University of Wroclaw

UWr 1

Krzysztof Boryczka, Antek Buraczewski, Łukasz Pluta

ICPC World Finals 2024

September 19, 2024

Contest (1)

template.cpp
84e42e, 37 lines

```cpp
#include "bits/stdc++.h"
using namespace std;

#define rep(i, b, e) for(int i = (b); i <= (e); i++)
#define per(i, b, e) for(int i = (e); i >= (b); i--)
#define FOR(i, b, e) rep(i, b, (e) - 1)
#define SZ(x) int(x.size())
#define all(x) x.begin(), x.end()
#define pb push_back
#define mp make_pair
#define st first
#define nd second
using ll = long long;
using vi = vector<int>;
using pii = pair<int, int>;

auto &operator<<(auto &o, pair<auto, auto> p) {
  return o << "(" << p.st << ", " << p.nd << ")"; }
auto operator<<(auto &o, auto x)->decltype(end(x), o) {
  o << "{"; int i=0; for(auto e: x) o << ", " + 2*!i++ << e;
  return o << "}"; }
#ifdef LOCAL
#define deb(x...) cerr << "[" #x "]: ", [](auto...$) { \
  ((cerr << $ << "; "),...) << endl; }(x)
#else
#define deb(...)
#endif

void solve() { }

int main() {
  cin.tie(0)->sync_with_stdio(0);
  int tt = 1;
  // cin >> tt;
  FOR(te, 0, tt) solve();
  return 0;
}
```

.bashrc
25 lines

```bash
# Slow compilation, mainly for debugging purposes
c() {
    g++ -std=gnu++20 -Wall -Wextra -Wshadow \
    -fsanitize=address,undefined -ggdb3 -DLOCAL $1.cpp -o $1
}
# Quick compilation, close to judge's one (w/o -static)
f() {
    g++ -std=gnu++20 -O2 -g $1.cpp -o $1
}
# Hashes a file, ignoring all whitespace and comments
# Use for verifying that code was correctly typed
hasz() {
    cpp -dD -P -fpreprocessed $1.cpp | tr -d '[:space:]' | \
    md5sum | cut -c-6
}
# Runs a tester
tester() {
  for ((i=0;;i++)); do
    ./gen > $1.test
    ./$1 < $1.test > $1.wzo
    ./brut < $1.test > $1.brut
    diff $1.wzo $1.brut || break
    echo "OK $i"
  done
}
```

valgrind.sh
7 lines

```bash
# Runtime
valgrind --tool=callgrind ./code < t.in
callgrind_annotate callgrind.out.12345
# Memory
valgrind --tool=massif ./code < t.in
ms_print massif.out.12345
ms_print massif.out.12345 | grep code
```

Data structures (2)

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). For PBDS $swap(a, b)$ works in $O(N)$, use $a.swap(b)$ instead!
3e474b, 12 lines

```cpp
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
// for places where hacking might be a problem:
const int R = chrono::high_resolution_clock::
  now().time_since_epoch().count();
struct chash { // to use most bits rather than just the lowest
  const uint64_t C = ll(4e18 * acos(0)) | 71;
  ll operator()(ll x) const {
    return __builtin_bswap64((x^R)*C); }
};
// 1 << 16 is initial size, don't put it if you want small map
gp_hash_table<ll, int, chash> h({},{},{},{}, {1 << 16});
```

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. For PBDS $swap(a, b)$ works in $O(N)$, use $a.swap(b)$ instead!
Time: $\mathcal{O}(\log N)$, join probably $\mathcal{O}(N)$
023ea1, 17 lines

```cpp
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
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).st;
  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); // O(N), assuming T<T2 or T>T2, merge t2 into t
}
```

UnionFindRollback.h
Description: Disjoint-set data structure with undo. If undo is not needed, skip st, time() and rollback().
Usage: int t = uf.time(); ...; uf.rollback(t);
Time: $\mathcal{O}(\log(N))$
2ccafc, 19 lines

```cpp
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].st] = st[i].nd;
    st.resize(t);
  }
  bool join(int a, int b) {
    a = find(a), b = find(b);
    if(a == b) return 0;
    if(e[a] > e[b]) swap(a, b);
    st.pb({a, e[a]}); st.pb({b, e[b]});
    e[a] += e[b]; e[b] = a;
    return 1;
  }
};
```

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, 29 lines

```cpp
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;
}
};
```

ExtendedLiChao.h
Description: Supports Range Line Insertion, Range Line Addition, Point Query. Can be modified to support Range Maximum Query, if Line Addition is replaced by Sum Addition.
Time: $\mathcal{O}\left(\log^2 N\right)$
<div align="right">d21694, 89 lines</div>

```
template<typename T>
struct Line {
    T a, b;
    Line() : a(0), b(-inf) {}
    Line(T a, T b) : a(a), b(b) {}
    T get(T x) { return a * x + b; }
    void add(Line x) { a += x.a; b += x.b; }
};

struct Node {
    Line<T> line, lazy(0, 0);
    Node *lc = nullptr, *rc = nullptr;
    void apply(T l, T r, Line<T> v) { line.add(v); lazy.add(v); }
};

void PushLazy(Node* &n, T tl, T tr) {
    if(n == nullptr) return;
    if(n->lc == nullptr) n->lc = new Node();
    if(n->rc == nullptr) n->rc = new Node();
    T mid = (tl + tr) / 2;
    n->lc->apply(tl, mid, n->lazy);
    n->rc->apply(mid + 1, tr, n->lazy);
    n->lazy = Line<T>(0, 0);
}

void PushLine(Node* &n, T tl, T tr) {
    if(n == nullptr) return;
    T mid = (tl + tr) / 2;
    InsertLineKnowingly(n->lc, tl, mid, n->line);
    InsertLineKnowingly(n->rc, mid + 1, tr, n->line);
    n->line = Line<T>();
}

void InsertLineKnowingly(Node* &n, T tl, T tr, Line<T> x) {
    if(n == nullptr) n = new Node();
    if(n->line.get(tl) < x.get(tl)) swap(n->line, x);
    if(n->line.get(tr) >= x.get(tr)) return;
    if(tl == tr) return;
    T mid = (tl + tr) / 2;
    PushLazy(n, tl, tr);
    if (n->line.get(mid) > x.get(mid)) {
        InsertLineKnowingly(n->rc, mid + 1, tr, x); }
    else {
        swap(n->line, x);
        InsertLineKnowingly(n->lc, tl, mid, x);
    }
}

void InsertLine(Node* &n, T tl, T tr, T l, T r, Line<T> x) {
    if(tr < l || r < tl || tl > tr || l > r) return;
    if(n == nullptr) n = new Node();
    if(l <= tl && tr <= r) return InsertLineKnowingly(n, tl, tr,
        x);
    T mid = (tl + tr) / 2;
    PushLazy(n, tl, tr);
    InsertLine(n->lc, tl, mid, l, r, x);
    InsertLine(n->rc, mid + 1, tr, l, r, x);
```

```
}

void AddLine(Node* &n, T tl, T tr, T l, T r, Line<T> x) {
    if(tr < l || r < tl || tl > tr || l > r) return;
    if(n == nullptr) n = new Node();
    if(l <= tl && tr <= r) return n->apply(tl, tr, x);
    T mid = (tl + tr) / 2;
    PushLazy(n, tl, tr);
    PushLine(n, tl, tr);
    AddLine(n->lc, tl, mid, l, r, x);
    AddLine(n->rc, mid + 1, tr, l, r, x);
}

T Query(Node* &n, T tl, T tr, T x) {
    if(n == nullptr) return -inf;
    if(tl == tr) return n->line.get(x);
    T res = n->line.get(x);
    T mid = (tl + tr) / 2;
    PushLazy(n, tl, tr);
    if(x <= mid) res = max(res, Query(n->lc, tl, mid, x));
    else res = max(res, Query(n->rc, mid + 1, tr, x));
    return res;
}

void InsertLine(T l, T r, Line<T> x) {
    return InsertLine(root, 0, sz - 1, l, r, x);
}

void AddLine(T l, T r, Line<T> x) {
    return AddLine(root, 0, sz - 1, l, r, x);
}

T Query(T x) { return Query(root, 0, sz - 1, x); }
```

FenwickTree.h
Description: Computes partial sums a[0] + a[1] + ... + a[pos - 1], and updates single elements a[i], taking the difference between the old and new value.
Time: $\mathcal{O}\left(\log N\right)$.
<div align="right">b06af0, 22 lines</div>

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

FenwickTree2D.h
Description: Computes sums a[i,j] for all i<I, j<J, and increases single elements a[i,j]. Requires that the elements to be updated are known in advance (call fakeUpdate() before init()).
Time: $\mathcal{O}\left(\log^2 N\right)$ (use persistent segment trees for $\mathcal{O}\left(\log N\right)$)
<div align="right">"FenwickTree.h"   cc92f9, 21 lines</div>

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

PersistentSegmentTree.h
Description: Persistent structures use a lot of memory in general, so remember to optimise the code whenever possible. Push function is responsible for creating copies of children, so whenever we enter a vertex (either add or query) it's already a fresh copy. When there is no lazy, it's better to modify this approach, and only create a vertex when we really visit it, so we can save memory.
Time: $\mathcal{O}\left(\log N\right)$ per operation
<div align="right">c5c0ae, 58 lines</div>

```
struct PersistentTree {
    using T = ll; // persistency needs a lot of memory
    struct Node { // so make sure to make it efficient
        int ch[2] = {-1, -1}; T sum, mini, add;
        Node() : sum(0), mini(0), add(0) {} // start / neutral
        Node(T val) : sum(val), mini(val), add(0) {}
        void apply(T val, int len) {
            sum += val * len; mini += val; add += val;
        }
        void pull(Node &l, Node &r) {
            sum = l.sum + r.sum; mini = min(l.mini, r.mini);
        }
    };
    vector<Node> t; int max_n;
    PersistentTree(int _max_n) : max_n(_max_n) {}
    PersistentTree(vector<T> vals) : max_n(SZ(vals) - 1) {
        function<int(int, int)> build = [&](int l, int r) {
            if(l == r) { t.pb(Node(vals[l])); return SZ(t)-1; }
            int mid = (l + r) / 2; Node a;
            a.ch[0] = build(l, mid), a.ch[1] = build(mid + 1, r);
            a.pull(t[a.ch[0]], t[a.ch[1]]);
            t.pb(a); return SZ(t) - 1;
        };
        build(0, max_n);
    }
    int copy(int v) { t.pb(~v ? t[v]:Node()); return SZ(t)-1; }
    void push(int v, int len) {
        FOR(i, 0, 2) {
            t[v].ch[i] = copy(t[v].ch[i]);
            t[t[v].ch[i]].apply(t[v].add, len / 2);
        }
        t[v].add = 0;
    }
    void add(int v, int l, int r, T val, int le, int re) {
        if(re < l || r < le) return;
        if(l <= le && re <= r) return t[v].apply(val, re-le + 1);
        push(v, re - le + 1); int mid = (le + re) / 2;
```

```
      add(t[v].ch[0], l, r, val, le, mid);
      add(t[v].ch[1], l, r, val, mid + 1, re);
      t[v].pull(t[t[v].ch[0]], t[t[v].ch[1]]);
   }
   Node query(int v, int l, int r, int le, int re) {
      if(re < l || r < le) return Node();
      if(l <= le && re <= r) return t[v];
      push(v, re - le + 1); int mid = (le + re) / 2;
      Node a = query(t[v].ch[0], l, r, le, mid);
      Node b = query(t[v].ch[1], l, r, mid + 1, re);
      Node res; res.pull(a, b); return res;
   }
   void add(int &v, int l, int r, T val) {
      v = copy(v); add(v, l, r, val, 0, max_n);
   }
   Node query(int v, int l, int r) {
      int sz = SZ(t);
      v = copy(v); Node res = query(v, l, r, 0, max_n);
      t.resize(sz); return res;
   }
};
```

Treap.h

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

2ad8fd, 122 lines

```
struct Treap {
   struct Node {
      int ch[2] = {0, 0};
      int rank = rand(), size = 0;
      int val = 0, mn = 1e9, sum = 0; // Subtree aggregates
      bool flip = 0; int add = 0;    // Lazy tags
      Node() {}
      Node(int v) : size(1), val(v), mn(v), sum(v) {}
   };
   vector<Node> t;
   Treap() : t(1) {}

   void pull(int v) {
      auto [l, r] = t[v].ch;
      t[v].size = t[l].size + 1 + t[r].size;
      t[v].mn = min({t[l].mn, t[v].val, t[r].mn});
      t[v].sum = t[l].sum + t[v].val + t[r].sum;
   }

   int apply(int v, bool flip, int add) {
      if(!v) return 0;
      // t.pb(t[v]), v = SZ(t) - 1;       // <- persistency
      if(flip) t[v].flip ^= 1, swap(t[v].ch[0], t[v].ch[1]);
      t[v].val += add; t[v].mn += add;
      t[v].sum += add * t[v].size;
      t[v].add += add;
      return v;
   }

   void push(int v) {
      FOR(i, 0, 2) {
         t[v].ch[i] = apply(t[v].ch[i], t[v].flip, t[v].add);
      }
      t[v].flip = 0; t[v].add = 0;
   }

   pii split(int v, int k) {
      if(!v) return {0, 0};
      push(v);
      auto [l, r] = t[v].ch;
      if(k <= t[l].size) {
      // if(k <= t[v].val) {              // <- by values
```

```
         auto [p, q] = split(l, k);
         t[v].ch[0] = q, pull(v);
         return {p, v};
      }
      else {
         auto [p, q] = split(r, k - t[l].size - 1);
         // auto [p, q] = split(r, k);    // <- by values
         t[v].ch[1] = p, pull(v);
         return {v, q};
      }
   }

   int merge(int v, int u) {
      if(!v || !u) return v ^ u;
      push(v), push(u);
      if(t[v].rank > t[u].rank) {
         t[v].ch[1] = merge(t[v].ch[1], u);
         return pull(v), v;
      }
      else {
         t[u].ch[0] = merge(v, t[u].ch[0]);
         return pull(u), u;
      }
   }

   void insert(int &v, int pos, int val) {
      // if(v) t.pb(t[v]), v = SZ(t) - 1;   // <- persistency
      auto [p, q] = split(v, pos);
      t.pb(Node(val)); int u = SZ(t) - 1;
      // t.pb(Node(pos)); int u = SZ(t) - 1;  // <- by values
      v = merge(merge(p, u), q);
   }

   void erase(int &v, int l, int r) {
      // if(v) t.pb(t[v]), v = SZ(t) - 1;   // <- persistency
      auto [p, q] = split(v, l);
      auto [u, s] = split(q, r - l + 1);
      // auto [u, s] = split(q, r + 1);    // <- by values
      v = merge(p, s);
   }

   void modify(int &v, int l, int r, bool flip, int add) {
      // if(v) t.pb(t[v]), v = SZ(t) - 1;   // <- persistency
      auto [p, q] = split(v, l);
      auto [u, s] = split(q, r - l + 1);
      // auto [u, s] = split(q, r + 1);    // <- by values
      u = apply(u, flip, add);
      v = merge(merge(p, u), s);
   }

   pii get(int &v, int l, int r) {
      // if(v) t.pb(t[v]), v = SZ(t) - 1;   // <- persistency
      auto [p, q] = split(v, l);
      auto [u, s] = split(q, r - l + 1);
      // auto [u, s] = split(q, r + 1);    // <- by values
      int mn = t[u].mn, sum = t[u].sum;
      v = merge(merge(p, u), s);
      return {mn, sum};
   }

// only when by values
   int join(int v, int u) {
      if(!v || !u) return v ^ u;
      if(t[v].rank < t[u].rank) swap(v, u);
      auto [p, q] = split(u, t[v].val);
      push(v);
      t[v].ch[0] = join(t[v].ch[0], p);
      t[v].ch[1] = join(t[v].ch[1], q);
      return pull(v), v;
```

```
   }

   // only when by values, persistency destroys complexity
   void increase(int &v, int l, int r, int increase) {
      // if(v) t.pb(t[v]), v = SZ(t) - 1; // <- persistency
      auto [p, q] = split(v, l);
      auto [u, s] = split(q, r + 1);
      u = apply(u, 0, increase);
      v = join(merge(p, s), u);
   }
};
```

RMQ.h

Description: Range Minimum Queries on an array. Returns min(V[a], V[a + 1], ... V[b - 1]) in constant time.
Time: $\mathcal{O}(|V| \log |V| + Q)$

1e70c8, 16 lines

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

MoQueries.h

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

c9cf1e, 48 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.st) add(--L, 0);
      while(R < q.nd) add(R++, 1);
      while(L < q.st) del(L++, 0);
      while(R > q.nd) 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) FOR(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;
}
```

HilbertOrder.h
Description: Useful speed up for MoQueries.
Time: $\mathcal{O}(\log N)$

8ce340, 14 lines

```
ll hilbertOrder(int x, int y, int pow = 20, int rotate = 0) {
  if(pow == 0) return 0;
  int hpow = 1 << (pow - 1);
  int seg = x < hpow ? (y < hpow ? 0:3) : (y < hpow ? 1:2);
  seg = (seg + rotate) & 3;
  const int rotateDelta[4] = {3, 0, 0, 1};
  int nx = x & (x ^ hpow), ny = y & (y ^ hpow);
  int nrot = (rotate + rotateDelta[seg]) & 3;
  ll subSquareSize = 1ll << (pow * 2 - 2);
  ll ans = seg * subSquareSize;
  ll add = hilbertOrder(nx, ny, pow - 1, nrot);
  ans += seg == 1 || seg == 2 ? add : (subSquareSize - add-1);
  return ans;
}
```

Numerical (3)

3.1 Polynomials and recurrences

Poly.h

fdbe2d, 18 lines

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

PolyRoots.h
Description: Finds the real roots to a polynomial.
Usage: polyRoots({{2,-3,1}},-1e9,1e9) // solve x^2-3x+2 = 0

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

"Poly.h"   9efa40, 21 lines

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

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

c57814, 13 lines

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

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

74d472, 17 lines

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

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

879be8, 21 lines

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

Polynomial.h
Description: Basic polynomial subroutines. Inputs must be in [0, mod). Operator % works ~1s for $N = 5 \cdot 10^5$ (using NTT). NTT is rather slow, FFTmod works ~2x faster.
Time: $\mathcal{O}(N \log N)$

"NumberTheoreticTransform.h"   19336f, 41 lines

```
struct Poly {
  vll a;
  Poly(ll a0 = 0) : a(1, a0) { normalize(); }
  Poly(vll _a) : a(_a) { normalize(); }
  void normalize() {
    while(SZ(a) && a.back() == 0) a.pop_back();
  }
  Poly rev() { auto b = a; reverse(all(b)); return b; }
  Poly trim(int n) {
    return vll(a.begin(), a.begin() + min(n, SZ(a)));
  }
  Poly inv(int n) {
    Poly q(modpow(a[0], mod - 2));
    for(int i = 1; i < n; i <<= 1) {
      Poly p = Poly(2 % mod) - q * trim(i * 2);
      q = (p * q).trim(i * 2);
    }
    return q.trim(n);
  }
  Poly operator+(Poly b) {
    vll r(a); r.resize(max(SZ(r), SZ(b.a)));
    FOR(i, 0, SZ(b.a))
      r[i] += b.a[i], r[i] -= r[i] >= mod ? mod : 0;
    return r;
  }
  Poly operator-(Poly b) {
    vll r(a); r.resize(max(SZ(r), SZ(b.a)));
    FOR(i, 0, SZ(b.a))
      r[i] -= b.a[i], r[i] += r[i] < 0 ? mod : 0;
    return r;
  }
  Poly operator*(Poly b) { return Poly(conv(a, b.a)); }
  Poly operator/(Poly b) {
    if(SZ(a) < SZ(b.a)) return Poly();
```

```
    int sz = SZ(a) - SZ(b.a) + 1;
    Poly ra = rev().trim(sz), rb = b.rev().trim(sz).inv(sz);
    vll res = (ra * rb).a; res.resize(sz); reverse(all(res));
    return res;
  }
  Poly operator%(Poly b) { return *this - *this / b * b; }
};
```

LinearRecurrenceFast.h
Description: Generates the k'th term of an n-order linear recurrence $a[i] = \sum_j a[i - j - 1]c[j]$, given $a[0\dots \geq n - 1]$ and $c[0\dots n - 1]$. Can be generalized to generate many consecutive terms. Works $\sim$5s for $N = 10^5$ (using FFTmod).
Time: $\mathcal{O}\left(N \log N \log K\right)$

"Polynomial.h"   ccc23a, 21 lines
```
ll linearRecFast(vll a, vll c, ll k) {
  reverse(all(c)); c.pb(-1);
  Poly d(1), xn({0, 1}), f(Poly() - c);
  Poly hf = f.rev().inv(SZ(f.a)); // compute inverse only once
  auto divFast = [&](Poly p) {
    if(SZ(p.a) < SZ(hf.a)) return Poly();
    int sz = SZ(p.a) - SZ(hf.a) + 1;
    Poly ra = p.rev().trim(sz), rb = hf.trim(sz);
    vll res = (ra * rb).a; res.resize(sz); reverse(all(res));
    return Poly(res);
  };
  auto modFast = [&](Poly p) { return p - divFast(p) * f; };
  for(; k; k >>= 1) {
    if(k & 1) d = modFast(d * xn);
    xn = modFast(xn * xn);
  }
  ll res = 0;
  FOR(i, 0, min(SZ(a), SZ(d.a)))
    res = (res + a[i] * d.a[i]) % mod;
  return res;
}
```

3.2 Optimization

Integrate.h
Description: Simple integration of a function over an interval using Simpson's rule. The error should be proportional to h^4, although in practice you will want to verify that the result is stable to desired precision when epsilon changes.
26598f, 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);
  FOR(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; });});});
aec921, 14 lines
```
using D = double;
#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, S1 = S(a, c), S2 = S(c, b), T = S1 + S2;
  if(abs(T - S) <= 15 * eps || b - a < 1e-10) {
    return T + (T - S) / 15; }
  return rec(f, a, c, eps / 2, S1) + rec(f, c, b, eps / 2, S2);
}
```

```
template<class F>
D quad(D a, D b, F f, D eps = 1e-8) {
  return rec(f, a, b, eps, S(a, b));
}
```

Simplex.h
Description: Solves a general linear maximization problem: maximize $c^T x$ subject to $Ax \leq b$, $x \geq 0$. Returns -inf if there is no solution, inf if there are arbitrarily good solutions, or the maximum value of $c^T x$ otherwise. The input vector is set to an optimal x (or in the unbounded case, an arbitrary solution fulfilling the constraints). Numerical stability is not guaranteed. For better performance, define variables such that $x = 0$ is viable.
Usage: vvd A = {{1,-1}, {-1,1}, {-1,-2}};
vd b = {1,1,-4}, c = {-1,-1}, x;
T val = LPSolver(A, b, c).solve(x);
Time: $\mathcal{O}(NM * \#pivots)$, where a pivot may be e.g. an edge relaxation. $\mathcal{O}\left(2^n\right)$ in the general case.
c62874, 64 lines
```
using T = double; // long double, Rational, double + mod<P>...
using vd = vector<T>;
using vvd = vector<vd>;

const T eps = 1e-8, inf = 1/.0;
#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)) {
      FOR(i,0,m) FOR(j,0,n) D[i][j] = A[i][j];
      FOR(i,0,m) { B[i] = n+i; D[i][n] = -1; D[i][n+1] = b[i];}
      FOR(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];
    FOR(i, 0, m+2) if(i != r && abs(D[i][s]) > eps) {
      T *b = D[i].data(), inv2 = b[s] * inv;
      FOR(j, 0, n+2) b[j] -= a[j] * inv2;
      b[s] = a[s] * inv2;
    }
    FOR(j, 0, n+2) if(j != s) D[r][j] *= inv;
    FOR(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;
      FOR(j, 0, n+1) if(N[j] != -phase) ltj(D[x]);
      if(D[x][s] >= -eps) return 1;
      int r = -1;
      FOR(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 0;
      pivot(r, s);
    }
  }

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

3.3 Matrices

Determinant.h
Description: Calculates determinant of a matrix. Destroys the matrix.
Time: $\mathcal{O}\left(N^3\right)$
1de5db, 15 lines
```
double det(vector<vector<double>> &a) {
  int n = SZ(a); double res = 1;
  FOR(i, 0, n) {
    int b = i;
    FOR(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;
    FOR(j, i+1, n) {
      double v = a[j][i] / a[i][i];
      if(v != 0) FOR(k, i+1, n) a[j][k] -= v * a[i][k];
    }
  }
  return res;
}
```

IntDeterminant.h
Description: Calculates determinant using modular arithmetics. Modulos can also be removed to get a pure-integer version.
Time: $\mathcal{O}\left(N^3\right)$
46be95, 16 lines
```
const ll mod = 12345;
ll det(vector<vector<ll>> &a) {
  int n = SZ(a); ll ans = 1;
  FOR(i, 0, n) {
    FOR(j, i+1, n) while(a[j][i] != 0) { // gcd step
      ll t = a[i][i] / a[j][i];
      if(t) FOR(k, i, n)
        a[i][k] = (a[i][k] - a[j][k] * t) % mod; }
      swap(a[i], a[j]);
      ans *= -1;
    }
    ans = ans * a[i][i] % mod;
    if(!ans) return 0;
  }
  return (ans + mod) % mod;
}
```

SolveLinear.h
Description: Solves $A * x = b$. If there are multiple solutions, an arbitrary one is returned. Returns rank, or -1 if no solutions. Data in A and b is lost.
Time: $\mathcal{O}\left(n^2 m\right)$
cfaaca, 33 lines
```
using vd = vector<double>;
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);
```

```
  FOR(i, 0, n) {
    double v, bv = 0;
    FOR(r, i, n) FOR(c, i, m) if((v = fabs(A[r][c])) > bv) {
      br = r, bc = c, bv = v; }
    if(bv <= eps) {
      FOR(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]);
    FOR(j, 0, n) swap(A[j][i], A[j][bc]);
    bv = 1 / A[i][i];
    FOR(j, i+1, n) {
      double fac = A[j][i] * bv;
      b[j] -= fac * b[i];
      FOR(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];
    FOR(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 Solve-Linear, make the following changes:

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

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

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

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

```
int matInv(vector<vector<double>> &A) {
  int n = SZ(A); vi col(n);
  vector<vector<double>> tmp(n, vector<double>(n));
  FOR(i, 0, n) tmp[i][i] = 1, col[i] = i;
  FOR(i, 0, n) {
    int r = i, c = i;
    FOR(j, i, n) FOR(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]);
    FOR(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];
    FOR(j, i+1, n) {
      double f = A[j][i] / v;
      A[j][i] = 0;
      FOR(k, i+1, n) A[j][k] -= f*A[i][k];
      FOR(k, 0, n) tmp[j][k] -= f*tmp[i][k];
    }
    FOR(j, i+1, n) A[i][j] /= v;
    FOR(j, 0, n) tmp[i][j] /= v;
    A[i][i] = 1;
  }
  per(i, 0, n-1) FOR(j, 0, i) {
    double v = A[j][i];
    FOR(k, 0, n) tmp[j][k] -= v*tmp[i][k];
  }
  FOR(i, 0, n) FOR(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 $A^{-1} = A^{-1}(2I - AA^{-1}) \pmod{p^k}$ where A^{-1} starts as the inverse of A mod p, and k is doubled in each step.
Time: $\mathcal{O}\left(n^3\right)$

```
int matInv(vector<vector<ll>>& A) {
  int n = SZ(A); vi col(n);
  vector<vector<ll>> tmp(n, vector<ll>(n));
  FOR(i, 0, n) tmp[i][i] = 1, col[i] = i;
  FOR(i, 0, n) {
    int r = i, c = i;
    FOR(j, i, n) FOR(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]);
    FOR(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);
    FOR(j, i+1, n) {
      ll f = A[j][i] * v % mod;
      A[j][i] = 0;
      FOR(k, i+1, n) A[j][k] = (A[j][k] - f*A[i][k]) % mod;
      FOR(k,0,n) tmp[j][k] = (tmp[j][k] - f*tmp[i][k]) % mod;
    }
```

```
    FOR(j, i+1, n) A[i][j] = A[i][j] * v % mod;
    FOR(j, 0, n) tmp[i][j] = tmp[i][j] * v % mod;
    A[i][i] = 1;
  }
  per(i, 0, n-1) FOR(j, 0, i) {
    ll v = A[j][i];
    FOR(k, 0, n) tmp[j][k] = (tmp[j][k] - v*tmp[i][k]) % mod;
  }
  FOR(i, 0, n) FOR(j, 0, n) {
    A[col[i]][col[j]] = tmp[i][j] % mod +
    (tmp[i][j] < 0 ? mod : 0); }
  return n;
}
```

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

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

This is useful for solving problems on the type

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

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

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

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

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

3.4 Fourier transforms
FastFourierTransform.h

Description: fft(a) computes $\hat{f}(k) = \sum_x a[x] \exp(2\pi i \cdot kx/N)$ for all k. N must be a power of 2. Useful for convolution: conv(a, b) = c, where $c[x] = \sum a[i]b[x-i]$. For convolution of complex numbers or more than two vectors: FFT, multiply pointwise, divide by n, reverse(start+1, end), FFT back. Rounding is safe if $(\sum a_i^2 + \sum b_i^2) \log_2 N < 9 \cdot 10^{14}$ (in practice 10^{16}; higher for random inputs). Otherwise, use NTT/FFTMod.
Time: $\mathcal{O}(N \log N)$ with $N = |A| + |B|$ (~1s for $N = 2^{22}$)   87d73c, 32 lines

```
using C = complex<double>;
using vd = vector<double>;
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);
    FOR(i, k, 2*k) rt[i] = R[i] = i&1 ? R[i/2] * x : R[i/2];
  }
  vi rev(n);
  FOR(i, 0, n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
  FOR(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) FOR(j,0,k) {
      C z = rt[j+k] * a[i+j+k]; // (25% faster if hand-rolled)
      a[i + j + k] = a[i + j] - z; a[i + j] += z;
    }
}
vd conv(const vd &a, const vd &b) {
  if(a.empty() || b.empty()) return {};
  vd res(SZ(a) + SZ(b) - 1);
  int L = 32 - __builtin_clz(SZ(res)), n = 1 << L;
  vector<C> in(n), out(n); copy(all(a), begin(in));
  FOR(i,0,SZ(b)) in[i].imag(b[i]);
  fft(in); for(C &x : in) x *= x;
  FOR(i, 0, n) out[i] = in[-i & (n - 1)] - conj(in[i]);
  fft(out);
  FOR(i, 0, SZ(res)) res[i] = imag(out[i]) / (4 * n);
  return res;
}
```

FastFourierTransformMod.h
Description: Higher precision FFT, can be used for convolutions modulo arbitrary integers as long as $N \log_2 N \cdot \text{mod} < 8.6 \cdot 10^{14}$ (in practice 10^{16} or higher). Inputs must be in $[0, \text{mod})$.
Time: $\mathcal{O}(N \log N)$, where $N = |A| + |B|$ (twice as slow as NTT or FFT)
"FastFourierTransform.h"   db1041, 22 lines

```
using vll = vector<ll>;
template<int M> vll convMod(const vll &a, const vll &b) {
  if(a.empty() || b.empty()) return {};
  vll 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);
  FOR(i,0,SZ(a)) L[i] = C((int)a[i] / cut, (int)a[i] % cut);
  FOR(i,0,SZ(b)) R[i] = C((int)b[i] / cut, (int)b[i] % cut);
  fft(L), fft(R);
  FOR(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);
  FOR(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;
}
```

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 $[0, \text{mod})$.
Time: $\mathcal{O}(N \log N)$ 16bf30, 33 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.
using vll = vector<ll>;
void ntt(vll &a) {
  int n = SZ(a), L = 31 - __builtin_clz(n);
  static vll 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)};
    FOR(i, k, 2*k) rt[i] = rt[i / 2] * z[i & 1] % mod;
  }
  vi rev(n);
  FOR(i, 0, n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
  FOR(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) FOR(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);
    }
}
vll conv(const vll &a, const vll &b) {
  if(a.empty() || b.empty()) return {};
  int s = SZ(a) + SZ(b) - 1, B = 32 - __builtin_clz(s);
  int n = 1 << B, inv = modpow(n, mod - 2);
  vll L(a), R(b), out(n);
  L.resize(n), R.resize(n);
  ntt(L), ntt(R);
  FOR(i,0,n) out[-i & (n-1)] = L[i] * R[i] % mod * inv % mod;
  ntt(out);
  return {out.begin(), out.begin() + s};
}
```

FastSubsetTransform.h
Description: Transform to a basis with fast convolutions of the form $c[z] = \sum_{z=x\oplus y} a[x] \cdot b[y]$, where $\oplus$ is one of AND, OR, XOR. The size of a must be a power of two.
Time: $\mathcal{O}(N \log N)$ 846b5e, 16 lines

```
void FST(vi &a, bool inv) {
  for(int n = SZ(a), step = 1; step < n; step *= 2) {
    for(int i = 0; i < n; i += 2 * step) FOR(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);
  FOR(i, 0, SZ(a)) a[i] *= b[i];
  FST(a, 1); return a;
}
```

Number theory (4)

4.1 Modular arithmetic

ModMulLL.h
Description: Calculate $a \cdot b \mod c$ (or $a^b \mod c$) 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   4c7c4a, 11 lines

```
using ull = unsigned long long;
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;
}
```

ModSum.h
Description: Sums of mod'ed arithmetic progressions.
modsum(to, c, k, m) = $\sum_{i=0}^{to-1} (ki+c)\%m$. divsum is similar but for floored division.
Time: $\log(m)$, with a large constant.   aba7f7, 14 lines

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

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

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

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: $\mathcal{O}(\sqrt{m})$ fd054b, 10 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)) FOR(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 19a793, 19 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;
    }
}
```

MinLinearMod.h
Description: MinRem calculates minimum value of $(ax + b) \pmod{m}$ for $x \in [0, k]$. FirstInRange returns minimum x s.t. $(ax + b) \pmod{m} \in [l, r]$. If there is no solution, both of them return -1.
Time: $\mathcal{O}(\log m)$

```
ll minRem(ll a, ll b, ll m, ll k) {
  if(k < 0) return -1;
  ll g = gcd(a, m);
  if(g > 1) return g * minRem(a/g, b/g, m/g, k) + b % g;
  auto posMod = [](ll x, ll mod) {
    ll res = x % mod; return res < 0 ? res + mod : res;
  };
  auto getSteps = [](ll t, ll ia, ll x, ll mod) {
    return (t - x + mod) * ia % mod;
  };
  for(ll b0 = b, m0 = m, ia0 = modInv(a, m); a;) {
    ll na, nb, nm;
    if(a > m - a) {
      na = a % (m - a); nb = b % (m - a); nm = m - a;
      for(ll steps = getSteps(nb, ia0, b0, m0); steps > k;) {
        ll add = steps - getSteps(nb + nm, ia0, b0, m0);
        return nb + nm * ((steps - k + (add - 1)) / add);
      }
    }
    else {
      na = posMod(-m, a);
      nb = (b < a ? b : posMod(b - m, a)); nm = a;
      if(getSteps(nb, ia0, b0, m0) > k) break;
    }
    a = na; b = nb; m = nm;
  }
  return b;
}

ll firstInRange(ll a, ll b, ll m, ll l, ll r) {
  ll g = gcd(a, m);
  if(g > 1) return firstInRange(a/g, b/g, m/g,
    l/g + (l % g > b % g), r/g - (r % g < b % g));
  ll ia = modInv(a, m);
  return minRem(ia, (l - b + m) * ia % m, m, r - l);
}
```

4.2 Primality

FastEratosthenes.h
Description: Prime sieve for generating all primes smaller than LIM.
Time: LIM=1e9 $\approx$ 1.5s

```
const int LIM = 1e6;
bitset<LIM> isPrime;
vi eratosthenes() {
  const int S = (int)round(sqrt(LIM)), R = LIM / 2;
  vi pr = {2}, sieve(S+1); pr.reserve(int(LIM/log(LIM)*1.1));
  vector<pii> cp;
  for(int i = 3; i <= S; i += 2) if(!sieve[i]) {
```

```
    cp.pb({i, i * i / 2});
    for(int j = i * i; j <= S; j += 2 * i) sieve[j] = 1;
  }
  for(int L = 1; L <= R; L += S) {
    array<bool, S> block{};
    for(auto &[p, idx]: cp)
      for(int i=idx; i < S+L; idx = (i+=p)) block[i-L] = 1;
    FOR(i, 0, min(S, R - L))
      if(!block[i]) pr.pb((L + i) * 2 + 1);
  }
  for(int i: pr) isPrime[i] = 1;
  return pr;
}
```

LinearSieve.h
Description: Usefull for computing values of multiplicative function and its prefix sums.
Time: $\mathcal{O}(N)$

```
struct LinearSieve {
  vector<bool> isComposite; vi prime, cnt;
  vector<ll> phi, prefPhi;
  ll dPhi(ll x, int p, int a) { // x / phi(p^(a-1)) * phi(p^a)
    return x * (a == 1 ? p - 1 : p);
  }
  LinearSieve(int n) : isComposite(n), cnt(n), phi(n) {
    if(n > 1) phi[1] = 1;
    FOR(i, 2, n) {
      if(!isComposite[i]) {
        prime.pb(i), cnt[i] = 1, phi[i] = dPhi(1, i, 1);
      }
      FOR(j, 0, SZ(prime)) {
        if(i * prime[j] >= n) break;
        isComposite[i * prime[j]] = 1;
        if(i % prime[j] == 0) {
          cnt[i * prime[j]] = cnt[i] + 1;
          phi[i*prime[j]] = dPhi(phi[i], prime[j], cnt[i]+1);
          break;
        }
        else {
          cnt[i * prime[j]] = 1;
          phi[i * prime[j]] = phi[i] * phi[prime[j]];
        }
      }
    }
    partial_sum(all(phi), back_inserter(prefPhi));
  }
};
```

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 $a^b \bmod c$.

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

Factor.h
Description: Pollard-rho randomized factorization algorithm. Returns prime factors of a number, in arbitrary order (e.g. 2299 -> {11, 19, 11}).
Time: $\mathcal{O}(n^{1/4})$, less for numbers with small factors.

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

4.3 Divisibility

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

```
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;
}

ll modInv(ll a, ll mod) {
  ll inv, b; euclid(a, mod, inv, b); inv %= mod;
  return inv < 0 ? inv + mod : inv;
}
```

CRT.h
Description: Chinese Remainder Theorem.
crt(a, m, b, n) computes x such that $x \equiv a \pmod{m}$, $x \equiv b \pmod{n}$. If $|a| < m$ and $|b| < n$, x will obey $0 \le x < \text{lcm}(m, n)$. Assumes $mn < 2^{62}$.
Time: $\log(n)$

```
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;
}
```

4.4 Other

FracBinarySearch.h
Description: Given f and N, finds the smallest fraction $p/q \in [0, 1]$ such that $f(p/q)$ is true, and $p, q \le N$. 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: $\mathcal{O}(\log(N))$

```
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;
}
```

PrefixSumMultiplicative.h
Description: Computes prefix sum $F(x)$ of multiplicative function $f(x)$ using Dirichlet inverse. To use it you need to find such a function $g(x)$, that for $c(x) = (f*g)(x)$ being their Dirichlet convolution, $G(x)$ and $C(x)$ can be quickly calculated. As $F(x)$ is multiplicative, standard trick is to compute small values using linear sieve. Provided $F(x)$ should work for $x < th$, $G(x)$ and $C(x)$ should work for $x \leq N$. Threshold th should be $\sim N^{\frac{2}{3}}$ (for single query). Hashmap can be changed to simple array (for single query), use $\frac{x}{i}$ as "hash function".
Time: $\mathcal{O}\left(N^{\frac{2}{3}}\right)$

f157e9, 16 lines

```
struct PrefSumMul {
    function<ll(ll)> F, G, C; ll th;
    unordered_map<ll, ll> mem;
    PrefSumMul(auto _F, auto _G, auto _C, ll _th) :
        F(_F), G(_G), C(_C), th(_th) {}
    ll calc(ll x) {
        if(x < th) return F(x);
        auto d = mem.find(x); if(d != mem.end()) return d->nd;
        ll res = 0;
        for(ll i = 2, la; i <= x; i = la + 1) {
            la = x / (x / i); // largest with same value
            res = (res + (G(la) - G(i-1) + mod) * calc(x/i)) % mod;
        } // if G(1) != 1 then use modInv there
        return mem[x] = (C(x) - res + mod) / G(1) % mod;
    }
};
```

4.5 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.

4.6 Mobius Function

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

Mobius Inversion:

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

Other useful formulas/forms:

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

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

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

Combinatorial (5)

5.1 Permutations

5.1.1 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

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

5.1.2 Young diagram
Given a partition λ, the Young diagram $Y(\lambda)$ is the set of pairs (i,j) of positive integers s.t. $i \leq |\lambda|$ and $j \leq \lambda_i$. Such pairs are called the squares of the diagram.

A normalised Young tableau P of size n is a pair (λ, p), where p is a permutation indexed by $Y(\lambda)$ s.t.
$p_{i,j} < \min(p_{i+1,j}, p_{i,j+1})$.

There is a bijection f from permutations π of length n to pairs $(P, Q) = ((\lambda, p), (\lambda, q))$ of Young tableaus of size n, which share the same partition.

This mapping (which can be computed by RSK) has the following properties:

- $\sum_{i \leq m} \lambda_i$ is the maximum total size of m disjoint **increasing** subsequences of the permutation

- $\sum_{j \leq m} \lambda_j^T$ is the maximum total size of m disjoint **decreasing** subsequences of the permutation

- $f(\pi^{-1}) = (Q, P)$

- $f(rev(\pi)) = (P^T, Sch(Q))$ – Sch is the normalised result of the Schützenberger involution

If $c(\lambda)$ is the number of valid p for the partition λ, then $n! = \sum_{\lambda \in P_n} c(\lambda)^2$.
The number of permutations s.t. $\pi = \pi^{-1}$ is $\sum_{\lambda \in P_n} c(\lambda)$.

5.1.3 Hook-length formula
For $\lambda \in P_n$ and $(i,j) \in Y(\lambda)$, define $h_\lambda(i,j)$ as the number of squares in the diagram directly below or to the right of square (i,j): $h_\lambda(i,j) = (\lambda_j^T - i) + (\lambda_i - j) + 1$.

The hook-length theorem states that:

$$c(\lambda) = \frac{n!}{\prod_{(i,j) \in Y(\lambda)} h_\lambda(i,j)}$$

5.2 General purpose numbers

5.2.1 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:

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

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

5.2.2 Stirling numbers of the first kind
Number of permutations on n items with k cycles.

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

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

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

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

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

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

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

5.2.4 Stirling numbers of the second kind
Partitions of n distinct elements into exactly k groups.

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

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

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

5.2.5 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,

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

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

RSK.h

Description: Computes mapping from permutations to pairs of normalised Young diagrams (Robinson-Schensted correspondence). Properties of this mapping are described in Young diagram section.

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

<div align="right">08bec9, 24 lines</div>

```cpp
vector<vi> boundedRSK(vi pi, int k) {
  vector<vi> res;
  for(auto x: pi) FOR(j, 0, k) {
    if(j >= SZ(res)) res.pb({});
    int pos = lower_bound(all(res[j]), x) - res[j].begin();
    if(pos == SZ(res[j])) { res[j].pb(x); break; }
    swap(x, res[j][pos]);
  }
  return res;
}

pair<vector<vi>, vector<vi>> fastRSK(vi pi) {
  int n = SZ(pi), sq = sqrt(n) + 1;
  vi invPi(n); FOR(i, 0, n) invPi[pi[i]] = i;
  auto p = boundedRSK(invPi, sq), q = boundedRSK(invPi, sq);
  reverse(all(pi)), reverse(all(invPi));
  auto pCol = boundedRSK(pi, sq), qCol = boundedRSK(invPi, sq);
  if(n > 0) p.resize(SZ(pCol[0])), q.resize(SZ(qCol[0]));
  FOR(i, sq, SZ(p)) FOR(j, 0, SZ(pCol)) {
    if(SZ(pCol[j]) <= i) break;
    p[i].pb(pCol[j][i]), q[i].pb(qCol[j][i]);
  }
  return {p, q};
}
```

RSKinverse.h

Description: Computes inverse of mapping from permutations to pairs of normalised Young diagrams (Robinson-Schensted correspondence).

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

<div align="right">8412c3, 18 lines</div>

```cpp
vi RSKinverse(vector<vi> p, vector<vi> q) {
  int n = 0; for(auto &x: p) n += SZ(x);
  vi pi(n);
  per(ind, 0, n-1) {
    int i, j;
    for(i = 0; i < SZ(q); i++) {
      for(j = 0; j < SZ(q[i]); j++) if(q[i][j] == ind) break;
      if(j < SZ(q[i])) break;
    }
    int x = p[i][j]; p[i].pop_back(), q[i].pop_back();
    for(i--; i >= 0; i--) {
      j = lower_bound(all(p[i]), x) - p[i].begin();
      swap(p[i][j - 1], x);
    }
    pi[ind] = x;
  }
  return pi;
}
```

RSKrecover.h

Description: Recovers k increasing disjoint subsequences, which cover the maximum possible number of elements from permutation pi.

Time: $\mathcal{O}(NM)$, where M is the length of the output

<div align="right">6865f3, 38 lines</div>

```cpp
vector<vi> RSKrecover(vi pi, int k) {
  int n = SZ(pi); vector<vi> h;
  vector<tuple<int, int, int>> swaps;
  for(auto x: pi) FOR(j, 0, k) {
    if(j >= SZ(h)) h.pb({});
    if(!SZ(h[j]) || h[j].back() < x) { h[j].pb(x); break; }
    per(y, 0, SZ(h[j])-1) {
      if(y == 0 || h[j][y-1] < x) { swap(x, h[j][y]); break; }
      swaps.pb({x, h[j][y-1], h[j][y]});
    }
  }
  vi nxt(n + 1, -1), prv(n + 1, -1);
  FOR(i, 0, SZ(h)) {
    prv[h[i][0]] = n;
    FOR(j, 1, SZ(h[i])) {
      int b = h[i][j-1], c = h[i][j]; nxt[b] = c, prv[c] = b;
    }
    nxt[h[i].back()] = n;
  }
  reverse(all(swaps));
  for(auto &[x, y, z]: swaps) {
    if(nxt[x] != z) continue;
    if(nxt[y] == -1) {
      prv[y] = prv[x]; nxt[prv[y]] = y; nxt[y] = z;
      prv[z] = y; prv[x] = nxt[x] = -1;
    }
    else {
      nxt[x] = nxt[y]; prv[nxt[x]] = x;
      nxt[y] = z; prv[z] = y;
    }
  }
  int cnt = 0; vi seq(n, -1); vector<vi> res(k);
  FOR(i, 0, n) if(prv[i] != -1) {
    seq[i] = prv[i] == n ? cnt++ : seq[prv[i]];
    res[seq[i]].pb(i);
  }
  return res;
}
```

MatroidIntersection.h

Description: Given two matroids, finds the largest common independent set. For the color and graph matroids, this would be the largest forest where no two edges are the same color. Pass the matroid with more expensive operations to M1.

Time: $\sim\mathcal{O}\left(N^3\right)$ oracle calls, in practice $\sim\mathcal{O}\left(N^2\right)$

<div align="right">858b54, 43 lines</div>

```cpp
template <class M1, class M2> struct MatroidIsect {
  M1 m1; M2 m2; int n; vi iset;
  MatroidIsect(M1 _m1, M2 _m2, int _n) :
    m1(_m1), m2(_m2), n(_n), iset(n + 1) {}
  vi solve() {
    per(i, 0, n-1) if(m1.check(i) && m2.check(i)) {
      iset[i] = 1, m1.add(i), m2.add(i);
    }
    while(augment());
    vi ans;
    FOR(i, 0, n) if(iset[i]) ans.pb(i);
    return ans;
  }
  bool augment() {
    vi frm(n, -1); queue<int> q({n}); // starts at dummy node
    auto fwdE = [&](int a) {
      vi ans; m1.clear();
      FOR(v, 0, n) if(iset[v] && v != a) m1.add(v);
      FOR(b, 0, n) if(!iset[b] && frm[b] == -1 && m1.check(b))
        ans.pb(b), frm[b] = a;
      return ans;
    };
    auto backE = [&](int b) {
      m2.clear();
      FOR(cas, 0, 2) FOR(v, 0, n)
        if((v == b || iset[v]) && (frm[v] == -1) == cas) {
          if(!m2.check(v))
            return cas ? q.push(v), frm[v] = b, v : -1;
          m2.add(v);
        }
      return n;
    };
    while(!q.empty()) {
      int a = q.front(), c; q.pop();
      for(int b: fwdE(a))
        while((c = backE(b)) >= 0) if(c == n) {
          while(b != n) iset[b] ^= 1, b = frm[b];
          return 1;
        }
    }
    return 0;
  }
};
```

MatroidOracles.h

Description: An oracle has 3 functions: check(int x): returns if current matroid can add x without becoming dependent; add(int x): adds an element to the matroid (guaranteed to never make it dependent); clear(): sets the matroid to the empty matroid.

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

<div align="right">7eef8c, 42 lines</div>

```cpp
struct ColorMat {
  vi clr, bound, cnt; int max_bound, offset = 0;
  ColorMat(vi _clr, vi _bound) : clr(_clr), bound(_bound) {
    cnt.resize(SZ(bound));
    max_bound = *max_element(all(bound));
  }
  void fix(int x) { cnt[clr[x]] = max(cnt[clr[x]], offset); }
  bool check(int x) {
    fix(x); return cnt[clr[x]] - offset < bound[clr[x]];
  }
  void add(int x) { fix(x); cnt[clr[x]]++; }
  void clear() { offset += max_bound; }
};

struct GraphicMat { // use normal UF to reduce memory
  vector<pii> edg; UF uf; vi vis;
  GraphicMat(vector<pii> _edg) : edg(_edg), uf(0) {
    int max_n = 0;
    for(auto &[x, y]: edg) max_n = max({max_n, x, y});
    uf = UF(max_n + 1);
  }
  bool check(int x) {
    return uf.find(edg[x].st) != uf.find(edg[x].nd);
  }
  void add(int x) { uf.join(edg[x].st,edg[x].nd); vis.pb(x); }
  void clear() {
    for(int x: vis) uf.e[edg[x].st] = uf.e[edg[x].nd] = -1;
    vis.clear();
  }
};

struct LinearMat {
  vector<ll> vals; set<ll, greater<ll>> basis, start;
  LinearMat(vector<ll> _vals, vector<ll> _s) : vals(_vals) {
    for(ll x: _s) basis.insert(red(x));
    start = basis;
  }
```

```
ll red(ll x) { for(ll y: basis) x = min(x, x^y); return x; }
bool check(int x) { return red(vals[x]); }
void add(int x) { basis.insert(red(vals[x])); }
void clear() { basis = start; }
};
```

NimMultiplication.h
Description: Nimber multiplication. To optimize use $(a \oplus b) * c = a * c \oplus b * c$.
Time: $\mathcal{O}\left((\log N)^{1.58}\right)$
<div align="right">ddb872, 10 lines</div>

```
using u64 = unsigned long long;
template<int L> inline u64 mulSlow(u64 a, u64 b) {
  static constexpr int l = L >> 1;
  const u64 a0 = a & ((1ull << l) - 1), a1 = a >> l;
  const u64 b0 = b & ((1ull << l) - 1), b1 = b >> l;
  const u64 a0b0 = mulSlow<l>(a0, b0);
  return (a0b0 ^ mulSlow<l>(1ull << (l-1), mulSlow<l>(a1,b1)))
    | (a0b0 ^ mulSlow<l>(a0 ^ a1, b0 ^ b1)) << l;
}
template<> inline u64 mulSlow<1>(u64 a, u64 b) {return a & b;}
```

Graph (6)

6.1   Network flow

MinCostMaxFlow.h
Description: Min-cost max-flow. If costs can be negative, call setpi before maxflow, but note that negative cost cycles are not supported. To obtain the actual flow, look at positive values only.
Time: $\mathcal{O}(FE \log(V))$ where F is max flow. $\mathcal{O}(VE)$ for setpi.
<div align="right">580207, 68 lines</div>

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

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

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

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

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

  void path(int s) {
    fill(all(seen), 0); fill(all(dist), INF);
    dist[s] = 0; ll di;
    __gnu_pbds::priority_queue<pair<ll, int>> q;
    vector<decltype(q)::point_iterator> its(N);
    q.push({0, s});
    while(!q.empty()) {
      s = q.top().nd; q.pop();
      seen[s] = 1; di = dist[s] + pi[s];
      for(edge &e: ed[s]) if(!seen[e.to]) {
        ll val = di - pi[e.to] + e.cost;
        if(e.cap - e.flow > 0 && val < dist[e.to]) {
          dist[e.to] = val; par[e.to] = &e;
          if(its[e.to] == q.end()) {
            its[e.to] = q.push({-dist[e.to], e.to}); }
          else q.modify(its[e.to], {-dist[e.to], e.to});
```

```
      }
    }
  }
  FOR(i, 0, N) pi[i] = min(pi[i] + dist[i], INF);
}

pair<ll, ll> maxflow(int s, int t) {
  ll totflow = 0, totcost = 0;
  while(path(s), seen[t]) {
    ll fl = INF;
    for(edge* x = par[t]; x; x = par[x->from]) {
      fl = min(fl, x->cap - x->flow); }
    totflow += fl;
    for(edge* x = par[t]; x; x = par[x->from]) {
      x->flow += fl; ed[x->to][x->rev].flow -= fl; }
  }
  FOR(i,0,N) for(edge &e: ed[i]) totcost += e.cost * e.flow;
  return {totflow, totcost / 2};
}

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

Dinic.h
Description: Flow algorithm with complexity $O(VE \log U)$ where $U = \max |cap|$. $O(\min(E^{1/2}, V^{2/3})E)$ if $U = 1$; $O(\sqrt{V}E)$ for bipartite matching.
<div align="right">995644, 42 lines</div>

```
struct Dinic {
  struct Edge {
    int to, rev; ll c, oc;
    ll flow() { return max(oc - c, 0ll); } // if flows needed
  };
  vi lvl, ptr, q;
  vector<vector<Edge>> adj;
  Dinic(int n) : lvl(n), ptr(n), q(n), adj(n) {}
  void addEdge(int a, int b, ll c, ll rcap = 0) {
    adj[a].pb({b, SZ(adj[b]), c, c});
    adj[b].pb({a, SZ(adj[a]) - 1, rcap, rcap});
  }
  ll dfs(int v, int t, ll f) {
    if(v == t || !f) return f;
    for(int &i = ptr[v]; i < SZ(adj[v]); i++) {
      Edge &e = adj[v][i];
      if(lvl[e.to] == lvl[v] + 1) {
        if(ll p = dfs(e.to, t, min(f, e.c))) {
          e.c -= p, adj[e.to][e.rev].c += p;
          return p;
        }
      }
    }
    return 0;
  }
  ll calc(int s, int t) {
    ll flow = 0; q[0] = s;
    FOR(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; }
};
```

GlobalMinCut.h
Description: Find a global minimum cut in an undirected graph, as represented by an adjacency matrix.
Time: $\mathcal{O}(V^3)$
<div align="right">9ebe69, 19 lines</div>

```
pair<int, vi> globalMinCut(vector<vi> mat) {
  pair<int, vi> best = {INT_MAX, {}};
  int n = SZ(mat); vector<vi> co(n);
  FOR(i, 0, n) co[i] = {i};
  FOR(ph, 1, n) {
    vi w = mat[0]; int s = 0, t = 0;
    FOR(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();
      FOR(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]));
    FOR(i, 0, n) mat[s][i] += mat[t][i];
    FOR(i, 0, n) mat[i][s] = mat[s][i];
    mat[0][t] = INT_MIN;
  }
  return best;
}
```

GomoryHu.h
Description: Given a list of edges representing an undirected flow graph, returns edges of the Gomory-Hu tree. The max flow between any pair of vertices is given by minimum edge weight along the Gomory-Hu tree path.
Time: $\mathcal{O}(V)$ Flow Computations
<div align="right">"Dinic.h"   ce350b, 12 lines</div>

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

6.2   Matching

hopcroftKarp.h
Description: Fast bipartite matching algorithm. Graph g should be a list of neighbors of the left partition, and *btoa* should be a vector full of -1's of the same size as the right partition. Returns the size of the matching. *btoa[i]* will be the match for vertex i on the right side, or -1 if it's not matched.
Usage: vi btoa(m, -1); hopcroftKarp(g, btoa);
Time: $\mathcal{O}\left(\sqrt{V}E\right)$
<div align="right">05aa1d, 33 lines</div>

```
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(SZ(g)), B(SZ(btoa)), cur, next;
  for(;;) {
    fill(all(A), 0); fill(all(B), 0);
    cur.clear();
    for(int a: btoa) if(a != -1) A[a] = -1;
    FOR(a, 0, SZ(g)) if(A[a] == 0) cur.pb(a);
    for(int lay = 1;; lay++) {
      bool islast = 0; next.clear();
      for(int a: cur) for(int b: g[a]) {
        if(btoa[b] == -1) { B[b] = lay; islast = 1; }
        else if (btoa[b] != a && !B[b]) {
          B[b] = lay; next.pb(btoa[b]); }
      }
      if(islast) break;
      if(next.empty()) return res;
      for(int a: next) A[a] = lay;
      cur.swap(next);
    }
    FOR(a, 0, SZ(g)) res += dfs(a, 0, g, btoa, A, B);
  }
}
```

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

```
"hopcroftKarp.h"                                    e38bb1, 17 lines
vi cover(vector<vi>& g, int n, int m) {
  vi match(m, -1);
  int res = hopcroftKarp(g, match);
  vector<bool> lfound(n, 1), seen(m);
  for(int it: match) if(it != -1) lfound[it] = 0;
  vi q, cover;
  FOR(i, 0, n) if(lfound[i]) q.pb(i);
  while(!q.empty()) {
    int i = q.back(); q.pop_back(); lfound[i] = 1;
    for(int e: g[i]) if(!seen[e] && match[e] != -1) {
      seen[e] = true; q.pb(match[e]); }
  }
  FOR(i, 0, n) if(!lfound[i]) cover.pb(i);
  FOR(i, 0, m) if(seen[i]) cover.pb(n + i);
  assert(SZ(cover) == res);
  return cover;
}
```

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

```
                                                    ff486d, 27 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);
  FOR(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] = 1; int i0 = p[j0], j1, delta = INT_MAX;
      FOR(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;
      }
      FOR(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; }
  }
  FOR(j, 1, m) if(p[j]) ans[p[j] - 1] = j - 1;
  return {-v[0], ans}; // min cost
}
```

GeneralMatching.h
Description: Unweighted matching for general graphs. If white[v] = 0 at the end, v is part of every max matching.
Time: $\mathcal{O}(NM)$, faster in practice

```
                                                    28552a, 48 lines
struct MaxMatching {          // 1-indexed
  vector<vi> G; int n;
  vi mate, par, white; vector<pii> l;
  MaxMatching(vector<vi> _G) : G(_G), n(SZ(G)), mate(n),
    par(n), white(n), l(n) {}
  int group(int x) {
    return par[x] = (white[par[x]] ? group(par[x]) : par[x]);
  }
  void match(int p, int b) {
    swap(mate[p], b); if(mate[b] != p) return;
    if(!l[p].nd) mate[b] = l[p].st, match(l[p].st, b);
    else match(l[p].st, l[p].nd), match(l[p].nd, l[p].st);
  }
  bool augment(int a) {
    white[a] = 1; par[a] = 0; l[a] = {0, 0};
    queue<int> q; q.push(a);
    while(!q.empty()) {
      a = q.front(); q.pop();
      for(int b: G[a]) {
        if(white[b]) {
          int x = group(a), y = group(b), lca = 0;
          while(x || y) {
            if(y) swap(x, y);
            if(l[x] == mp(a, b)) { lca = x; break; }
            l[x] = {a, b}; x = group(l[mate[x]].st);
          }
          for(int v: {group(a), group(b)}) while(v != lca) {
            q.push(v); white[v] = 1; par[v] = lca;
            v = group(l[mate[v]].st);
          }
        } else if(!mate[b]) {
          mate[b] = a; match(a, b); fill(all(white), 0);
          return 1;
        } else if(!white[mate[b]]) {
          white[mate[b]] = 1; par[mate[b]] = b;
          l[b] = {0, 0}; l[mate[b]] = {a, 0};
          q.push(mate[b]);
        }
      }
    }
    return 0;
  }
  int max_matching() {
```

```
    int ans = 0;
    FOR(v, 1, n) if(!mate[v]) ans += augment(v);
    return ans;
  }
};
```

6.3   DFS algorithms

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

```
                                                    43b75d, 30 lines
vi num, st;
vector<vector<pii>> ed;
int Time;
template<class F>
int dfs(int at, int par, F& f) {
  int me = num[at] = ++Time, top = me;
  for(auto [y, e]: ed[at]) if(e != par) {
    if(num[y]) {
      top = min(top, num[y]);
      if(num[y] < me) st.pb(e); }
    else {
      int si = SZ(st), up = dfs(y, e, f);
      top = min(top, up);
      if(up == me) {
        st.pb(e);
        f(vi(st.begin() + si, st.end()));
        st.resize(si);
      }
      else if(up < me) st.pb(e);
      else { /* e is a bridge */ }
    }
  }
  return top;
}

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

2sat.h
Description: Calculates a valid assignment to boolean variables a, b, c,... to a 2-SAT problem, so that an expression of the type $(a||b)\&\&(!a||c)\&\&(d||!b)\&\&...$ 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.

```
                                                    89d617, 43 lines
struct TwoSat {
  int n; vector<vi> gr; vi values; // 0 = false, 1 = true
  TwoSat(int _n = 0) : n(_n), gr(n * 2) {}
```

```
int addVar() { FOR(i, 0, 2) gr.pb({}); return n++; }

void either(int f, int j) {
  f = max(2*f, -1-2*f); j = max(2*j, -1-2*j);
  gr[f].pb(j^1); gr[j].pb(f^1);
}

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

void atMostOne(const vi& li) { // (optional)
  if(SZ(li) <= 1) return;
  int cur = ~li[0];
  FOR(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.pb(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(n * 2, 0); comp = val;
  FOR(i, 0, n * 2) if(!comp[i]) dfs(i);
  FOR(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: $\mathcal{O}(V + E)$
ecaa22, 12 lines

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

6.4 Trees

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: $\mathcal{O}\left((\log N)^2\right)$
ab0cb2, 32 lines

```
struct HLD {
  int N, tim = 0, VALS_EDGES = 0; // change to 1 if needed
  vector<vi> adj; vi par, sz, depth, rt, pos;
  HLD(vector<vi> _adj) : N(SZ(_adj)), adj(_adj), par(N, -1),
    sz(N, 1), depth(N), rt(N), pos(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); sz[v] += sz[u];
      if(sz[u] > sz[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); }
  }
  vector<pii> path(int u, int v) {
    vector<pii> paths;
    for(; rt[u] != rt[v]; v = par[rt[v]]) {
      if(depth[rt[u]] > depth[rt[v]]) swap(u, v);
      paths.pb({pos[rt[v]], pos[v]});
    }
    if(depth[u] > depth[v]) swap(u, v);
    paths.pb({pos[u] + VALS_EDGES, pos[v]});
    return paths;
  }
  pii subtree(int v) {
    return {pos[v] + VALS_EDGES, pos[v] + sz[v] - 1};
  }
};
```

CentroidTree.h

Description: Builds a centroid tree.
Time: $\mathcal{O}(N \log N)$
568b6f, 33 lines

```
struct CentroidTree {    // 0-indexed
  vector<vector<pii>> G;  // {neighbour, distance}
  vector<ll> dist[LOG];
  vi sz, block, par, dpth;
  CentroidTree(vector<vector<pii>> &_G) : G(_G) {
    int n = SZ(G); sz = block = par = dpth = vi(n);
    FOR(i, 0, LOG) dist[i].resize(n);
    decomp(0, -1);
  }
  void sz_dfs(int v, int p) {
    sz[v] = 1;
    for(auto &[u, _]: G[v])
      if(u != p && !block[u]) sz_dfs(u, v), sz[v] += sz[u];
  }
  int find_centr(int v, int p, int sum) {
    for(auto &[u, _]: G[v])
      if(u != p && !block[u] && sz[u] > sum / 2) {
        return find_centr(u, v, sum); }
    return v;
  }
  void dist_dfs(int v, int p, int gleb, ll akt) {
    dist[gleb][v] = akt;
    for(auto &[u, d]: G[v])
      if(u != p && !block[u]) dist_dfs(u, v, gleb, akt + d);
```

```
  }
  void decomp(int v, int p) {
    sz_dfs(v, -1); v = find_centr(v, -1, sz[v]);
    par[v] = p;
    if(p != -1) dpth[v] = dpth[p] + 1;
    dist_dfs(v, -1, dpth[v], 0); block[v] = 1;
    for(auto &[u, _]: G[v]) if(!block[u]) decomp(u, v);
  }
};
```

LinkCutTree.h

Description: One-indexed. Represents a forest of unrooted trees. You can add and remove edges (as long as the result is still a forest), and check whether two nodes are in the same tree. Also you can update a value in a vertex, and do standard path and subtree queries. Operation on subtrees must be invertible.
Time: All operations take amortized $\mathcal{O}(\log N)$.
42b461, 86 lines

```
struct SplayTree {
  struct Node {
    int p = 0, ch[2] = {0, 0};
    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) {}

  void push(int v) {
    if(!v || !t[v].flip) return;
    auto &[l, r] = t[v].ch;
    t[l].flip ^= 1, t[r].flip ^= 1;
    swap(l, r), t[v].flip = 0;
  }

  void pull(int v) {
    auto [l, r] = t[v].ch; push(l), push(r);
    t[v].path = t[l].path + t[v].self + t[r].path;
    t[v].sub = t[v].vir + t[l].sub + t[v].self + t[r].sub;
  }

  void set(int u, int d, int v) {
    t[u].ch[d] = v, t[v].p = u, pull(u);
  }

  void splay(int v) {
    auto dir = [&](int x) {
      int u = t[x].p;
      return t[u].ch[0] == x ? 0 : t[u].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(v); ~dir(v); ) {
      int y = t[v].p, z = t[y].p;
      push(z), push(y), push(v);
      int dv = dir(v), dy = dir(y);
      if(~dy) rotate(dv == dy ? y : v);
      rotate(v);
    }
  }
};

struct LinkCut : SplayTree {  // 1-indexed
  LinkCut(int n) : SplayTree(n) {}
  int access(int v) {
```

```cpp
    int u = v, x = 0;
    for(; u; x = u, u = t[u].p) {
      splay(u);
      int &ox = t[u].ch[1];
      t[u].vir += t[ox].sub;
      t[u].vir -= t[x].sub;
      ox = x, pull(u);
    }
    return splay(v), x;
  }
  void reroot(int v) { access(v), t[v].flip ^= 1, push(v); }
  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 are not 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 getSub(int u, int v) {
    reroot(v), access(u); return t[u].vir + t[u].self;
  }
  ll getPath(int u, int v) {
    reroot(u), access(v); return t[v].path;
  }
  // Update vertex u with value val
  void update(int u, ll val) {
    access(u), t[u].self = val, pull(u);
  }
};
```

DirectedMST.h
Description: Finds a minimum spanning tree/arborescence of a directed graph, given a root node. If no MST exists, returns -1.
Time: $\mathcal{O}(E \log V)$

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

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

```cpp
  }
  FOR(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]}});
      }
    }
    FOR(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;
  }
  FOR(i, 0, n) par[i] = in[i].a;
  return {res, par};
}
```

DominatorTree.h
Description: Computes Dominator tree.
Time: $\mathcal{O}(V + E)$

```cpp
struct DominatorTree { // 0-indexed
  int n, t;
  vector<basic_string<int>> g, rg, bucket;
  basic_string<int> arr, par, rev, sdom, dom, dsu, label;
  DominatorTree(int _n) : n(_n), t(0), g(n), rg(n),
    bucket(n), arr(n, -1), par(n, -1), rev(n, -1),
    sdom(n, -1), dom(n, -1), dsu(n, 0), label(n, 0) {}
  void add_edge(int u, int v) { g[u] += v; }
  void dfs(int u) {
    arr[u] = t; rev[t] = u;
    label[t] = sdom[t] = dsu[t] = t; t++;
    for(int w: g[u]) {
      if(arr[w] == -1) dfs(w), par[arr[w]] = arr[u];
      rg[arr[w]] += arr[u];
    }
  }
  int find(int u, int x = 0) {
    if(u == dsu[u]) return x ? -1 : u;
    int v = find(dsu[u], x + 1);
    if(v < 0) return u;
    if(sdom[label[dsu[u]]] < sdom[label[u]]) {
      label[u] = label[dsu[u]]; }
    dsu[u] = v;
    return x ? v : label[u];
  }  // returns -1 for unreachable
  vi run(int root) {
    dfs(root); iota(all(dom), 0);
    per(i, 0, t-1) {
      for(int w: rg[i]) sdom[i] = min(sdom[i], sdom[find(w)]);
      if(i) bucket[sdom[i]] += i;
      for(int w: bucket[i]) {
        int v = find(w);
        if(sdom[v] == sdom[w]) dom[w] = sdom[w];
        else dom[w] = v;
      }
      if(i > 1) dsu[i] = par[i];
```

```cpp
    }
    FOR(i, 1, t) if(dom[i] != sdom[i]) dom[i] = dom[dom[i]];
    vi outside_dom(n, -1);
    FOR(i, 1, t) outside_dom[rev[i]] = rev[dom[i]];
    return outside_dom;
  }
};
```

6.5   Various

kShortestWalks.h
Description: Given a non-negative weighted directed graph, computes lengths of K shortest walks (vertices can repeat). For graphs with negative weights, try your luck or change Dijkstra to SPFA.
Time: $\mathcal{O}((M + K) \log N)$

```cpp
struct PersistentHeap {
  struct Node {
    int id, l = -1, r = -1; ll cost;
    Node(int _id, ll _cost) : id(_id), cost(_cost) {}
  };
  vector<Node> ds;
  int add(int id, ll cost) {
    ds.pb(Node(id, cost)); return SZ(ds) - 1;
  }
  int ins(int v, int u) {
    if(v == -1) return u;
    ds.pb(ds[v]); v = SZ(ds) - 1; swap(ds[v].l, ds[v].r);
    if(ds[v].cost > ds[u].cost) {
      swap(ds[v].cost, ds[u].cost), swap(ds[v].id, ds[u].id);
    }
    ds[v].r = ins(ds[v].r, u); return v;
  }
  void insert(int &v, int u) { v = ins(v, u); }
};

vector<ll> kWalk(vector<vector<pii>> G, int s, int t, int k) {
  int n = SZ(G); vector<vector<pii>> GR(n);
  FOR(v, 0, n) for(auto &[u, d]: G[v]) GR[u].pb({v, d});
  const ll INF = 1e18; vector<ll> dist(n, INF); vi par(n, -1);
  using T = pair<ll, int>; dist[t] = 0;
  priority_queue<T, vector<T>, greater<T>> q; q.push({0, t});
  while(!q.empty()) {
    auto [dv, v] = q.top(); q.pop();
    if(dv != dist[v]) continue;
    for(auto &[u, d]: GR[v]) if(dv + d < dist[u]) {
      par[u] = v; dist[u] = dv + d; q.push({dist[u], u});
    }
  }
  vector<vi> tree(n);
  FOR(v, 0, n) if(~par[v]) tree[par[v]].pb(v);
  PersistentHeap heap; vi head(n, -1);
  function<void(int)> dfs = [&](int v) {
    bool skip = 0;
    for(auto &[u, d]: G[v]) if(dist[u] != INF) {
      if(dist[v] == dist[u] + d && par[v] == u && !skip)
        skip = 1;
      else
        heap.insert(head[v], heap.add(u, dist[u]-dist[v]+d));
    }
    for(int u: tree[v]) head[u] = head[v], dfs(u);
  }; dfs(t);
  vector<ll> ans(k, -1); q.push({dist[s], heap.add(s, 0)});
  FOR(i, 0, k) {
    if(q.empty() || dist[s] == INF) break;
    auto [dv, v] = q.top(); q.pop();
    ans[i] = dv; auto &node = heap.ds[v]; ll diff = 0;
    for(int u: {head[node.id], node.l, node.r}) {
      if(~u) q.push({dv + heap.ds[u].cost - diff, u});
      diff = node.cost;
```

```
    }
  }
  return ans;
}
```

SubgraphsCounting.h
Description: Given simple undirected graph, counts all 3- and 4-edges subgraphs.

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

4ca57a, 50 lines
```
struct Subgraphs {
  int triangles3 = 0;
  ll stars3 = 0, paths3 = 0;
  ll ps4 = 0, rectangles4 = 0, paths4 = 0;
  __int128 ys4 = 0, stars4 = 0;
  Subgraphs(vector<vi> &G) {
    int n = SZ(G); vector<pii> deg(n);
    FOR(i, 0, n) deg[i] = {SZ(G[i]), i};
    sort(all(deg)); vi id(n), cnt(n);
    FOR(i, 0, n) id[deg[i].nd] = i;
    FOR(v, 0, n) {
      for(int u: G[v]) if(id[v] > id[u]) cnt[u] = 1;
      for(int u: G[v]) if(id[v] > id[u])
        for(int w: G[u]) if(id[w] > id[u] && cnt[w]) {
          triangles3++;
          for(int x: {v, u, w}) ps4 += SZ(G[x]) - 2;
        }
      for(int u: G[v]) if(id[v] > id[u]) cnt[u] = 0;
      for(int u: G[v]) if(id[v] > id[u])
        for(int w: G[u]) if(id[v] > id[w]) {
          rectangles4 += cnt[w]++;
        }
      for(int u: G[v]) if(id[v] > id[u])
        for(int w: G[u]) cnt[w] = 0;
    }
    paths3 = -3 * triangles3;
    FOR(v, 0, n) for(int u: G[v])
      if(v < u) paths3 += ll(SZ(G[v]) - 1) * (SZ(G[u]) - 1);
    ys4 = -2 * ps4;
    auto choose2 = [&](int x) { return x * ll(x - 1) / 2; };
    FOR(v, 0, n) for(int u: G[v]) {
      ys4 += (SZ(G[v]) - 1) * choose2(SZ(G[u]) - 1);
    }
    paths4 = -(4 * rectangles4 + 2 * ps4 + 3 * triangles3);
    FOR(v, 0, n) {
      int x = 0;
      for(int u: G[v]) {
        x += SZ(G[u]) - 1;
        paths4 -= choose2(SZ(G[u]) - 1);
      }
      paths4 += choose2(x);
    }
    FOR(v, 0, n) {
      int s = SZ(G[v]);
      stars3 += s * ll(s - 1) * (s - 2);
      stars4 += s * (__int128)(s - 1) * (s - 2) * (s - 3);
    }
    stars3 /= 6; stars4 /= 24;
  }
};
```

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: $\mathcal{O}(NM)$

e1c352, 26 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++);
  }
  FOR(i, 0, SZ(eds))
    for(tie(u, v) = eds[i]; adj[u][ret[i]] != v;) ++ret[i];
  return ret;
}
```

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.

e1d2b4, 46 lines
```
using vb = vector<bitset<200>>;
struct Maxclique {
  double limit = 0.025, pk = 0;
  struct Vertex { int i, d=0; };
  using vv = vector<Vertex>;
  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;
    FOR(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.pb(R.back().i); vv T;
      for(auto v: R) if (e[R.back().i][v.i]) T.pb({v.i});
      if(SZ(T)) {
        if(S[lev]++ / ++pk < limit) init(T);
        int j = 0, mxk = 1, mnk = max(SZ(qmax) - SZ(q)+1, 1);
        C[1].clear(), C[2].clear();
        for(auto v: T) {
          int k = 1;
          auto f = [&](int i) { return e[v.i][i]; };
          while(any_of(all(C[k]), f)) k++;
          if(k > mxk) mxk = k, C[mxk + 1].clear();
          if(k < mnk) T[j++].i = v.i;
          C[k].pb(v.i);
        }
        if(j > 0) T[j - 1].d = 0;
        FOR(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) {
    FOR(i, 0, SZ(e)) V.pb({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

745132, 11 lines
```
using B = bitset<128>;
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];
  FOR(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;
  }
}
```

GreenHackenbush.h
Description: Given a rooted graph computes the nimber of Green Hackenbush game. Rule one: for a tree, nimber of a root is a xorsum of (son's nimber + 1). Rule two: when u and v lies on a cycle, then they can be contracted.

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

17609f, 19 lines
```
struct GreenHack {
  vector<vi> G; vi pre, low; int T = 0;
  GreenHack(vector<vi> _G) : G(_G), pre(SZ(G)), low(SZ(G)) {}
  int dfs(int v, int p) {
    pre[v] = low[v] = ++T; int ans = 0;
    for(int u: G[v]) {
      if(u == p) { p += SZ(G) + 1; continue; }
      if(!pre[u]) {
        int son = dfs(u, v); low[v] = min(low[v], low[u]);
        if(pre[v] < low[u]) ans ^= (son + 1);
        else ans ^= son ^ 1;
      }
      else if(pre[v] <= pre[u]) ans ^= 1;
      else low[v] = min(low[v], pre[u]);
    }
    return ans;
  }
  int run() { return dfs(0, -1); }
};
```

RedBlueHackenbush.h
Description: Given a rooted tree computes the result of Red-Blue Hackenbush game. If result != 0 then {result} always wins, otherwise the second player to go wins.

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

72d802, 48 lines
```
struct Surreal {
  int value = 0, offset = 0; set<int> powers;
  int sign() {
    int tmp = 2 * value + !powers.empty();
    return tmp < 0 ? -1 : (tmp > 0);
  }
  int add_power(int power) {
```

```
  while(power) {
    if(!powers.count(power - offset)) {
      powers.insert(power - offset); break; }
    powers.erase(power - offset); power--;
  }
  return !power;
}
void operator+=(const Surreal &v) {
  value += v.value;
  for(int power: v.powers)
    value += add_power(power + v.offset);
}
void divide(int power) {
  offset += power; int to_add = 0;
  FOR(i, 0, power) {
    if(value & 1) to_add += add_power(power - i);
    value >>= 1;
  }
  value += to_add;
}
void get_next(int t) {
  int power = max(0, -t * value); value += t * (power + 1);
  if(value == -1 || (value == 1 && powers.empty())) {
    power++, value += t; }
  divide(power);
}
};

struct RedBlueHack { /* Weights on edges should be -1 or 1 */
  vector<vector<pii>> G; vector<Surreal> ans;
  RedBlueHack(vector<vector<pii>> _G) : G(_G), ans(SZ(G)) {}
  void dfs(int u, int p) {
    for(auto &[v, w]: G[u]) if(v != p) {
      dfs(v, u); ans[v].get_next(w);
      if(SZ(ans[u].powers) < SZ(ans[v].powers)) {
        swap(ans[u], ans[v]); }
      ans[u] += ans[v];
    }
  }
  int run() { dfs(0, 0); return ans[0].sign(); }
};
```

6.6 Math

6.6.1 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).

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

6.6.3 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 $t_i = 1 + \sum_{k \in \mathbf{G}} p_{ki} t_k$.

Geometry (7)

7.1 Geometric primitives

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

b1a082, 29 lines

```
using D = double;
template<class T> int sgn(T x) { return (x > 0) - (x < 0); }
template<class T>
struct Point {
  using P = Point;
  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; }
  D dist() const { return sqrt((D)dist2()); }
  // angle to x-axis in interval [-pi, pi]
  D 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(D 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 << ")"; }
};
```
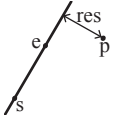
lineDistance.h
Description:
Returns the signed distance between point p and the line containing points a and b. Positive value on left side and negative on right as seen from a towards b. a==b gives nan. P is supposed to be Point<T> or Point3D<T> where T is e.g. double or long long. It uses products 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.

0052e6, 4 lines

```
template<class P>
D lineDist(const P &a, const P &b, const P &p) {
  return (D)(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.
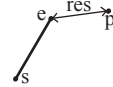**Usage:** Point<D> a, b(2,2), p(1,1);
bool onSegment = segDist(a,b,p) < 1e-10;

677ad0, 6 lines

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

SegmentIntersection.h
Description:
If a unique intersection point between the line segments going from s1 to e1 and from s2 to e2 exists then it is returned. If no intersection point exists an empty vector is returned. If infinitely many exist a vector with 2 elements is returned, containing the endpoints of the common line segment. The wrong position will be returned if P is Point<ll> and the intersection point does not have integer coordinates. Products of three coordinates are used in intermediate steps so watch out for overflow if using int or long long.
**Usage:** vector<P> inter = segInter(s1,e1,s2,e2);
if (sz(inter)==1)
cout << "segments intersect at " << inter[0] << endl;
"Point.h", "OnSegment.h"

9d57f2, 13 lines

```
template<class P> vector<P> segInter(P a, P b, P c, P d) {
  auto oa = c.cross(d, a), ob = c.cross(d, b),
       oc = a.cross(b, c), od = a.cross(b, d);
  // Checks if intersection is single non-endpoint point.
  if (sgn(oa) * sgn(ob) < 0 && sgn(oc) * sgn(od) < 0)
    return {(a * ob - b * oa) / (ob - oa)};
  set<P> s;
  if (onSegment(c, d, a)) s.insert(a);
  if (onSegment(c, d, b)) s.insert(b);
  if (onSegment(a, b, c)) s.insert(c);
  if (onSegment(a, b, d)) s.insert(d);
  return {all(s)};
}
```
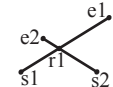
lineIntersection.h
Description:
If a unique intersection point of the lines going through s1,e1 and s2,e2 exists {1, point} is returned. If no intersection point exists {0, (0,0)} is returned and if infinitely many exists {-1, (0,0)} is returned. The wrong position will be returned if P is Point<ll> and the intersection point does not have integer coordinates. Products of three coordinates are used in intermediate steps so watch out for overflow if using int or ll.
Usage: `auto res = lineInter(s1,e1,s2,e2);`
`if (res.first == 1)`
`cout << "intersection point at " << res.second << endl;`
`"Point.h"` a01f81, 8 lines

```
template<class P>
pair<int, P> lineInter(P s1, P e1, P s2, P e2) {
  auto d = (e1 - s1).cross(e2 - s2);
  if(d == 0) // if parallel
    return {-(s1.cross(e1, s2) == 0), P(0, 0)};
  auto p = s2.cross(e1, e2), q = s2.cross(e2, s1);
  return {1, (s1 * p + e1 * q) / d};
}
```
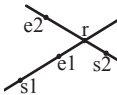
sideOf.h
Description: Returns where p is as seen from s towards e. 1/0/-1 ⇔ left/on line/right. If the optional argument eps is given 0 is returned if p is within distance eps from the line. P is supposed to be Point<T> where T is e.g. double or long long. It uses products in intermediate steps so watch out for overflow if using int or long long.
Usage: `bool left = sideOf(p1,p2,q)==1;`
`"Point.h"` c2efb9, 8 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, D eps) {
  auto a = (e-s).cross(p-s); D 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<D>.
`"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;
}
```
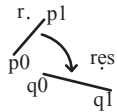
linearTransformation.h
Description:

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

`"Point.h"` 1ef732, 6 lines

```
using P = Point<D>;
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 instead. The wrong point will be returned if P is an integer point and the desired point doesn't have integer coordinates. Products of three coordinates are used in intermediate steps so watch out for overflow.
`"Point.h"` b5562d, 5 lines

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

7.2 Circles

CircleIntersection.h
Description: Computes the pair of points at which two circles intersect. Returns false in case of no intersection.
`"Point.h"` 9dd9e4, 11 lines

```
using P = Point<D>;
bool circleInter(P a, P b, D r1, D r2, pair<P, P>* out) {
  if(a == b) { assert(r1 != r2); return 0; }
  P vec = b - a;
  D 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 0;
  P mid = a + vec*p, per = vec.perp() * sqrt(fmax(0, h2) / d2);
  *out = {mid - per, mid + per};
  return 1;
}
```

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"` c6361c, 13 lines

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

CircleLine.h
Description: Finds the intersection between a circle and a line. Returns a vector of either 0, 1, or 2 intersection points. P is intended to be Point<D>.
`"Point.h"` f4015d, 9 lines

```
template<class P>
vector<P> circleLine(P c, D r, P a, P b) {
  P ab = b - a, p = a + ab * (c-a).dot(ab) / ab.dist2();
  D 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: $\mathcal{O}(n)$
`"Point.h"` de735b, 19 lines

```
using P = Point<D>;
```

```
#define arg(p, q) atan2(p.cross(q), p.dot(q))
D circlePoly(P c, D 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 = fmax(0, -a-sqrt(det)), t = fmin(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;
  };
  D sum = 0;
  FOR(i, 0, SZ(ps))
    sum += tri(ps[i] - c, ps[(i + 1) % SZ(ps)] - c);
  return sum;
}
```

CirclesUnionArea.h
Description: Returns the area of the sum of circles.
Time: $\mathcal{O}(n^2 \log n)$
`"CircleIntersection.h"` 243537, 33 lines

```
D circlesArea(vector<pair<P, D>> c) {
  const D PI = acos((D)-1);
  sort(all(c)); c.erase(unique(all(c)), c.end());
  D res = 0;
  for(auto &[p, r]: c) {
    int cnt = 0, cover = 0;
    vector<pair<D, int>> eve = {{-PI, 0}};
    for(auto &[q, s]: c) if(mp(p, r) != mp(q, s)) {
      D dst = (p - q).dist();
      if(r + dst <= s) { cover = 1; break; }
      pair<P, P> inters;
      if(!circleInter(p, q, r, s, &inters)) continue;
      D le = (inters.st - p).angle();
      D re = (inters.nd - p).angle();
      cnt += le > re;
      eve.pb({le, 1}), eve.pb({re, -1});
    }
    if(cover) continue;
    sort(eve.begin() + 1, eve.end());
    eve.pb({PI, 0});
    D loc = 0;
    FOR(i, 1, SZ(eve)) {
      if(!cnt) {
        D a = eve[i-1].st, b = eve[i].st;
        loc += r * (b - a) +
          p.cross(P(cos(b)-cos(a), sin(b)-sin(a)));
      }
      cnt += eve[i].nd;
    }
    res += r * loc;
  }
  return res / 2;
}
```
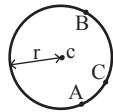
circumcircle.h
Description:

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"` 162089, 9 lines

```
using P = Point<D>;
D 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"   81b8ec, 16 lines

```
pair<P, D> mec(vector<P> ps) {
  shuffle(all(ps), mt19937(1337));
  P o = ps[0]; D r = 0, EPS = 1 + 1e-8;
  FOR(i, 0, SZ(ps)) if((o - ps[i]).dist() > r * EPS) {
    o = ps[i], r = 0;
    FOR(j, 0, i) if((o - ps[j]).dist() > r * EPS) {
      o = (ps[i] + ps[j]) / 2;
      r = (o - ps[i]).dist();
      FOR(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};
}
```

7.3   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"   da435c, 11 lines

```
template<class P>
bool inPolygon(vector<P> &p, P a, bool strict = true) {
  int cnt = 0, n = SZ(p);
  FOR(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"   362a7f, 6 lines

```
template<class T>
T polygonArea2(vector<Point<T>> &v) {
  T a = v.back().cross(v[0]);
  FOR(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"   619437, 9 lines

```
using P = Point<D>;
P polygonCenter(const vector<P>& v) {
```

```
  P res(0, 0); D 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"   ad7124, 12 lines

```
using P = Point<D>;
vector<P> polygonCut(const vector<P> &poly, P s, P e) {
  vector<P> res;
  FOR(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.pb(lineInter(s, e, cur, prev).nd);
    if(side) res.pb(cur);
  }
  return res;
}
```

PolygonUnion.h
Description: Calculates the area of the union of n polygons (not necessarily convex). The points within each polygon must be given in CCW order. (Epsilon checks may optionally be added to sideOf/sgn, but shouldn't be needed.)
Time: $\mathcal{O}(N^2)$, where N is the total number of points
"Point.h", "sideOf.h"   55d77a, 30 lines

```
using P = Point<D>;
D rat(P a, P b) { return sgn(b.x) ? a.x/b.x : a.y/b.y; }
D polyUnion(vector<vector<P>> &poly) {
  D ret = 0;
  FOR(i,0,SZ(poly)) FOR(v,0,SZ(poly[i])) {
    P A = poly[i][v], B = poly[i][(v + 1) % SZ(poly[i])];
    vector<pair<D, int>> segs = {{0, 0}, {1, 0}};
    FOR(j,0,SZ(poly)) if(i != j) FOR(u, 0, SZ(poly[j])) {
      P C = poly[j][u], E = poly[j][(u + 1) % SZ(poly[j])];
      int sc = sideOf(A, B, C), sd = sideOf(A, B, E);
      if(sc != sd) {
        D sa = C.cross(E, A), sb = C.cross(E, 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(E-C))>0) {
        segs.emplace_back(rat(C - A, B - A), 1);
        segs.emplace_back(rat(E - A, B - A), -1);
      }
    }
    sort(all(segs));
    for(auto &s: segs) s.st = min(max(s.st, 0.0), 1.0);
    D sum = 0; int cnt = segs[0].nd;
    FOR(j, 1, SZ(segs)) {
      if(!cnt) sum += segs[j].st - segs[j - 1].st;
      cnt += segs[j].nd;
    }
    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"   c06f0e, 12 lines

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

HullDiameter.h
Description: Returns the two points with max distance on a convex hull (ccw, no duplicate/collinear points).
Time: $\mathcal{O}(n)$
"Point.h"   2aeea8, 11 lines

```
using P = Point<ll>;
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]}});
  FOR(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.nd;
}
```

MinkowskiSum.h
Description: Computes the Minkowski sum of two convex polygons. Distance between two convex polygons can be computed by finding the closest edge of P - Q to the origin.
**Time:** $\mathcal{O}(N)$
"Point.h"   d461a3, 14 lines

```
template<class P>
vector<P> minkowski(vector<P> a, vector<P> b) {
  rotate(a.begin(), min_element(all(a)), a.end());
  rotate(b.begin(), min_element(all(b)), b.end());
  FOR(i, 0, 2) a.pb(a[i]), b.pb(b[i]);
  vector<P> res;
  for(int i = 0, j = 0; i < SZ(a) - 2 || j < SZ(b) - 2; ) {
    res.pb(a[i] + b[j]);
    auto cross = (a[i + 1] - a[i]).cross(b[j + 1] - b[j]);
    if(cross >= 0 && i < SZ(a)) i++;
    if(cross <= 0 && j < SZ(b)) j++;
  }
  return res;
}
```

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"   550f0d, 13 lines

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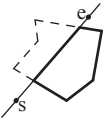
```
  if(SZ(l) < 3) return r && onSegment(l[0], l.back(), p);
  if(sideOf(l[0], l[a], l[b]) > 0) swap(a, b);
  if(sideOf(l[0], l[a], p) >= r || sideOf(l[0], l[b], p)<= -r)
    return 0;
  while(abs(a - b) > 1) {
    int c = (a + b) / 2;
    (sideOf(l[0], l[c], p) > 0 ? b : a) = c;
  }
  return sgn(l[a].cross(l[b],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

LineHullIntersection.h

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

"Point.h"   e112e1, 39 lines
```
#define cmp(i, j) sgn(dir.perp().cross(poly[(i)%n]-poly[(j)%n]))
#define extr(i) cmp(i + 1, i) >= 0 && cmp(i, i - 1 + n) < 0
template <class P> int extrVertex(vector<P> &poly, P dir) {
  int n = SZ(poly), lo = 0, hi = n;
  if(extr(0)) return 0;
  while(lo + 1 < hi) {
    int m = (lo + hi) / 2;
    if(extr(m)) return m;
    int ls = cmp(lo + 1, lo), ms = cmp(m + 1, m);
    (ls < ms || (ls == ms && ls == cmp(lo, m)) ? hi : lo) = m;
  }
  return lo;
}

#define cmpL(i) sgn(a.cross(poly[i], b))
template <class P>
array<int, 2> lineHull(P a, P b, vector<P> &poly) {
  int endA = extrVertex(poly, (a - b).perp());
  int endB = extrVertex(poly, (b - a).perp());
  if(cmpL(endA) < 0 || cmpL(endB) > 0) return {-1, -1};
  array<int, 2> res;
  FOR(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;
}
```

PolygonCutter.h

Description: Returns area of polygon halfplane intersection.

"LineHullIntersection.h", "lineIntersection.h"   8f8a19, 18 lines
```
using P = Point<D>;
struct PolyCutter {
  vector<D> pArea = {0}; vector<P> poly; int n;
```

```
  PolyCutter(vector<P> &_poly) : poly(_poly), n(SZ(poly)) {
    FOR(i, 0, n)
      pArea.pb(pArea.back() + poly[i].cross(poly[(i+1)%n]));
  }
  D cutArea(P a, P b) {
    auto [i, j] = lineHull(a, b, poly);
    if(j == -1 || i == j)
      return cmpL((i + 2) % n) < 0 ? pArea[n] / 2 : 0;
    D res = pArea[i] - pArea[j + 1] + (i < j ? pArea[n] : 0);
    P p = lineInter(poly[i], poly[(i + 1) % n], a, b).nd;
    P q = lineInter(poly[j], poly[(j + 1) % n], a, b).nd;
    res += poly[i].cross(p)+p.cross(q)+q.cross(poly[(j+1)%n]);
    return res / 2;
  }
};
```

PolygonTangents.h

Description: Polygon tangents from a given point. The polygon must be ccw and have no collinear points. Returns a pair of indices of the given polygon. Should work for a point on border (for a point being polygon vertex returns previous and next one).
Time: $\mathcal{O}(\log n)$

"Point.h"   4d1a00, 21 lines
```
#define pdir(i) (ph ? p - poly[(i)%n] : poly[(i)%n] - p)
#define cmp(i,j) sgn(pdir(i).cross(poly[(i)%n]-poly[(j)%n]))
#define extr(i) cmp(i + 1, i) >= 0 && cmp(i, i - 1 + n) < 0
template <class P>
array<int, 2> polygonTangents(vector<P>& poly, P p) {
  auto bs = [&](int ph) {
    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;
  };
  array<int, 2> res = {bs(0), bs(1)};
  if(res[0] == res[1]) res[0] = (res[0] + 1) % SZ(poly);
  if(poly[res[0]] == p) res[0] = (res[0] + 1) % SZ(poly);
  return res;
}
```

HalfplaneIntersectionDynamic.h

Description: Data structure that dynamically keeps track of the intersection of half-planes.
Time: amortized $\mathcal{O}(\log n)$

"lineIntersection.h"   6d30ae, 57 lines
```
using P = Point<D>;
int hf(P a) { return a.y < 0 || (a.y == 0 && a.x < 0); }
struct polarCmp {
  bool operator()(const P &a, const P &b) const {
    return hf(a) == hf(b) ? a.cross(b) > 0 : hf(a) < hf(b);
  }
};

struct HalfplaneSet : map<P, P, polarCmp> {
  D INF = 1e6, area = 8 * INF * INF;
  HalfplaneSet() {
    P p(-INF, -INF), d(1, 0);
    FOR(k, 0, 4) {
      insert({d, p}); p = p + d * 2 * INF; d = d.perp(); }
  }
  auto fix(auto it) { return it == end() ? begin() : it; }
  auto getNext(auto it) { return fix(next(it)); }
  auto getPrev(auto it) {
```

```
    return it == begin() ? prev(end()) : prev(it);
  }
  auto uSide(auto it, int change) { // 1 - add, -1 - del
    area += change * it->nd.cross(getNext(it)->nd);
  }
  auto del(auto it) {
    uSide(getPrev(it), -1), uSide(it, -1);
    it = fix(erase(it));
    if(size()) uSide(getPrev(it), 1);
    return it;
  }
  void add(P s, P e) {
    auto eval = [&](auto it) {
      return sgn(s.cross(e, it->nd));
    };
    auto intersect = [&](auto it) {
      return lineInter(s, e, it->nd, it->st + it->nd).nd;
    };
    auto it = fix(lower_bound(e - s));
    if(empty() || eval(it) >= 0) return;
    while(size() && eval(getPrev(it)) < 0) del(getPrev(it));
    while(size() && eval(getNext(it)) < 0) it = del(it);
    if(empty()) return;
    if(eval(getNext(it)) > 0) {
      uSide(getPrev(it), -1), uSide(it, -1);
      it->nd = intersect(it);
      uSide(getPrev(it), 1), uSide(it, 1);
    }
    else it = del(it);
    it = getPrev(it);
    uSide(it, -1); insert(it, {e - s, intersect(it)});
    uSide(it, 1), uSide(getNext(it), 1);
    if(eval(it) == 0) del(it);
  }
  D maxDot(P a) {
    return a.dot(fix(lower_bound(a.perp()))->nd);
  }
  D getArea() { return area / 2; }
};
```

7.4 Misc. Point Set Problems

ClosestPair.h

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

"Point.h"   5c381c, 16 lines
```
using P = Point<ll>;
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.nd;
}
```

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"   3dd86e, 22 lines

```cpp
using P = Point<int>;
vector<array<int, 3>> manhattanMST(vector<P> ps) {
  vi id(SZ(ps)); iota(all(id), 0);
  vector<array<int, 3>> edges;
  FOR(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.pb({d.y + d.x, i, j});
      }
      sweep[-ps[i].y] = i;
    }
    for(P &p : ps) if(k & 1) p.x = -p.x; else swap(p.x, p.y);
  }
  return edges;
}
```

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

"Point.h"   ee5448, 57 lines

```cpp
using T = ll;
using P = Point<T>;
const T INF = numeric_limits<T>::max();

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

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

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

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

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

  pair<T, P> search(Node *node, const P& p) {
```

```cpp
    if(!node->first) {
      // uncomment if we should not find the point itself:
      // if (p == node->pt) return {INF, P()};
      return mp((p - node->pt).dist2(), node->pt);
    }

    Node *f = node->first, *s = node->second;
    T bfirst = f->distance(p), bsec = s->distance(p);
    if(bfirst > bsec) swap(bsec, bfirst), swap(f, s);
    // search closest side first, other side if needed
    auto best = search(f, p);
    if(bsec < best.st) best = min(best, search(s, p));
    return best;
  }
  // find nearest point to a point, and its squared distance
  // (requires an arbitrary operator< for Point)
  pair<T, P> nearest(const P &p) { return search(root, p); }
};
```

DelaunayTriangulation.h
Description: Computes the Delaunay triangulation of a set of points. Each circumcircle contains none of the input points. If any three points are collinear or any four are on the same circle, behavior is undefined.
Time: $\mathcal{O}(n^2)$

"Point.h", "3DHull.h"   80635c, 10 lines

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

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"   38ce5d, 86 lines

```cpp
using P = Point<ll>;
using Q = struct Quad*;
using lll = __int128_t; // (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;
  FOR(i,0,4) r = r->rot, r->p = arb, r->o = i & 1 ? r : r->r();
  r->p = orig; r->F() = dest;
  return r;
}
```

```cpp
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).st;
  vector<Q> q = {e};
  int qi = 0;
  while (e->o->F().cross(e->F(), e->p) < 0) e = e->o;
#define ADD { Q c = e; do { c->mark = 1; pts.pb(c->p); \
  q.pb(c->r()); c = c->next(); } while (c != e); }
  ADD; pts.clear();
  while(qi < SZ(q)) if (!(e = q[qi++])->mark) ADD;
  return pts;
}
```

directedSegment.h
Description: Segment representation usefull for sweeping. Compares two disjoint (can touch) segments on the sweep line (OY projection). Transitivity breaks if three segments touch in one point and are on different side of the sweep line. Can be easily fixed by comparing mean X coordinates.

"Point.h"   d144f8, 15 lines

```cpp
template<class P>
```

```cpp
struct dirSeg {
  P s, e; int rev;
  dirSeg(P _s, P _e) : s(_s), e(_e), rev(0) {
    if(e < s) swap(s, e), rev = 1;
  }
  P getY(P X) { // takes x * 2, returns y * 2 as a fraction
    P d = (e - s);
    return !sgn(d.x) ? P(s.y+e.y, 1) : P(d.cross(s*2-X), d.x);
  }
  int cmp(dirSeg b) { // needs ~64 * M^3 !
    P X(max(s.x, b.s.x) + min(e.x, b.e.x), 0);
    return sgn(getY(X).cross(b.getY(X)));
  }
};
```

PointLocation.h

Description: Computes (not necessarily convex) polygon tree structure.
Also for each query point computes its location (including boundaries).
**Time:** $\mathcal{O}(n \log n)$

"directedSegment.h"   edc289, 47 lines

```cpp
template<class P>
pair<vi,vi> pointLoc(vector<vector<P>> polys, vector<P> pts) {
  vector<tuple<P, int, int>> eve; // {point, event_type, id}
  vector<pair<dirSeg<P>, int>> segs;  // {s, e, poly_id}
  FOR(i, 0, SZ(polys)) FOR(j, 0, SZ(polys[i])) {
    dirSeg<P> seg(polys[i][j], polys[i][(j+1)%SZ(polys[i])]);
    eve.pb({seg.s,0,SZ(segs)}), eve.pb({seg.e,2,SZ(segs)});
    segs.pb({seg, i});
  }
  FOR(i, 0, SZ(pts)) eve.pb({pts[i], 1, i});
  sort(all(eve));
  vi par(SZ(polys), -2), ans(SZ(pts), -1);
  auto cmp = [](auto a, auto b) {
    return mp(a.st.cmp(b.st), a.nd) < mp(0, b.nd);
  };
  set<pair<dirSeg<P>, int>, decltype(cmp)> s(cmp);
  for(auto &[_, eve_tp, id]: eve) {
    if(eve_tp == 1) { // point query
      P p = pts[id];
      auto it = s.lower_bound({dirSeg(p, p), 0});
      if(it != s.begin()) { // on vertical segment?
        auto prv = prev(it);
        if(!sgn(p.cross(prv->st.s, prv->st.e))) it--;
      }
      if(it == s.end()) ans[id] = -1;
      else {
        auto [seg, seg_id] = *it;
        int poly_id = segs[seg_id].nd; // strictness there!
        ans[id] = !seg.rev && sgn(p.cross(seg.s, seg.e))
          ? par[poly_id] : poly_id;
      }
    }
    if(eve_tp == 0) { // add segment
      auto it = next(s.insert({segs[id].st, id}).st);
      int poly_id = segs[id].nd;
      if(par[poly_id] == -2) {
        if(it == s.end()) par[poly_id] = -1;
        else {
          int up_rev = it->st.rev, up_id = segs[it->nd].nd;
          par[poly_id] = !up_rev ? par[up_id] : up_id;
        }
      }
    }
    if(eve_tp == 2) s.erase({segs[id].st, id}); // del segment
  }
  return {par, ans};
}
```

SegmentsIntersection.h

Description: Finds one of segments intersections. You should change
dirSeg's comparator, to compare segments at their left end.
Time: $\mathcal{O}(N \log N)$

"SegmentIntersection.h", "directedSegment.h"   79d27e, 33 lines

```cpp
template<class P>
pii segmentsIntersect(vector<pair<P, P>> segments) {
  vector<tuple<P, int, int>> eve; // {point, event_type, id}
  vector<dirSeg<P>> segs;
  for(auto &[s, e]: segments) {
    dirSeg<P> seg(s, e);
    eve.pb({seg.s,0,SZ(segs)}), eve.pb({seg.e,1,SZ(segs)});
    segs.pb(seg);
  }
  sort(all(eve));
  auto inter = [](auto a, auto b) {
    return SZ(segInter(a->st.s, a->st.e, b->st.s, b->st.e));
  };
  auto cmp = [](auto a, auto b) {
    return mp(a.st.cmp(b.st), a.nd) < mp(0, b.nd);
  };
  set<pair<dirSeg<P>, int>, decltype(cmp)> s(cmp);
  for(auto &[_, eve_tp, id]: eve) {
    if(eve_tp == 0) { // add segment
      auto it = s.insert({segs[id], id}).st;
      if(next(it) != s.end() && inter(it, next(it)))
        return {it->nd, next(it)->nd};
      if(it != s.begin() && inter(it, prev(it)))
        return {it->nd, prev(it)->nd};
    }
    if(eve_tp == 1) { // del segment
      auto it = s.erase(s.find({segs[id], id}));
      if(it!=s.begin() && it!=s.end() && inter(it, prev(it)))
        return {it->nd, prev(it)->nd};
    }
  }
  return {-1, -1};
}
```

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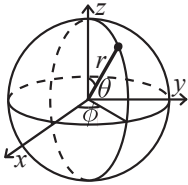
$$x = r \sin\theta \cos\phi \qquad r = \sqrt{x^2 + y^2 + z^2}$$
$$y = r \sin\theta \sin\phi \qquad \theta = \text{acos}(z/\sqrt{x^2 + y^2 + z^2})$$
$$z = r \cos\theta \qquad \phi = \text{atan2}(y, x)$$

7.6 3D

PolyhedronVolume.h

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

3058c3, 6 lines

```cpp
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.

e193d4, 34 lines

```cpp
using D = double;
template<class T> struct Point3D {
  using P = Point3D;
  using R = const P&;
  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; }
  D dist() const { return sqrt((D)dist2()); }
  //Azimuthal angle (longitude) to x-axis in interval [-pi, pi]
  D phi() const { return atan2(y, x); }
  //Zenith angle (latitude) to the z-axis in interval [0, pi]
  D 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(D angle, P axis) const {
    D 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"   892282, 41 lines

```cpp
using P3 = Point3D<D>;
struct PR {
  void ins(int x) { (a == -1 ? a : b) = x; }
  void rem(int x) { (a == x ? a : b) = -1; }
  int cnt() { return (a != -1) + (b != -1); }
  int a, b;
};
struct F { P3 q; int a, b, c; };
vector<F> hull3d(const vector<P3> &A) {
  assert(SZ(A) >= 4);
  vector<vector<PR>> E(SZ(A), vector<PR>(SZ(A), {-1, -1}));
#define E(x,y) E[f.x][f.y]
  vector<F> FS;
  auto mf = [&](int i, int j, int k, int l) {
    P3 q = (A[j] - A[i]).cross((A[k] - A[i]));
    if(q.dot(A[l]) > q.dot(A[i])) q = q * -1;
    F f{q, i, j, k};
    E(a,b).ins(k); E(a,c).ins(j); E(b,c).ins(i);
    FS.pb(f);
  };
  FOR(i, 0, 4) FOR(j, i+1, 4) FOR(k, j+1, 4) {
    mf(i, j, k, 6 - i - j - k); }
  FOR(i, 4, SZ(A)) {
    FOR(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);
   FOR(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.

232b64, 8 lines

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

SpheresIntersection.h
Description: Computes the intersection points of 3 spheres.

"Point3D.h"   738a7f, 17 lines

```
using P = Point3D<D>;
vector<P> trilaterate(P a, P b, P c, D p, D q, D r) {
  b = b - a, c = c - a;
  P e_x = b.unit();
  D i = e_x.dot(c);
  P e_y = (c - e_x * i).unit(), e_z = e_x.cross(e_y);
  D d = b.dist(), j = e_y.dot(c);
  D x = (p * p - q * q + d * d) / 2 / d;
  D y = (p * p - r * r - 2 * i * x + i * i + j * j) / 2 / j;
  D z2 = p * p - x * x - y * y;
  const D EPS = 1e-8; // take care!
  if(z2 < -EPS) return {};
  D z = sqrt(fmax(z2, 0));
  P sol = a + e_x * x + e_y * y;
  if(z2 < EPS) return {sol};
  return {sol - e_z * z, sol + e_z * z};
}
```

Strings (8)

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: $\mathcal{O}(n)$

e144f7, 16 lines

```
vi pi(const string &s) {
  vi p(SZ(s));
```

```
  FOR(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;
  FOR(i, SZ(p)-SZ(s), SZ(p))
    if(p[i] == SZ(pat)) res.pb(i - 2 * SZ(pat));
  return res;
}
```

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

19b0b9, 9 lines

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

Manacher.h
Description: Computes palindromic ranges.

```
s: a   b   a   a   b   a   a   c   a   a   b   b   b   b   a   a
r: 0 0 1 0 0 3 0 0 2 0 0 1 0 0 3 0 0 1 0 0 0 1 1 4 1 1 0 0 0 1 0
```
Time: $\mathcal{O}(N)$

a86c39, 10 lines

```
vi Manacher(string s) {
  vi r(SZ(s) * 2 - 1);
  for(int i=0, m=0, k=0, p=0; i < 2*SZ(s) - 1; m = i++ - 1) {
    while(p < k && i / 2 + r[m] != k) {
      r[i++] = min(r[m--], (k + 1 - p++) / 2); }
    while(k+1 < SZ(s) && p > 0 && s[k+1] == s[p-1]) k++, p--;
    r[i] = (k + 1 - p++) / 2;
  }
  return r;
}
```

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

d987b7, 34 lines

```
int s[N]; /* s[i] > 0, n = len(s), A = sigma(s) */
int n, A, sa[N], lcp[N], cnt[N];
vi x, y;

bool dif(int a, int b, int k) {
  return y[a] != y[b] ||
  (a + k <= n ? y[a + k] : -1) != (b + k <= n ? y[b + k] : -1);
}
/* 1-indexed */
void build() {
  x.clear(); x.resize(max(A, n) + 2); y = x; // +2 enough?
  int j = 0;
  rep(i, 1, n) cnt[x[i] = s[i]]++;
  rep(i, 1, A) cnt[i] += cnt[i - 1];
  per(i, 1, n) sa[cnt[x[i]]--] = i;
```

```
  for(int k = 1; k < n; k *= 2) {
    int p = 0;
    rep(i, n - k + 1, n) y[++p] = i;
    rep(i, 1, n) if(sa[i] > k) y[++p] = sa[i] - k;
    rep(i, 1, A) cnt[i] = 0;
    rep(i, 1, n) cnt[x[i]]++;
    rep(i, 1, A) cnt[i] += cnt[i - 1];
    per(i, 1, n) sa[cnt[x[y[i]]]--] = y[i];
    swap(x, y); A = x[sa[1]] = 1;
    rep(i, 2, n) x[sa[i]] = (A += dif(sa[i - 1], sa[i], k));
    if(n == A) break;
  }
  rep(i, 1, n) {
    if(x[i] == n) { lcp[x[i]] = 0; continue; }
    int nxt = sa[x[i] + 1];
    while(max(i, nxt) + j <= n && s[i + j] == s[nxt + j]) j++;
    lcp[x[i]] = j; j = max(j - 1, 0);
  }
}
```

TandemRepeats.h
Description: Find all (i, p) such that s.substr(i,p) == s.substr(i+p,p). No two intervals with the same period intersect or touch.
Usage: solve("aaababab") // {{0, 1, 1}, {2, 5, 2}}
Time: $\mathcal{O}(N \log N)$

3e559b, 13 lines

```
vector<array<int, 3>> solve(string s) {
  int N = SZ(s); SuffixArray A, B;
  A.init(s); reverse(all(s)); B.init(s);
  vector<array<int, 3>> runs;
  for(int p = 1; 2*p <= N; ++p) { // do in O(N/p) for period p
    for(int i = 0, lst = -1; i+p <= N; i += p) {
      int l = i-B.getLCP(N-i-p,N-i), r = i-p+A.getLCP(i,i+p);
      if (l > r || l == lst) continue;
      runs.pb({lst = l,r,p}); // for each i in [l,r],
    } // s.substr(i,p) == s.substr(i+p,p)
  }
  return runs;
}
```

Hasher.h
Description: Self-explanatory methods for string hashing.

19f1f5, 22 lines

```
const __int128 C = 311;
const ll mod = ll(1e18) + 31;

struct Hasher {
  vector<ll> ha, pw;
  Hasher(string &str) : ha(SZ(str)+1), pw(SZ(str)+1, 1) {
    FOR(i, 0, SZ(str)) {
      ha[i+1] = (ha[i] * C + str[i]) % mod;
      pw[i+1] = pw[i] * C % mod;
    }
  }
  ll hashInterval(int a, int b) { // hash [a, b]
    ll res = (ha[b+1] - __int128(ha[a]) * pw[b+1 - a]) % mod;
    return res < 0 ? res + mod : res;
  }
};

ll hashString(string &s) {
  ll h = 0;
  for(char c: s) h = (h * C + c) % mod;
  return h;
}
```

FastHasher.h
Description: Self-explanatory methods for string hashing.
602bc4, 17 lines

```cpp
// 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).
// "using H = ull;" instead if you think test data is random,
// or work mod 10^9+7 if the Birthday paradox is not a problem.
using ull = unsigned long long;
struct H {
  ull x; H(ull _x=0) : x(_x) {}
  H operator+(H o) { return x + o.x + (x + o.x < x); }
  H operator-(H o) { return *this + ~o.x; }
  H operator*(H o) { auto m = (__uint128_t)x * o.x;
    return H((ull)m) + (ull)(m >> 64); }
  ull get() const { return x + !~x; }
  bool operator==(H o) const { return get() == o.get(); }
  bool operator<(H o) const { return get() < o.get(); }
};
static const H C = (ll)1e11+3; // (order ~ 3e9; random also ok)
```

AhoCorasick.h
Description: Famous Bohun's Aho
f89aac, 26 lines

```cpp
const int N = 1e6 + 5;
int cnt, fail[N], go[N][26];

void add(string s) {
  int u = 0;
  for(auto ch : s) {
    int c = ch - 'a';
    if(!go[u][c]) go[u][c] = ++cnt;
    u = go[u][c];
  }
}

void build() {
  queue<int> q; q.push(0);
  while(!q.empty()) {
    int u = q.front(); q.pop();
    FOR(c, 0, 26) {
      int &v = go[u][c];
      if(v == 0) v = go[fail[u]][c];
      else {
        fail[v] = u == 0 ? 0 : go[fail[u]][c];
        q.push(v);
      }
    }
  }
}
```

Eertree.h
Description: TODO
26a87b, 24 lines

```cpp
const int N = 1e6 + 5, A = 26;
int nxt[N][A], fail[N], last[N], len[N], cnt, par[N];
char s[N];

void prepare(int n) { // 0 <= i <= n + 1
  rep(i, 0, n + 1) FOR(j, 0, A) nxt[i][j] = 0;
  s[0] = '#'; // CUSTOM
  last[0] = cnt = fail[0] = fail[1] = 1;
  len[1] = -1;
}

void add(int n){
  int c = s[n] - 'a'; // CUSTOM
  int v = last[n - 1];
  while(s[n - len[v] - 1] != s[n]) v = fail[v];
  if(!nxt[v][c]){
```

```cpp
    int now = ++cnt, k = fail[v];
    len[now] = len[v] + 2;
    while(s[n - len[k] - 1] != s[n]) k = fail[k];
    fail[now] = nxt[k][c]; nxt[v][c] = now;
    par[now] = v;
  }
  last[n] = nxt[v][c];
}
```

LyndonFactor.h
Description: A string is "simple" if it is strictly smaller than any of its own nontrivial suffixes. The Lyndon factorization of the string s is a factorization $s = w_1 w_2 \ldots w_k$ where all strings w_i are simple and $w_1 \geq w_2 \geq \cdots \geq w_k$. Min rotation gets min index i such that cyclic shift of s starting at i is minimum.
Time: $\mathcal{O}(N)$
be259a, 19 lines

```cpp
vector<string> duval(string s) {
  vector<string> facts;
  for(int i = 0; i < SZ(s); ) {
    int j = i + 1, k = i;
    while(j < SZ(s) && s[k] <= s[j]) {
      if(s[k] < s[j++]) k = i;
      else k++;
    }
    for(; i <= k; i += j-k) facts.pb(s.substr(i, j-k));
  }
  return facts;
}

int minRotation(string s) {
  vector<string> d = duval(s + s); int ind = 0, ans = 0;
  while(ans + SZ(d[ind]) < SZ(s)) ans += SZ(d[ind++]);
  while(ind && d[ind] == d[ind-1]) ans -= SZ(d[ind--]);
  return ans;
}
```

CyclicLCS.h
Description: Cyclic Longest Common Subsequence. maximum of lcs(any cyclic shift of s, any cyclic shift of t)
Time: $\mathcal{O}(NM)$
bd8b93, 39 lines

```cpp
const int N = 2010;

int dp[N * 2][N], from[N * 2][N];
int cyclic_lcs(string s, string t) {
  int n = SZ(s), m = SZ(t);
  auto eq = [&](int a, int b) {
    return s[(a - 1) % n] == t[(b - 1) % m]; };
  dp[0][0] = from[0][0] = 0;
  rep(i, 0, n * 2) rep(j, 0, m) {
    dp[i][j] = 0;
    if(j && dp[i][j - 1] > dp[i][j]) {
      dp[i][j] = dp[i][j - 1]; from[i][j] = 0;
    }
    if(i && j && eq(i, j) && dp[i-1][j-1] + 1 > dp[i][j]) {
      dp[i][j] = dp[i - 1][j - 1] + 1; from[i][j] = 1;
    }
    if(i && dp[i - 1][j] > dp[i][j]) {
      dp[i][j] = dp[i - 1][j]; from[i][j] = 2;
    }
  }
  int ret = 0;
  FOR(i, 0, n) {
    ret = max(ret, dp[i + n][m]);
    // re-root
    int x = i + 1, y = 0;
    while(y <= m && from[x][y] == 0) ++y;
    for(; y <= m && x <= n * 2; ++x) {
```

```cpp
      from[x][y] = 0, --dp[x][m];
      if(x == n * 2) break;
      for(; y <= m; ++y) {
        if(from[x + 1][y] == 2) break;
        if(y + 1 <= m && from[x + 1][y + 1] == 1) {
          ++y; break;
        }
      }
    }
  }
  return ret;
}
```

AllSubstringLCS.h
Description: Computes LCS of s and t[i, j]
Time: $\mathcal{O}(NM)$
23ce75, 17 lines

```cpp
const int N = 2002;
int f[N][N], g[N][N], ans[N][N];
void all_substring_lcs(string s, string t) {
  int n = SZ(s), m = SZ(t);
  s = "#" + s; t = "#" + t;
  rep(i, 1, m) f[0][i] = i;
  rep(i, 1, n) rep(j, 1, m) {
    if(s[i] == t[j]) {
      f[i][j] = g[i][j - 1]; g[i][j] = f[i - 1][j];
    } else {
      f[i][j] = max(f[i - 1][j], g[i][j - 1]);
      g[i][j] = min(g[i][j - 1], f[i - 1][j]);
    }
  }
  rep(i, 1, m) for(int j = i, a = 0; j <= m; ++j) {
    a += i > f[n][j]; ans[i][j] = a; }
}
```

Various (9)

9.1 Misc. algorithms

FastKnapsack.h
Description: Given N non-negative integer weights w and a non-negative target $S \leq t$, computes the maximum $S <= t$ such that S is the sum of some subset of the weights.
Time: $\mathcal{O}(N \max(w_i))$
b21633, 15 lines

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

KnightMoves.h
Description: Finds minimum number of knight moves from (x,y) to (0,0) in non-negative infinite chessboard.
Time: $\mathcal{O}(1)$
cc32ee, 8 lines

```cpp
ll knightMoves(ll x, ll y) {
  ll cnt = max({(x + 1) / 2, (y + 1) / 2, (x + y + 2) / 3});
```

```
    while((cnt % 2) != (x + y) % 2) cnt++;
    if(x == 1 && !y) return 3;
    if(y == 1 && !x) return 3;
    if(x == y && x == 2) return 4;
    return cnt;
}
```

9.2   Optimization tricks

9.2.1   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.

- `FOR(b,0,K) FOR(i,0,(1 << K))`
  `    if(i & 1 << b) D[i] += D[i^(1 << b)];`
  computes all sums of subsets.