University of Central Florida

UCF Guioco Piano

Lior Barak, Cameron Custer, Chris Gittings

2023 ICPC Southeast USA Regionals

2023

Contest (1)

template.cpp

14 lines

```
#include <bits/stdc++.h>
using namespace std;

#define rep(i, a, b) for(int i = a; i < (b); ++i)
#define all(x) begin(x), end(x)
#define sz(x) (int)(x).size()
typedef long long ll;
typedef pair<int, int> pii;
typedef vector<int> vi;

int main() {
  cin.tie(0)->sync_with_stdio(0);
  cin.exceptions(cin.failbit);
}
```

hash.sh

5 lines

```
# Hashes a file, ignoring all whitespace and comments. Use for
# verifying that code was correctly typed.
# compile by typing in a hash.sh file and run cmd: chmod +x ./
    hash.sh
# execute with ./hash.sh < template.cpp
cpp -dD -P -fpreprocessed | tr -d '[:space:]'| md5sum |cut -c-6
```

Data structures (2)

DSU.h

Description: Disjoint-set data structure.
Time: $\mathcal{O}(\alpha(N))$

95dc47, 14 lines

```
struct UF {
  vi e;
  UF(int n) : e(n, -1) {}
  bool same(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;
  }
};
```

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

de4ad0, 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.push_back({a, e[a]});
    st.push_back({b, e[b]});
    e[a] += e[b]; e[b] = a;
    return true;
  }
};
```

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. Works for (Ranges Updates and Point Queries) OR (Point Updates and Range Queries). For range queries use `query(end) - query(start - 1)`. For interval updates, `update(start, +1)` and `update(end + 1, -1)`.
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" 157f07, 22 lines

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

RMQ.h

Description: Constant time subarray min/max queries for a fixed array. Queries are inclusive-exclusive.
Time: $\mathcal{O}(nlogn)$ initialization and $\mathcal{O}(1)$ queries.

c0579a, 14 lines

```
template<class T, class C = less<T>> struct RMQ {
  vector<vector<T>> t;
  C cmp;
  RMQ(vector<T>& a, C cmp = C{}):
    t(__lg(sz(a)) + 1, a), cmp(cmp) {
    for (int k = 1, len = 1; k < sz(t); k++, len <<= 1)
      for (int i = 0; i + 2 * len - 1 < sz(a); i++)
        t[k][i] = min(t[k - 1][i], t[k - 1][i + len], cmp);
  }
  T query(int a, int b) { // inclusive-exclusive [a, b)
    int k = __lg(b - a), len = 1 << k;
    return min(t[k][a], t[k][b - len], cmp);
  }
};
```

HashMap.h

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

a04980, 12 lines

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

OrderStatisticTree.h

Description: A set (not multiset!) with support for finding the n'th element, and finding the index of an element. To get a map, change `null_type`.
Time: $\mathcal{O}(\log N)$

782797, 15 lines

```
#include <bits/extc++.h>
using namespace __gnu_pbds;
template<class T>
using Tree = tree<T, null_type, less<T>, rb_tree_tag,
  tree_order_statistics_node_update>;
void example() {
  Tree<int> t, t2;
  t.insert(8);
  auto it = t.insert(10).first;
  assert(it == t.lower_bound(9));
  assert(t.order_of_key(10) == 1);
  assert(t.order_of_key(11) == 2);
  assert(*t.find_by_order(0) == 8);
  t.join(t2); // assuming T < T2 or T > T2, merge t2 into t
}
```

IterativeSegmentTree.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, 20 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);
  }
};
```

LazyIterativeSegTree.h

fcc5f1, 56 lines

```
template<class T, T (*e)(), T (*op)(T, T), class F, F (*id)(),
  T (*onto)(F, T), F (*comp)(F, F)>
struct lazy_segtree {
  int N, log, S;
  vector<T> d;
  vector<F> lz;
  lazy_segtree(const vector<T>& v):
    N(sz(v)), log(__lg(2 * N - 1)), S(1 << log), d(2 * S, e()),
    lz(S, id()) {
    for (int i = 0; i < N; i++) d[S + i] = v[i];
    for (int i = S - 1; i >= 1; i--) pull(i);
  }
  void apply(int k, F f) {
    d[k] = onto(f, d[k]);
    if (k < S) lz[k] = comp(f, lz[k]);
  }
  void push(int k) {
    apply(2 * k, lz[k]), apply(2 * k + 1, lz[k]), lz[k] = id();
  }
  void push(int l, int r) {
    int zl = __builtin_ctz(l), zr = __builtin_ctz(r);
    for (int i = log; i > min(zl, zr); i--) {
      if (i > zl) push(l >> i);
      if (i > zr) push((r - 1) >> i);
    }
  }
  void pull(int k) { d[k] = op(d[2 * k], d[2 * k + 1]); }
  void set(int p, T x) {
    p += S;
    for (int i = log; i >= 1; i--) push(p >> i);
    for (d[p] = x; p /= 2;) pull(p);
  }
  T query(int l, int r) {
    if (l == r) return T{};
    push(l += S, r += S);
    T vl = e(), vr = e();
    for (; l < r; l /= 2, r /= 2) {
      if (l & 1) vl = op(vl, d[l++]);
      if (r & 1) vr = op(d[--r], vr);
    }
    return op(vl, vr);
  }
  void update(int l, int r, F f) {
    if (l == r) return;
    push(l += S, r += S);
    for (int a = l, b = r; a < b; a /= 2, b /= 2) {
      if (a & 1) apply(a++, f);
      if (b & 1) apply(--b, f);
    }
    int zl = __builtin_ctz(l), zr = __builtin_ctz(r);
    for (int i = min(zl, zr) + 1; i <= log; i++) {
      if (i > zl) pull(l >> i);
      if (i > zr) pull((r - 1) >> i);
    }
  }
};
```

SegTreeWalk.h

8f9536, 40 lines

```
// hash = 8f9536
template<class G> int max_right(int l, G g) { // bool g(T);
  assert(g(e()));
  if (l == N) return N;
  l += S;
  for (int i = log; i >= 1; i--) push(l >> i);
  T sm = e();
  do {
    while (l % 2 == 0) l >>= 1;
    if (!g(op(sm, d[l]))) {
      while (l < S) {
        push(l), l *= 2;
        if (g(op(sm, d[l]))) sm = op(sm, d[l++]);
      }
      return l - S;
    }
    sm = op(sm, d[l++]);
  } while ((l & -l) != l);
  return N;
}
template<class G> int min_left(int r, G g) {
  assert(g(e()));
  if (r == 0) return 0;
  r += S;
  for (int i = log; i >= 1; i--) push((r - 1) >> i);
  T sm = e();
  do {
    r--;
    while (r > 1 && (r % 2)) r >>= 1;
    if (!g(op(d[r], sm))) {
      while (r < S) {
        push(r), r *= 2;
        if (g(op(d[++r], sm))) sm = op(d[r--], sm);
      }
      return r + 1 - S;
    }
    sm = op(d[r], sm);
  } while ((r & -r) != r);
  return 0;
}
```

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;
  }
};
```

MonotonicQueue.h

Description: Queue that maintains its minimum/maximum element.
Usage: Works exactly like std::queue;
monotonic_queue<T> gives a min queue,
and monotonic_queue<T, greater<T>> gives a max queue.
Time: Amortized $\mathcal{O}(1)$ for push(), true $\mathcal{O}(1)$ for pop()/min().

e46cb6, 23 lines

```
template<class T, class Compare = less<T>>
struct monotonic_queue: queue<T> {
  using q = queue<T>;
  deque<T> mq;
  Compare cmp;
  const T& min() { return assert(!q::empty()), mq.front(); }
  void update() {
    while (!mq.empty() && cmp(q::back(), mq.back()))
      mq.pop_back();
    mq.push_back(q::back());
  }
  void pop() {
    assert(!q::empty());
    if (!mq.empty() && !cmp(mq.front(), q::front()))
      mq.pop_front();
```

```
    q::pop();
  }
  void push(const T& val) { queue<T>::push(val), update(); }
  void push(T&& val) { queue<T>::push(val), update(); }
  template<class... Args> void emplace(Args&&... args) {
    q::emplace(args...), update();
  }
};
```

SplayTree.h

14f6b8, 76 lines

```
// https://codeforces.com/blog/entry/75885
struct SplayTree {
  struct Node {
    int ch[2] = {0, 0}, p = 0, cnt = 1;
    ll self = 0, path = 0; // Path aggregates
    ll sub = 0, vir = 0; // Subtree aggregates
    bool flip = 0; // Lazy tags
  };
  vector<Node> T;
  SplayTree(int n): T(n + 1) { T[0].cnt = 0; }
  void push(int x) {
    if (!x || !T[x].flip) return;
    int l = T[x].ch[0], r = T[x].ch[1];
    T[l].flip ^= 1, T[r].flip ^= 1;
    swap(T[x].ch[0], T[x].ch[1]);
    T[x].flip = 0;
  }
  void pull(int x) {
    int l = T[x].ch[0], r = T[x].ch[1];
    push(l);
    push(r);
    T[x].path = T[l].path + T[x].self + T[r].path;
    T[x].sub = T[x].vir + T[l].sub + T[r].sub + T[x].self;
    T[x].cnt = T[l].cnt + 1 + T[r].cnt;
  }
  void set(int x, int d, int y) {
    T[x].ch[d] = y;
    T[y].p = x;
    pull(x);
  }
  void splay(int x) {
    auto dir = [&](int x) {
      int p = T[x].p;
      if (!p) return -1;
      return T[p].ch[0] == x ? 0 : T[p].ch[1] == x ? 1 : -1;
    };
    auto rotate = [&](int x) {
      int y = T[x].p, z = T[y].p, dx = dir(x), dy = dir(y);
      set(y, dx, T[x].ch[!dx]);
      set(x, !dx, y);
      if (~dy) set(z, dy, x);
      T[x].p = z;
    };
    for (push(x); ~dir(x);) {
      int y = T[x].p, z = T[y].p;
      push(z);
      push(y);
      push(x);
      int dx = dir(x), dy = dir(y);
      if (~dy) rotate(dx != dy ? x : y);
      rotate(x);
    }
  }
  // TEST THIS STUFF
  int kth(int x, int k) {
    if (!x) return 0;
    k = min(max(k, 0), T[x].cnt - 1);
    for (int l = T[x].ch[0]; T[l].cnt != k; l = T[x].cnt[0])
      if (T[l].cnt < k) x = l;
      else k -= T[l].cnt + 1, x = T[x].ch[1];
    return splay(x), x;
  }
  // k = size of left subtree
  pair<int, int> split(int x, int k) {
    if (!x || k <= 0) return {0, x};
    if (k >= T[x].cnt) return {x, 0};
    int r = kth(x, k), l = T[r].ch[0];
    T[r].ch[0] = 0, T[l].p = 0;
    return {l, r};
  }
  int merge(int l, int r) {
    if (!l || !r) return !l ? r : l;
    r = kth(r, 0), set(r, 0, l);
    return r;
  }
};
```

LinkCutTree.h

98a056, 55 lines

```
// https://codeforces.com/blog/entry/75885
struct LinkCut: SplayTree {
  LinkCut(int n): SplayTree(n) {}
  int access(int x) {
    int u = x, v = 0;
    for (; u; v = u, u = T[u].p) {
      splay(u);
      int& ov = T[u].ch[1];
      T[u].vir += T[ov].sub;
      T[u].vir -= T[v].sub;
      ov = v;
      pull(u);
    }
    return splay(x), v;
  }
  void reroot(int x) {
    access(x);
    T[x].flip ^= 1;
    push(x);
  }
  void Link(int u, int v) {
    reroot(u), access(v);
    T[v].vir += T[u].sub;
    T[u].p = v;
    pull(v);
  }
  void Cut(int u, int v) {
    reroot(u), access(v);
    T[v].ch[0] = T[u].p = 0;
    pull(v);
  }
  // Rooted tree LCA. Returns 0 if u and v arent connected.
  int LCA(int u, int v) {
    if (u == v) return u;
    access(u);
    int ret = access(v);
    return T[u].p ? ret : 0;
  }
  // Query subtree of u where v is outside the subtree.
  ll Subtree(int u, int v) {
    reroot(v), access(u);
    return T[u].vir + T[u].self;
  }
  // Query path [u..v]
  ll Path(int u, int v) {
    reroot(u), access(v);
    return T[v].path;
  }
  // Update vertex u with value v
  void Update(int u, ll v) {
    access(u);
    T[u].self = v;
    pull(u);
  }
};
```

Treaps.h

e71bf7, 65 lines

```
struct node {
  int pri, sz = 1;
  ll val, sum, min, lazy = 0;
  node *l = NULL, *r = NULL;
  node(ll v): val(v), sum(v), min(v) { pri = rand(); }
};
int getSz(node *in) { return in ? in->sz : 0; }
ll getSum(node *in) { return in ? in->sum : 0; }
ll getMin(node *in) { return in ? in->min : 4e18; }
void pushNode(node *in, ll val) {
  in->sum += val;
  in->min += val;
  in->lazy += val;
}
void push(node *in) {
  if (in->l) pushNode(in->l, in->lazy);
  if (in->r) pushNode(in->r, in->lazy);
  in->lazy = 0;
}
void pull(node *in) {
  in->sz = getSz(in->l) + 1 + getSz(in->r);
  in->sum = getSum(in->l) + in->val + getSum(in->r);
  in->min = min(getMin(in->l), min(in->val, getMin(in->r)));
}
pair<node *, node *> split(node *in, int leftIdx) {
  if (!in) return {NULL, NULL};
  push(in);
  if (leftIdx < getSz(in->l)) {
    auto [lLeft, lRight] = split(in->l, leftIdx);
    in->l = lRight;
    pull(in);
    return {lLeft, in};
  }
  auto [rLeft, rRight] =
    split(in->r, leftIdx - (getSz(in->l) + 1));
  in->r = rLeft;
  pull(in);
  return {in, rRight};
}
node *merge(node *a, node *b) {
  if (!a) return b;
  if (!b) return a;
  push(a), push(b);
  if (a->pri < b->pri) {
    a->r = merge(a->r, b);
    pull(a);
    return a;
  }
  b->l = merge(a, b->l);
  pull(b);
  return b;
}
ll getSum(node *in, int l, int r) {
  auto [rest, right] = split(in, r);
  auto [left, root] = split(rest, l - 1);
  ll out = getSum(root);
  in = merge(left, merge(root, right));
  return out;
}
```

```
void add(node *in, int l, int r, ll val) {
  auto [rest, right] = split(in, r);
  auto [left, root] = split(rest, l - 1);
  pushNode(root, val);
  in = merge(merge(left, root), right);
}
```

Geometry (3)

3.1 Geometric primitives

Point.h

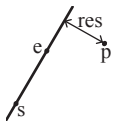
Description: Class to handle points in the plane. T can be e.g. double or long long. (Avoid int.)

47ec0a, 35 lines

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

lineDistance.h

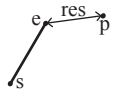
Description:
Returns the signed distance between point p and the line containing points a and b. Positive value on left side and negative on right as seen from a towards b. a==b gives nan. P is supposed to be Point<T> or Point3D<T> where T is e.g. double or long long. It uses products in intermediate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



"Point.h" f6bf6b, 4 lines

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

SegmentDistance.h



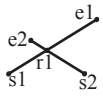
Description:
Returns the shortest distance between point p and the line segment from point s to e.
Usage: `Point<double> a, b(2,2), p(1,1);`
`bool onSegment = segDist(a,b,p) < 1e-10;`

"Point.h" 5c88f4, 7 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.



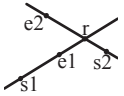
Usage: `vector<P> inter = segInter(s1,e1,s2,e2);`
`if (sz(inter)==1)`
`cout << "segments intersect at " << inter[0] << endl;`

"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 $\Leftrightarrow$ left/on line/right. If the optional argument *eps* is given 0 is returned if p is within distance *eps* from the line. P is supposed to be Point<T> where T is e.g. double or long long. It uses products in intermediate steps so watch out for overflow if using int or long long.
Usage: `bool left = sideOf(p1,p2,q)==1;`

"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

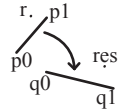
Description: Returns true iff p lies on the line segment from s to e. Use `(segDist(s,e,p)<=epsilon)` instead when using Point<double>.

"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, 7 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, 6 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,0,n) { while (v[j] < v[i].t180()) ++j; }`
`// sweeps j such that (j-i) represents the number of positively oriented triangles with vertices at 0 and i`

0f0602, 38 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)};
}
```

angleCmp.h

Description: Useful utilities for dealing with angles of rays from origin. OK for integers, only uses cross product. Doesn't support (0,0).

6edd25, 18 lines

```
template<class P> bool sameDir(P s, P t) {
  return s.cross(t) == 0 && s.dot(t) > 0;
}
// checks 180 <= s..t < 360?
template<class P> bool isReflex(P s, P t) {
  auto c = s.cross(t);
  return c ? (c < 0) : (s.dot(t) < 0);
}
// operator< (s,t) for angles in [base,base+2pi)
template<class P> bool angleCmp(P base, P s, P t) {
  int r = isReflex(base, s) - isReflex(base, t);
  return r ? (r < 0) : (0 < s.cross(t));
}
// is x in [s,t] taken ccw? 1/0/-1 for in/border/out
template<class P> int angleBetween(P s, P t, P x) {
  if (sameDir(x, s) || sameDir(x, t)) return 0;
  return angleCmp(s, x, t) ? 1 : -1;
}
```

3.2 Circles

CircleIntersection.h

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

"Point.h" b84f93, 14 lines

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

CircleTangents.h

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

"Point.h" b0153d, 13 lines

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

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: $\mathcal{O}(n)$

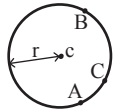
"../../content/geometry/Point.h" a1ee63, 21 lines

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

circumcircle.h

Description:

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



"Point.h" 1caa3a, 10 lines

```
typedef Point<double> P;
double ccRadius(const P& A, const P& B, const P& C) {
  return (B - A).dist() * (C - B).dist() * (A - C).dist() /
    abs((B - A).cross(C - A)) / 2;
}
P ccCenter(const P& A, const P& B, const P& C) {
  P b = C - A, c = B - A;
  return A +
    (b * c.dist2() - c * b.dist2()).perp() / b.cross(c) / 2;
}
```

MinimumEnclosingCircle.h

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

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

"circumcircle.h" 09dd0a, 17 lines

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

3.3 Polygons

InsidePolygon.h

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

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

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

"Point.h", "OnSegment.h", "SegmentDistance.h" 2bf504, 12 lines

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

PolygonArea.h

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

"Point.h" f12300, 5 lines

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

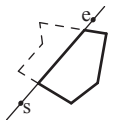
PolygonCenter.h

Description: Returns the center of mass for a polygon.
Time: $\mathcal{O}(n)$

"Point.h" 9706dc, 10 lines

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

PolygonCut.h



Description:
Returns a vector with the vertices of a polygon with everything to the left of the line going from s to e cut away.
Usage: `vector<P> p = ...;`
`p = polygonCut(p, P(0,0), P(1,0));`

"Point.h", "lineIntersection.h" f2b7d4, 12 lines

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

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" 3931c6, 36 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, 0, sz(poly)) rep(v, 0, 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, 0, sz(poly)) if (i != j) {
      rep(u, 0, 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.emplace_back(sa / (sa - sb), sgn(sc - sd));
        } else if (
          !sc && !sd && j < i && sgn((B - A).dot(D - C)) > 0) {
          segs.emplace_back(rat(C - A, B - A), 1);
          segs.emplace_back(rat(D - A, B - A), -1);
        }
      }
    }
    sort(all(segs));
    for (auto& s : segs) s.first = min(max(s.first, 0.0), 1.0);
    double sum = 0;
    int cnt = segs[0].second;
    rep(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 counterclockwise order. Points on the edge of the hull between two other points are not considered part of the hull.
Time: $\mathcal{O}(n \log n)$

"Point.h" 310954, 14 lines

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

HullDiameter.h

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

"Point.h" c571b8, 11 lines

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

PointInsideHull.h

Description: Determine whether a point t lies inside a convex hull (CCW order, with no collinear points). Returns true if point lies within the hull. If strict is true, points on the boundary aren't included.
Time: $\mathcal{O}(\log N)$

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

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

LineHullIntersection.h

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

"Point.h" e5fd61, 38 lines

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

halfPlane.h

Description: Halfplane intersection area

"Point.h", "lineIntersection.h" 5aff1a, 64 lines

```
#define eps 1e-8
typedef Point<double> P;
```

```
struct Line {
  P P1, P2;
  // Right hand side of the ray P1 -> P2
  explicit Line(P a = P(), P b = P()): P1(a), P2(b){};
  P intpo(Line y) {
    P r;
    assert(lineIntersection(P1, P2, y.P1, y.P2, r) == 1);
    return r;
  }
  P dir() { return P2 - P1; }
  bool contains(P x) { return (P2 - P1).cross(x - P1) < eps; }
  bool out(P x) { return !contains(x); }
};
template<class T> bool mycmp(Point<T> a, Point<T> b) {
  // return atan2(a.y, a.x) < atan2(b.y, b.x);
  if (a.x * b.x < 0) return a.x < 0;
  if (abs(a.x) < eps) {
    if (abs(b.x) < eps) return a.y > 0 && b.y < 0;
    if (b.x < 0) return a.y > 0;
    if (b.x > 0) return true;
  }
  if (abs(b.x) < eps) {
    if (a.x < 0) return b.y < 0;
    if (a.x > 0) return false;
  }
  return a.cross(b) > 0;
}
bool cmp(Line a, Line b) { return mycmp(a.dir(), b.dir()); }
double Intersection_Area(vector<Line> b) {
  sort(b.begin(), b.end(), cmp);
  int n = b.size();
  int q = 1, h = 0, i;
  vector<Line> c(b.size() + 10);
  for (i = 0; i < n; i++) {
    while (q < h && b[i].out(c[h].intpo(c[h - 1]))) h--;
    while (q < h && b[i].out(c[q].intpo(c[q + 1]))) q++;
    c[++h] = b[i];
    if (q < h && abs(c[h].dir().cross(c[h - 1].dir())) < eps) {
      if (c[h].dir().dot(c[h - 1].dir()) > 0) {
        h--;
        if (b[i].out(c[h].P1)) c[h] = b[i];
      } else {
        // The area is either 0 or infinite.
        // If you have a bounding box, then the area is
             definitely 0.
        return 0;
      }
    }
  }
  while (q < h - 1 && c[q].out(c[h].intpo(c[h - 1]))) h--;
  while (q < h - 1 && c[h].out(c[q].intpo(c[q + 1]))) q++;
  // Intersection is empty. This is sometimes different from
       the case when
  // the intersection area is 0.
  if (h - q <= 1) return 0;
  c[h + 1] = c[q];
  vector<P> s;
  for (i = q; i <= h; i++) s.push_back(c[i].intpo(c[i + 1]));
  s.push_back(s[0]);
  double ans = 0;
  for (i = 0; i < (int)s.size() - 1; i++)
    ans += s[i].cross(s[i + 1]);
  return ans / 2;
}
```

mitlineIntersection.h

Description: If a unique intersection point of the lines going through s1,e1 and s2,e2 exists r is set to this point and 1 is returned. If no intersection point exists 0 is returned and if infinitely many exists -1 is returned. If s1==e1 or s2==e2 -1 is returned. The wrong position will be returned if P is Point<int> and the intersection point does not have integer coordinates. Products of three coordinates are used in intermediate steps so watch out for overflow if using int or long long.
Usage: point<double> intersection;
if (1 == LineIntersection(s1,e1,s2,e2,intersection))
cout << "intersection point at " << intersection << endl;

"Point.h" aa1f17, 10 lines

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

PlanarFaceExtraction.h

Description: Takes a bunch of points and adjacency array. No lines formed by adjacent points can cross! Returns an array list of polygons formed by these points and adjs No two points can be the same. Points will be assigned IDs in order given. Will not form polygons with holes (there may be nested polygons you need to check for)

"Point.h" 72d650, 74 lines

```
template<class P> struct Edge {
  int id;
  P a, b, ab;
  Edge *rev, *prev;
  bool used, isBorder;
  Edge(P a, P b):
    id(0), a(a), b(b), ab(b - a), rev(NULL), prev(NULL),
    used(0), isBorder(0) {}
  friend ostream &operator<<(ostream &os, Edge e) {
    return os << e.id;
  }
};
// Takes a bunch of points and adjacency array. No lines formed
     by adjacent points can cross!
// Returns an array list of polygons formed by these points and
     adjs
// No two points can be the same. Points will be assigned IDs
    in order given.
// Will not form polygons with holes (there may be nested
     polygons you need to check for)
// O(v + m log m)
template<class P>
vector<vector<Edge<P> *>> extractPolygons(vector<P> &points,
  vector<vi> &adjs) {
  using Edge = Edge<P>;
  int n = sz(points),
      curEId = 0; // # of poly-poly edges; can keep global
  vector<vector<Edge *>> edges(n);
  vi idxs(n);
  rep(i, 0, n) edges[i].resize(sz(adjs[i]));
  for (int i = 0; i < n; i++) {
    P p = points[i];
    for (int next : adjs[i]) {
      if (next < i) continue;
      P q = points[next];
      Edge *a = new Edge(p, q), *b = new Edge(q, p);
      a->id = b->id = curEId++;
      edges[i][idxs[i]++] = b->rev = a;
      edges[next][idxs[next]++] = a->rev = b;
    }
  }
  rep(i, 0, n) {
    int len = sz(edges[i]);
    sort(all(edges[i]), [&](auto ea, auto eb) {
      // or another more stable radial sort of your choosing
      return atan2l(ea->ab.y, ea->ab.x) <
        atan2l(eb->ab.y, eb->ab.x);
    });
    rep(j, 0, len) edges[i][(j + 1) % len]->prev = edges[i][j];
  }
  vector<vector<Edge *>> polys;
  for (int i = 0; i < n; i++) {
    P cur = points[i];
    for (Edge *e : edges[i]) {
      if (e->used) continue;
      e->used = true;
      vector<Edge *> edgeList{e};
      cur = e->b;
      while (true) {
        e = e->rev->prev;
        if (e->used) break;
        e->used = true;
        edgeList.pb(e);
        cur = e->b;
      }
      polys.pb({edgeList});
    }
  }
  vector<vector<Edge *>> res;
  for (vector<Edge *> &p : polys) {
    ld a = 0;
    for (Edge *e : p) a = a + e->a.cross(e->b);
    if (a >= 0) res.pb(p); // Normal polygon (maybe 0 area)
    else // Else, this the border polygon (vs in reverse order)
      for (Edge *e : p) e->isBorder = true;
  }
  return res;
}
```

3.4 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" df6f59, 26 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, 0, 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.push_back({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;
}
```

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" 551b62, 98 lines

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

3.5 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, 43 lines

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

3dHull.h

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

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

"Point3D.h" 95372a, 49 lines

```
typedef Point3D<double> P3;
struct PR {
```

```
  void ins(int x) { (a == -1 ? a : b) = x; }
  void rem(int x) { (a == x ? a : b) = -1; }
  int cnt() { return (a != -1) + (b != -1); }
  int a, b;
};
struct F {
  P3 q;
  int a, b, c;
};
vector<F> hull3d(const vector<P3>& A) {
  assert(sz(A) >= 4);
  vector<vector<PR>> E(sz(A), vector<PR>(sz(A), {-1, -1}));
#define E(x, y) E[f.x][f.y]
  vector<F> FS;
  auto mf = [&](int i, int j, int k, int l) {
    P3 q = (A[j] - A[i]).cross((A[k] - A[i]));
    if (q.dot(A[l]) > q.dot(A[i])) q = q * -1;
    F f{q, i, j, k};
    E(a, b).ins(k), E(a, c).ins(j), E(b, c).ins(i);
    FS.push_back(f);
  };
  rep(i, 0, 4) rep(j, i + 1, 4) rep(k, j + 1, 4)
    mf(i, j, k, 6 - i - j - k);
  rep(i, 4, sz(A)) {
    rep(j, 0, sz(FS)) {
      F f = FS[j];
      if (f.q.dot(A[i]) > f.q.dot(A[f.a])) {
        E(a, b).rem(f.c), E(a, c).rem(f.b), E(b, c).rem(f.a);
        swap(FS[j--], FS.back());
        FS.pop_back();
      }
    }
    int nw = sz(FS);
    rep(j, 0, nw) {
      F f = FS[j];
#define C(a, b, c) \
  if (E(a, b).cnt() != 2) mf(f.a, f.b, i, f.c);
      C(a, b, c), C(a, c, b), C(b, c, a);
    }
  }
  for (F& it : FS)
    if (
      (A[it.b] - A[it.a]).cross(A[it.c] - A[it.a]).dot(it.q) <=
      0)
      swap(it.c, it.b);
  return FS;
};
```

sphericalDistance.h

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

611f07, 8 lines

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

Graphs (4)

Euler's planar graph formula: $V - E + F - C = 1$

4.1 Fundamentals

BellmanFord.h

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

830a8f, 23 lines

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

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

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

FloydWarshall.h

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

531245, 12 lines

```
const ll inf = 1LL << 62;
void floydWarshall(vector<vector<ll>>& m) {
  int n = sz(m);
  rep(i,0,n) m[i][i] = min(m[i][i], 0LL);
  rep(k,0,n) rep(i,0,n) rep(j,0,n)
    if (m[i][k] != inf && m[k][j] != inf) {
      auto newDist = max(m[i][k] + m[k][j], -inf);
      m[i][j] = min(m[i][j], newDist);
    }
  rep(k,0,n) if (m[k][k] < 0) rep(i,0,n) rep(j,0,n)
    if (m[i][k] != inf && m[k][j] != inf) m[i][j] = -inf;
}
```

4.2 DFS Algorithsm

SCCTarjan.h

Description: Finds strongly connected components of a directed graph. Visits/indexes SCCs in reverse topological order.
Usage: `scc(graph)` returns an array that has the ID of each node's SCC. `scc(graph, [&](vector<int>& v) { ... })` calls the lambda on each SCC, and returns the same array. Multiple edges are not allowed.
Time: $\mathcal{O}(|V| + |E|)$

358d18, 35 lines

```
namespace SCCTarjan {
vector<int> val, comp, z, cont;
int Time, ncomps;
template<class G, class F> int dfs(int j, G& g, F& f) {
  int low = val[j] = ++Time, x;
  z.push_back(j);
  for (auto e : g[j])
    if (comp[e] < 0) low = min(low, val[e] ?: dfs(e, g, f));
  if (low == val[j]) {
    do {
      x = z.back();
      z.pop_back();
      comp[x] = ncomps;
      cont.push_back(x);
    } while (x != j);
    f(cont);
    cont.clear();
    ncomps++;
  }
  return val[j] = low;
}
template<class G, class F> vector<int> scc(G& g, F f) {
  int n = g.size();
  val.assign(n, 0);
  comp.assign(n, -1);
  Time = ncomps = 0;
  for (int i = 0; i < n; i++)
    if (comp[i] < 0) dfs(i, g, f);
  return comp;
}
template<class G> // convenience function w/o lambda
vector<int> scc(G& g) {
  return scc(g, [](auto& v) {});
}
} // namespace SCCTarjan
```

2sat.h

Description: Calculates a valid assignment to boolean variables a, b, c,... to a 2-SAT problem, so that an expression of the type $(a||b)\&\&(!a||c)\&\&(d||!b)\&\&\ldots$ becomes true, or reports that it is unsatisfiable. Negated variables are represented by bit-inversions (~x).
Usage:
```
TwoSat ts(number of boolean variables);
ts.either(0, ~3); // Var 0 is true or var 3 is false
ts.setValue(2); // Var 2 is true
ts.atMostOne({0,~1,2}); // <= 1 of vars 0, ~1 and 2 are true
ts.solve(); // Returns true iff it is solvable
ts.values[0..N-1] holds the assigned values to the vars
```
Time: $\mathcal{O}(N + E)$, where N is the number of boolean variables, and E is the number of clauses.

0e20f5, 50 lines

```
struct TwoSat {
  int N;
  vector<vi> gr;
  vi values; // 0 = false, 1 = true
  TwoSat(int n = 0): N(n), gr(2 * n) {}
  int addVar() { // (optional)
    gr.emplace_back();
    gr.emplace_back();
    return N++;
  }
  void either(int f, int j) {
    f = max(2 * f, -1 - 2 * f);
    j = max(2 * j, -1 - 2 * j);
    gr[f].push_back(j ^ 1);
    gr[j].push_back(f ^ 1);
  }
  void setValue(int x) { either(x, x); }
  void atMostOne(const vi& li) { // (optional)
    if (sz(li) <= 1) return;
    int cur = ~li[0];
    rep(i, 2, sz(li)) {
      int next = addVar();
      either(cur, ~li[i]);
      either(cur, next);
```

```
      either(~li[i], next);
      cur = ~next;
    }
    either(cur, ~li[1]);
  }
  vi val, comp, z;
  int time = 0;
  int dfs(int i) {
    int low = val[i] = ++time, x;
    z.push_back(i);
    for (int e : gr[i])
      if (!comp[e]) low = min(low, val[e] ?: dfs(e));
    if (low == val[i]) do {
        x = z.back(), z.pop_back();
        comp[x] = low;
        if (values[x >> 1] == -1) values[x >> 1] = x & 1;
      } while (x != i);
    return val[i] = low;
  }
  bool solve() {
    values.assign(N, -1), val.assign(2 * N, 0), comp = val;
    rep(i, 0, 2 * N) if (!comp[i]) dfs(i);
    rep(i, 0, N) if (comp[2 * i] == comp[2 * i + 1]) return 0;
    return 1;
  }
};
```

EulerWalk.h

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

5f39ba, 21 lines

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

MaximumClique.h

Description: Quickly finds a maximum clique of a graph (given as symmetric bitset matrix; self-edges not allowed). Can be used to find a maximum independent set by finding a clique of the complement graph.
Time: Runs in about 1s for n=155 and worst case random graphs (p=.90). Runs faster for sparse graphs.

f7c0bc, 56 lines

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

MaximalCliques.h

Description: Runs a callback for all maximal cliques in a graph (given as a symmetric bitset matrix; self-edges not allowed). Callback is given a bitset representing the maximal clique.
Time: $\mathcal{O}\left(3^{n/3}\right)$, much faster for sparse graphs

b0d5b1, 17 lines

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

EdgeColoring.h

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

e210e2, 32 lines

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

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" 6dec70, 61 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() { return prop(), key; }
};
Node* merge(Node* a, Node* b) {
```

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

DominatorTree.h

Description: Given a digraph, return the edges of the dominator tree given as an adj. list (directed tree downwards from the root)
Time: $\mathcal{O}((n+m) * logn)$ where n is the number of verticies in the graph and m is the number of edges

36a500, 39 lines

```
vector<vi> dominator_tree(const vector<vi>& adj, int root) {
  int n = sz(adj) + 1, co = 0;
  vector<vi> ans(n), radj(n), child(n), sdomChild(n);
  vi label(n), rlabel(n), sdom(n), dom(n), par(n), bes(n);
  auto get = [&](auto self, int x) -> int {
    if (par[x] != x) {
      int t = self(self, par[x]);
      par[x] = par[par[x]];
      if (sdom[t] < sdom[bes[x]]) bes[x] = t;
    }
    return bes[x];
  };
  auto dfs = [&](auto self, int x) -> void {
    label[x] = ++co, rlabel[co] = x;
    sdom[co] = par[co] = bes[co] = co;
    for (auto y : adj[x]) {
      if (!label[y])
        self(self, y), child[label[x]].push_back(label[y]);
      radj[label[y]].push_back(label[x]);
    }
  };
  dfs(dfs, root);
  for (int i = co; i >= 1; --i) {
    for (auto j : radj[i])
      sdom[i] = min(sdom[i], sdom[get(get, j)]);
    if (i > 1) sdomChild[sdom[i]].push_back(i);
    for (auto j : sdomChild[i]) {
      int k = get(get, j);
      if (sdom[j] == sdom[k]) dom[j] = sdom[j];
      else dom[j] = k;
    }
    for (auto j : child[i]) par[j] = i;
  }
  for (int i = 2; i < co + 1; ++i) {
    if (dom[i] != sdom[i]) dom[i] = dom[dom[i]];
    ans[rlabel[dom[i]]].push_back(rlabel[i]);
  }
  return ans;
}
```

4.3 Trees

LCA.h

Description: Answers lowest common ancestor queries on a rooted tree using RMQ. Works with both directed and undirected adjacency lists.
Time: $\mathcal{O}(1)$ queries with an $\mathcal{O}(n \log n)$ precomp (RMQ).

"../../content/data-structures/RMQ.h" d11e3d, 17 lines

```
struct LCA {
  int T = 0;
  vector<int> pre, inv, tour;
  RMQ<int> rmq;
  LCA(vector<vi>& adj, int root = 0):
    pre(sz(adj)), inv(pre), rmq((dfs(adj, root), tour)) {}
  void dfs(vector<vi>& adj, int u, int p = -1) {
    inv[pre[u] = T++] = u;
    for (int v : adj[u])
      if (v != p) tour.push_back(pre[u]), dfs(adj, v, u);
  }
  int lca(int u, int v) {
    if (u == v) return u;
    tie(u, v) = minmax(pre[u], pre[v]);
    return inv[rmq.query(u, v)];
  }
};
```

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

"../data-structures/LazySegmentTree.h" 5d976e, 48 lines

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

TreeLifting.h

Description: Calculate jumps up a tree, to support fast upward jumps and LCAs. Assumes the root node points to itself.
Time: construction $\mathcal{O}(N)$, queries $\mathcal{O}(\log N)$

8410b6, 27 lines

```
struct lift {
  vi d, p, j;
  lift(vector<vi>& adj): d(sz(adj)), p(d), j(d) {
    dfs(0, adj);
  }
  void dfs(int u, vector<vi>& adj) {
    int jmp = (d[u] + d[j[j[u]]] == 2 * d[j[u]]) ? j[j[u]] : u;
    for (int v : adj[u])
      if (v != p[u])
        d[v] = d[p[v] = u] + 1, j[v] = jmp, dfs(v, adj);
  }
  int lca(int u, int v) {
    if (d[u] < d[v]) swap(u, v);
    while (d[u] > d[v]) u = d[j[u]] >= d[v] ? j[u] : p[u];
    if (u == v) return u;
    while (p[u] != p[v])
      if (j[u] != j[v]) u = j[u], v = j[v];
      else u = p[u], v = p[v];
    return p[u];
  }
  int kth(int u, int k) {
    if (k > d[u]) return -1;
    k = d[u] - k;
    while (d[u] > k) u = d[j[u]] >= k ? j[u] : p[u];
    return u;
  }
};
```

CentroidDecomp.h

Description: Computes centroid decomposition on connected tree, and runs callback function
Usage: `centroidDecomp(adj, [&] (int root, vector<int>& isIn) { ... });`
all nodes with isIn[i] = 1 connected to root are part of root's centroid
Time: $\mathcal{O}(n \log n)$

fffdc2, 19 lines

```
template<class G, class F>
void centroidDecomp(G g, F f) {
  vi s(sz(g), 1), par(sz(g)), is(s);
  auto go = [&] (int u, int p, auto& go) -> void {
    if ((par[u] = p) != -1) g[u].erase(find(all(g[u]), p));
    for (int v : g[u]) go(v, u, go), s[u] += s[v];
  };
  go(0, -1, go); queue<int> q({0});
  while (sz(q)) {
    int x = q.front(), b = x, ss, c; q.pop();
    do for(int v : g[c = b]) if(s[v] > s[x]/2) b = v;
      while(c != b);
    f(c, is);
    is[c] = 0, ss = s[c];
    for (int v : g[c]) if (s[v] > 0) q.push(v);
    if (c != x) q.push(x);
    do s[c] -= ss; while ((c = par[c]) != par[x]);
  }
}
```

TwoEdgeCCs.h

Description: Finds all Two Edge Connected Components and constructs a meta graph. The meta tree constructed has all nodes of the same 2ECC compressed into one, with bridges connecting them. Multi-edges are NOT bridges.
Time: $\mathcal{O}(V + E)$

91bd2e, 41 lines

```
vector<vector<pii>> ed;
int preCnt, metaN, stkSz;
vi pre, metaMap, stk;
void pop(int stop) {
  while (stkSz > 0) {
    int curr = stk[--stkSz];
    metaMap[curr] = metaN;
    if (curr == stop) break;
  }
  ++metaN;
}
int dfs(int idx, int par) {
  int low = pre[idx] = ++preCnt;
  stk[stkSz++] = idx;
  for (auto [v, id] : ed[idx]) {
    if (id == par) continue;
    if (pre[v] != -1) low = min(low, pre[v]); // back edge
    else {
      int get = dfs(v, id);
      low = min(low, get);
      if (get == pre[v]) pop(v); // bridge
    }
  }
  if (par == -1 && stkSz) pop(idx); // root case
  return low;
}
vector<vector<pii>> getMeta() {
  int n = sz(ed);
  preCnt = -1;
  metaN = 0;
  pre.assign(n, -1);
  metaMap.resize(n);
  stk.resize(n);
  stkSz = 0;
  rep(i, 0, n) if (pre[i] == -1) dfs(i, -1);
  vector<vector<pii>> metaEdges(metaN);
  rep(u, 0, n) for (auto [v, id] : ed[u]) if (
    metaMap[u] != metaMap[v]) metaEdges[metaMap[u]]
    .emplace_back(metaMap[v], id);
  return metaEdges;
}
```

TwoVertexCCs.h

Description: Finds all Two Vertex Connected Components and runs a callback for the edges in each. Note that a node can be in several components. An edge which is not in a component is a bridge, i.e., not part of any cycle.
Usage: `int eid = 0; ed.resize(N);`
`for each edge (a,b) {`
`ed[a].emplace_back(b, eid);`
`ed[b].emplace_back(a, eid++); }`
`twoVCCs([&](const vi& edgeIDs) {...});`
Time: $\mathcal{O}(E + V)$

45f60f, 29 lines

```
vi num, st;
vector<vector<pii>> ed;
int Time;
template<class F> int dfs(int at, int par, F& f) {
  int me = num[at] = ++Time, e, y, top = me;
  for (auto pa : ed[at])
    if (pa.second != par) {
      tie(y, e) = pa;
      if (num[y]) {
        top = min(top, num[y]);
        if (num[y] < me) st.push_back(e);
      } else {
        int si = sz(st);
        int up = dfs(y, e, f);
        top = min(top, up);
        if (up == me) {
          st.push_back(e);
          f(vi(st.begin() + si, st.end()));
          st.resize(si);
        } else if (up < me) st.push_back(e);
        else { /* e is a bridge */ }
      }
    }
  return top;
}
template<class F> void twoVCCs(F f) {
  num.assign(sz(ed), 0);
  rep(i, 0, sz(ed)) if (!num[i]) dfs(i, -1, f);
}
```

2VCCs.h

c8d930, 74 lines

```
struct graph {
  int n;
  vector<vector<pii>> edges;
  graph(int n): n(n), edges(n) {}
  void addEdge(int a, int b, int id) {
    edges[a].emplace_back(b, id);
    edges[b].emplace_back(a, id);
  }
};
int preCnt, metaN;
vi pre, low, used, stk, lowMap, artMap;
void pop(int stopId) {
  while (size(stk)) {
    int id = stk.back();
    stk.pop_back();
    lowMap[id] = metaN;
    if (id == stopId) break;
  }
  ++metaN;
}
void dfs(int idx, int par, graph &g) {
  pre[idx] = low[idx] = ++preCnt;
  bool hasFwd = false;
  for (auto [next, id] : g.edges[idx]) {
    if (next != par && !used[id]) {
      used[id] = true, stk.push_back(id);
      // back edge
      if (pre[next] != -1) {
        low[idx] = min(low[idx], pre[next]);
        continue;
      }
      // forward edge
      dfs(next, idx, g);
      low[idx] = min(low[idx], low[next]);
      if (par == -1 && hasFwd ||
        par != -1 && low[next] >= pre[idx]) {
        // idx is an articulation point
        if (artMap[idx] == -1) artMap[idx] = metaN++;
        pop(id);
      }
      hasFwd = true;
    }
  }
  if (par == -1 && size(stk)) pop(-1);
}
pair<graph, vi> getMeta(graph &g, int numEdges) {
  int n = g.n;
  preCnt = -1, metaN = 0;
  pre = vi(n, -1), low = vi(n), artMap = vi(n, -1);
  used = vi(numEdges), lowMap = vi(numEdges), stk = vi();
  for (int i = 0; i < n; ++i)
    if (pre[i] == -1) dfs(i, -1, g);
  graph meta = graph(metaN);
  vi seen(metaN, false);
  for (int u = 0; u < n; ++u) {
    if (artMap[u] == -1) continue;
    for (auto [v, id] : g.edges[u]) {
      if (!seen[lowMap[id]]) {
        seen[lowMap[id]] = true;
        meta.addEdge(artMap[u], lowMap[id], -1);
      }
    }
    for (auto [v, id] : g.edges[u]) seen[lowMap[id]] = false;
  }
  // use and return if needed:
  // gives a original node contribution to each meta node
  // articulation points are only counted toward their
  // individual meta nodes
  vi cont(metaN);
  for (int u = 0; u < n; ++u)
    if (artMap[u] != -1) cont[artMap[u]] = 1;
    else ++cont[lowMap[g.edges[u].begin()->second]];
  return {meta, cont};
}
```

CompressTree.h

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

"LCA.h" 9775a0, 22 lines

```
typedef vector<pair<int, int>> vpi;
vpi compressTree(LCA& lca, const vi& subset) {
  static vi rev;
  rev.resize(sz(lca.time));
  vi li = subset, &T = lca.time;
```

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

4.4 Flow

PushRelabel.h

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

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

0ae1d4, 56 lines

```
struct PushRelabel {
  struct Edge {
    int dest, back;
    ll f, c;
  };
  vector<vector<Edge>> g;
  vector<ll> ec;
  vector<Edge*> cur;
  vector<vi> hs;
  vi H;
  PushRelabel(int n): g(n), ec(n), cur(n), hs(2 * n), H(n) {}
  void addEdge(int s, int t, ll cap, ll rcap = 0) {
    if (s == t) return;
    g[s].push_back({t, sz(g[t]), 0, cap});
    g[t].push_back({s, sz(g[s]) - 1, 0, rcap});
  }
  void addFlow(Edge& e, ll f) {
    Edge& back = g[e.dest][e.back];
    if (!ec[e.dest] && f) hs[H[e.dest]].push_back(e.dest);
    e.f += f;
    e.c -= f;
    ec[e.dest] += f;
    back.f -= f;
    back.c += f;
    ec[back.dest] -= f;
  }
  ll calc(int s, int t) {
    int v = sz(g);
    H[s] = v;
    ec[t] = 1;
    vi co(2 * v);
    co[0] = v - 1;
    rep(i, 0, v) cur[i] = g[i].data();
    for (Edge& e : g[s]) addFlow(e, e.c);
    for (int hi = 0;;) {
      while (hs[hi].empty())
        if (!hi--) return -ec[s];
      int u = hs[hi].back();
      hs[hi].pop_back();
      while (ec[u] > 0) // discharge u
        if (cur[u] == g[u].data() + sz(g[u])) {
          H[u] = 1e9;
          for (Edge& e : g[u])
            if (e.c && H[u] > H[e.dest] + 1)
              H[u] = H[e.dest] + 1, cur[u] = &e;
          if (++co[H[u]], !--co[hi] && hi < v)
            rep(i, 0, v) if (hi < H[i] && H[i] < v)-- co[H[i]],
              H[i] = v + 1;
          hi = H[u];
        } else if (cur[u]->c && H[u] == H[cur[u]->dest] + 1)
          addFlow(*cur[u], min(ec[u], cur[u]->c));
        else ++cur[u];
    }
  }
  bool leftOfMinCut(int a) { return H[a] >= sz(g); }
};
```

hopcroftKarp.h

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

Usage: `vi btoa(m, -1); hopcroftKarp(g, btoa);`

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

f612e4, 42 lines

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

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.

d7f0f1, 42 lines

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

SebaDinic.h

Description: Max flow algorithm. Can find a valid circulation given vertex and/or edge demands. Time: $O(VE\log U)$

b2e5d6, 81 lines

```
// disable scaling when max flow/capacity is small, or
// sometimes on random data
template<bool SCALING = true> struct Dinic {
  struct Edge {
    int v, dual;
    ll cap, res;
    constexpr ll flow() { return max(cap - res, 0LL); }
  };
  int n, s, t;
  vi lvl, q, ptr;
  vector<vector<Edge>> adj;
  vector<pii> edges;
  Dinic(int n): n(n + 2), s(n++), t(n++), q(n), adj(n) {}
  int add(int u, int v, ll cap, ll flow = 0) {
    adj[u].push_back({v, sz(adj[v]), cap, cap - flow});
    adj[v].push_back({u, sz(adj[u]) - 1, 0, 0});
    edges.emplace_back(u, adj[u].size() - 1);
```

```
    return edges.size() - 1; // this Edge's ID
  }
  ll dfs(int u, ll in) {
    if (u == t || !in) return in;
    ll flow = 0;
    for (int& i = ptr[u]; i < sz(adj[u]); i++) {
      auto& e = adj[u][i];
      if (e.res && lvl[e.v] == lvl[u] - 1)
        if (ll out = dfs(e.v, min(in, e.res))) {
          flow += out, in -= out, e.res -= out;
          adj[e.v][e.dual].res += out;
          if (!in) return flow;
        }
    }
    return flow;
  }
  ll flow() {
    ll flow = 0;
    q[0] = t;
    for (int B = SCALING * 30; B >= 0; B--) do {
        lvl = ptr = vi(n);
        int qi = 0, qe = lvl[t] = 1;
        while (qi < qe && !lvl[s]) {
          int u = q[qi++];
          for (auto& e : adj[u])
            if (!lvl[e.v] && adj[e.v][e.dual].res >> B)
              q[qe++] = e.v, lvl[e.v] = lvl[u] + 1;
        }
        if (lvl[s]) flow += dfs(s, LLONG_MAX);
      } while (lvl[s]);
    return flow;
  }
  Edge& get(int id) { // get Edge object from its ID
    return adj[edges[id].first][edges[id].second];
  }
  void clear() {
    for (auto& it : adj)
      for (auto& e : it) e.res = e.cap;
  }
  bool leftOfMinCut(int u) { return lvl[u] == 0; }
  // d is a list of vertex demands, d[u] = flow in - flow out
  // negative if u is a source, positive if u is a sink
  bool circulation(vector<ll> d = {}) {
    d.resize(n);
    vector<int> circEdges;
    Dinic g(n);
    for (int u = 0; u < n; u++)
      for (auto& e : adj[u]) {
        d[u] += e.flow(), d[e.v] -= e.flow();
        if (e.res) circEdges.push_back(g.add(u, e.v, e.res));
      }
    int tylerEdge = g.add(t, s, LLONG_MAX, 0);
    ll flow = 0;
    for (int u = 0; u < n; u++)
      if (d[u] < 0) g.add(g.s, u, -d[u]);
      else if (d[u] > 0) g.add(u, g.t, d[u]), flow += d[u];
    if (flow != g.flow()) return false;
    int i = 0; // reconstruct the flow into this graph
    for (int u = 0; u < n; u++)
      for (auto& e : adj[u])
        if (e.res) e.res -= g.get(circEdges[i++]).flow();
    return true;
  }
};
```

DFSMatching.h

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

522b98, 22 lines

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

MinCostMaxFlow.h

Description: Min-cost max-flow. cap[i][j] != cap[j][i] is allowed; double edges are not. If costs can be negative, call setpi before maxflow, but note that negative cost cycles are not supported. To obtain the actual flow, look at positive values only.
Time: Approximately $\mathcal{O}(E^2)$

fe85cc, 78 lines

```
#include <bits/extc++.h>
const ll INF = numeric_limits<ll>::max() / 4;
typedef vector<ll> VL;
struct MCMF {
  int N;
  vector<vi> ed, red;
  vector<VL> cap, flow, cost;
  vi seen;
  VL dist, pi;
  vector<pii> par;
  MCMF(int N):
    N(N), ed(N), red(N), cap(N, VL(N)), flow(cap), cost(cap),
    seen(N), dist(N), pi(N), par(N) {}
  void addEdge(int from, int to, ll cap, ll cost) {
    this->cap[from][to] = cap;
    this->cost[from][to] = cost;
    ed[from].push_back(to);
    red[to].push_back(from);
  }
  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});
    auto relax = [&](int i, ll cap, ll cost, int dir) {
      ll val = di - pi[i] + cost;
      if (cap && val < dist[i]) {
        dist[i] = val;
        par[i] = {s, dir};
        if (its[i] == q.end()) its[i] = q.push({-dist[i], i});
        else q.modify(its[i], {-dist[i], i});
      }
    };
    while (!q.empty()) {
      s = q.top().second;
      q.pop();
      seen[s] = 1;
      di = dist[s] + pi[s];
      for (int i : ed[s])
        if (!seen[i])
          relax(i, cap[s][i] - flow[s][i], cost[s][i], 1);
      for (int i : red[s])
        if (!seen[i]) relax(i, flow[i][s], -cost[i][s], 0);
    }
    rep(i, 0, N) pi[i] = min(pi[i] + dist[i], INF);
  }
  pair<ll, ll> maxflow(int s, int t) {
    ll totflow = 0, totcost = 0;
    while (path(s), seen[t]) {
      ll fl = INF;
      for (int p, r, x = t; tie(p, r) = par[x], x != s; x = p)
        fl = min(fl, r ? cap[p][x] - flow[p][x] : flow[x][p]);
      totflow += fl;
      for (int p, r, x = t; tie(p, r) = par[x], x != s; x = p)
        if (r) flow[p][x] += fl;
        else flow[x][p] -= fl;
    }
    rep(i, 0, N) rep(j, 0, N) totcost +=
      cost[i][j] * flow[i][j];
    return {totflow, totcost};
  }
  // If some costs can be negative, call this before maxflow:
  void setpi(int s) { // (otherwise, leave this out)
    fill(all(pi), INF);
    pi[s] = 0;
    int it = N, ch = 1;
    ll v;
    while (ch-- && it--)
      rep(i, 0, N) if (pi[i] != INF) for (int to : ed[i]) if (
        cap[i][to]) if ((v = pi[i] + cost[i][to]) < pi[to])
        pi[to] = v,
        ch = 1;
    assert(it >= 0); // negative cost cycle
  }
};
```

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

1e0fe9, 31 lines

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

```
      if (dist[j] < delta) delta = dist[j], j1 = j;
    }
    rep(j, 0, m) {
      if (done[j]) u[p[j]] += delta, v[j] -= delta;
      else dist[j] -= delta;
    }
    j0 = j1;
  } while (p[j0]);
  while (j0) { // update alternating path
    int j1 = pre[j0];
    p[j0] = p[j1], j0 = j1;
  }
 }
 rep(j, 1, m) if (p[j]) ans[p[j] - 1] = j - 1;
 return {-v[0], ans}; // min cost
}
```

GlobalMinCut.h

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

8b0e19, 21 lines

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

Blossom.h

Description: General matching in $O(nm)$.

1fa809, 46 lines

```
vi Blossom(vector<vi>& adj) {
 int n = adj.size(), T = -1;
 vi mate(n, -1), label(n), par(n), orig(n), aux(n, -1), q;
 auto lca = [&](int x, int y) {
  for (T++;; swap(x, y)) {
   if (x == -1) continue;
   if (aux[x] == T) return x;
   aux[x] = T;
   x = (mate[x] == -1 ? -1 : orig[par[mate[x]]]);
  }
 };
 auto blossom = [&](int v, int w, int a) {
  while (orig[v] != a) {
   par[v] = w;
   w = mate[v];
   if (label[w] == 1) label[w] = 0, q.push_back(w);
   orig[v] = orig[w] = a, v = par[w];
  }
 };
 auto augment = [&](int v) {
  while (v != -1) {
   int pv = par[v], nv = mate[pv];
   mate[v] = pv, mate[pv] = v, v = nv;
  }
 };
 auto bfs = [&](int root) {
  fill(all(label), -1), iota(all(orig), 0);
  q.clear(), q.push_back(root), label[root] = 0;
  for (int i = 0; i < sz(q); i++) {
   int v = q[i];
   for (auto x : adj[v])
    if (label[x] == -1) {
     label[x] = 1, par[x] = v;
     if (mate[x] == -1) return augment(x);
     label[mate[x]] = 0, q.push_back(mate[x]);
    } else if (label[x] == 0 && orig[v] != orig[x]) {
     int a = lca(orig[v], orig[x]);
     blossom(x, v, a), blossom(v, x, a);
    }
  }
 };
 // Time halves if you start with (any) maximal matching.
 for (int i = 0; i < n; i++)
  if (mate[i] == -1) bfs(i);
 return mate;
}
```

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

"PushRelabel.h" 0418b3, 13 lines

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

4.5 Math

4.5.1 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).

4.5.2 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 $k = 1 \ldots n$,

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

Strings (5)

KMP.h

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

d4375c, 15 lines

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

ahocorasick.h

023d61, 46 lines

```
const int ALPHA = 26;
struct AhoCorasick {
 int nodeCount;
 vector<vector<int>> transition, children;
 vector<int> term, leaf;
 queue<int> q;
 AhoCorasick() {}
 AhoCorasick(vector<string> &strs) {
  int maxNodes = 1;
  for (string &s : strs) maxNodes += (int)s.length();
  transition.resize(ALPHA);
  children.resize(ALPHA, vector<int>(maxNodes));
  leaf.resize(maxNodes);
  nodeCount = 1;
  for (string &s : strs) {
   int node = 0;
   for (char &ch : s) {
    int c = ch - 'a';
    if (children[c][node] == 0)
     children[c][node] = nodeCount++;
    node = children[c][node];
   }
   leaf[node]++;
  }
  for (int i = 0; i < ALPHA; ++i)
   transition[i].resize(nodeCount);
  term.resize(nodeCount);
  q.push(0), q.push(0);
  while (q.size()) {
   int node = q.front();
   q.pop();
   int suffLink = q.front();
   q.pop();
   term[node] = leaf[node] + term[suffLink];
   for (int ch = 0; ch < ALPHA; ++ch) {
    if (children[ch][node] != 0) {
     transition[ch][node] = children[ch][node];
     q.push(children[ch][node]);
     q.push(node == 0 ? 0 : transition[ch][suffLink]);
    } else {
     transition[ch][node] = transition[ch][suffLink];
    }
   }
```

```
    }
  }
};
```

MinRotation.h

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

d07a42, 15 lines

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

SuffixArray.h

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

ad76fb, 27 lines

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

FMIndex.h

80cb9c, 23 lines

```
const int LIM = 26, A = 'A';
struct FMIndex: SuffixArray {
  vector<array<int, LIM>> freq;
  array<int, LIM + 1> occ = {1};
  FMIndex(string& s): SuffixArray(s), freq(sz(s) + 2) {
    rep(i, 0, sz(sa)) {
      freq[i + 1] = freq[i];
      int p = sa[i] - 1;
      if (p >= 0) freq[i + 1][s[p] - A]++, occ[s[p] - A + 1]++;
    }
    rep(i, 0, LIM) occ[i + 1] += occ[i];
  } // prepend c at pos i
  int add(int i, char c) {
    return occ[c - A] + freq[i][c - A];
  }
  vector<pii> query(string& q) { // inclusive-exclusive
    vector<pii> ret(sz(q) + 1);
    auto [L, R] = ret.back() = {0, sz(sa)};
    for (int i = sz(q) - 1; i >= 0; i--)
      ret[i] = {L = add(L, q[i]), R = add(R, q[i])};
    return ret;
  }
};
```

KactlSuffixTree.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: $\mathcal{O}(26N)$

aae0b8, 71 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, 0, sz(a)) ukkadd(i, toi(a[i]));
  }
  // example: find longest common substring (uses ALPHA = 28)
  pii best;
  int lcs(int node, int i1, int i2, int olen) {
    if (l[node] <= i1 && i1 < r[node]) return 1;
    if (l[node] <= i2 && i2 < r[node]) return 2;
    int mask = 0, len = node ? olen + (r[node] - l[node]) : 0;
    rep(c, 0, ALPHA) if (t[node][c] != -1) mask |=
      lcs(t[node][c], i1, i2, len);
    if (mask == 3) best = max(best, {len, r[node] - len});
    return mask;
  }
  static pii LCS(string s, string t) {
    SuffixTree st(s + (char)('z' + 1) + t + (char)('z' + 2));
    st.lcs(0, sz(s), sz(s) + 1 + sz(t), 0);
    return st.best;
  }
};
```

SuffixAutomaton.h

Description: Generates a suffix automaton. len is the longest-length substring ending here pos is the first index in the string matching here term is whether this node is a terminal (aka a suffix)
Time: $\mathcal{O}(|S|)$ (unsure)

1914a9, 22 lines

```
struct st { int len, pos, term; st *link; map<char, st*> next;
    };
st *suffixAutomaton(string &str) {
  st *last = new st(), *root = last;
  for(auto c : str) {
    st *p = last, *cur = last = new st{last->len + 1, last->len
        };
    while(p && !p->next.count(c))
      p->next[c] = cur, p = p->link;
    if (!p) cur->link = root;
    else {
      st *q = p->next[c];
      if (p->len + 1 == q->len) cur->link = q;
      else {
        st *clone = new st{p->len+1, q->pos, 0, q->link, q->
            next};
        for (; p && p->next[c] == q; p = p->link)
          p->next[c] = clone;
        q->link = cur->link = clone;
      }
    }
  }
  while(last) last->term = 1, last = last->link;
  return root;
}
```

Eertree.h

Description: Generates an eertree on str. *cur* is accurate at the end of the main loop before the final assignment to t.
Time: $\mathcal{O}(|S|)$

e1fb21, 19 lines

```
vector<int> slink = {0, 0}, len = {-1, 0};
vector<vector<int>> down(2, vector<int>(26, -1));
```

```
int cur = 0, t = 0;
for(int i = 0; i < str.size(); i++) {
  char c = str[i]; int ci = c - 'a';
  while(t <= 0 || str[t-1] != c) t = i - len[cur = slink[cur]];
  if(down[cur][ci] == -1) {
    down[cur][ci] = slink.size();
    down.emplace_back(26, -1);
    len.push_back(len[cur] + 2);
    if(len.back() > 1){
        do t = i - len[cur = slink[cur]];
        while(t <= 0 || str[t-1] != c);
        slink.push_back(down[cur][ci]);
    } else slink.push_back(1);
    cur = slink.size() - 1;
  } else cur = down[cur][ci];
  t = i - len[cur] + 1;
}
```

ZValues.h

333c4f, 10 lines

```
vector<int> zValues(string& s) {
  int n = (int)s.length();
  vector<int> z(n);
  for (int i = 1, L = 0, R = 0; i < n; ++i) {
    if (i <= R) z[i] = min(R - i + 1, z[i - L]);
    while (i + z[i] < n && s[z[i]] == s[i + z[i]]) ++z[i];
    if (i + z[i] - 1 > R) L = i, R = i + z[i] - 1;
  }
  return z;
}
```

Manacher.h

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

e7ad79, 13 lines

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

Hashing.h

Description: Self-explanatory methods for string hashing.

2d2a67, 47 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 x + ~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, 0, sz(str)) ha[i + 1] = ha[i] * C + str[i],
                       pw[i + 1] = pw[i] * C;
  }
  H hashInterval(int a, int b) { // hash [a, b)
    return ha[b] - ha[a] * pw[b - a];
  }
};
vector<H> getHashes(string& str, int length) {
  if (sz(str) < length) return {};
  H h = 0, pw = 1;
  rep(i, 0, length) h = h * C + str[i], pw = pw * C;
  vector<H> ret = {h};
  rep(i, length, sz(str)) {
    ret.push_back(h = h * C + str[i] - pw * str[i - length]);
  }
  return ret;
}
H hashString(string& s) {
  H h{};
  for (char c : s) h = h * C + c;
  return h;
}
```

Miscellaneous (6)

LowerHigher.h

6a2498, 9 lines

```
template<class S, class T>
auto lower(const S& s, const T& x, bool strict = 0) {
  auto it = strict ? s.lower_bound(x) : s.upper_bound(x);
  return it == begin(s) ? end(s) : prev(it);
}
template<class S, class T>
auto higher(const S& s, const T& x, bool strict = 0) {
  return strict ? s.upper_bound(x) : s.lower_bound(x);
}
```

HilbertMos.h

Description: Maps points on a $2^k \times 2^k$ matrix to their index on the Hilbert curve.

447ee3, 16 lines

```
const int logn = 21, maxn = 1 << logn;
ll hilbert(int x, int y) {
  ll d = 0;
  for (int s = 1 << (logn - 1); s > 0; s >>= 1) {
    int rx = x & s, ry = y & s;
    d = d << 2 | rx * 3 ^ ry;
    if (ry == 0) {
      if (rx != 0) {
        x = maxn - x;
        y = maxn - y;
      }
      swap(x, y);
    }
  }
  return d;
}
```

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 (mod 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;
  }
};
```

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, 22 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);
}
```

LIS.h

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

2932a0, 18 lines

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

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

8bc1cc, 61 lines

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

KnuthDP.h

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

SubmasksSupermasks.h

21f9fe, 13 lines

```
void submasks(int n) {
  for (int mask = 0; mask < (1 << n); mask++)
    for (int sub = mask; sub; sub = (sub - 1) & mask) {
      // do thing
    }
}
void supermasks(int n) {
  for (int mask = 0; mask < (1 << n); mask++)
    for (int sup = mask; sup < (1 << n);
         sup = (sup + 1) | mask) {
      // do thing
    }
}
```

6.1 Debugging tricks

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

6.2 Optimization tricks

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

6.2.1 Bit hacks

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

6.2.2 Pragmas

- `#pragma GCC optimize ("Ofast")` will make GCC auto-vectorize loops and optimizes floating points better.
- `#pragma GCC target ("avx2")` can double performance of vectorized code, but causes crashes on old machines.
- `#pragma GCC optimize ("trapv")` kills the program on integer overflows (but is really slow).
- `#pragma GCC optimize("O3,unroll-loops")`
- `#pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")`

Mathematics (7)

7.1 Equations

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

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

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

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

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

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

7.2 Recurrences

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

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

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

7.3 Trigonometry

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

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

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

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

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

7.4 Geometry

7.4.1 Triangles

Side lengths: a, b, c

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

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

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

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

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

Length of bisector (divides angles in two):

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

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

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

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

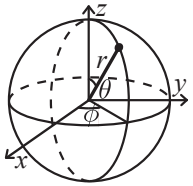
7.4.2 Quadrilaterals

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

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

7.4.3 Spherical coordinates

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



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

7.5 Derivatives/Integrals

$$\frac{d}{dx}\arcsin x = \frac{1}{\sqrt{1-x^2}} \qquad \frac{d}{dx}\arccos x = -\frac{1}{\sqrt{1-x^2}}$$

$$\frac{d}{dx}\tan x = 1 + \tan^2 x \qquad \frac{d}{dx}\arctan x = \frac{1}{1+x^2}$$

$$\int \tan ax = -\frac{\ln|\cos ax|}{a} \qquad \int x\sin ax = \frac{\sin ax - ax\cos ax}{a^2}$$

$$\int e^{-x^2} = \frac{\sqrt{\pi}}{2}\operatorname{erf}(x) \qquad \int xe^{ax}dx = \frac{e^{ax}}{a^2}(ax-1)$$

Integration by parts:

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

7.6 Sums

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

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

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

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

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

7.7 Series

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

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

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

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

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

7.8 Probability theory

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

Expectation is linear:

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

For independent X and Y,

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

7.8.1 Discrete distributions

Binomial distribution

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

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

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

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

First success distribution

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

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

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

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

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

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

7.8.2 Continuous distributions

Uniform distribution

If the probability density function is constant between a and b and 0 elsewhere it is 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 Exp(λ), $\lambda > 0$.

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

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

Normal distribution

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

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

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

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

7.9 Markov chains

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

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

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

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

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

7.10 Fractions

FractionOverflow.h

Description: Safer struct for representing fractions/rationals. Comparison is 100% overflow safe; other ops are safer but can still overflow. All ops are $O(\log N)$.

8ff7f8, 40 lines

```
template<class T> struct QO {
  T a, b;
  QO(T p, T q = 1) {
    T g = gcd(p, q);
    a = p / g;
    b = q / g;
    if (b < 0) a = -a, b = -b;
  }
  T gcd(T x, T y) const { return __gcd(x, y); }
  QO operator+(const QO& o) const {
    T g = gcd(b, o.b), bb = b / g, obb = o.b / g;
    return {a * obb + o.a * bb, b * obb};
  }
  QO operator-(const QO& o) const {
    return *this + QO(-o.a, o.b);
  }
  QO operator*(const QO& o) const {
    T g1 = gcd(a, o.b), g2 = gcd(o.a, b);
    return {(a / g1) * (o.a / g2), (b / g2) * (o.b / g1)};
  }
  QO operator/(const QO& o) const {
    return *this * QO(o.b, o.a);
  }
  QO recip() const { return {b, a}; }
  int signum() const { return (a > 0) - (a < 0); }
  static bool lessThan(T a, T b, T x, T y) {
    if (a / b != x / y) return a / b < x / y;
    if (x % y == 0) return false;
    if (a % b == 0) return true;
    return lessThan(y, x % y, b, a % b);
  }
  bool operator<(const QO& o) const {
    if (this->signum() != o.signum() || a == 0) return a < o.a;
    if (a < 0) return lessThan(abs(o.a), o.b, abs(a), b);
    else return lessThan(a, b, o.a, o.b);
  }
  friend ostream& operator<<(ostream& cout, const QO& o) {
    return cout << o.a << "/" << o.b;
  }
};
```

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, 24 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, 27 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;
}
```

Combinatorial (8)

8.1 Permutations

8.1.1 Factorial

factorialMod.h

b5773c, 20 lines

```
int factmod(int n, int p) {
  vector<int> f(p);
  f[0] = 1;
  for (int i = 1; i < p; i++) f[i] = f[i - 1] * i % p;
  int res = 1;
```

```
  while (n > 1) {
    if ((n / p) % 2) res = p - res;
    res = res * f[n % p] % p;
    n /= p;
  }
  return res;
}
int multiplicityFactorial(int n, int p) {
  int count = 0;
  do {
    n /= p;
    count += n;
  } while (n);
  return count;
}
```

IntPerm.h

Description: Permutation -> integer conversion. (Not order preserving.) Integer -> permutation can use a lookup table.
Time: $\mathcal{O}(n)$ 044568, 7 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;
}
```

binomialModPrime.h

Description: Lucas' thm: Let n, m be non-negative integers and p a prime. Write $n = n_k p^k + ... + n_1 p + n_0$ and $m = m_k p^k + ... + m_1 p + m_0$. Then $\binom{n}{m} \equiv \prod_{i=0}^{k} \binom{n_i}{m_i} \pmod p$. fact and invfact must hold pre-computed factorials / inverse factorials, e.g. from ModInverse.h.
Time: $\mathcal{O}\left(\log_p n\right)$ c16cde, 10 lines

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

schreier-sims.cpp

Description: Check group membership of permutation groups 949a6e, 54 lines

```
struct Perm {
  int a[N];
  Perm() {
    for (int i = 1; i <= n; ++i) a[i] = i;
  }
  friend Perm operator*(const Perm &lhs, const Perm &rhs) {
    static Perm res;
    for (int i = 1; i <= n; ++i) res.a[i] = lhs.a[rhs.a[i]];
    return res;
  }
  friend Perm inv(const Perm &cur) {
    static Perm res;
    for (int i = 1; i <= n; ++i) res.a[cur.a[i]] = i;
    return res;
  }
};
class Group {
  bool flag[N];
  Perm w[N];
  std::vector<Perm> x;
public:
  void clear(int p) {
    memset(flag, 0, sizeof flag);
    for (int i = 1; i <= n; ++i) w[i] = Perm();
    flag[p] = true;
    x.clear();
  }
  friend bool check(const Perm &, int);
  friend void insert(const Perm &, int);
  friend void updateX(const Perm &, int);
} g[N];
bool check(const Perm &cur, int k) {
  if (!k) return true;
  int t = cur.a[k];
  return g[k].flag[t] ? check(g[k].w[t] * cur, k - 1) : false;
}
void updateX(const Perm &, int);
void insert(const Perm &cur, int k) {
  if (check(cur, k)) return;
  g[k].x.push_back(cur);
  for (int i = 1; i <= n; ++i)
    if (g[k].flag[i]) updateX(cur * inv(g[k].w[i]), k);
}
void updateX(const Perm &cur, int k) {
  int t = cur.a[k];
  if (g[k].flag[t]) {
    insert(g[k].w[t] * cur, k - 1);
  } else {
    g[k].w[t] = inv(cur);
    g[k].flag[t] = true;
    for (int i = 0; i < g[k].x.size(); ++i)
      updateX(g[k].x[i] * cur, k);
  }
}
```

8.1.2 Cycles

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

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

8.1.3 Derangements

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

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

8.1.4 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

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

8.2 Partitions and subsets

8.2.1 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

8.2.2 Lucas' Theorem

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

8.2.3 Binomials

multinomial.h

Description: Computes $\binom{k_1 + \cdots + k_n}{k_1, k_2, \ldots, k_n} = \frac{(\sum k_i)!}{k_1! k_2! \ldots k_n!}$. a0a312, 5 lines

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

8.3 General purpose numbers

8.3.1 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:

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

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

8.3.2 Stirling numbers of the first kind

Number of permutations on n items with k cycles.

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

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

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

8.3.3 Eulerian numbers

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

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

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

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

8.3.4 Stirling numbers of the second kind

Partitions of n distinct elements into exactly k groups.

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

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

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

8.3.5 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,

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

8.3.6 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: $(n-2)!/((d_1-1)! \cdots (d_n-1)!)$

8.3.7 Catalan numbers

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

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

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

- 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.

Number Theory (9)

9.0.1 Multiplicative Functions

A function $f : \mathbb{N} \to \mathbb{C}$ is multiplicative if $f(ab) = f(a)f(b)$ when a, b are coprime. Common multiplicative functions:
Unit function: $\epsilon(n) = [n = 1]$
Constant function: $1(n) = 1$
Identity function: $\mathrm{Id}(n) = n$
Power function: $\mathrm{Id}_a(n) = n^a$
Divisor function $\sigma_a(p^k) = \frac{p^{a(k+1)}-1}{p^a-1}$
Möbius function $\mu(p^k) = [k = 0] - [k = 1]$
Euler's totient function $\phi(p^k) = p^k - p^{k-1}$
Dirichlet convolution of multiplicative functions f and g is multiplicative,

$$(f * g)(n) = \sum_{d|n} f(d) g\left(\frac{n}{d}\right)$$

$$\begin{array}{ccc} 1 * \mu = \epsilon & g = f * 1 \text{ iff } f = g * \mu & \sigma_k = \mathrm{Id}_k * 1 \\ \sigma_0 = d = 1 * 1 & \phi * 1 = \mathrm{Id} & \phi^k * 1 = \mathrm{Id}_k \\ \sigma = \phi * d & d^3 * 1 = (d * 1)^2 & \end{array}$$

$\pi(x) = \sum_{n \leq x} (\omega * \mu)(n)$, π = prime counting function, ω = num. of prime factors

9.0.2 From kactl

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

Mobius Inversion:

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

Other useful formulas/forms:

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

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

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

9.0.3 Möbius Inversion Formula

If for any two functions $f, g : \mathbb{N} \to \mathbb{C}$, $g(n) = \sum_{d|n} f(n)$,

$$f(n) = (\mu * g)(n) = \sum_{d|n} \mu(d) g\left(\frac{n}{d}\right)$$

FloorRange.h — 14b147, 4 lines

```
for (int i = 1, la; i <= n; i = la + 1) {
  la = n / (n / i);
  //n / x yields the same value for i <= x <= la.
}
```

MultiplicativeSieve.h — b7fecc, 36 lines

```
const int LIM = 1e6 + 10;
vi primes;
bitset<LIM> notPrime;
int ps[LIM], ks[LIM];
ll f[LIM];
template<ll (*g)(int, int, int)> void sieve(int n = LIM) {
  f[1] = 1;
  for (int i = 2; i < n; i++) {
    if (!notPrime[i]) {
      primes.push_back(i);
      ps[i] = i, ks[i] = 1;
      f[i] = g(i, i, 1);
      for (ll pk = i, x = 1ll * i * i; x < n;
           pk *= i, x *= i) {
        ps[x] = x, ks[x] = ks[pk] + 1;
        f[x] = g(x, i, ks[x]);
      }
    }
    for (ll j = 0, p;
         j < sz(primes) && (p = primes[j]) * i < n; j++) {
      ll x = i * p;
      if (x >= n) break;
      notPrime[x] = 1;
      if (i % p == 0) {
        if (i != ps[i]) {
          ps[x] = ps[i] * p, ks[x] = ks[i] + 1;
          f[x] = f[i / ps[i]] * f[ps[x]];
        }
        break;
      } else {
        f[x] = f[i] * f[p];
        ps[x] = p, ks[x] = 1;
      }
    }
  }
}
```

MultiplicativePrefixSum.h — 2c0d18, 32 lines

```
/*  Prefix sum of multiplicative functions :
    p_f : the prefix sum of f (x) (1 <= x <= th).
    p_g : the prefix sum of g (x) (0 <= x <= N).
    p_c : the prefix sum of f * g (x) (0 <= x <= N).
    th : the thereshold, generally should be n ^ (2 / 3).
*/
typedef ll (*func)(ll);
template<func p_f, func p_g, func p_c, ll TH>
struct prefix_mul {
  ll n, inv;
  unordered_map<ll, ll> mem;
  ll calc(ll x) {
    if (x < TH) return p_f(x);
    auto it = mem.find(x);
    if (it != mem.end()) return it->second;
    ll ans = 0;
    for (ll i = 2, la; i <= x; i = la + 1) {
      la = x / (x / i);
      ans =
        (ans + (p_g(la) - p_g(i - 1) + mod) * calc(x / i)) %
        mod;
    }
    ans = (p_c(x) - ans + mod) * inv % mod;
    return mem[x] = ans;
  }
  ll solve(ll n) {
    if (n <= 0) return 0;
    this->n = n;
    inv = binpow(p_g(1), mod - 2);
```

```
    return calc(n);
  }
};
```

phiFunction.h

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

cf7d6d, 8 lines

```
const int LIM = 5000000;
int phi[LIM];
void calculatePhi() {
  rep(i, 0, LIM) phi[i] = i & 1 ? i : i / 2;
  for (int i = 3; i < LIM; i += 2)
    if (phi[i] == i)
      for (int j = i; j < LIM; j += i) phi[j] -= phi[j] / i;
}
```

9.1 Primality

MillerRabin.h

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

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

"ModMulLL.h" 60dcd1, 12 lines

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

PrimeSieveFast.h

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

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

a1933d, 23 lines

```
const int LIM = 1e8;
bitset<LIM> isPrime;
vector<int> primeSieve() {
  const int S = round(sqrt(LIM)), R = LIM / 2;
  vector<int> pr = {2}, sieve(S + 1);
  pr.reserve(int(LIM / log(LIM) * 1.1));
  vector<pair<int, int>> cp;
  for (int i = 3; i <= S; i += 2)
    if (!sieve[i]) {
      cp.push_back({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;
    for (int i = 0; i < min(S, R - L); i++)
      if (!block[i]) pr.push_back((L + i) * 2 + 1);
  }
  for (int i : pr) isPrime[i] = 1;
  return pr;
}
```

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" a33cf6, 18 lines

```
ull pollard(ull n) {
  auto f = [n](ull x) { return modmul(x, x, n) + 1; };
  ull x = 0, y = 0, t = 30, prd = 2, i = 1, q;
  while (t++ % 40 || __gcd(prd, n) == 1) {
    if (x == y) x = ++i, y = f(x);
    if ((q = modmul(prd, max(x, y) - min(x, y), n))) prd = q;
    x = f(x), y = f(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;
}
```

CountPrimes.h

Description: Count primes in $O(N^{\frac{3}{4}})$.

85fe69, 27 lines

```
const int SQN = 320'000;
bool notPrime[SQN];
ll countprimes(ll n) {
  vector<ll> divs;
  for (ll i = 1; i * i <= n; i++) {
    divs.push_back(i);
    divs.push_back(n / i);
  }
  sort(all(divs));
  divs.erase(unique(all(divs)), end(divs));
  vector<ll> dp(sz(divs));
  for (int i = 0; i < sz(divs); i++) dp[i] = divs[i] - 1;
  ll sq = sqrt(n), sum = 0;
  auto idx = [&](ll x) -> int {
    return x <= sq ? x - 1 : (sz(divs) - n / x);
  };
  for (ll p = 2; p * p <= n; p++)
    if (!notPrime[p]) {
      ll p2 = p * p;
      for (ll i = sz(divs) - 1; i >= 0 && divs[i] >= p2; i--)
        dp[i] -= dp[idx(divs[i] / p)] - sum;
      sum += 1;
      for (ll i = p * p; i < SQN && i * i <= n; i += p)
        notPrime[i] = 1;
    }
  return dp.back();
}
```

9.2 Divisibility

ExtendedEuclidean.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 $0 \leq x < \text{lcm}(m, n)$. Assumes $mn < 2^{62}$.

Time: $\log(n)$

"ExtendedEuclidean.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;
}
```

9.2.1 Bézout's identity

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

$$ax + by = d$$

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

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

9.3 Modular arithmetic

ModInverse.h

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

6f684f, 4 lines

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

ChooseUnderMod.h

Description: Returns the value of $nCr \ \% \ mod$ using Fermat's little theorem. [kactl version not yet tested]

"ModPow.h" 8349a5, 19 lines

```
// could also use kactl's modInverse
int modInverse(int n)
{
    return modpow(n, mod - 2); // set
}

int nCrModPFermat(int n, int r)
{
   if (n<r) return 0;
   if (r == 0) return 1;

   vi fac (n + 1);
   fac[0] = 1;

   for (int i = 1; i <= n; i++)
       fac[i] = fac[i - 1] * i % mod;

   return (fac[n] * modInverse(fac[r]) % mod * modInverse(fac[n
         - r]) % mod) % mod;
}
```

ModMulLL.h

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

ModPow.h

b83e45, 7 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: $\mathcal{O}(\sqrt{m})$

c040b8, 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))
    rep(i, 2, n + 2) if (A.count(e = e * f % m)) return n * i -
      A[e];
  return -1;
}
```

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

"ModPow.h" 19a793, 23 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;
  }
}
```

9.4 Misc

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 $\le n$.
Usage: `berlekampMassey({0, 1, 1, 3, 5, 11}) // {1, 2}`
Time: $\mathcal{O}(N^2)$

"../number-theory/ModPow.h" 96548b, 24 lines

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

ContinuedFractions.h

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

dd6c5e, 21 lines

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

9.5 Pythagorean Triples

The Pythagorean triples are uniquely generated by

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

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

9.6 Primes

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

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

9.7 Estimates

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

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

Numerical (10)

FastFourierTransform.h

Description: fft(a) computes $\hat{f}(k) = \sum_x a[x] \exp(2\pi i \cdot kx/N)$ for all k. N must be a power of 2. Useful for convolution: `conv(a, b) = c`, where $c[x] = \sum a[i]b[x-i]$. For convolution of complex numbers or more than two vectors: FFT, multiply pointwise, divide by n, reverse(start+1, end), FFT back. Rounding is safe if $(\sum a_i^2 + \sum b_i^2) \log_2 N < 9 \cdot 10^{14}$ (in practice 10^{16}; higher for random inputs). Otherwise, use NTT/FFTMod.
Time: $\mathcal{O}(N \log N)$ with $N = |A| + |B|$ ($\sim 1s$ for $N = 2^{22}$)

c4db99, 39 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);
    rep(i, k, 2 * k) rt[i] = R[i] =
      i & 1 ? R[i / 2] * x : R[i / 2];
  }
  vi rev(n);
  rep(i, 0, n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
  rep(i, 0, 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, 0, k) {
      C z = rt[j+k] * a[i+j+k]; // (25% faster if hand-rolled
          )
      auto x = (double*)&rt[j + k],
      C z(x[0] * y[0] - x[1] * y[1],
      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, 0, sz(b)) in[i].imag(b[i]);
  fft(in);
  for (C& x : in) x *= x;
  rep(i, 0, n) out[i] = in[-i & (n - 1)] - conj(in[i]);
  fft(out);
  rep(i, 0, sz(res)) res[i] = imag(out[i]) / (4 * n);
  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" ced03d, 35 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.
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)};
    rep(i, k, 2 * k) rt[i] = rt[i / 2] * z[i & 1] % mod;
  }
  vi rev(n);
  rep(i, 0, n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
  rep(i, 0, 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, 0, 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;
  int inv = modpow(n, mod - 2);
  vl L(a), R(b), out(n);
  L.resize(n), R.resize(n);
  ntt(L), ntt(R);
  rep(i, 0, n) out[-i & (n - 1)] =
    (ll)L[i] * R[i] % mod * inv % mod;
  ntt(out);
  return {out.begin(), out.begin() + s};
}
```

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" b82773, 24 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, 0, sz(a)) L[i] = C((int)a[i] / cut, (int)a[i] % cut);
  rep(i, 0, sz(b)) R[i] = C((int)b[i] / cut, (int)b[i] % cut);
  fft(L), fft(R);
  rep(i, 0, 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, 0, 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;
}
```

FastSubsetTransform.h

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

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

464cf3, 18 lines

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

SumOverSubsets.h

56b6c0, 24 lines

```
// in place sum over subsets (give desc.)
// works with vector, array, and std::array
template<class Arr> void sos(Arr& f, int n, bool inv = 0) {
  for (int i = 0, b = 1; i < n; i++, b *= 2)
    rep(mask, 0, 1 << n) if (mask & b) f[mask] +=
      inv ? -f[mask ^ b] : f[mask ^ b];
}
template<class T>
vector<T> conv(const vector<T>& a, const vector<T>& b) {
  int n = __lg(sz(a)), len = 1 << n;
  vector<vector<T>> f(n + 1, vector<T>(len)), g = f, h = f;
  rep(mask, 0, len) {
    int pc = __builtin_popcount(mask);
    f[pc][mask] = a[mask], g[pc][mask] = b[mask];
  }
  rep(i, 0, n + 1) sos(f[i], n), sos(g[i], n);
  rep(mask, 0, len) rep(i, 0, n + 1) rep(j, 0, i + 1)
    h[i][mask] += f[j][mask] * g[i - j][mask];
  rep(i, 0, n + 1) sos(h[i], n, 1);
  vector<T> c(len);
  rep(mask, 0, len) c[mask] =
    h[__builtin_popcount(mask)][mask];
  return c;
}
```

Multipoint.h

e84187, 52 lines

```
vl inv(const vl& a, int N) {
  vl b = {modpow(a[0], mod - 2)};
  int k = 1;
  for (int k = 2; k / 2 < N; k *= 2) {
    vl temp = {begin(a), sz(a) > k ? begin(a) + k : end(a)};
    temp = conv(temp, b);
    for (int i = 0; i < sz(temp); i++)
      if (temp[i]) temp[i] = mod - temp[i];
    temp[0] += 2;
    if (temp[0] >= mod) temp[0] -= mod;
    b = conv(temp, b);
    b.resize(min(k, N));
  }
  return b;
}
pair<vl, vl> div(vl& a, vl& b) {
  if (sz(a) < sz(b)) return {{}, a};
  reverse(all(a)), reverse(all(b));
  vl q = conv(a, inv(b, sz(a) - sz(b) + 1));
  q.resize(sz(a) - sz(b) + 1);
  reverse(all(a)), reverse(all(b)), reverse(all(q));
  int cap = 0;
  vl r = conv(b, q);
  rep(i, 0, sz(a)) {
    r[i] = a[i] + mod - r[i];
    if (r[i] >= mod) r[i] -= mod;
    if (r[i]) cap = i + 1;
  }
  r.resize(cap);
  return {q, r};
}
const int M = 1 << 17;
vl t[4 * M]; // about 2 sec
void build(int v, int lo, int hi, vl& x) {
  if (hi - lo == 1) // handle if negative or w/e
    t[v] = {x[lo] ? mod - x[lo] : 0, 1};
  else {
    int mid = (lo + hi) / 2;
    build(2 * v, lo, mid, x), build(2 * v + 1, mid, hi, x);
    t[v] = conv(t[2 * v], t[2 * v + 1]);
  }
}
void calc(int v, int lo, int hi, vl a, vl& ans) {
  a = div(a, t[v]).second;
  if (hi - lo == 1) {
    ans[lo] = sz(a) ? a[0] : 0;
    return;
  }
  int mid = (lo + hi) / 2;
  calc(2 * v, lo, mid, a, ans);
  calc(2 * v + 1, mid, hi, a, ans);
}
```

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.

0b79e2, 74 lines

```
typedef double T; // long double, Rational, double + mod<P>...
typedef vector<T> vd;
typedef vector<vd> vvd;
const T eps = 1e-8, inf = 1 / .0;
#define MP make_pair
#define ltj(X) \
  if (s == -1 || MP(X[j], N[j]) < MP(X[s], N[s])) s = j
struct LPSolver {
  int m, n;
  vi N, B;
  vvd D;
  LPSolver(const vvd& A, const vd& b, const vd& c):
    m(sz(b)), n(sz(c)), N(n + 1), B(m), D(m + 2, vd(n + 2)) {
    rep(i, 0, m) rep(j, 0, n) D[i][j] = A[i][j];
    rep(i, 0, m) {
      B[i] = n + i;
      D[i][n] = -1;
      D[i][n + 1] = b[i];
    }
    rep(j, 0, n) {
      N[j] = j;
      D[m][j] = -c[j];
    }
    N[n] = -1;
    D[m + 1][n] = 1;
  }
  void pivot(int r, int s) {
    T *a = D[r].data(), inv = 1 / a[s];
    rep(i, 0, m + 2) if (i != r && abs(D[i][s]) > eps) {
      T *b = D[i].data(), inv2 = b[s] * inv;
      rep(j, 0, n + 2) b[j] -= a[j] * inv2;
      b[s] = a[s] * inv2;
    }
    rep(j, 0, n + 2) if (j != s) D[r][j] *= inv;
    rep(i, 0, m + 2) if (i != r) D[i][s] *= -inv;
    D[r][s] = inv;
    swap(B[r], N[s]);
  }
  bool simplex(int phase) {
    int x = m + phase - 1;
    for (;;) {
      int s = -1;
      rep(j, 0, n + 1) if (N[j] != -phase) ltj(D[x]);
      if (D[x][s] >= -eps) return true;
      int r = -1;
      rep(i, 0, m) {
        if (D[i][s] <= eps) continue;
        if (r == -1 ||
          MP(D[i][n + 1] / D[i][s], B[i]) <
            MP(D[r][n + 1] / D[r][s], B[r]))
          r = i;
      }
      if (r == -1) return false;
      pivot(r, s);
    }
  }
  T solve(vd& x) {
    int r = 0;
    rep(i, 1, m) if (D[i][n + 1] < D[r][n + 1]) r = i;
    if (D[r][n + 1] < -eps) {
      pivot(r, n);
      if (!simplex(2) || D[m + 1][n + 1] < -eps) return -inf;
      rep(i, 0, m) if (B[i] == -1) {
        int s = 0;
        rep(j, 1, n + 1) ltj(D[i]);
        pivot(i, s);
      }
    }
    bool ok = simplex(1);
    x = vd(n);
    rep(i, 0, m) if (B[i] < n) x[B[i]] = D[i][n + 1];
    return ok ? D[m][n + 1] : inf;
  }
};
```

Integrate.h

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

4756fc, 6 lines

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

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; });});});

bbac0d, 13 lines

```
typedef double d;
#define S(a, b) \
  (f(a) + 4 * f((a + b) / 2) + f(b)) * (b - a) / 6
template<class F> d rec(F& f, d a, d b, d eps, d S) {
  d c = (a + b) / 2;
  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));
}
```

SolveLinear.h

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

44c9ab, 36 lines

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

SolveLinear2.h

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

"SolveLinear.h" 08e495, 8 lines

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

Determinant.h

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

bd5cec, 16 lines

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

IntDeterminant.h

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

3313dc, 19 lines

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

```
    ans = ans * a[i][i] % mod;
    if (!ans) return 0;
  }
  return (ans + mod) % mod;
}
```

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 $A^{-1} = A^{-1}(2I - AA^{-1})$ (mod p^k) where A^{-1} starts as the inverse of A mod p, and k is doubled in each step.
Time: $\mathcal{O}\left(n^3\right)$

ebfff6, 34 lines

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

MatrixInverse-mod.h

Description: Invert matrix A modulo a prime. Returns rank; result is stored in A unless singular (rank < n). For prime powers, repeatedly set $A^{-1} = A^{-1}(2I - AA^{-1})$ (mod p^k) where A^{-1} starts as the inverse of A mod p, and k is doubled in each step.
Time: $\mathcal{O}\left(n^3\right)$

"../number-theory/ModPow.h" a6f68f, 40 lines

```
int matInv(vector<vector<ll>>& A) {
  int n = sz(A);
  vi col(n);
  vector<vector<ll>> tmp(n, vector<ll>(n));
  rep(i, 0, n) tmp[i][i] = 1, col[i] = i;
  rep(i, 0, n) {
    int r = i, c = i;
    rep(j, i, n) rep(k, i, n) if (A[j][k]) {
      r = j;
      c = k;
      goto found;
    }
    return i;
found:
    A[i].swap(A[r]);
    tmp[i].swap(tmp[r]);
    rep(j, 0, n) swap(A[j][i], A[j][c]),
      swap(tmp[j][i], tmp[j][c]);
    swap(col[i], col[c]);
    ll v = modpow(A[i][i], mod - 2);
    rep(j, i + 1, n) {
      ll f = A[j][i] * v % mod;
      A[j][i] = 0;
      rep(k, i + 1, n) A[j][k] = (A[j][k] - f * A[i][k]) % mod;
      rep(k, 0, n) tmp[j][k] =
        (tmp[j][k] - f * tmp[i][k]) % mod;
    }
    rep(j, i + 1, n) A[i][j] = A[i][j] * v % mod;
    rep(j, 0, n) tmp[i][j] = tmp[i][j] * v % mod;
    A[i][i] = 1;
  }
  for (int i = n - 1; i > 0; --i) rep(j, 0, i) {
      ll v = A[j][i];
      rep(k, 0, n) tmp[j][k] =
        (tmp[j][k] - v * tmp[i][k]) % mod;
    }
  rep(i, 0, n) rep(j, 0, n) A[col[i]][col[j]] =
    tmp[i][j] % mod + (tmp[i][j] < 0 ? mod : 0);
  return n;
}
```