
      rep(i,N) if (pi[i] != INF)
        for (edge& e : ed[i]) if (e.cap)
          if ((v = pi[i] + e.cost) < pi[e.to])
            pi[e.to] = v, ch = 1;
    assert(it >= 0); // negative cost cycle
  }
};
```

Dinic.h

Description: Flow algorithm with complexity $O(VE\log U)$ where $U = \max|\text{cap}|$. $O(\min(E^{1/2}, V^{2/3})E)$ if $U = 1$; $O(\sqrt{V}E)$ for bipartite matching. fe1a74, 42 lines

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

GlobalMinCut.h

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

Time: $\mathcal{O}(V^3)$ 81c2ad, 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,n) co[i] = {i};
  fwd(ph,1,n) {
    vi w = mat[0];
    size_t s = 0, t = 0;
    fwd(it,0,n-ph) { // O(V^2) -> O(E log V) with prio.
         queue
      w[t] = INT_MIN;
      s = t, t = max_element(all(w)) - w.begin();
      rep(i,n) w[i] += mat[t][i];
    }
    best = min(best, {w[t] - mat[t][t], co[t]});
    co[s].insert(co[s].end(), all(co[t]));
    rep(i,n) mat[s][i] += mat[t][i];
    rep(i,n) mat[i][s] = mat[s][i];
    mat[0][t] = INT_MIN;
  }
  return best;
}
```

GomoryHu.h

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

Time: $\mathcal{O}(V)$ Flow Computations

"Dinic.h" a977b2, 13 lines

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

Flow with demands

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

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

For min flow, replace ∞ with L and find smallest L such that flow is saturated.

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: $\mathcal{O}(\sqrt{V}E)$ a7d791, 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,sz(g)) if(A[a] == 0) cur.pb(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.pb(btoa[b]);
        }
      }
      if (islast) break;
      if (next.empty()) return res;
      for (int a : next) A[a] = lay;
      cur.swap(next);
    }
    rep(a,sz(g))
      res += dfs(a, 0, g, btoa, A, B);
  }
}
```

TurboMatching.h

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

Time: $\mathcal{O}(?)$ 39e36c, 33 lines

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

BoskiMatching.h

Description: Bosek's algorithm for partially online bipartite maximum matching - white vertices (right side) are fixed, black vertices (left) are added one by one. • match[v] = index of black vertex matched to white vertex v or -1 if unmatched • Black vertices are indexed in order they were added, from 0.

Time: $\mathcal{O}\left(E\sqrt{V}\right)$ 962c39, 32 lines

```
struct Matching : vi { // Usage: Matching match(
      num_white);
  vector<vi> adj; vi rank, low, pos, vis, seen; int k{0
      };
  Matching(int n = 0) : vi(n, -1), rank(n) {}
  bool add(vi vec) { //match.add(
      indices_of_white_neighbours);
    adj.pb(move(vec));
    low.pb(0); pos.pb(0); vis.pb(0);
    if (!adj.back().empty()) {
      int i = k;
    nxt:
      seen.clear();
      if (dfs(sz(adj)-1, ++k-i)) return 1;
      for(auto v: seen) for(auto e: adj[v])
        if (rank[e] < 1e9 && vis[at(e)] < k)
          goto nxt;
      for(auto v: seen) for(auto w: adj[v])
        rank[w] = low[v] = 1e9;
    }
    return 0;
  } //returns 1 if matching size increased
  bool dfs(int v, int g) {
    if (vis[v] < k) vis[v] = k, seen.pb(v);
    while (low[v] < g) {
      int e = adj[v][pos[v]];
      if (at(e) != v && low[v] == rank[e]) {
        rank[e]++;
        if (at(e) == -1 || dfs(at(e), rank[e]))
          return at(e) = v, 1;
      } else if (++pos[v] == sz(adj[v])) {
        pos[v] = 0, low[v]++;
      }
    }
    return 0; } };
```

WeightedMatching.h

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

Time: $\mathcal{O}\left(N^2M\right)$ deee37, 31 lines

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

GeneralMatching.h

Description: Matching for general graphs using Blossom algorithm.

Time: $\mathcal{O}(NM, fastinpractice)$ 46c85b, 46 lines

```
int blossom(vector<vi>& G, vi& match) {
  int n = sz(G), cnt = -1, ans = 0;match.assign(n, -1);
  vi lab(n), par(n), orig(n), aux(n, -1), q;
  auto blos = [&](int v, int w, int a) {
    while (orig[v] != a) {
      par[v] = w; w = match[v];
      if (lab[w] == 1) lab[w] = 0, q.pb(w);
      orig[v] = orig[w] = a; v = par[w];
    }
  };
  rep(i, n) if (match[i] == -1)
    for (auto e : G[i]) if (match[e] == -1) {
      match[match[e] = i] = e; ans++; break;
    }
  rep(root, n) if (match[root] == -1) {
    fill(all(lab), -1);
    iota(all(orig), 0);
    lab[root] = 0;
    q = {root};
    rep(i, sz(q)) {
      int v = q[i];
      for (auto x : G[v]) if (lab[x] == -1) {
        lab[x] = 1; par[x] = v;
        if (match[x] == -1) {
          for (int y = x; y+1;) {
            int p = par[y], w = match[p];
            match[match[p] = y] = p; y = w;
          }
          ans++;
          goto nxt;
        }
        lab[match[x]] = 0; q.pb(match[x]);
      } else if (lab[x] == 0 && orig[v]!=orig[x]) {
        int a = orig[v], b = orig[x];
        for (cnt++;; swap(a, b)) if (a+1) {
          if (aux[a] == cnt) break;
          aux[a] = cnt;
          a = (match[a]+1 ?
            orig[par[match[a]]] : -1);
        }
        blos(x, v, a); blos(v, x, a);
      }
    }
    nxt:;
  }
  return ans; }
```

WeightedBlossom.h

Description: Read below.

Time: $\mathcal{O}\left(N^3\right)$ 2fd5d6, 232 lines

```
// Edmond's Blossom algorithm for weighted
// maximum matching in general graphs; O(n^3)?
// Weights must be positive (I believe).
//! Source: https://github.com/koosaga/
      DeobureoMinkyuParty/blob/master/teamnote.pdf
struct WeightedBlossom {
  struct edge { int u, v, w; };
  int n, s, nx;
  vector<vector<edge>> g;
  vi lab, match, slack, st, pa, S, vis;
  vector<vi> flo, floFrom;
  queue<int> q;
  // Initialize for k vertices
  WeightedBlossom(int k)
      : n(k), s(n*2+1),
        g(s, vector<edge>(s)),
        lab(s), match(s), slack(s), st(s),
        pa(s), S(s), vis(s), flo(s),
        floFrom(s, vi(n+1)) {
    fwd(u, 1, n+1) fwd(v, 1, n+1)
      g[u][v] = {u, v, 0};
  }
  // Add edge between u and v with weight w
  void addEdge(int u, int v, int w) {
    u++; v++;
    g[u][v].w = g[v][u].w = max(g[u][v].w, w);
  }
  // Compute max weight matching.
  // `count` is set to matching size,
  // `weight` is set to matching weight.
  // Returns vector `match` such that:
  // match[v] = vert matched to v or -1
  vi solve(int& count, ll& weight) {
    fill(all(match), 0);
    nx = n;
    weight = count = 0;
    fwd(u, 0, n+1) flo[st[u] = u].clear();
    int tmp = 0;
    fwd(u, 1, n+1) fwd(v, 1, n+1) {
      floFrom[u][v] = (u-v ? 0 : v);
      tmp = max(tmp, g[u][v].w);
    }
    fwd(u, 1, n+1) lab[u] = tmp;
    while (matching()) count++;
    fwd(u, 1, n+1)
      if (match[u] && match[u] < u)
        weight += g[u][match[u]].w;
    vi ans(n);
    fwd(i, 0, n) ans[i] = match[i+1]-1;
    return ans;
  }
  int delta(edge& e) {
    return lab[e.u]+lab[e.v]-g[e.u][e.v].w*2;
  }
  void updateSlack(int u, int x) {
    if (!slack[x] || delta(g[u][x]) <
      delta(g[slack[x]][x])) slack[x] = u;
  }
  void setSlack(int x) {
    slack[x] = 0;
    fwd(u, 1, n+1) if (g[u][x].w > 0 &&
      st[u] != x && !S[st[u]])
        updateSlack(u, x);
  }
  void push(int x) {
    if (x <= n) q.push(x);
    else fwd(i, 0, sz(flo[x])) push(flo[x][i]);
  }
  void setSt(int x, int b) {
    st[x] = b;
    if (x > n) fwd(i, 0, sz(flo[x]))
      setSt(flo[x][i],b);
  }
  int getPr(int b, int xr) {
    int pr = int(find(all(flo[b]), xr) -
      flo[b].begin());
    if (pr % 2) {
      reverse(flo[b].begin()+1, flo[b].end());
      return sz(flo[b]) - pr;
    } else return pr;
  }
  void setMatch(int u, int v) {
    match[u] = g[u][v].v;
    if (u <= n) return;
    edge e = g[u][v];
    int xr = floFrom[u][e.u], pr = getPr(u,xr);
    fwd(i, 0, pr)
      setMatch(flo[u][i], flo[u][i^1]);
    setMatch(xr, v);
    rotate(flo[u].begin(), flo[u].begin()+pr,
      flo[u].end());
  }
  void augment(int u, int v) {
    while (1) {
      int xnv = st[match[u]];
      setMatch(u, v);
      if (!xnv) return;
      setMatch(xnv, st[pa[xnv]]);
      u = st[pa[xnv]], v = xnv;
    }
  }
  int getLca(int u, int v) {
    static int t = 0;
    for (++t; u||v; swap(u, v)) {
      if (!u) continue;
      if (vis[u] == t) return u;
      vis[u] = t;
      u = st[match[u]];
      if (u) u = st[pa[u]];
    }
    return 0;
  }
  void blossom(int u, int lca, int v) {
    int b = n+1;
    while (b <= nx && st[b]) ++b;
    if (b > nx) ++nx;
    lab[b] = S[b] = 0;
    match[b] = match[lca];
    flo[b].clear();
    flo[b].pb(lca);
    for (int x=u, y; x != lca; x = st[pa[y]]) {
      flo[b].pb(x);
      flo[b].pb(y = st[match[x]]);
      push(y);
    }
    reverse(flo[b].begin()+1, flo[b].end());
    for (int x=v, y; x != lca; x = st[pa[y]]) {
      flo[b].pb(x);
      flo[b].pb(y = st[match[x]]);
      push(y);
    }
    setSt(b, b);
    fwd(x, 1, nx+1) g[b][x].w = g[x][b].w = 0;
    fwd(x, 1, n+1) floFrom[b][x] = 0;
    fwd(i, 0, sz(flo[b])) {
      int xs = flo[b][i];
      fwd(x, 1, nx+1) if (!g[b][x].w ||
        delta(g[xs][x]) < delta(g[b][x]))
          g[b][x]=g[xs][x], g[x][b]=g[x][xs];
      fwd(x, 1, n+1) if (floFrom[xs][x])
        floFrom[b][x] = xs;
    }
    setSlack(b);
  }
  void blossom(int b) {
    for (auto &e : flo[b]) setSt(e, e);
    int xr = floFrom[b][g[b][pa[b]].u];
    int pr = getPr(b, xr);
    for (int i = 0; i < pr; i += 2) {
      int xs = flo[b][i], xns = flo[b][i+1];
      pa[xs] = g[xns][xs].u;
      S[xs] = 1; S[xns] = slack[xs] = 0;
      setSlack(xns); push(xns);
    }
    S[xr] = 1; pa[xr] = pa[b];
    fwd(i, pr+1, sz(flo[b])) {
      int xs = flo[b][i];
      S[xs] = -1; setSlack(xs);
    }
    st[b] = 0;
  }
  bool found(const edge& e) {
    int u = st[e.u], v = st[e.v];
    if (S[v] == -1) {
      pa[v] = e.u; S[v] = 1;
      int nu = st[match[v]];
      slack[v] = slack[nu] = S[nu] = 0;
      push(nu);
    } else if (!S[v]) {
      int lca = getLca(u, v);
      if (!lca) return augment(u, v),
        augment(v, u), 1;
      else blossom(u, lca, v);
    }
    return 0;
```

```
  }
  bool matching() {
    fill(S.begin(), S.begin()+nx+1, -1);
    fill(slack.begin(), slack.begin()+nx+1, 0);
    q = {};
    fwd(x, 1, nx+1)
      if (st[x] == x && !match[x])
        pa[x] = S[x] = 0, push(x);
    if (q.empty()) return 0;
    while (1) {
      while (q.size()) {
        int u = q.front(); q.pop();
        if (S[st[u]] == 1) continue;
        fwd(v, 1, n+1)
          if (g[u][v].w > 0 && st[u] != st[v]){
            if (!delta(g[u][v])) {
              if (found(g[u][v])) return 1;
            } else updateSlack(u, st[v]);
          }
      }
      int d = INT_MAX;
      fwd(b, n+1, nx+1)
        if (st[b] == b && S[b] == 1)
          d = min(d, lab[b]/2);
      fwd(x, 1, nx+1)
        if (st[x] == x && slack[x]) {
          if (S[x] == -1)
            d = min(d, delta(g[slack[x]][x]));
          else if (!S[x])
            d = min(d,delta(g[slack[x]][x])/2);
        }
      fwd(u, 1, n+1) {
        if (!S[st[u]]) {
          if (lab[u] <= d) return 0;
          lab[u] -= d;
        } else if (S[st[u]] == 1) lab[u] += d;
      }
      fwd(b, n+1, nx+1) if (st[b] == b) {
        if (!S[st[b]]) lab[b] += d*2;
        else if (S[st[b]] == 1) lab[b] -= d*2;
      }
      q = {};
      fwd(x, 1, nx+1)
        if (st[x] == x && slack[x] &&
          st[slack[x]] != x &&
          !delta(g[slack[x]][x]) &&
          found(g[slack[x]][x])) return 1;
      fwd(b, n+1, nx+1)
        if (st[b] == b && S[b] == 1 && !lab[b])
          blossom(b);
    }
    return 0;
  }
};
```

MatroidIntersection.h

Description: Find largest subset S of [n] such that S is independent in both matroid A and B, given by their oracles, see example implementations below. Returns vector V such that V[i] = 1 iff i-th element is included in found set;
Time: $\mathcal{O}\left(r^2 \cdot (init + n \cdot add)\right)$, where r is max independent set. e3ed3b, 152 lines

```
template<class T, class U>
vector<bool> intersectMatroids(T& A, U& B, int n) {
  vector<bool> ans(n);
  bool ok = 1;
// NOTE: for weighted matroid intersection find
// shortest augmenting paths first by weight change,
// then by length using Bellman-Ford,
  // Speedup trick (only for unweighted):
  A.init(ans); B.init(ans);
  rep(i, n)
    if (A.canAdd(i) && B.canAdd(i))
      ans[i] = 1, A.init(ans), B.init(ans);
  //End of speedup
  while (ok) {
    vector<vi> G(n);
    vector<bool> good(n);
    queue<int> que;
    vi prev(n, -1);
    A.init(ans); B.init(ans); ok = 0;
    rep(i, n) if (!ans[i]) {
      if (A.canAdd(i)) que.push(i), prev[i]=-2;
      good[i] = B.canAdd(i);
    }
    rep(i, n) if (ans[i]) {
      ans[i] = 0;
      A.init(ans); B.init(ans);
      rep(j, n) if (i != j && !ans[j]) {
        if (A.canAdd(j)) G[i].pb(j); //-cost[j]
        if (B.canAdd(j)) G[j].pb(i); // cost[i]
      }
      ans[i] = 1;
    }
    while (!que.empty()) {
      int i = que.front();
      que.pop();
      if (good[i]) { // best found (unweighted =
          shortest path)
        ans[i] = 1;
        while (prev[i] >= 0) { // alternate matching
          ans[i = prev[i]] = 0;
          ans[i = prev[i]] = 1;
        }
        ok = 1; break;
      }
      for(auto j: G[i]) if (prev[j] == -1)
        que.push(j), prev[j] = i;
    }
  }
  return ans;
}
// Matroid where each element has color
// and set is independent iff for each color c
// #{elements of color c} <= maxAllowed[c].
struct LimOracle {
  vi color; // color[i] = color of i-th element
  vi maxAllowed; // Limits for colors
  vi tmp;
  // Init oracle for independent set S; O(n)
  void init(vector<bool>& S) {
    tmp = maxAllowed;
    rep(i, sz(S)) tmp[color[i]] -= S[i];
  }
  // Check if S+{k} is independent; time: O(1)
  bool canAdd(int k) { return tmp[color[k]] > 0;}
};
// Graphic matroid - each element is edge,
// set is independent iff subgraph is acyclic.
struct GraphOracle {
  vector<pii> elems; // Ground set: graph edges
  int n; // Number of vertices, indexed [0;n-1]
  vi par;
  int find(int i) {
    return par[i] == -1 ? i : par[i] = find(par[i]);
  }
  // Init oracle for independent set S; ~O(n)
  void init(vector<bool>& S) {
    par.assign(n, -1);
    rep(i, sz(S)) if (S[i])
      par[find(elems[i].st)] = find(elems[i].nd);
  }
  // Check if S+{k} is independent; time: ~O(1)
  bool canAdd(int k) {
    return find(elems[k].st) != find(elems[k].nd);
  }
};
// Co-graphic matroid - each element is edge,
// set is independent iff after removing edges
// from graph number of connected components
// doesn't change.
struct CographOracle {
  vector<pii> elems; // Ground set: graph edges
  int n; // Number of vertices, indexed [0;n-1]
  vector<vi> G;
  vi pre, low;
  int cnt;
  int dfs(int v, int p) {
    pre[v] = low[v] = ++cnt;
    for(auto e: G[v]) if (e != p)
      low[v] = min(low[v], pre[e] ?: dfs(e,v));
    return low[v];
  }
  // Init oracle for independent set S; O(n)
  void init(vector<bool>& S) {
    G.assign(n, {});
    pre.assign(n, 0);
    low.resize(n);
    cnt = 0;
    rep(i,sz(S)) if (!S[i]) {
      pii e = elems[i];
      G[e.st].pb(e.nd);
      G[e.nd].pb(e.st);
    }
    rep(v, n) if (!pre[v]) dfs(v, -1);
  }
  // Check if S+{k} is independent; time: O(1)
  bool canAdd(int k) {
    pii e = elems[k];
    return max(pre[e.st], pre[e.nd]) != max(low[e.st],
        low[e.nd]);
  }
};
// Matroid equivalent to linear space with XOR
struct XorOracle {
  vector<ll> elems; // Ground set: numbers
  vector<ll> base;
  // Init for independent set S; O(n+r^2)
  void init(vector<bool>& S) {
    base.assign(63, 0);
    rep(i, sz(S)) if (S[i]) {
      ll e = elems[i];
      rep(j, sz(base)) if ((e >> j) & 1) {
        if (!base[j]) {
          base[j] = e;
          break;
        }
        e ^= base[j];
      }
    }
  }
  // Check if S+{k} is independent; time: O(r)
  bool canAdd(int k) {
    ll e = elems[k];
    rep(i, sz(base)) if ((e >> i) & 1) {
      if (!base[i]) return 1;
      e ^= base[i];
    }
    return 0;
  }
};
```

7.4 DFS algorithms

StronglyConnected.h

Description: SCC scc(graph), scc[v] = index of SCC of vertex v, scc.comps[i] = vertices of i-th scc, components in reverse topological order
Time: $\mathcal{O}(|E| + |V|)$ a648bc, 17 lines

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

Biconnected.h

Description: Biconnected bi(graph), bi[v] = indices of components containing v, bi.verts[i] = vertices of i-th component, bi.edges[i] = edges of i-th component, Bridges $\iff$ components with 2 vertices, Articulation points $\iff$ vertices in > 1 comp, Isolated vertex $\iff$ empty component list
Time: $\mathcal{O}(|E| + |V|)$ ebf680, 26 lines

```
struct Biconnected : vector<vi> {
  vector<vi> verts; vector<pii> S;
  vector<vector<pii>> edges;
  Biconnected() {}
  Biconnected(vector<vi>& G) : S(sz(G)) {
    resize(sz(G));
    rep(i, sz(G)) S[i].x ?: dfs(G, i, -1);
    rep(c, sz(verts)) for(auto v : verts[c])
      at(v).pb(c);
  }
  int dfs(vector<vi>& G, int v, int p) {
    int low = S[v].x = sz(S)-1; S.pb({v, -1});
    for (auto e : G[v]) if (e != p) {
      if (S[e].x < S[v].x) S.pb({v, e});
      low = min(low, S[e].x ?: dfs(G, e, v));
    }
    if (p + 1 && low >= S[p].x) {
      verts.pb({p}); edges.pb({});
      fwd(i, S[v].x, sz(S)) {
        if (S[i].y == -1)
          verts.back().pb(S[i].x);
        else edges.back().pb(S[i]);
      }
      S.resize(S[v].x);
    }
    return low; } };
```

2Sat.h

Description: sat.either(x, ỹ). Uses kosaraju
Usage: `SAT2 sat(variable_cnt), sat.solve(),`
`sat[i] = value of i-th variable (0 or 1),`
`(internally: i_false_id = 2i, i_true_id = 2i + 1)`
Time: $\mathcal{O}(|SantaClauses| + |Variables|)$ 2414ec, 37 lines

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

EulerWalk.h

Description: Eulerian undirected/directed path/cycle algorithm. Input should be a vector of (dest, global edge index), where for undirected graphs, forward/backward edges have the same index. Returns a list of nodes in the Eulerian path/cycle with src at both start and end, or empty list if no cycle/path exists. To get edge indices back, add .second to s and ret.
Time: $\mathcal{O}(V + E)$ 3e0eb1, 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.pb(x); s.pop_back(); continue;
        }
    tie(y, e) = gr[x][it++];
    if (!eu[e]) {
      D[x]--, D[y]++;
      eu[e] = 1; s.pb(y);
    }}
  for (int x : D) if (x < 0 || sz(ret) != nedges+1)
      return {};
  return {ret.rbegin(), ret.rend()};
}
```

Dominators.h

Description: Tarjan's dominators in directed graph Returns tree (as array of parents) of immediate dominators idom. idom[root] = root, idom[v] = -1 if v is unreachable from root
Time: $\mathcal{O}(|E| log|V|)$ 2613e6, 32 lines

```
vi dominators(vector<vi>& G, int root) {
  int n = sz(G); vector<vi> in(n), bucket(n);
  vi pre(n, -1), anc(n, -1), par(n), best(n);
  vi ord, idom(n, -1), sdom(n, n), rdom(n);
  auto dfs = [&] (auto f, int v, int p) -> void {
    if (pre[v] == -1) {
      par[v] = p; pre[v] = sz(ord);
      ord.pb(v);
```

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

KthShortest.h

Description: Given directed weighted graph with non-negative edge weights gets K-th shortest walk (not necessarily simple) in O(log—E—). -1 if no next path (can only happen in DAG). WARNING: USES KLOGM memory and persistent heaps!

2d9393, 57 lines

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

DenseDFS.h

Description: DFS over dense graph. Suddenly DFS over N <= 1000 graph many times becomes feasible

6e9645, 68 lines

```
// DFS over bit-packed adjacency matrix
// G = NxN adjacency matrix of graph
//     G(i,j) <=> (i,j) is edge
// V = 1xN matrix containing unvisited vertices
//     V(0,i) <=> i-th vertex is not visited
// Total DFS time: O(n^2/64)
using ull = uint64_t;
// Matrix over Z_2 (bits and xor)
// TODO: arithmetic operations //!HIDE
struct BitMatrix {
  vector<ull> M;
  int rows, cols, stride;
  // Create matrix with n rows and m columns
  BitMatrix(int n = 0, int m = 0) {
    rows = n; cols = m;
    stride = (m+63)/64;
    M.resize(n*stride);
  }
  // Get pointer to bit-packed data of i-th row
  ull* row(int i) { return &M[i*stride]; }
  // Get value in i-th row and j-th column
  bool operator()(int i, int j) {
    return (row(i)[j/64] >> (j%64)) & 1;
  }
  // Set value in i-th row and j-th column
  void set(int i, int j, bool val) {
    ull &w = row(i)[j/64], m = 1ull << (j%64);
    if (val) w |= m;
    else w &= ~m;
  }
};
struct DenseDFS {
  BitMatrix G, V; // space: O(n^2/64)
  // Initialize structure for n vertices
  DenseDFS(int n = 0) : G(n, n), V(1, n) {
    reset();
  }
  // Mark all vertices as unvisited
  void reset() { for(auto &x : V.M) x = -1; }
  // Get/set visited flag for i-th vertex
  void setVisited(int i) { V.set(0, i, 0); }
  bool isVisited(int i)  { return !V(0, i); }
  // DFS step: func is called on each unvisited
  // neighbour of i. You need to manually call
  // setVisited(child) to mark it visited
  // or this function will call the callback
  // with the same vertex again.
  void step(int i, auto func) {
    ull* E = G.row(i);
    for (int w = 0; w < G.stride;) {
      ull x = E[w] & V.row(0)[w];
      if (x) func((w<<6) | __builtin_ctzll(x));
      else   w++;
    }
  }
};
```

PlanarFaces.h

Description: Read desc below.

a391b4, 102 lines

```
/*
 * complexity mlogm, assumes that you are given an
       embedding
 * graph is drawn straightline non-intersecting
 * returns combinatorial embedding (inner face vertices
        clockwise, outer counter clockwise).
 * WAZNE czasem trzeba zlaczyc wszystkie sciany
       zewnetrzne (chodzi o kmine do konkretnego
       zadania)
 * (ktorych moze byc kilka, gdy jest wiele spojnych) w
       jedna sciane.
 * Zewnetrzne sciany moga wygladac jak kaktusy, a
       wewnetrzne zawsze sa niezdegenerowanym
       wielokatem.
 */
struct Edge {
  int e, from, to;
  // face is on the right of "from -> to"
};
ostream& operator<<(ostream &o, Edge e) {
  return o << vector{e.e, e.from, e.to};
}
struct Face {
  bool is_outside;
  vector<Edge> sorted_edges;
  // edges are sorted clockwise for inside and cc for
        outside faces
};
ostream& operator<<(ostream &o, Face f) {
  return o << pair(f.is_outside, f.sorted_edges);
}
vector<Face> split_planar_to_faces(vector<pii> coord,
      vector<pii> edges) {
  int n = sz(coord);
  int E = sz(edges);
  vector<vi> graph(n);
  rep(e, E) {
    auto [v, u] = edges[e];
    graph[v].eb(e);
    graph[u].eb(e);
  }
  vi lead(2 * E);
  iota(lead.begin(), lead.end(), 0);
  function<int (int)> find = [&](int v) {
    return lead[v] == v ? v : lead[v] = find(lead[v]);
  };
  auto side_of_edge = [&](int e, int v, bool outward) {
    return 2 * e + ((v != min(edges[e].first, edges[e].
          second)) ^ outward);
  };
  rep(v, n) {
    vector<pair<pii, int>> sorted;
    for(int e : graph[v]) {
      auto p = coord[edges[e].first ^ edges[e].second ^
            v];
      auto center = coord[v];
      sorted.eb(pair(p.first - center.first, p.second -
            center.second), e);
    }
    sort(all(sorted), [&](pair<pii, int> l0, pair<pii,
          int> r0) {
      auto l = l0.first;
      auto r = r0.first;
      bool half_l = l > pair(0, 0);
      bool half_r = r > pair(0, 0);
      if(half_l != half_r)
        return half_l;
      return l.first * LL(r.second) - l.second * LL(r.
            first) > 0;
    });
    rep(i, sz(sorted)) {
      int e0 = sorted[i].second;
      int e1 = sorted[(i + 1) % sz(sorted)].second;
      int side_e0 = side_of_edge(e0, v, true);
      int side_e1 = side_of_edge(e1, v, false);
      lead[find(side_e0)] = find(side_e1);
    }
  }
  vector<vi> comps(2 * E);
  rep(i, 2 * E)
    comps[find(i)].eb(i);
  vector<Face> polygons;
  vector<vector<pii>> outgoing_for_face(n);
  rep(leader, 2 * E)
    if(sz(comps[leader])) {
      for(int id : comps[leader]) {
        int v = edges[id / 2].first;
        int u = edges[id / 2].second;
        if(v > u)
          swap(v, u);
        if(id % 2 == 1)
          swap(v, u);
        outgoing_for_face[v].eb(u, id / 2);
      }
      vector<Edge> sorted_edges;
      function<void (int)> dfs = [&](int v) {
        while(sz(outgoing_for_face[v])) {
          auto [u, e] = outgoing_for_face[v].back();
          outgoing_for_face[v].pop_back();
          dfs(u);
          sorted_edges.eb(e, v, u);
        }
      };
      dfs(edges[comps[leader].front() / 2].first);
      reverse(all(sorted_edges));
      LL area = 0;
      for(auto edge : sorted_edges) {
        auto l = coord[edge.from];
        auto r = coord[edge.to];
        area += l.first * LL(r.second) - l.second * LL(
              r.first);
      }
      polygons.eb(area >= 0, sorted_edges);
    }
  // Remember that there can be multiple outside faces.
  return polygons;
}
```

PlanarityCheck.h

Description: Read desc below.

cc4508, 93 lines

```
/*
 * Opis: O(szybko) ale istnieja przyklady O(n2),
       przyjmuje graf nieskierowany bez petelek i
       multikrawedzi.
 */
bool is_planar(vector<vi> g) {
  int n = sz(g), m = 0;
  rep(v, n) m += sz(g[v]);
  m /= 2;
  if(n <= 3) return true;
  if(m > 3 * n - 6) return false;
  vector<vi> up(n), dn(n);
  vi low(n, -1), pre(n);
  rep(start, n)
    if(low[start] == -1) {
      vector<pii> e_up;
      int tm = 0;
      function<void (int, int)> dfs_low = [&](int v,
            int p) {
        low[v] = pre[v] = tm++;
        for(int u : g[v])
          if(u != p and low[u] == -1) {
            dn[v].eb(u);
            dfs_low(u, v);
            low[v] = min(low[v], low[u]);
          }
          else if(u != p and pre[u] < pre[v]) {
            up[v].eb(ssize(e_up));
            e_up.eb(v, u);
            low[v] = min(low[v], pre[u]);
          }
      };
      dfs_low(start, -1);
      vector<pair<int, bool>> dsu(sz(e_up));
      rep(v, sz(dsu)) dsu[v].first = v;
      function<pair<int, bool> (int)> find = [&](int v)
            {
        if(dsu[v].first == v)
          return pair(v, false);
        auto [u, ub] = find(dsu[v].first);
        return dsu[v] = pair(u, ub ^ dsu[v].second);
      };
      auto onion = [&](int x, int y, bool flip) {
        auto [v, vb] = find(x);
        auto [u, ub] = find(y);
        if(v == u)
          return not (vb ^ ub ^ flip);
        dsu[v] = {u, vb ^ ub ^ flip};
        return true;
      };
      auto interlace = [&](const vi &ids, int lo) {
        vi ans;
        for(int e : ids)
          if(pre[e_up[e].second] > lo)
            ans.eb(e);
        return ans;
      };
      auto add_fu = [&](const vi &a, const vi &b) {
        fwd(k, 1, sz(a))
          if(not onion(a[k - 1], a[k], 0))
            return false;
        fwd(k, 1, sz(b))
          if(not onion(b[k - 1], b[k], 0))
            return false;
        return a.empty() or b.empty() or onion(a[0], b
              [0], 1);
      };
      function<bool (int, int)> dfs_planar = [&](int v,
             int p) {
        for(int u : dn[v])
          if(not dfs_planar(u, v))
            return false;
        rep(i, sz(dn[v])) {
          fwd(j, i + 1, sz(dn[v]))
            if(not add_fu(interlace(up[dn[v][i]], low[
                  dn[v][j]]),
                    interlace(up[dn[v][j]], low[dn[v][i
                          ]])))
              return false;
          for(int j : up[v]) {
            if(e_up[j].first != v)
              continue;
            if(not add_fu(interlace(up[dn[v][i]], pre[
                  e_up[j].second]),
                    interlace({j}, low[dn[v][i]])))
              return false;
          }
        }
        for(int u : dn[v]) {
```

```
        for(int idx : up[u])
          if(pre[e_up[idx].second] < pre[p])
            up[v].eb(idx);
        exchange(up[u], {});
      }
      return true;
    };
    if(not dfs_planar(start, -1))
      return false;
  }
  return true;
}
```

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.

Time: $\mathcal{O}(NM)$ be7d13, 31 lines

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

EdgeColoringBipartite.h

Description: Bipartite edge coloring, edges is list of (left vert, right vert). Returns number of used colors, which is equal to max degree. col[i] = color of i-th edge [0..max_deg-1]

Time: $\mathcal{O}(NM)$ 850e16, 34 lines

```
int colorEdges(vector<pii>& edges, int n, vi& col) {
  int m = sz(edges), c[2] = {}, ans = 0;
  vi deg[2];
  vector<vector<pii>> has[2];
  col.assign(m, 0);
  rep(i, 2) {
    deg[i].resize(n+1);
    has[i].resize(n+1, vector<pii>(n+1));
  }
  auto dfs = [&](auto f, int x, int p)->void {
    pii i = has[p][x][c[!p]];
    if (has[!p][i.x][c[p]].y) f(f, i.x, !p);
    else has[!p][i.x][c[!p]] = {};
    has[p][x][c[p]] = i;
    has[!p][i.x][c[p]] = {x, i.y};
    if (i.y) col[i.y-1] = c[p]-1;
  };
  rep(i, m) {
    int x[2] = {edges[i].x+1, edges[i].y+1};
    rep(d, 2) {
      deg[d][x[d]]++;
      ans = max(ans, deg[d][x[d]]);
      for (c[d] = 1; has[d][x[d]][c[d]].y;)
        c[d]++;
    }
    if (c[0]-c[1]) dfs(dfs, x[1], 1);
    rep(d, 2)
      has[d][x[d]][c[0]] = {x[!d], i+1};
    col[i] = c[0]-1;
  }
  return ans;
}
```

ChordalGraph.h

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

Time: $\mathcal{O}(n+m)$ 34f560, 38 lines

```
vi perfectEliminationOrder(vector<vi>& g) { // 0-
    indexed, adj list
  int top = 0, n = sz(g);
  vi ord, vis(n), indeg(n);
  vector<vi> bucket(n);
  rep(i, n) bucket[0].pb(i);
  for(int i = 0; i < n; ) {
    while(bucket[top].empty()) --top;
    int u = bucket[top].back();
    bucket[top].pop_back();
    if(vis[u]) continue;
    ord.pb(u);
    vis[u] = 1;
    ++i;
    for(int v : g[u]) {
      if(vis[v]) continue;
      bucket[++indeg[v]].pb(v);
      top = max(top, indeg[v]);
    }
  }
  reverse(all(ord));
  return ord;
}
bool isChordal(vector<vi>& g, vi ord) {//ord =
    perfectEliminationOrder(g)
  int n = sz(g);
  set<pii> edg;
  rep(i, n) for(auto v:g[i]) edg.insert({i,v});
  vi pos(n); rep(i, n) pos[ord[i]] = i;
  rep(u, n){
    int mn = n;
    for(auto v : g[u]) if(pos[u] < pos[v]) mn = min(mn,
        pos[v]);
    if(mn != n) {
      int p = ord[mn];
      for(auto v : g[u]) if(pos[v] > pos[u] && v != p
          && !edg.count({v, p})) return 0;
    }
  }
  return 1;
}
```

7.6 Heuristics

MaximalCliques.h

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

Time: $\mathcal{O}\left(3^{n/3}\right)$, much faster for sparse graphs 6effc5, 12 lines

```
typedef bitset<128> B;
template<class F>
void cliques(vector<B>& eds, F f, B P = ~B(), B X={}, B
    R={}) {
  if (!P.any()) { if (!X.any()) f(R); return; }
  auto q = (P | X)._Find_first();
  auto cands = P & ~eds[q];
  rep(i,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. 448c39, 49 lines

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

MaximumCliqueChinese.h

Description: Chinese max clique heuristic, good for geometric packing problems. Vertices should be ordered by (X, Y) (not shuffled!). daa1f3, 45 lines

```
constexpr int N = 405;
struct MaxClique {
  bool g[N][N];
  int n, dp[N], st[N][N], ans, res[N], stk[N];
  void init(int n_) {
    n = n_; memset(g, 0, sizeof(g));
  }
  void addEdge(int u, int v, int w) {
    g[u][v] = w;
  }
  bool dfs(int siz, int num) {
    if (siz == 0) {
      if (num > ans) {
        ans = num;
        copy(stk+1, stk+1+num, res+1);
        return 1;
      }
      return 0;
    }
    rep(i, siz) {
      if (siz-i+num <= ans) return 0;
      int u = st[num][i];
      if (dp[u]+num <= ans) return 0;
      int cnt = 0;
      fwd(j, i+1, siz) if (g[u][st[num][j]])
        st[num+1][cnt++] = st[num][j];
      stk[num+1] = u;
      if (dfs(cnt, num + 1)) return 1;
    }
    return 0;
  }
  int solve() {
    ans = 0;
    memset(dp, 0, sizeof(dp));
    for (int i = n; i >= 1; i--) {
      int cnt = 0;
      fwd(j, i+1, n+1)
        if (g[i][j]) st[1][cnt++] = j;
      stk[1] = i;
      dfs(cnt, 1);
      dp[i] = ans;
    }
    return ans;
  }
};
```

7.7 Trees

TreeJumps.h

Description: Provides LCA, K-th ancestor and isAncestor queries in log(n) time with O(n) memory. 271bb1, 51 lines

```
struct LCA {
  vi par, jmp, depth, pre, post;
  int cnt = 0; LCA() {}
  LCA(vector<vi>& g, int v = 0) :
  par(sz(g), -1), jmp(sz(g), v),
  depth(sz(g)), pre(sz(g)), post(sz(g)) {
    dfs(g, v);
  }
  void dfs(vector<vi>& g, int v) {
    int j = jmp[v], k = jmp[j], x =
      depth[v]+depth[k] == depth[j]*2 ? k : v;
    pre[v] = ++cnt;
    for (auto e : g[v]) if (!pre[e]) {
      par[e] = v; jmp[e] = x;
      depth[e] = depth[v]+1;
      dfs(g, e);
    }
    post[v] = ++cnt;
  }
  int laq(int v, int d) {
    while (depth[v] > d)
      v = depth[jmp[v]] < d ? par[v] : jmp[v];
    return v;
  } // Lowest Common Ancestor; time: O(lg n)
  int operator()(int a, int b) {
    if (depth[a] > depth[b]) swap(a, b);
    b = laq(b, depth[a]);
    while (a != b) {
      if (jmp[a] == jmp[b])
        a = par[a], b = par[b];
      else
        a = jmp[a], b = jmp[b];
    }
    return a;
  } // Check if a is ancestor of b; time: O(1)
  bool isAncestor(int a, int b) {
    return pre[a] <= pre[b] &&
           post[b] <= post[a];
  } // Get distance from a to b; time: O(lg n)
  int distance(int a, int b) {
    return depth[a] + depth[b] -
           depth[operator()(a, b)]*2;
  } // Get k-th vertex on path from a to b,
    // a is 0, b is last; time: O(lg n)
    // Returns -1 if k > distance(a, b)
  int kthVertex(int a, int b, int k) {
    int c = operator()(a, b);
    if (depth[a]-k >= depth[c])
      return laq(a, depth[a]-k);
    k += depth[c]*2 - depth[a];
    return (k > depth[b] ? -1 : laq(b, k)); } };
```

LCA.h

Description: Provides lca(v, u) and compress(ss) functions. Compress returns a list of (par, orig_index) representing a tree rooted at 0. ss is the subset of nodes to be compressed.

"../data-structures/RMQ.h" c3ae23, 31 lines

```
struct LCA { // takes up 5 * N memory + RMQ (be careful
    )
  vi t, it, et, rv; // time, inverse-time, euler-tour
      times
  RMQ<int> rmq;  // times are given in dfs preorder
  LCA(vector<vi> &g, int r = 0) : t(sz(g)), rv(sz(g)) {
```

```
    if (sz(g) == 0) return;
    dfs(g, r, -1);
    rmq = RMQ<int>(et);
  }
  void dfs(vector<vi> &g, int v, int p) {
    t[v] = sz(it); it.pb(v);
    for (int u : g[v]) if (u != p)
      et.pb(t[v]), dfs(g, u, v);
  }
  int lca(int v, int u) {
    if (v == u) return v;
    if (t[v] > t[u]) swap(v, u);
    return it[rmq.get(t[v], t[u] - 1)];
  }
  vector<pii> compress(vi &ss) { // 3 * sz(ss) lca
      queries
    if (sz(ss) == 0) return {};
    auto cmp = [&] (int v, int u) { return t[v] < t[u];
        };
    sort(all(ss), cmp); int m = sz(ss) - 1;
    rep(i, m) ss.pb(lca(ss[i], ss[i + 1]));
    sort(all(ss), cmp);
    ss.erase(unique(all(ss)), ss.end());
    vector<pii> r; int v = ss[0], u;
    rep(i, sz(ss)) {
      rv[u = ss[i]] = i;
      r.pb({rv[lca(v, u)], u}), v = u;
    }
    return r; } };
```

HLD.h

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

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

"MyLazyTree.h" // make some sort of tree or whatever you like
80e958, 48 lines

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

Centroid.h

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

Time: $\mathcal{O}(n \log n)$
5ba6c3, 47 lines

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

LinkCutTree.h

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

Time: All operations take amortized $\mathcal{O}(\log N)$
0fb462, 90 lines

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

DirectedMST.h

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

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

"../data-structures/UnionFindRollback.h"
84db4b, 60 lines

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

7.8 Math

7.8.1 Number of Spanning Trees

Create an $N \times N$ matrix `mat`, and for each edge $a \to b \in G$, 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.)
fc73e8, 28 lines

```
int sgn(long long x) { return (x>0) - (x<0); } //
    floats compare with eps
```

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

lineDistance.h

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

"Point.h" f6bf6b, 4 lines

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

SegmentDistance.h

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

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

"Point.h" 5c88f4, 6 lines

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

SegmentIntersection.h

Description: If a unique intersection point between the line segments going from 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);
}
```

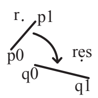
OnSegment.h

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();
}
```

LineProjectionReflection.h

Description: Projects point p onto line ab. Set refl=true to get reflection of point p across line ab insted. The wrong point will be returned if P is an integer point and the desired point doesn't have integer coordinates. Products of three coordinates are used in intermediate steps so watch out for overflow.

"Point.h" b5562d, 5 lines

```
template<class P>
P lineProj(P a, P b, P p, bool refl=false) {
  P v = b - a;
  return p - v.perp()*(1+refl)*v.cross(p-a)/v.dist2();
}
```

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,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, 10 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" 31cca4, 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.pb({c1 + v * r1, c2 + v * r2});
  }
  if (h2 == 0) out.pop_back();
  return out;
}
```

CircleLine.h

Description: Finds the intersection between a circle and a line. Returns a vector of either 0, 1, or 2 intersection points. P is intended to be Point<double>.

"Point.h" e0cfba, 9 lines

```
template<class P>
vector<P> circleLine(P c, double r, P a, P b) {
  P ab = b - a, p = a + ab * (c-a).dot(ab) / ab.dist2()
      ;
  double s = a.cross(b, c), h2 = r*r - s*s / ab.dist2()
      ;
  if (h2 < 0) return {};
  if (h2 == 0) return {p};
  P h = ab.unit() * sqrt(h2);
  return {p - h, p + h};
}
```

CirclePolygonIntersection.h

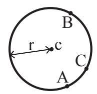
Description: Returns the area of the intersection of a circle with a ccw polygon.

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

"../../content/geometry/Point.h" 9f3c45, 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,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" 3a52a7, 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,sz(ps)) if ((o - ps[i]).dist() > r * EPS) {
    o = ps[i], r = 0;
    rep(j,i) if ((o - ps[j]).dist() > r * EPS) {
      o = (ps[i] + ps[j]) / 2;
      r = (o - ps[i]).dist();
      rep(k,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" ba8e07, 11 lines

```
template<class P>
bool inPolygon(vector<P> &p, P a, bool strict = true) {
  int cnt = 0, n = sz(p);
  rep(i,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" b775a2, 6 lines

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

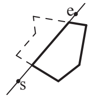
Minkowski.h

Description: Computes Minkowski sum of two convex polygons in ccw order. Vertices are required to be in ccw order.
Time: $\mathcal{O}(n+m)$
"Point.h", "Angle.h" e0df19, 18 lines

```
P edgeSeq(vector<P> p, vector<P>& edges) {
  int i = 0, n = sz(p);
  rep(j, n) if (tie(p[i].y, p[i].x) > tie(p[j].y, p[j].
      x)) i = j;
  rep(j, n) edges.pb(p[(i+j+1)%n] - p[(i+j)%n]);
  return p[i];
}
vector<P> hullSum(vector<P> A, vector<P> B) {
  vector<P> sum, e1, e2, es(sz(A) + sz(B));
  P pivot = edgeSeq(A, e1) + edgeSeq(B, e2);
  merge(all(e1), all(e2), es.begin(), [&](P a, P b){
    return Angle(a.x, a.y) < Angle(b.x,b.y);
  });
  sum.pb(pivot);
  for(auto e: es) sum.pb(sum.back() + e);
  sum.pop_back();
  return sum; //can have collinear vertices!
}
```

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" e9fce4, 13 lines

```
typedef Point<double> P;
vector<P> polygonCut(const vector<P>& poly, P s, P e) {
  vector<P> res;
  rep(i,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.pb(lineInter(s, e, cur, prev).second);
    if (side)
      res.pb(cur);
  }
  return res;
}
```

PolygonUnion.h

Description: Calculates the area of the union of n polygons (not necessarily convex). The points within each polygon must be given in CCW order. (Epsilon checks may optionally be added to sideOf/sgn, but shouldn't be needed.)
Time: $\mathcal{O}\left(N^2\right)$, where N is the total number of points
"Point.h", "sideOf.h" e76126, 33 lines

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

ConvexHull.h

Description: Returns a vector of the points of the convex hull in counter-clockwise order. Points on the edge of the hull between two other points are not considered part of the hull.
Time: $\mathcal{O}(n \log n)$
"Point.h" 0c88c1, 13 lines

```
using P = Point<ll>;
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])};
}
```

ConvexHullOnline.h

Description: Allows online point insertion. If exists, left vertical segment is included; right one is excluded. To get a lower hull add (-x, -y) instead of (x, y).
Time: amortized $\mathcal{O}(\log n)$ per add
"Point.h" 10c55b, 16 lines

```
using P = Point<ll>;
struct UpperHull : set<P> {
  bool rm(auto it) {
    if (it==begin() || it==end() || next(it)==end() ||
      it->cross(*prev(it), *next(it)) > 0)
      return false;
    erase(it); return true;
  }
  bool add(P p) { // true iff added
    auto [it, ok] = emplace(p);
    if (!ok || rm(it)) return false;
    while (rm(next(it)));
    while (it != begin() && rm(prev(it)));
    return true;
  }
};
```

HullDiameter.h

Description: Returns the two points with max distance on a convex hull (ccw, no duplicate/collinear points).
Time: $\mathcal{O}(n)$
"Point.h" 8edf58, 12 lines

```
typedef Point<ll> P;
array<P, 2> hullDiameter(vector<P> S) {
  int n = sz(S), j = n < 2 ? 0 : 1;
  pair<ll, array<P, 2>> res({0, {S[0], S[0]}});
  rep(i,j)
    for (;; j = (j + 1) % n) {
      res = max(res, {(S[i] - S[j]).dist2(), {S[i], S[j
          ]}});
      if ((S[(j + 1) % n] - S[j]).cross(S[i + 1] - S[i
          ]) >= 0)
        break;
    }
  return res.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.
Time: $\mathcal{O}(\log n)$
"Point.h" 46fb73, 44 lines

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

HalfplaneIntersection.h

Description: Online half plane intersection. Works both for ll and long double. Bounding box is optional, but needed for distinguishing bounded vs unbounded. Halfplanes are sorted ccw in HPI.s. Time: O(log n) per add.
"Point.h" f5db92, 98 lines

```
using T = ll; // has to fit 2*|pts|**2
using P = Point<T>; // only cross needed
using SuperT = __int128_t; // has to fit 6*|pts|**3
```

```
const SuperT EPS = 1e-12; // |pts| <= 10^6 (for T=dbl)
struct Line {
  T a,b,c;
  Line(T a_=0, T b_=0, T c_=0): a(a_), b(b_), c(c_) {}
        //ax + by + c >= 0 (coords <= 10^9)
  Line(P p, P q): a(p.y-q.y), b(q.x-p.x), c(p.cross(q))
        {} //p->q ccw (coords <= 10^6)
  Line operator- () const {return Line(-a, -b, -c); }
  bool up() const { return a?(a<0):(b>0);}
  P v() const {return P(a,b);}
  P vx() {return P(b,c);} P vy() {return P(a,c);}
  T wek(Line p) const {return v().cross(p.v());}
  bool operator<(Line b) const {
    if (up() != b.up()) return up() > b.up();
    return wek(b) > 0;
  }
};
bool parallel(Line a, Line b) {return !a.wek(b);}
bool same(Line a, Line b) {
  return parallel(a,b) && !a.vy().cross(b.vy()) && !a.
        vx().cross(b.vx());
}
T weaker (Line a, Line b) {
  if (abs(a.a) > abs(a.b)) return a.c*abs(b.a) - b.c*
        abs(a.a);
  return a.c*abs(b.b) - b.c*abs(a.b);
}
array<SuperT, 3> intersect(Line a, Line b) {
  SuperT det = a.wek(b);
  SuperT x = a.vx().cross(b.vx());
  SuperT y = a.vy().cross(b.vy());
  // if (T=dbl) return {x / det, -y / det, 1.0};
  if (det > 0) return {x, -y, det};
  return {-x, y, -det};
}
struct HPI {
  bool empty=0, pek=0;
  set<Line> s;
  typedef set<Line>::iterator iter;
  iter next(iter it){return ++it == s.end() ? s.begin()
        : it;}
  iter prev(iter it){return it == s.begin() ? --s.end()
        : --it;}
  bool hide(Line a, Line b, Line c) { // do a,b hide c?
    if (parallel(a,b)) {
      if (weaker(a, -b) < 0) empty = 1;
      return 0;
    }
    if (a.wek(b) < 0) swap(a,b);
    auto [rx, ry, rdet] = intersect(a,b);
    auto v = rx*c.a + ry*c.b + rdet*c.c;
    if (a.wek(c) >=0  && c.wek(b) >=0 && v >= -EPS)
        return 1;
    if (a.wek(c) < 0  && c.wek(b) < 0) {
      if (v < -EPS) empty = 1;
      else if (v <= EPS) pek = 1;
    }
    return 0;
  }
  void delAndMove(iter& i, int nxt) {
    iter j = i;
    if(nxt==1) i = next(i);
    else i = prev(i);
    s.erase(j);
  }
  void add(Line l) {
    if (empty) return;
    if (l.a == 0 && l.b == 0) {
      if (l.c < 0) empty = 1;
      return;
    }
    iter it = s.lower_bound(l);  //parallel
    if(it != s.end() && parallel(*it, l) && it->up() ==
        l.up()) {
      if (weaker(l, *it)>=0) return;
      delAndMove(it,1);
    }
    if(it == s.end()) it = s.begin(); //*it>p
    while(sz(s) >= 2 && hide(l, *next(it), *it))
      delAndMove(it,1);
    if(sz(s)) it = prev(it); //*it<p
    while(sz(s) >= 2 && hide(l, *prev(it), *it))
      delAndMove(it,0);
    if(sz(s) < 2 || !hide(*it, *next(it), l)) s.insert(
        l);
  }
  int type() { // 0=empty, 1=point, 2=segment,
    if(empty) return 0; // 3=halfline, 4=line,
    if(sz(s) <= 4) { // 5=polygon or unbounded
      vector<Line> r(all(s));
      if(sz(r) == 2 && parallel(r[0], r[1]) && weaker(r
          [0],-r[1])<0)
        return 0;
      rep(i, sz(r)) rep(j, i) if(same(r[i], r[j])) {
        if(sz(r) == 2) return 4;
        if(sz(r) == 3) return 3;
        if(sz(r) == 4 && same(r[0], r[2]) && same(r[1],
            r[3])) return 1;
        return 2;
      }
      if(sz(r) == 3 && pek) return 1;
    }
    return 5;
  }
};
```

8.4 Misc. Point Set Problems

ClosestPair.h

Description: Finds the closest pair of points.
Time: $\mathcal{O}(n \log n)$

"Point.h" ac41a6, 17 lines

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

ManhattanMST.h

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

"Point.h" 4c1e22, 22 lines

```
typedef Point<int> P;
vector<array<int, 3>> manhattanMST(vector<P> ps) {
  vi id(sz(ps));
  iota(all(id), 0);
  vector<array<int, 3>> edges;
  rep(k,4) {
    sort(all(id), [&](int i, int j) {
         return (ps[i]-ps[j]).x < (ps[j]-ps[i]).y;});
    map<int, int> sweep;
    for (int i : id) {
      for (auto it = sweep.lower_bound(-ps[i].y); it !=
            sweep.end(); sweep.erase(it++)) {
        int j = it->second;
        P d = ps[i] - ps[j];
        if (d.y > d.x) break;
        edges.pb({d.y + d.x, i, j});
      }
      sweep[-ps[i].y] = i;
    }
    for (P& p : ps) if (k & 1) p.x = -p.x; else swap(p.
        x, p.y);
  }
  return edges;
}
```

kdTree.h

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

"Point.h" bac5b0, 63 lines

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

FastDelaunay.h

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

"Point.h" caa383, 88 lines

```
typedef Point<ll> P;
typedef struct Quad* Q;
typedef __int128_t lll; // (can be ll if coords are < 2
      e4)
P arb(LLONG_MAX,LLONG_MAX); // not equal to any other
      point
struct Quad {
  Q rot, o; P p = arb; bool mark;
  P& F() { return r()->p; }
  Q& r() { return rot->rot; }
  Q prev() { return rot->o->rot; }
  Q next() { return r()->prev(); }
} *H;
bool circ(P p, P a, P b, P c) { // is p in the
      circumcircle?
  lll p2 = p.dist2(), A = a.dist2()-p2,
      B = b.dist2()-p2, C = c.dist2()-p2;
  return p.cross(a,b)*C + p.cross(b,c)*A + p.cross(c,a)
        *B > 0;
}
Q makeEdge(P orig, P dest) {
  Q r = H ? H : new Quad{new Quad{new Quad{new Quad{0}}
        }};
  H = r->o; r->r()->r() = r;
  rep(i,4) r = r->rot, r->p = arb, r->o = i & 1 ? r : r
        ->r();
  r->p = orig; r->F() = dest;
  return r;
}
void splice(Q a, Q b) {
  swap(a->o->rot->o, b->o->rot->o); swap(a->o, b->o);
}
Q connect(Q a, Q b) {
  Q q = makeEdge(a->F(), b->p);
  splice(q, a->next());
  splice(q->r(), b);
  return q;
}
pair<Q,Q> rec(const vector<P>& s) {
  if (sz(s) <= 3) {
    Q a = makeEdge(s[0], s[1]), b = makeEdge(s[1], s.
          back());
    if (sz(s) == 2) return { a, a->r() };
    splice(a->r(), b);
    auto side = s[0].cross(s[1], s[2]);
    Q c = side ? connect(b, a) : 0;
    return {side < 0 ? c->r() : a, side < 0 ? c : b->r
          () };
  }
#define H(e) e->F(), e->p
#define valid(e) (e->F().cross(H(base)) > 0)
  Q A, B, ra, rb;
  int half = sz(s) / 2;
  tie(ra, A) = rec({all(s) - half});
  tie(B, rb) = rec({sz(s) - half + all(s)});
  while ((B->p.cross(H(A)) < 0 && (A = A->next())) ||
         (A->p.cross(H(B)) > 0 && (B = B->r()->o)));
  Q base = connect(B->r(), A);
  if (A->p == ra->p) ra = base->r();
  if (B->p == rb->p) rb = base;
#define DEL(e, init, dir) Q e = init->dir; if (valid(e)
      ) \
    while (circ(e->dir->F(), H(base), e->F())) { \
      Q t = e->dir; \
      splice(e, e->prev()); \
      splice(e->r(), e->r()->prev()); \
      e->o = H; H = e; e = t; \
    }
  for (;;) {
    DEL(LC, base->r(), o);  DEL(RC, base, prev());
    if (!valid(LC) && !valid(RC)) break;
    if (!valid(LC) || (valid(RC) && circ(H(RC), H(LC)))
          )
      base = connect(RC, base->r());
    else
      base = connect(base->r(), LC->r());
  }
  return { ra, rb };
}
vector<P> triangulate(vector<P> pts) {
  sort(all(pts));  assert(unique(all(pts)) == pts.end()
        );
  if (sz(pts) < 2) return {};
  Q e = rec(pts).first;
  vector<Q> q = {e};
  int qi = 0;
  while (e->o->F().cross(e->F(), e->p) < 0) e = e->o;
#define ADD { Q c = e; do { c->mark = 1; pts.pb(c->p);
      \
  q.pb(c->r()); c = c->next(); } while (c != e); }
  ADD; pts.clear();
  while (qi < sz(q)) if (!(e = q[qi++])->mark) ADD;
  return pts;
}
```

8.5 3D

PolyhedronVolume.h

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

3058c3, 6 lines

```
template<class V, class L>
double signedPolyVolume(const V& p, const L& 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: $\mathcal{O}\left(n^2\right)$

"Point3D.h" ae1f07, 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.pb(f);
  };
  rep(i,4) fwd(j,i+1,4) fwd(k,j+1,4)
    mf(i, j, k, 6 - i - j - k);
  fwd(i,4,sz(A)) {
    rep(j,sz(FS)) {
      F f = FS[j];
      if(f.q.dot(A[i]) > f.q.dot(A[f.a])) {
        E(a,b).rem(f.c);
        E(a,c).rem(f.b);
        E(b,c).rem(f.a);
        swap(FS[j--], FS.back());
        FS.pop_back();
      }
    }
    int nw = sz(FS);
    rep(j,nw) {
      F f = FS[j];
#define C(a, b, c) if (E(a,b).cnt() != 2) mf(f.a, f.b, i, f.c);
      C(a, b, c); C(a, c, b); C(b, c, a);
    }
  }
  for (F& it : FS) if ((A[it.b] - A[it.a]).cross(
    A[it.c] - A[it.a]).dot(it.q) <= 0) swap(it.c, it.b);
  return FS;
};
```

sphericalDistance.h

Description: Returns the shortest distance on the sphere with radius radius between the points with azimuthal angles (longitude) f1 (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: $\mathcal{O}(n)$ e814aa, 15 lines

```
vi pi(const string& s) {
  vi p(sz(s));
  fwd(i, 1, sz(s)) {
    int g = p[i-1];
    while (g && s[i] != s[g]) g = p[g-1];
    p[i] = g + (s[i] == s[g]);
  }
  return p;
}
vi match(const string& s, const string& pat) {
  vi p = pi(pat + '\0' + s), res;
  fwd(i,sz(p)-sz(s),sz(p))
    if (p[i] == sz(pat)) res.pb(i - 2 * sz(pat));
  return res;
}
```

Zfunc.h

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

Time: $\mathcal{O}(n)$ 011e28, 14 lines

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

Manacher.h

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

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

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

ALCS.h

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

Time: $\mathcal{O}(nm)$ 8aadab, 58 lines

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

MainLorentz.h

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

Time: $\mathcal{O}(nlgn)$ 46fbbc, 46 lines

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

Lyndon.h

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

Time: $\mathcal{O}(n)$ 688c1c, 12 lines

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

MinRotation.h

Description: Finds the lexicographically smallest rotation of a string.

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

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

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

KMR.h

Description: KMR algorithm for lexical string comparison.

Time: $\mathcal{O}\left(n\log^2 n\right)$, but one of the logs is from std::sort a7962f, 40 lines

```
struct KMR {
  vector<vi> ids; KMR() {}
  // You can change str type to vi freely.
  KMR(const string& str) {
    ids.clear(); ids.pb(vi(all(str)));
    for (int h = 1; h <= sz(str); h *= 2) {
      vector<pair<pii, int>> tmp;
      rep(j, sz(str)) {
        int a = ids.back()[j], b = -1;
        if (j+h < sz(str)) b = ids.back()[j+h];
        tmp.pb({ {a, b}, j });
      }
      sort(all(tmp));
      ids.emplace_back(sz(tmp));
      int n = 0;
      rep(j, sz(tmp)) {
        if (j > 0 && tmp[j-1].st != tmp[j].st)
          n++;
        ids.back()[tmp[j].nd] = n;
      }
    }
  } // Get representative of [begin;end); O(1)
  pii get(int begin, int end) {
```

```
    if (begin >= end) return {0, 0};
    int k = __lg(end-begin);
    return {ids[k][begin], ids[k][end-(1<<k)]};
  } // Compare [b1;e1) with [b2;e2); O(1)
  // Returns -1 if <, 0 if ==, 1 if >
  int cmp(int b1, int e1, int b2, int e2) {
    int l1 = e1-b1, l2 = e2-b2;
    int l = min(l1, l2);
    pii x = get(b1, b1+l), y = get(b2, b2+l);
    if (x == y) return (l1 > l2) - (l1 < l2);
    return (x > y) - (x < y);
  } // Compute suffix array of string; O(n)
  vi sufArray() {
    vi sufs(sz(ids.back()));
    rep(i, sz(ids.back()))
      sufs[ids.back()[i]] = i;
    return sufs; } };
```

SuffixArray.h

Description: Builds suffix array for a string. `sa[i]` is the starting index of the suffix which is i'th in the sorted suffix array. The returned vector is of size $n+1$, and `sa[0] = n`. The `lcp` array contains longest common prefixes for neighbouring strings in the suffix array: `lcp[i] = lcp(sa[i], sa[i-1])`, `lcp[0] = 0`. The input string must not contain any zero bytes.
Time: $\mathcal{O}(n \log n)$ 9ff92c, 23 lines

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

SuffixTree.h

Description: Ukkonen's algorithm for online suffix tree construction. Each node contains indices [l, r) into the string, and a list of child nodes. Suffixes are given by traversals of this tree, joining [l, r) substrings. The root is 0 (has l = -1, r = 0), non-existent children are -1. To get a complete tree, append a dummy symbol – otherwise it may contain an incomplete path (still useful for substring matching, though).
Time: $\mathcal{O}(26N)$ f2f561, 50 lines

```
struct SuffixTree {
  enum { N = 200010, ALPHA = 26 }; // N ~ 2*maxlen+10
  int toi(char c) { return c - 'a'; }
  string a; // v = cur node, q = cur position
  int t[N][ALPHA],l[N],r[N],p[N],s[N],v=0,q=0,m=2;
  void ukkadd(int i, int c) { suff:
    if (r[v]<=q) {
      if (t[v][c]==-1) { t[v][c]=m;  l[m]=i;
        p[m++]=v; v=s[v]; q=r[v];  goto suff; }
      v=t[v][c]; q=l[v];
    }
    if (q==-1 || c==toi(a[q])) q++; else {
      l[m+1]=i;  p[m+1]=m;  l[m]=l[v];  r[m]=q;
      p[m]=p[v];  t[m][c]=m+1;  t[m][toi(a[q])]=v;
      l[v]=q;  p[v]=m;  t[p[m]][toi(a[l[m]])]=m;
      v=s[p[m]];  q=l[m];
      while (q<r[m]) { v=t[v][toi(a[q])];  q+=r[v]-l[v
          ]; }
      if (q==r[m])  s[m]=v;  else s[m]=m+2;
      q=r[v]-(q-r[m]);  m+=2;  goto suff;
    }
  }
  SuffixTree(string a) : a(a) {
    fill(r,r+N,sz(a));
    memset(s, 0, sizeof s);
    memset(t, -1, sizeof t);
    fill(t[1],t[1]+ALPHA,0);
    s[0] = 1; l[0] = l[1] = -1; r[0] = r[1] = p[0] = p
         [1] = 0;
    rep(i,sz(a)) ukkadd(i, toi(a[i]));
  }
  // example: find longest common substring (uses ALPHA
       = 28)
  pii best;
  int lcs(int node, int i1, int i2, int olen) {
    if (l[node] <= i1 && i1 < r[node]) return 1;
    if (l[node] <= i2 && i2 < r[node]) return 2;
    int mask = 0, len = node ? olen + (r[node] - l[node
         ]) : 0;
    rep(c,ALPHA) if (t[node][c] != -1)
      mask |= lcs(t[node][c], i1, i2, len);
    if (mask == 3)
      best = max(best, {len, r[node] - len});
    return mask;
  }
  static pii LCS(string s, string t) {
    SuffixTree st(s + (char)('z' + 1) + t + (char)('z'
         + 2));
    st.lcs(0, sz(s), sz(s) + 1 + sz(t), 0);
    return st.best;
  }
};
```

Hashing.h

Description: Self-explanatory methods for string hashing. 4e8ed3, 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 ~ 3e9; random
     also ok)
struct HashInterval {
  vector<H> ha, pw;
  HashInterval(string& str) : ha(sz(str)+1), pw(ha) {
    pw[0] = 1;
    rep(i,sz(str))
      ha[i+1] = ha[i] * C + str[i],
      pw[i+1] = pw[i] * C;
  }
  H hashInterval(int a, int b) { // hash [a, b)
    return ha[b] - ha[a] * pw[b - a];
  }
};
vector<H> getHashes(string& str, int length) {
  if (sz(str) < length) return {};
  H h = 0, pw = 1;
  rep(i,length)
    h = h * C + str[i], pw = pw * C;
  vector<H> ret = {h};
  fwd(i,length,sz(str)) {
    ret.pb(h = h * C + str[i] - pw * str[i-length]);
  }
  return ret;
}
H hashString(string& s){H h{}; for(char c:s) h=h*C+c;
     return h;}
```

AhoCorasick.h

Description: Aho-Corasick algorithm for linear-time multi-pattern matching. Self explanatory. Call Build after adding all patterns.
Time: $\mathcal{O}(26n)$ *build* e74d08, 47 lines

```
constexpr char AMIN = 'a'; // Smallest letter
constexpr int ALPHA = 26;  // Alphabet size
struct Aho {
  vector<array<int, ALPHA>> nxt{1};
  vi suf = {-1}, accLink = {-1};
  vector<vi> accept{1};
  // Add string with given ID to structure
  // Returns index of accepting node
  int add(const string& str, int id) {
    int i = 0;
    for (auto c : str) {
      if (!nxt[i][c-AMIN]) {
        nxt[i][c-AMIN] = sz(nxt);
        nxt.pb({}); suf.pb(-1);
        accLink.pb(1); accept.pb({});
      } // creates new node above
      i = nxt[i][c-AMIN];
    }
    accept[i].pb(id);
    return i;
  } // Build automata; time: O(V*ALPHA)
  void build() {
    queue<int> que;
    for (auto e : nxt[0]) if (e) {
      suf[e] = 0; que.push(e);
    }
    while (!que.empty()) {
      int i = que.front(), s = suf[i], j = 0;
      que.pop();
      for (auto &e : nxt[i]) {
        if (e) que.push(e);
        (e ? suf[e] : e) = nxt[s][j++];
      }
      accLink[i] = (accept[s].empty() ?
          accLink[s] : s);
    } // propagate link information above
  } // Append `c` to state `i`
  int next(int i, char c) {
    return nxt[i][c-AMIN];
  } // Call `f` for each pattern accepted
    // when in state `i` with its ID as argument.
    // Return true from `f` to terminate early.
    // Calls are in descreasing length order.
  void accepted(int i, auto f) {
    while (i != -1) {
      for (auto a :  accept[i]) if (f(a)) return;
      i = accLink[i]; } } };
```

PalindromicTree.h

Description: Computes plaindromic tree: for each end position in the string we store longest palindrome ending in that position. `link` is the suffix palindrome links, eg ababa -> aba. Can be used to compute shortest decomposition of strings to palindromes in O(n log n) time - use [DP] lines.
Time: $\mathcal{O}(N)$ eb3607, 38 lines

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

Monge.h

Description: Jak to czytasz to kurwa kaplica XD :prayge:
Time: ??? 1697b2, 158 lines

```
// NxN matrix A is simple (sub-)unit-Monge
// iff there exists a (sub-)permutation
// (N-1)x(N-1) matrix P such that:
//   A[x,y] = sum i>=x, j<y: P[i,j]
// The first column and last row are always 0.
// We represent these matrices implicitly
// using permutations p s.t. P[i,p(i)] = 1.
// (min, +) product of simple unit-Monge
// matrices represented by permutations P, Q,
// is also a simple unit-Monge matrix.
// The permutation that describes the product
// can be obtained by the following procedure:
// 1. Decompose P, Q into minimal sequences of
//    elementary transpositions.
// 2. Concatenate the transposition sequences.
// 3. Scan from left to right and remove
//    transpositions that decrease
//    inversion count (i.e. second crossings).
// 4. The reduced sequence represents result.
// Invert sub-permutation with values [0;n).
// Missing values should have value `def`.
vi invert(const vi& P, int n, int def) {
  vi ret(n, def);
  rep(i, sz(P)) if (P[i] != def)
    ret[P[i]] = i;
  return ret;
} // Split permutation P into half `lo`
// with values less than `k`, and half `hi`
// with remaining values, shifted by `k`.
// Missing rows from `lo` and `hi` are removed,
// original indices are in `loInd` and `hiInd`.
void split(const vi& P, int k, vi& lo, vi& hi,
           vi& loInd, vi& hiInd) {
  int i = 0;
  for (auto e : P) {
    if (e < k) lo.pb(e), loInd.pb(i++);
    else hi.pb(e-k), hiInd.pb(i++);
  }
} // Map sub-permutation into sub-permutation
// of length `n` on given indices sets.
vi expand(const vi& P, vi& ind1, vi& ind2,
          int n, int def) {
  vi ret(n, def);
  rep(k, sz(P)) if (P[k] != def)
    ret[ind1[k]] = ind2[P[k]];
  return ret;
} // Compute (min, +) product of square
// simple unit-Monge matrices given their
// permutation representations; time: O(n lg n)
// Permutation of second matrix is inverted!
//! Source: https://arxiv.org/pdf/0707.3619.pdf
vi comb(const vi& P, const vi& invQ) {
  int n = sz(P);
  if (n < 100) {
    // 5s -> 1s speedup for ALIS for n = 10^5
    vi ret = invert(P, n, -1);
    rep(i, sz(invQ)) {
      int from = invQ[i];
      rep(j, i) from += invQ[j] > invQ[i];
      for (int j = from; j > i; j--)
        if (ret[j-1] < ret[j])
          swap(ret[j-1], ret[j]);
    }
    return invert(ret, n, -1);
  }
  vi p1, p2, q1, q2, i1, i2, j1, j2;
  split(P, n/2, p1, p2, i1, i2);
  split(invQ, n/2, q1, q2, j1, j2);
  p1 = expand(comb(p1, q1), i1, j1, n, -1);
  p2 = expand(comb(p2, q2), i2, j2, n, n);
  q1 = invert(p1, n, -1);
  q2 = invert(p2, n, n);
```

```
  vi ans(n, -1);
  int delta = 0, j = n;
  rep(i, n) {
    ans[i] = (p1[i] < 0 ? p2[i] : p1[i]);
    while (j > 0 && delta >= 0)
      delta -= (q2[--j] < i || q1[j] >= i);
    if (p2[i] < j || p1[i] >= j)
      if (delta++ < 0)
        if (q2[j] < i || q1[j] >= i)
          ans[i] = j;
  }
  return ans;
} // Helper function for `mongeMul`.
void padPerm(const vi& P, vi& has, vi& pad,
             vi& ind, int n) {
  vector<bool> seen(n);
  rep(i, sz(P)) if (P[i] != -1) {
    ind.pb(i);
    has.pb(P[i]);
    seen[P[i]] = 1;
  }
  rep(i, n) if (!seen[i]) pad.pb(i);
} // Compute (min, +) product of
// simple sub-unit-Monge matrices given their
// permutation representations; time: O(n lg n)
// Left matrix has size sz(P) x sz(Q).
// Right matrix has size sz(Q) x n.
// Output matrix has size sz(P) x n.
// NON-SQUARE MATRICES ARE NOT TESTED!
//! Source: https://arxiv.org/pdf/0707.3619.pdf
vi mongeMul(const vi& P, const vi& Q, int n) {
  vi h1, p1, i1, h2, p2, i2;
  padPerm(P, h1, p1, i1, sz(Q));
  padPerm(invert(Q, n, -1), h2, p2, i2, sz(Q));
  h1.insert(h1.begin(), all(p1));
  h2.insert(h2.end(), all(p2));
  vi ans(sz(P), -1), tmp = comb(h1, h2);
  rep(i, sz(i1)) {
    int j = tmp[i+sz(p1)];
    if (j < sz(i2)) {
      ans[i1[i]] = i2[j];
    }
  }
  return ans;
}
// values must be small. If not small overflow
// scale yuor valoes
// Range Longest Increasing Subsequence Query;
// preprocessing: O(n lg^2 n), query: O(lg n)
// #include "../structures/wavelet_tree.h"
struct ALIS {
  WaveletTree tree;
  ALIS() {}
  // Precompute data structure; O(n lg^2 n)
  ALIS(const vi& seq) {
    vi P = build(seq);
    for (auto &k : P) if (k == -1) k = sz(seq);
    tree = {P, sz(seq)+1};
  }
  // Query LIS of s[b;e); time: O(lg n)
  int operator()(int b, int e) {
    return e - b -
      tree.count(b, sz(tree.seq[1]), 0, e);
  }
  vi build(const vi& seq) {
    int n = sz(seq);
    if (!n) return {};
    int lo = *min_element(all(seq));
    int hi = *max_element(all(seq));
    if (lo == hi) {
      vi tmp(n);
      iota(all(tmp), 1);
      tmp.back() = -1;
      return tmp;
    }
    int mid = (lo+hi+1) / 2;
    vi p1, p2, i1, i2;
    split(seq, mid, p1, p2, i1, i2);
    p1 = expand(build(p1), i1, i1, n, -1);
    p2 = expand(build(p2), i2, i2, n, -1);
    for(auto j : i1) p2[j] = j;
    for(auto j : i2) p1[j] = j;
    return mongeMul(p1, p2, n); } };
```

Various (10)

10.1 Intervals

IntervalContainer.h

Description: Add and remove intervals from a set of disjoint intervals. Will merge the added interval with any overlapping intervals in the set when adding. Intervals are [inclusive, exclusive).
Time: $\mathcal{O}(\log N)$ edce47, 23 lines

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

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: $\mathcal{O}(N \log N)$ a9491c, 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.pb(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

LIS.h

Description: Compute indices for the longest increasing subsequence.
Time: $\mathcal{O}(N \log N)$ a282a6, 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,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.eb(), 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: $\mathcal{O}(N \max(w_i))$ e74d03, 16 lines

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

FastMod.h

Description: Compute $a\%b$ about 5 times faster than usual, where b is constant but not known at compile time. Returns a value congruent to $a \pmod b$ in the range $[0, 2b)$. 751a02, 8 lines

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

BumpAllocator.h

Description: When you need to dynamically allocate many objects and don't care about freeing them. "new X" otherwise has an overhead of something like 0.05us + 16 bytes per allocation. 745db2, 8 lines

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

BumpAllocatorSTL.h

Description: BumpAllocator for STL containers.
Usage: vector<vector<int, small<int>>> ed(N); bb66d4, 14 lines

```
char buf[450 << 20] alignas(16);
size_t buf_ind = sizeof buf;
template<class T> struct small {
  typedef T value_type;
  small() {}
  template<class U> small(const U&) {}
  T* allocate(size_t n) {
    buf_ind -= n * sizeof(T);
    buf_ind &= 0 - alignof(T);
    return (T*)(buf + buf_ind);
  }
  void deallocate(T*, size_t) {}
};
```

FastInput.h

Description: Read an integer from stdin. Usage requires your program to pipe in input from file.
Usage: ./a.out < input.txt
Time: About 5x as fast as cin/scanf. 731361, 98 lines

```
inline char gc() { // like getchar()
  static char buf[1 << 16];
  static size_t bc, be;
  if (bc >= be) {
    buf[0] = 0, bc = 0;
    be = fread(buf, 1, sizeof(buf), stdin);
  }
  return buf[bc++]; // returns 0 on EOF
}
int readInt() {
  int a, c;
  while ((a = gc()) < 40);
  if (a == '-') return -readInt();
  while ((c = gc()) >= 48) a = a * 10 + c - 480;
  return a - 48;
}
//UWLIB Fast Input
#ifdef ONLINE_JUDGE
// write this when judge is on Windows
inline int getchar_unlocked() { return _getchar_nolock
     (); }
inline void putchar_unlocked(char c) { _putchar_nolock(
     c); }
#endif
// BEGIN HASH
int fastin() {
  int n = 0, c = getchar_unlocked();
  while(isspace(c))
    c = getchar_unlocked();
  while(isdigit(c)) {
    n = 10 * n + (c - '0');
    c = getchar_unlocked();
  }
  return n;
} // END HASH
// BEGIN HASH
int fastin_negative() {
  int n = 0, negative = false, c = getchar_unlocked();
  while(isspace(c))
    c = getchar_unlocked();
  if(c == '-') {
    negative = true;
    c = getchar_unlocked();
  }
  while(isdigit(c)) {
    n = 10 * n + (c - '0');
    c = getchar_unlocked();
  }
  return negative ? -n : n;
} // END HASH
// BEGIN HASH
double fastin_double() {
  double x = 0, t = 1;
  int negative = false, c = getchar_unlocked();
  while(isspace(c))
    c = getchar_unlocked();
  if (c == '-') {
    negative = true;
    c = getchar_unlocked();
  }
  while (isdigit(c)) {
    x = x * 10 + (c - '0');
    c = getchar_unlocked();
  }
  if (c == '.') {
    c = getchar_unlocked();
    while (isdigit(c)) {
      t /= 10;
      x = x + t * (c - '0');
      c = getchar_unlocked();
    }
  }
  return negative ? -x : x;
} // END HASH
// BEGIN HASH
void fastout(int x) {
  if(x == 0) {
    putchar_unlocked('0');
```

```
    putchar_unlocked(' ');
    return;
  }
  if(x < 0) {
    putchar_unlocked('-');
    x *= -1;
  }
  static char t[10];
  int i = 0;
  while(x) {
    t[i++] = char('0' + (x % 10));
    x /= 10;
  }
  while(--i >= 0)
    putchar_unlocked(t[i]);
  putchar_unlocked(' ');
}
void nl() { putchar_unlocked('\n'); }
// END HASH
```

Packing.h

Description: Packing. 03b70d, 63 lines

```
// Utilities for packing precomputed tables.
// Encodes 13 bits using two characters.
// Example usage:
//   Writer out;
//   out.ints(-123, 8);
//   out.done();
//   cout << out.buf;
struct Writer {
  string buf;
  int cur = 0, has = 0;
  void done() {
    buf.pb(char(cur%91 + 35));
    buf.pb(char(cur/91 + 35));
    cur = has = 0;
  }
  // Write unsigned b-bit integer.
  void intu(uint64_t v, int b) {
    assert(b == 64 || v < (1ull<<b));
    while (b--) {
      cur |= (v & 1) << has;
      if (++has == 13) done();
      v >>= 1;
    }
  }
  // Write signed b-bit integer (sign included)
  void ints(ll v, int b) {
    intu(v < 0 ? -v*2+1 : v*2, b);
  }
};
// Example usage:
//   Reader in("packed_data");
//   int firstValue = in.ints(8);
struct Reader {
  const char *buf;
  ll cur = 0;
  Reader(const char *s) : buf(s) {}
  // Read unsigned b-bit integer.
  uint64_t intu(int b) {
    uint64_t n = 0;
    rep(i, b) {
      if (cur < 2) {
        cur = *buf++ + 4972;
        cur += *buf++ * 91;
      }
      n |= (cur & 1) << i;
      cur >>= 1;
    }
    return n;
  }
  // Read signed b-bit integer (sign included).
  ll ints(int b) {
    auto v = intu(b);
    return (v%2 ? -1 : 1) * ll(v/2);
  }
};
```

HilbertMO.h

Description: Packing. 7646cf, 35 lines

```
// Modified MO's queries sorting algorithm,
// slightly better results than standard.
// Allows to process q queries in O(n*sqrt(q))
struct Query {
  int begin, end;
};
// Get point index on Hilbert curve
ll hilbert(int x, int y, int s, ll c = 0) {
  if (s <= 1) return c;
  s /= 2; c *= 4;
  if (y < s)
    return hilbert(x&(s-1), y, s, c+(x>=s)+1);
  if (x < s)
    return hilbert(2*s-y-1, s-x-1, s, c);
  return hilbert(y-s, x-s, s, c+3);
}
// Get good order of queries; time: O(n lg n)
vi moOrder(vector<Query>& queries, int maxN) {
  int s = 1;
  while (s < maxN) s *= 2;
  vector<ll> ord;
  for( auto &q : queries)
    ord.pb(hilbert(q.begin, q.end, s));
  vi ret(sz(ord));
  iota(all(ret), 0);
  sort(all(ret), [&](int l, int r) {
    return ord[l] < ord[r];
  });
  return ret;
}
```

Int128IO.h

Description: Packing. a481d3, 15 lines

```
istream& operator>>(istream& i, __int128& x) {
  char s[50], *p = s;
  for (i >> s, x = 0, p += *p < 48; *p;)
    x = x*10 + *p++ - 48;
  if (*s == 45) x = -x;
  return i;
}
// Note: Doesn't work for INT128_MIN!
ostream& operator<<(ostream& o, __int128 x) {
  if (x < 0) o << '-', x = -x;
  char s[50] = {}, *p = s+49;
  for (; x > 9; x /= 10) *--p = char(x%10+48);
  return o << ll(x) << p;
}
```

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

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: $\mathcal{O}\left((N + (hi - lo))\log N\right)$ f816e3, 18 lines

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

AliensTrick.h

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

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