Table of Contents

contest

hash.sh

```
# Hashes a file, ignoring all whitespace and comments.
Use for
# verifying that code was correctly typed.
cpp -dD -P -fpreprocessed | tr -d '[:space:]'| md5sum
|cut -c-6
```

template.cpp

```
#pragma GCC optimize("O3")
#pragma GCC optimize("unroll-loops")
#include <bits/stdc++.h>

#define pb push_back
#define all(x) (x).begin(), (x).end()
#define debug(x) { cerr << #x << " = " << x << endl; }
#define IO { ios_base::sync_with_stdio(false);
cin.tie(0); }
#define read(x) freopen(x, "r", stdin)
#define write(x) freopen(x, "w", stdout)
#define endl '\n'
#define int long long
using namespace std;

typedef long long ll;
typedef pair<int, int> ii;
typedef vector<int> vi;

void solve() {

}

signed main() {
 IO;
 int tc = 1;

  #ifndef ONLINE_JUDGE
 read("input.txt");
 write("output.txt");
  #endif

 // cin >> tc;
 for (int cs = 1; cs <= tc; cs++) {
  solve();
 }

 return 0;
}
```

data-structures

FenwickTree.cpp

```
class maxsegtree {
 public:
  int n;
  vector<int> t;
  maxsegtree(vi &res) {
   n = res.size();
   t.assign(2 * n, -INF);
   for (int i = 0; i < n; i++) {
    t[n + i] = res[i];
   }
   build();
  }
  void build() {
   for (int i = n - 1; i > 0; --i) t[i] = max(t[i<<1],
t[i<<1|1]);
  }
  void modify(int p, int value) {
   for (t[p += n] = value; p > 1; p >>= 1) t[p>>1] =
max(t[p] , t[p^1]);
  }
  int query(int l, int r) {
   int res = -INF;
   for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
    if (l&1) res = max(res,t[l++]);
    if (r&1) res = max(res,t[--r]);
   }
   return res;
  }
 };
```

HashMap.cpp

```
#pragma once

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

LazySegmentTree.cpp

```
struct LazySegTree {
    int n;
    vector<int> tree, lazy;
    LazySegTree(int size) {
        n = size;
        tree.assign(4 * n, 0); // stores max in a segment
        lazy.assign(4 * n, 0); // stores pending
additions
    }
    void push(int node, int l, int r) {
        if (lazy[node] != 0) {
            tree[node] += lazy[node]; // apply the lazy
value
            if (l != r) { // not a leaf node
                lazy[2*node] += lazy[node];
                lazy[2*node + 1] += lazy[node];
            }
            lazy[node] = 0; // clear the lazy value
        }
    }
    void range_add(int node, int l, int r, int ql, int
qr, int val) {
        push(node, l, r);
        if (qr < l || r < ql) return; // no overlap
        if (ql <= l && r <= qr) { // total overlap
            lazy[node] += val;
            push(node, l, r);
            return;
        }
        // partial overlap
        int mid = (l + r) / 2;
        range_add(2*node, l, mid, ql, qr, val);
        range_add(2*node+1, mid+1, r, ql, qr, val);
        tree[node] = max(tree[2*node], tree[2*node+1]);
    }
    int range_query(int node, int l, int r, int ql, int
qr) {
        push(node, l, r);
        if (qr < l || r < ql) return INT_MIN; // no
overlap
        if (ql <= l && r <= qr) return tree[node]; //
total overlap
        // partial overlap
        int mid = (l + r) / 2;
        int left = range_query(2*node, l, mid, ql, qr);
        int right = range_query(2*node+1, mid+1, r, ql,
qr);
        return max(left, right);
    }
    // Public API
    void add(int l, int r, int val) {
        range_add(1, 0, n-1, l, r, val);
```

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

LiChaoTree.cpp

```
struct Line {
    long long m, b;
    Line(long long _m = 0, long long _b = LLONG_MAX) :
m(_m), b(_b) {}
    long long eval(long long x) const {
        return m * x + b;
    }
};
struct LiChaoTree {
    struct Node {
        Line line;
        Node *left = nullptr, *right = nullptr;
    };
    int low, high;
    Node* root;
    LiChaoTree(int l, int r) {
        low = l;
        high = r;
        root = nullptr;
    }
    void add_line(Line newLine) {insert(root, low, high,
newLine);}
    void insert(Node* &node, int l, int r, Line newLine)
{
        if (!node) {
            node = new Node();
            node->line = newLine;
            return;
        }
        int mid = (l + r) / 2;
        bool leftBetter = newLine.eval(l) <
node->line.eval(l);
        bool midBetter = newLine.eval(mid) <
node->line.eval(mid);
        if (midBetter) {swap(node->line, newLine);}
        if (r - l == 0) return;
        if (leftBetter != midBetter) {
            insert(node->left, l, mid, newLine);
        } else {
            insert(node->right, mid + 1, r, newLine);
        }
    }
    long long query(int x) {
        return query(root, low, high, x);
    }
    long long query(Node* node, int l, int r, int x) {
        if (!node) return LLONG_MAX;
        long long res = node->line.eval(x);
        if (l == r) return res;
        int mid = (l + r) / 2;
        if (x <= mid)
            return min(res, query(node->left, l, mid,
x));
        else
            return min(res, query(node->right, mid + 1,
r, x));
    }
};
int main() {
    LiChaoTree cht(-1e6, 1e6); // define x-range

    cht.add_line({2, 3}); // add line y = 2x + 3
    cht.add_line({-1, 4}); // add line y = -x + 4

    cout << cht.query(1) << '\n'; // returns min value at
x = 1
}
```

OrderStatisticTree.cpp

```
#include <bits/extc++.h>
using namespace __gnu_pbds;
// Pair<int, int> -> (value, unique_id)
template<typename T>
using OrderedMultiSet = tree<
    pair<T, int>,
    null_type,
    less<pair<T, int>>,
    rb_tree_tag,
    tree_order_statistics_node_update
>;
struct MultiSet {
    OrderedMultiSet<int> s;
    int uid = 0; // Unique ID to distinguish duplicates
    void insert(int x) {s.insert({x, uid++});}
    void erase(int x) {
        auto it = s.lower_bound({x, -1}); // find first
occurrence
        if (it != s.end() && it->first == x) s.erase(it);
    }
    int count(int x) {
        return s.upper_bound({x, INT_MAX}) -
s.lower_bound({x, -1});
    }

    int order_of_key(int x) {return s.order_of_key({x,
-1});}

    int find_by_order(int k) {
  if (k >= (int)s.size()) return -1;
        return s.find_by_order(k)->first;
    }

    int size() {return s.size();}
};
```

RMQ.cpp

```
struct RMQ {
    vector<vector<int>> t;
    int n;
    void init(const vector<int>& a) {
        n = a.size();
        int K = 32 - __builtin_clz(n);
        t.assign(n + 20, vector<int>(K + 5, -inf));
        for (int i = 0; i < n; ++i) {t[i][0] = a[i];}
        for (int j = 1; (1 << j) <= n; ++j) {
            for (int i = 0; i + (1 << j) <= n; ++i) {
                t[i][j] = max(t[i][j - 1], t[i + (1 << (j
- 1))][j - 1]);
            }
        }
    }
    int q(int l, int r) {
        int len = r - l + 1;
        int k = 31 - __builtin_clz(len);
        return max(t[l][k], t[r - (1 << k) + 1][k]);
    }
};
```

SegmentTree.cpp

```
const pair<int, int> NEUTRAL = {INT_MIN, -1};
struct node {
    pair<int, int> val;
    node *L, *R;
    node() {
        val = NEUTRAL;
        L = R = nullptr;
```

```cpp
    }
};
pair<int, int> combine(pair<int, int> a, pair<int, int>
b) {
    return max(a, b);
}
struct SegmentTree {
    node* root;
    int lbound, rbound;

    SegmentTree(int l, int r) {
        lbound = l;
        rbound = r;
        root = new node();
    }

    void update(node* cur, int l, int r, int pos,
pair<int, int> value) {
        if (l == r) {
            cur->val = value;
            return;
        }
        int mid = (l + r) / 2;
        if (pos <= mid) {
            if (!cur->L) cur->L = new node();
            update(cur->L, l, mid, pos, value);
        } else {
            if (!cur->R) cur->R = new node();
            update(cur->R, mid + 1, r, pos, value);
        }
        pair<int, int> left = cur->L ? cur->L->val :
make_pair(MIN, -1);
        pair<int, int> right = cur->R ? cur->R->val :
make_pair(MIN, -1);
        cur->val = combine(left, right);
    }

    pair<int, int> query(node* cur, int l, int r, int ql,
int qr) {
        if (!cur || r < ql || l > qr) return {MIN, -1};
        if (ql <= l && r <= qr) return cur->val;
        int mid = (l + r) / 2;
        pair<int, int> left = query(cur->L, l, mid, ql,
qr);
        pair<int, int> right = query(cur->R, mid + 1, r,
ql, qr);
        return combine(left, right);
    }
    void update(int pos, int val) {
        update(root, lbound, rbound, pos, {val, pos});
    }
    pair<int, int> query(int l, int r) {
        return query(root, lbound, rbound, l, r);
    }
};
```

UnionFind.cpp

```cpp
struct dsu{
 vector<int> p;
 vector<int> sz;
 int comp;
 void init(int n){
  comp = n;
  p.resize(n);
  sz.resize(n);
  for(int i=0; i < n; i++)p[i] = i;
  for(int i=0; i < n; i++)sz[i] = 1;
 }
 int parent(int at){
  if(p[at] == at) return at;
  return p[at] = parent(p[at]);
 }
 void add(int u, int v){
  u = parent(u);
  v = parent(v);
  if(u == v) return;
  comp--;
  p[u] = v;
  sz[v] += sz[u];
 }
 int size(int u){
  return sz[parent(u)];
 }
};
```

UnionFindRollback.cpp

```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].first] = st[i].second;
  st.resize(t);
 }
 bool join(int a, int b) {
  a = find(a), b = find(b);
  if (a == b) return false;
  if (e[a] > e[b]) swap(a, b);
  st.push_back({a, e[a]});
  st.push_back({b, e[b]});
  e[a] += e[b]; e[b] = a;
  return true;
 }
};
```

mergeSortTree.cpp

```cpp
const int MAXN = 1e5 + 5;
int a[MAXN];
multiset<int> t[4 * MAXN];
void build(int v, int tl, int tr) {
    if (tl == tr) {
        t[v].insert(a[tl]);
    } else {
        int tm = (tl + tr) / 2;
        build(v*2, tl, tm);
        build(v*2+1, tm+1, tr);
        t[v].insert(t[v*2].begin(), t[v*2].end());
        t[v].insert(t[v*2+1].begin(), t[v*2+1].end());
    }
}
int query(int v, int tl, int tr, int l, int r, int x) {
    if (l > r) return INT_MAX;
    if (tl == l && tr == r) {
        auto it = t[v].lower_bound(x);
        return (it != t[v].end() ? *it : INT_MAX);
    }
    int tm = (tl + tr) / 2;
    return min(
        query(v*2, tl, tm, l, min(r, tm), x),
        query(v*2+1, tm+1, tr, max(l, tm+1), r, x)
    );
}

void update(int v, int tl, int tr, int pos, int old_val,
int new_val) {
    t[v].erase(t[v].find(old_val));
    t[v].insert(new_val);
    if (tl != tr) {
        int tm = (tl + tr) / 2;
        if (pos <= tm)
            update(v*2, tl, tm, pos, old_val, new_val);
        else
            update(v*2+1, tm+1, tr, pos, old_val,
new_val);
```

```
    }
}

int main() {
    // Initialize array
    int n = 5;
    a[0] = 5;
    a[1] = 1;
    a[2] = 7;
    a[3] = 3;
    a[4] = 6;
    // Build merge sort tree
    build(1, 0, n - 1);
    // Query: Find min element >= 4 in range [1, 4]
    int res = query(1, 0, n - 1, 1, 4, 4);
    if (res != INT_MAX)
        cout << "Query result (>= 4 in [1,4]) = " << res
<< "\n";
    else
        cout << "No value >= 4 in [1,4]\n";
    update(1, 0, n - 1, 2, 7, 0);
    a[2] = 0;
    // Query again
    res = query(1, 0, n - 1, 1, 4, 4);
    if (res != INT_MAX)
        cout << "After update, query result = " << res <<
"\n";
    else
        cout << "After update, no value >= 4 in [1,4]\n";
}
```

geometry

Angle.cpp

```
struct Angle {
 int x, y;
 int t;
 Angle(int x, int y, int t=0) : x(x), y(y), t(t) {}
 Angle operator-(Angle b) const { return {x-b.x, y-b.y,
t}; }
 int half() const {
  assert(x || y);
  return y < 0 || (y == 0 && x < 0);
 }
 Angle t90() const { return {-y, x, t + (half() && x >=
0)}; }
 Angle t180() const { return {-x, -y, t + half()}; }
 Angle t360() const { return {x, y, t + 1}; }
};
bool operator<(Angle a, Angle b) {
 // add a.dist2() and b.dist2() to also compare distances
 return make_tuple(a.t, a.half(), a.y * (ll)b.x) <
        make_tuple(b.t, b.half(), a.x * (ll)b.y);
}
// Given two points, this calculates the smallest angle
between
// them, i.e., the angle that covers the defined line
segment.
pair<Angle, Angle> segmentAngles(Angle a, Angle b) {
 if (b < a) swap(a, b);
 return (b < a.t180() ?
         make_pair(a, b) : make_pair(b, a.t360()));
}
Angle operator+(Angle a, Angle b) { // point a + vector b
 Angle r(a.x + b.x, a.y + b.y, a.t);
 if (a.t180() < r) r.t--;
 return r.t180() < a ? r.t360() : r;
}
Angle angleDiff(Angle a, Angle b) { // angle b - angle a
 int tu = b.t - a.t; a.t = b.t;
 return {a.x*b.x + a.y*b.y, a.x*b.y - a.y*b.x, tu - (b <
a)};
}
```

CircleIntersection.cpp

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

CircleLine.cpp

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

CirclePolygonIntersection.cpp

```
#pragma once

#include "../../content/geometry/Point.h"

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

CircleTangents.cpp

```
#include "Point.h"

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

```
 return out;
}
```

ClosestPair.cpp

```
#include "Point.h"

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

ConvexHull.cpp

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

FastDelaunay.cpp

```
typedef Point<ll> P;
typedef struct Quad* Q;
typedef __int128_t lll; // (can be ll if coords are <
2e4)
P arb(LLONG_MAX,LLONG_MAX); // not equal to any other
point

struct Quad {
 Q rot, o; P p = arb; bool mark;
 P& F() { return r()->p; }
 Q& r() { return rot->rot; }
 Q prev() { return rot->o->rot; }
 Q next() { return r()->prev(); }
} *H;

bool circ(P p, P a, P b, P c) { // is p in the
circumcircle?
 lll p2 = p.dist2(), A = a.dist2()-p2,
     B = b.dist2()-p2, C = c.dist2()-p2;
 return p.cross(a,b)*C + p.cross(b,c)*A + p.cross(c,a)*B
> 0;
}
Q makeEdge(P orig, P dest) {
 Q r = H ? H : new Quad{new Quad{new Quad{new Quad{0}}}};
 H = r->o; r->r()->r() = r;
 rep(i,0,4) r = r->rot, r->p = arb, r->o = i & 1 ? r :
r->r();
 r->p = orig; r->F() = dest;
 return r;
}
void splice(Q a, Q b) {
 swap(a->o->rot->o, b->o->rot->o); swap(a->o, b->o);
}
Q connect(Q a, Q b) {
 Q q = makeEdge(a->F(), b->p);
 splice(q, a->next());
 splice(q->r(), b);
 return q;
}

pair<Q,Q> rec(const vector<P>& s) {
 if (sz(s) <= 3) {
  Q a = makeEdge(s[0], s[1]), b = makeEdge(s[1],
s.back());
  if (sz(s) == 2) return { a, a->r() };
  splice(a->r(), b);
  auto side = s[0].cross(s[1], s[2]);
  Q c = side ? connect(b, a) : 0;
  return {side < 0 ? c->r() : a, side < 0 ? c : b->r() };
 }

#define H(e) e->F(), e->p
#define valid(e) (e->F().cross(H(base)) > 0)
 Q A, B, ra, rb;
 int half = sz(s) / 2;
 tie(ra, A) = rec({all(s) - half});
 tie(B, rb) = rec({sz(s) - half + all(s)});
 while ((B->p.cross(H(A)) < 0 && (A = A->next())) ||
        (A->p.cross(H(B)) > 0 && (B = B->r()->o)));
 Q base = connect(B->r(), A);
 if (A->p == ra->p) ra = base->r();
 if (B->p == rb->p) rb = base;

#define DEL(e, init, dir) Q e = init->dir; if (valid(e))
\
  while (circ(e->dir->F(), H(base), e->F())) { \
   Q t = e->dir; \
   splice(e, e->prev()); \
   splice(e->r(), e->r()->prev()); \
   e->o = H; H = e; e = t; \
  }
 for (;;) {
  DEL(LC, base->r(), o); DEL(RC, base, prev());
  if (!valid(LC) && !valid(RC)) break;
  if (!valid(LC) || (valid(RC) && circ(H(RC), H(LC))))
   base = connect(RC, base->r());
  else
   base = connect(base->r(), LC->r());
 }
 return { ra, rb };
}

vector<P> triangulate(vector<P> pts) {
 sort(all(pts)); assert(unique(all(pts)) == pts.end());
 if (sz(pts) < 2) return {};
 Q e = rec(pts).first;
 vector<Q> q = {e};
 int qi = 0;
 while (e->o->F().cross(e->F(), e->p) < 0) e = e->o;
#define ADD { Q c = e; do { c->mark = 1;
pts.push_back(c->p); \
 q.push_back(c->r()); c = c->next(); } while (c != e); }
 ADD; pts.clear();
 while (qi < sz(q)) if (!(e = q[qi++])->mark) ADD;
 return pts;
}
```

HullDiameter.cpp

```
#pragma once
#include "Point.h"
```

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

InsidePolygon.cpp

```
#pragma once

#include "Point.h"
#include "OnSegment.h"
#include "SegmentDistance.h"

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

LineHullIntersection.cpp

```
 * 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:

#include "Point.h"

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

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

LineProjectionReflection.cpp

```
 * 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.

#pragma once

#include "Point.h"

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();
}
```

MinimumEnclosingCircle.cpp

```
 * Description: Computes the minimum circle that encloses
a set of points.
 * Time: expected O(n)

#pragma once

#include "circumcircle.h"

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

Point.cpp

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

#pragma once
```

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

PointInsideHull.cpp

```
 * 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: O(\log N)

#pragma once

#include "Point.h"
#include "sideOf.h"
#include "OnSegment.h"

typedef Point<ll> P;

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

PolygonArea.cpp

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

#include "Point.h"

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

PolygonCenter.cpp

```
 * Description: Returns the center of mass for a polygon.
 * Time: O(n)
#pragma once

#include "Point.h"

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

PolygonCut.cpp

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

#pragma once

#include "Point.h"

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

PolygonUnion.cpp

```
 * 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: $O(N^2)$, where $N$ is the total number of
points

#pragma once

#include "Point.h"
#include "sideOf.h"

typedef Point<double> P;
double rat(P a, P b) { return sgn(b.x) ? a.x/b.x :
a.y/b.y; }
double polyUnion(vector<vector<P>>& poly) {
 double ret = 0;
 rep(i,0,sz(poly)) rep(v,0,sz(poly[i])) {
```

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

SegmentDistance.cpp

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

```
#pragma once

#include "Point.h"

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

SegmentIntersection.cpp

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.

```
#pragma once

#include "Point.h"
#include "OnSegment.h"

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)};
}
```

circumcircle.cpp

```
#pragma once

#include "Point.h"

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

kdTree.cpp

```
#pragma once

#include "Point.h"

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

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

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

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

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

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

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

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

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

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

lineDistance.cpp

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.

```
#include "Point.h"

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

lineIntersection.cpp

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.

```
#pragma once

#include "Point.h"

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};
}
```

linearTransformation.cpp

```
/**
 * Author: Per Austrin, Ulf Lundstrom
 * Date: 2009-04-09
 * License: CC0
 * Source:
 * Description:\\
\begin{minipage}{75mm}
 Apply the linear transformation (translation, rotation
and scaling) which takes line p0-p1 to line q0-q1 to
point r.
\end{minipage}
\begin{minipage}{15mm}
\vspace{-8mm}
\includegraphics[width=\textwidth]{content/geometry/linea
rTransformation}
\vspace{-2mm}
\end{minipage}
 * Status: not tested
 */
#pragma once

#include "Point.h"

typedef Point<double> P;
P linearTransformation(const P& p0, const P& p1,
  const P& q0, const P& q1, const P& r) {
 P dp = p1-p0, dq = q1-q0, num(dp.cross(dq), dp.dot(dq));
 return q0 + P((r-p0).cross(num),
(r-p0).dot(num))/dp.dist2();
}
```

sideOf.cpp

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

```
#pragma once

#include "Point.h"

template<class P>
int sideOf(P s, P e, P p) { return sgn(s.cross(e, p)); }

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

sphericalDistance.cpp

 * 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.

```
#pragma once
```

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

graph

2sat.cpp

```
 * 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 (\texttt{\tilde{}x}).
 * Usage:
 * TwoSat ts(number of boolean variables);
 * ts.either(0, \tilde3); // Var 0 is true or var 3 is false
 * ts.setValue(2); // Var 2 is true
 * ts.atMostOne({0,\tilde1,2}); // <= 1 of vars 0, \tilde1 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: O(N+E), where N is the number of boolean variables, and E is the number of clauses.

#pragma once

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

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

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

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

 void atMostOne(const vi& li) { // (optional)
  if (sz(li) <= 1) return;
  int cur = ~li[0];
  rep(i,2,sz(li)) {
   int next = addVar();
   either(cur, ~li[i]);
   either(cur, next);
   either(~li[i], next);
   cur = ~next;
  }
  either(cur, ~li[1]);
 }

 vi val, comp, z; int time = 0;
 int dfs(int i) {
  int low = val[i] = ++time, x; z.push_back(i);
  for(int e : gr[i]) if (!comp[e])
   low = min(low, val[e] ?: dfs(e));
  if (low == val[i]) do {
   x = z.back(); z.pop_back();
   comp[x] = low;
   if (values[x>>1] == -1)
    values[x>>1] = x&1;
  } while (x != i);
  return val[i] = low;
 }

 bool solve() {
  values.assign(N, -1);
  val.assign(2*N, 0); comp = val;
  rep(i,0,2*N) if (!comp[i]) dfs(i);
  rep(i,0,N) if (comp[2*i] == comp[2*i+1]) return 0;
  return 1;
 }
};
```

BellmanFord.cpp

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

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

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

BinaryLifting.cpp

```
#pragma once

vector<vi> treeJump(vi& P){
 int on = 1, d = 1;
 while(on < sz(P)) on *= 2, d++;
 vector<vi> jmp(d, P);
 rep(i,1,d) rep(j,0,sz(P))
  jmp[i][j] = jmp[i-1][jmp[i-1][j]];
 return jmp;
}

int jmp(vector<vi>& tbl, int nod, int steps){
 rep(i,0,sz(tbl))
  if(steps&(1<<i)) nod = tbl[i][nod];
 return nod;
}

int lca(vector<vi>& tbl, vi& depth, int a, int b) {
 if (depth[a] < depth[b]) swap(a, b);
 a = jmp(tbl, a, depth[a] - depth[b]);
 if (a == b) return a;
 for (int i = sz(tbl); i--;) {
  int c = tbl[i][a], d = tbl[i][b];
  if (c != d) a = c, b = d;
```

```
 }
 return tbl[0][a];
}
```

Dinic.cpp

```
/**
 * Author: chilli
 * Date: 2019-04-26
 * License: CC0
 * Source: https://cp-algorithms.com/graph/dinic.html
 * Description: Flow algorithm with complexity $O(VE\log
U)$ where $U = \max |\text{cap}|$.
 * $O(\min(E^{1/2}, V^{2/3})E)$ if $U = 1$;
$O(\sqrt{V}E)$ for bipartite matching.
 * Status: Tested on SPOJ FASTFLOW and SPOJ MATCHING,
stress-tested
 */
#pragma once

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

DirectedMST.cpp

```
 * Source: https://github.com/spaghetti-source/algorithm/
blob/master/graph/arborescence.cc
 * and https://github.com/bqi343/USACO/blob/42d177dfb9d6c
e350389583cfa71484eb8ae614c/Implementations/content/graph
s%20(12)/Advanced/DirectedMST.h for the reconstruction
 * Description: Finds a minimum spanning
 * tree/arborescence of a directed graph, given a root
node. If no MST exists, returns -1.
 * Time: O(E \log V)

#pragma once

#include "../data-structures/UnionFindRollback.h"

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

pair<ll, vi> dmst(int n, int r, vector<Edge>& g) {
 RollbackUF uf(n);
 vector<Node*> heap(n);
 for (Edge e : g) heap[e.b] = merge(heap[e.b], new
Node{e});
 ll res = 0;
 vi seen(n, -1), path(n), par(n);
 seen[r] = r;
 vector<Edge> Q(n), in(n, {-1,-1}), comp;
 deque<tuple<int, int, vector<Edge>>> cycs;
 rep(s,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) { /// found cycle, contract
    Node* cyc = 0;
    int end = qi, time = uf.time();
    do cyc = merge(cyc, heap[w = path[--qi]]);
    while (uf.join(u, w));
    u = uf.find(u), heap[u] = cyc, seen[u] = -1;
    cycs.push_front({u, time, {&Q[qi], &Q[end]}});
   }
  }
  rep(i,0,qi) in[uf.find(Q[i].b)] = Q[i];
 }

 for (auto& [u,t,comp] : cycs) { // restore sol
(optional)
  uf.rollback(t);
  Edge inEdge = in[u];
  for (auto& e : comp) in[uf.find(e.b)] = e;
  in[uf.find(inEdge.b)] = inEdge;
 }
 rep(i,0,n) par[i] = in[i].a;
 return {res, par};
}
```

EdgeColoring.cpp

```
 * 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: O(NM)

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

EulerWalk.cpp

```
 * 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: O(V + E)

#pragma once

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

FloydWarshall.cpp

```
#pragma once

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

GlobalMinCut.cpp

```
 * Description: Find a global minimum cut in an
undirected graph, as represented by an adjacency matrix.
 * Time: O(V^3)
#pragma once

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

HLD.cpp

```
 * 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: O((\log N)^2)

#pragma once

#include "../data-structures/LazySegmentTree.h"

template <bool VALS_EDGES> struct HLD {
 int N, tim = 0;
 vector<vi> adj;
 vi par, siz, rt, pos;
 Node *tree;
 HLD(vector<vi> adj_)
  : N(sz(adj_)), adj(adj_), par(N, -1), siz(N, 1),
```

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

LCA.cpp

```
#pragma once

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

struct LCA {
 int T = 0;
 vi time, path, ret;
 RMQ<int> rmq;

 LCA(vector<vi>& C) : time(sz(C)), rmq((dfs(C,0,-1),
ret)) {}
 void dfs(vector<vi>& C, int v, int par) {
  time[v] = T++;
  for (int y : C[v]) if (y != par) {
   path.push_back(v), ret.push_back(time[v]);
   dfs(C, y, v);
  }
 }

 int lca(int a, int b) {
  if (a == b) return a;
  tie(a, b) = minmax(time[a], time[b]);
  return path[rmq.query(a, b)];
 }
 //dist(a,b){return depth[a] + depth[b] -
2*depth[lca(a,b)];}
};
```

LinkCutTree.cpp

```
 * Description: Represents a forest of unrooted trees.
You can add and remove
 * edges (as long as the result is still a forest), and
check whether
 * two nodes are in the same tree.
 * Time: All operations take amortized O(\log N).
#pragma once

struct Node { // Splay tree. Root's pp contains tree's
parent.
 Node *p = 0, *pp = 0, *c[2];
 bool flip = 0;
 Node() { c[0] = c[1] = 0; fix(); }
 void fix() {
  if (c[0]) c[0]->p = this;
  if (c[1]) c[1]->p = this;
  // (+ update sum of subtree elements etc. if wanted)
 }
 void pushFlip() {
  if (!flip) return;
  flip = 0; swap(c[0], c[1]);
  if (c[0]) c[0]->flip ^= 1;
  if (c[1]) c[1]->flip ^= 1;
 }
 int up() { return p ? p->c[1] == this : -1; }
 void rot(int i, int b) {
  int h = i ^ b;
  Node *x = c[i], *y = b == 2 ? x : x->c[h], *z = b ? y :
x;
  if ((y->p = p)) p->c[up()] = y;
  c[i] = z->c[i ^ 1];
  if (b < 2) {
   x->c[h] = y->c[h ^ 1];
   y->c[h ^ 1] = x;
  }
  z->c[i ^ 1] = this;
  fix(); x->fix(); y->fix();
  if (p) p->fix();
  swap(pp, y->pp);
 }
 void splay() { /// Splay this up to the root. Always
finishes without flip set.
  for (pushFlip(); p; ) {
   if (p->p) p->p->pushFlip();
   p->pushFlip(); pushFlip();
   int c1 = up(), c2 = p->up();
   if (c2 == -1) p->rot(c1, 2);
   else p->p->rot(c2, c1 != c2);
  }
 }
 Node* first() { /// Return the min element of the
subtree rooted at this, splayed to the top.
  pushFlip();
  return c[0] ? c[0]->first() : (splay(), this);
 }
};

struct LinkCut {
 vector<Node> node;
 LinkCut(int N) : node(N) {}

 void link(int u, int v) { // add an edge (u, v)
  assert(!connected(u, v));
  makeRoot(&node[u]);
  node[u].pp = &node[v];
 }
 void cut(int u, int v) { // remove an edge (u, v)
  Node *x = &node[u], *top = &node[v];
  makeRoot(top); x->splay();
  assert(top == (x->pp ?: x->c[0]));
  if (x->pp) x->pp = 0;
  else {
   x->c[0] = top->p = 0;
```

```
   x->fix();
  }
 }
 bool connected(int u, int v) { // are u, v in the same
tree?
  Node* nu = access(&node[u])->first();
  return nu == access(&node[v])->first();
 }
 void makeRoot(Node* u) { /// Move u to root of
represented tree.
  access(u);
  u->splay();
  if(u->c[0]) {
   u->c[0]->p = 0;
   u->c[0]->flip ^= 1;
   u->c[0]->pp = u;
   u->c[0] = 0;
   u->fix();
  }
 }
 Node* access(Node* u) { /// Move u to root aux tree.
Return the root of the root aux tree.
  u->splay();
  while (Node* pp = u->pp) {
   pp->splay(); u->pp = 0;
   if (pp->c[1]) {
    pp->c[1]->p = 0; pp->c[1]->pp = pp; }
   pp->c[1] = u; pp->fix(); u = pp;
  }
  return u;
 }
};
```

MinCostMaxFlow.cpp

```
 * 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.
 * Status: Tested on kattis:mincostmaxflow, stress-tested
against another implementation
 * Time: $O(F E \log(V))$ where F is max flow. $O(VE)$
for setpi.
#pragma once

// #include <bits/extc++.h> /// include-line,
keep-include

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

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

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

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

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

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

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

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

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

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

MinimumVertexCover.cpp

```
 * 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.

#include "DFSMatching.h"

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

```
  int i = q.back(); q.pop_back();
  lfound[i] = 1;
  for (int e : g[i]) if (!seen[e] && match[e] != -1) {
   seen[e] = true;
   q.push_back(match[e]);
  }
 }
 rep(i,0,n) if (!lfound[i]) cover.push_back(i);
 rep(i,0,m) if (seen[i]) cover.push_back(n+i);
 assert(sz(cover) == res);
 return cover;
}
```

SCC.cpp

```
#pragma once

vi val, comp, z, cont;
int Time, ncomps;
template<class G, class F> int dfs(int j, G& g, F& f) {
 int low = val[j] = ++Time, x; z.push_back(j);
 for (auto e : g[j]) if (comp[e] < 0)
  low = min(low, val[e] ?: dfs(e,g,f));

 if (low == val[j]) {
  do {
   x = z.back(); z.pop_back();
   comp[x] = ncomps;
   cont.push_back(x);
  } while (x != j);
  f(cont); cont.clear();
  ncomps++;
 }
 return val[j] = low;
}
template<class G, class F> void scc(G& g, F f) {
 int n = sz(g);
 val.assign(n, 0); comp.assign(n, -1);
 Time = ncomps = 0;
 rep(i,0,n) if (comp[i] < 0) dfs(i, g, f);
}
```

WeightedMatching.cpp

```
 * 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: O(N^2M)

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

math

Totient.cpp

```
int phi(int n) { //O(sqrt(n))
    int result = n;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            while (n % i == 0)
                n /= i;
            result -= result / i;
        }
    }
    if (n > 1)
        result -= result / n;
    return result;
}

void phi_1_to_n(int n) { //O(nloglogn)
    vector<int> phi(n + 1);
    for (int i = 0; i <= n; i++)
        phi[i] = i;

    for (int i = 2; i <= n; i++) {
        if (phi[i] == i) {
            for (int j = i; j <= n; j += i)
                phi[j] -= phi[j] / i;
        }
    }
}
```

mobius.cpp

```
const int MAXN = 1e6 + 5;
int mu[MAXN]; // Möbius values
bool isPrime[MAXN];
vector<int> primes;

void computeMobius() {
    for (int i = 0; i < MAXN; i++) mu[i] = 1;
    vector<int> cnt(MAXN, 0);

    for (int i = 2; i < MAXN; ++i) {
        if (!cnt[i]) {
            for (int j = i; j < MAXN; j += i) {
                cnt[j]++;
                mu[j] *= -1;
            }
            for (long long j = 1LL * i * i; j < MAXN; j += 1LL * i * i) {
                mu[j] = 0;
            }
        }
    }
}
```

pollardRho.cpp

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

```
using LL = long long;
mt19937_64 rng(chrono::steady_clock::now().time_since_epo
ch().count());

LL modmul(LL a, LL b, LL mod) {
    return (__int128)a * b % mod; // safe for a * b up to
1e18
}

LL modpow(LL a, LL b, LL mod) {
    LL res = 1;
    while (b > 0) {
        if (b & 1) res = modmul(res, a, mod);
        a = modmul(a, a, mod);
        b >>= 1;
    }
    return res;
}

bool isPrime(LL n) {
    if (n < 2) return false;
    for (LL x : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
37}) {
        if (x == n) return true;
        if (n % x == 0) return false;
        LL d = n - 1, s = 0;
        while (d % 2 == 0) d /= 2, ++s;
        LL cur = modpow(x, d, n);
        if (cur == 1 || cur == n - 1) continue;
        bool passed = false;
        for (int r = 0; r < s; ++r) {
            cur = modmul(cur, cur, n);
            if (cur == n - 1) {
                passed = true;
                break;
            }
        }
        if (!passed) return false;
    }
    return true;
}

LL pollard(LL n) {
    if (n % 2 == 0) return 2;
    if (isPrime(n)) return n;

    while (true) {
        LL c = uniform_int_distribution<LL>(1, n -
1)(rng);
        LL x = uniform_int_distribution<LL>(2, n -
1)(rng);
        LL y = x, d = 1;

        auto f = [&](LL x) {
            return (modmul(x, x, n) + c) % n;
        };

        while (d == 1) {
            x = f(x);
            y = f(f(y));
            d = gcd(abs(x - y), n);
        }

        if (d != n) return d;
    }
}

vector<LL> factorize(LL n) {
    vector<LL> factors;
    function<void(LL)> factor = [&](LL x) {
        if (x == 1) return;
        if (isPrime(x)) {
            factors.push_back(x);
            return;
        }
        LL d = pollard(x);
        factor(d);
        factor(x / d);
    };
    factor(n);
    sort(factors.begin(), factors.end());
    return factors;
}
int main() {
    LL n;
    cin >> n;

    vector<LL> factors = factorize(n);
    cout << "Factors of " << n << ": ";
    for (LL x : factors) cout << x << ' ';
    cout << "\n";
}
```

number-theory

Binomial.cpp

```
#pragma once
#include "ModOperations.h"

const int MX = 2e5 + 10;
int factorial[MX];
bool isPrecomputed = false;

void precomputeFactorials() {
    isPrecomputed = true;
    factorial[0] = 1;
    for (int i = 1; i < MX; i++) {
        factorial[i] = mul(factorial[i - 1], i);
    }
}

int divide(int a, int b) {
    return mul(a, modInverse(b, MOD));
}

int nCr(int n, int r) {
    if (!isPrecomputed) precomputeFactorials();
    if (r < 0 || n < r) return 0;
    return divide(factorial[n], mul(factorial[r],
factorial[n - r]));
}

int nPr(int n, int r) {
    if (!isPrecomputed) precomputeFactorials();
    if (r < 0 || n < r) return 0;
    return divide(factorial[n], factorial[n - r]);
}
```

ExtendedGCD.cpp

```
#pragma once

// Returns gcd(a, b) and finds x, y such that: a*x + b*y
= gcd(a, b)
int egcd(int a, int b, int &x, int &y) {
    x = 1, y = 0;
    int x1 = 0, y1 = 1;
    while (b) {
        int q = a / b;
        int t = b;
        b = a % b;
        a = t;

        t = x1;
        x1 = x - q * x1;
        x = t;

        t = y1;
```

```
        y1 = y - q * y1;
        y = t;
    }
    return a;
}
```

ModOperations.cpp

```
#pragma once

const int MOD = 1e9 + 7;

inline int add(int a, int b) { return (a + b) % MOD; }
inline int sub(int a, int b) { return (a - b + MOD) % MOD;
}
inline int mul(int a, int b) { return (1LL * a * b) % MOD;
}

int power(int a, int b) {
    int res = 1;
    while (b > 0) {
        if (b & 1) res = mul(res, a);
        a = mul(a, a);
        b >>= 1;
    }
    return res;
}
```

numerical

Determinant.cpp

```
 * Description: Calculates determinant of a matrix.
Destroys the matrix.
 * Time: $O(N^3)$
#pragma once

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

FastFourierTransform.cpp

```
 * Description: fft(a) computes $\hat f(k) = \sum_x a[x]
\exp(2\pi i \cdot k x / N)$ for all $k$. N must be a power
of 2.
   Useful for convolution:
   \texttt{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 $(\sum a_i^2 + \sum
b_i^2)\log_2{N} < 9\cdot10^{14}$
   (in practice $10^{16}$; higher for random inputs).
   Otherwise, use NTT/FFTMod.
 * Time: O(N \log N) with $N = |A|+|B|$ ($\tilde 1s$ for
$N=2^{22}$)

#pragma once

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

FastFourierTransformMod.cpp

```
 * Description: Higher precision FFT, can be used for
convolutions modulo arbitrary integers
 * as long as $N\log_2N\cdot \text{mod} < 8.6 \cdot
10^{14}$ (in practice $10^{16}$ or higher).
 * Inputs must be in $[0, \text{mod})$.
 * Time: O(N \log N), where $N = |A|+|B|$ (twice as slow
as NTT or FFT)
#pragma once

#include "FastFourierTransform.h"

typedef vector<ll> vl;
template<int M> vl convMod(const vl &a, const vl &b) {
 if (a.empty() || b.empty()) return {};
 vl res(sz(a) + sz(b) - 1);
 int B=32-__builtin_clz(sz(res)), n=1<<B,
cut=int(sqrt(M));
 vector<C> L(n), R(n), outs(n), outl(n);
 rep(i,0,sz(a)) L[i] = C((int)a[i] / cut, (int)a[i] %
cut);
 rep(i,0,sz(b)) R[i] = C((int)b[i] / cut, (int)b[i] %
cut);
 fft(L), fft(R);
 rep(i,0,n) {
  int j = -i & (n - 1);
  outl[j] = (L[i] + conj(L[j])) * R[i] / (2.0 * n);
  outs[j] = (L[i] - conj(L[j])) * R[i] / (2.0 * n) / 1i;
 }
 fft(outl), fft(outs);
 rep(i,0,sz(res)) {
  ll av = ll(real(outl[i])+.5), cv =
ll(imag(outs[i])+.5);
  ll bv = ll(imag(outl[i])+.5) + ll(real(outs[i])+.5);
```

```
  res[i] = ((av % M * cut + bv) % M * cut + cv) % M;
 }
 return res;
}
```

FastSubsetTransform.cpp

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

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

LinearRecurrence.cpp

```
 * Description: Generates the $k$'th term of an $n$-order
 * linear recurrence $S[i] = \sum_j S[i-j-1]tr[j]$,
 * given $S[0 \ldots \ge n-1]$ 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: O(n^2 \log k)

const ll mod = 5; /** exclude-line */

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

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

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

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

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

PolyInterpolate.cpp

```
/**
 * Author: Simon Lindholm
 * Date: 2017-05-10
 * License: CC0
 * Source: Wikipedia
 * Description: Given $n$ points (x[i], y[i]), computes
an n-1-degree polynomial $p$ that
 * passes through them: $p(x) = a[0]*x^0 + ... +
a[n-1]*x^{n-1}$.
 * For numerical precision, pick $x[k] =
c*\cos(k/(n-1)*\pi), k=0 \dots n-1$.
 * Time: O(n^2)
 */
#pragma once

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

PolyRoots.cpp

```
 * Description: Finds the real roots to a polynomial.
 * Usage: polyRoots({{2,-3,1}},-1e9,1e9) // solve
x^2-3x+2 = 0
 * Time: O(n^2 \log(1/\epsilon))
#pragma once

#include "Polynomial.h"

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

Polynomial.cpp

```
#pragma once

struct Poly {
 vector<double> a;
 double operator()(double x) const {
```

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

SolveLinear.cpp

```
/**
 * Author: Per Austrin, Simon Lindholm
 * Date: 2004-02-08
 * License: CC0
 * 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: O(n^2 m)
 * Status: tested on kattis:equationsolver, and
bruteforce-tested mod 3 and 5 for n,m <= 3
 */
#pragma once

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

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

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

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

SolveLinear2.cpp

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

#include "SolveLinear.h"

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

SolveLinearBinary.cpp

```
 * 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: O(n^2 m)

#pragma once

typedef bitset<1000> bs;

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

 x = bs();
 for (int i = rank; i--;) {
  if (!b[i]) continue;
  x[col[i]] = 1;
  rep(j,0,i) b[j] ^= A[j][i];
 }
 return rank; // (multiple solutions if rank < m)
}
```

subset.cpp

```
Description: Various subset convolutions
Time: $O(2^K * K^2)$ for conv, $O(2^K * K)$ for others
bf4921, 91 lines

vector<LL> XorTransform(vector<LL> p, bool inverse) {
  int n = p.size();
  assert(((n & (n-1))==0));
  for (int len = 1; 2*len <= n; len <<= 1) {
    for (int i = 0; i < n; i += len+len) {
      for (int j = 0; j < len; j++) {
        LL u = p[i+j], v = p[i+len+j];
        if (!inverse) p[i+j] = u+v, p[i+len+j] = u-v;
        else p[i+j] = (u+v)/2, p[i+len+j] = (u-v)/2;
      }
    }
```

```
  }
  return p;
}

vector<LL> SOS(vector<LL> p, bool inverse, bool subset) {
  int k = __builtin_ctz(p.size());
  assert(p.size() == (1<<k));
  for (int i=0; i<k; i++)
    for (int mask=0; mask<(1<<k); mask++)
      if (bool(mask & (1<<i)) == subset) {
        if (!inverse) p[mask] += p[mask^(1<<i)];
        else p[mask] -= p[mask^(1<<i)];
      }
  return p;
}

vector<LL> product(const vector<LL> &a, const vector<LL>
&b) {
  assert(a.size() == b.size());
  vector<LL> ans(a.size());
  for (int i=0; i<a.size(); i++) ans[i] = a[i] * b[i];
  return ans;
}

vector<LL> XorConvolution(vector<vector<LL>> vs) {
    int n = vs.size();
    for (int i=0; i<n; i++) vs[i] = XorTransform(vs[i],
0);
    vector<LL> ans = vs[0];
    for (int i=1; i<n; i++) ans = product(ans, vs[i]);
    ans = XorTransform(ans, 1);
    return ans;
}

vector<LL> ORConvolution(vector<vector<LL>> vs) {
  int n = vs.size();
  for (int i=0; i<n; i++) vs[i] = SOS(vs[i], 0, 1);

  vector<LL> ans = vs[0];
  for (int i=1; i<n; i++) ans = product(ans, vs[i]);
  ans = SOS(ans, 1, 1);
  return ans;
}

vector<LL> AndConvolution(vector<vector<LL>> vs) {
  int n = vs.size();
  for (int i=0; i<n; i++) vs[i] = SOS(vs[i], 0, 0);

  vector<LL> ans = vs[0];
  for (int i=1; i<n; i++) ans = product(ans, vs[i]);
  ans = SOS(ans, 1, 0);
  return ans;
}

vector<LL> SubsetConvolution(const vector<LL> &a, const
vector<LL> &b) {
  int k = __builtin_ctz(a.size());
  assert(a.size() == (1<<k) && b.size() == (1<<k));

  vector<vector<LL>> A(k+1, Z), B(k+1, Z), C(k+1, Z);
  for (int mask=0; mask<(1<<k); mask++) {
    A[__builtin_popcount(mask)][mask] = a[mask];
    B[__builtin_popcount(mask)][mask] = b[mask];
  }

  for (int i=0; i<k; i++) {
    A[i] = SOS(A[i], 0, 1);
    B[i] = SOS(B[i], 0, 1);
    for (int j=0; j<=i; j++)
      for (int mask = 0; mask < (1<<k); mask++)
        C[i][mask] += A[j][mask]*B[i-j][mask];
  }

  vector<LL> ans(1<<k);
  for (int mask=0; mask<(1<<k); mask++) {
    ans[mask] = C[__builtin_popcount(mask)][mask];
  }
  return ans;
}
```

strings

Hashing-codeforces.cpp

```
#pragma once

typedef uint64_t ull;
static int C; // initialized below

// Arithmetic mod two primes and 2^32 simultaneously.
// "typedef uint64_t H;" instead if Thue-Morse does not
apply.
template<int M, class B>
struct A {
 int x; B b; A(int x=0) : x(x), b(x) {}
 A(int x, B b) : x(x), b(b) {}
 A operator+(A o){int y = x+o.x; return{y - (y>=M)*M,
b+o.b};}
 A operator-(A o){int y = x-o.x; return{y + (y< 0)*M,
b-o.b};}
 A operator*(A o) { return {(int)(1LL*x*o.x % M), b*o.b};
}
 explicit operator ull() { return x ^ (ull) b << 21; }
 bool operator==(A o) const { return (ull)*this ==
(ull)o; }
 bool operator<(A o) const { return (ull)*this < (ull)o;
}
};
typedef A<1000000007, A<1000000009, unsigned>> H;

struct HashInterval {
 vector<H> ha, pw;
 HashInterval(string& str) : ha(sz(str)+1), pw(ha) {
  pw[0] = 1;
  rep(i,0,sz(str))
   ha[i+1] = ha[i] * C + str[i],
   pw[i+1] = pw[i] * C;
 }
 H hashInterval(int a, int b) { // hash [a, b)
  return ha[b] - ha[a] * pw[b - a];
 }
};

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

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

#include <sys/time.h>
int main() {
 timeval tp;
 gettimeofday(&tp, 0);
 C = (int)tp.tv_usec; // (less than modulo)
 assert((ull)(H(1)*2+1-3) == 0);
 // ...
}
```

Hashing.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).
// "typedef ull H;" instead if you think test data is
random,
// or work mod 10^9+7 if the Birthday paradox is not a
problem.
typedef uint64_t ull;
struct H {
 ull x; H(ull x=0) : x(x) {}
 H operator+(H o) { return x + o.x + (x + o.x < x); }
 H operator-(H o) { return *this + ~o.x; }
 H operator*(H o) { auto m = (__uint128_t)x * o.x;
  return H((ull)m) + (ull)(m >> 64); }
 ull get() const { return x + !~x; }
 bool operator==(H o) const { return get() == o.get(); }
 bool operator<(H o) const { return get() < o.get(); }
};
static const H C = (ll)1e11+3; // (order ~ 3e9; random
also ok)

struct HashInterval {
 vector<H> ha, pw;
 HashInterval(string& str) : ha(sz(str)+1), pw(ha) {
  pw[0] = 1;
  rep(i,0,sz(str))
   ha[i+1] = ha[i] * C + str[i],
   pw[i+1] = pw[i] * C;
 }
 H hashInterval(int a, int b) { // hash [a, b)
  return ha[b] - ha[a] * pw[b - a];
 }
};

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

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

KMP.cpp

```
#pragma once

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

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

Manacher.cpp

```
/**
 * Author: User adamant on CodeForces
 * Source: http://codeforces.com/blog/entry/12143
 * Description: For each position in a string, computes
p[0][i] = half length of
 * longest even palindrome around pos i, p[1][i] =
longest odd (half rounded down).
 * Time: O(N)
 * Status: Stress-tested
 */
#pragma once

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

MinRotation.cpp

```
 * Description: Finds the lexicographically smallest
rotation of a string.
 * Time: O(N)

#pragma once

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

SuffixArray.cpp

```
/**
 * Author:    , chilli
 * Date: 2019-04-11
 * License: Unknown
 * Source: Suffix array - a powerful tool for dealing
with strings
 * (Chinese IOI National team training paper, 2009)
 * Description: Builds suffix array for a string.
 * \texttt{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
\texttt{sa[0] = n}.
 * The \texttt{lcp} array contains longest common
prefixes for
 * neighbouring strings in the suffix array:
 * \texttt{lcp[i] = lcp(sa[i], sa[i-1])}, \texttt{lcp[0]
= 0}.
 * The input string must not contain any nul chars.
 * Time: O(n \log n)
 * Status: stress-tested
 */
#pragma once

struct SuffixArray {
 vi sa, lcp;
 SuffixArray(string s, int lim=256) { // or vector<int>
```

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

Zfunc.cpp

```
/**
 * Author: chilli
 * License: CC0
 * Description: z[i] computes the length of the longest
common prefix of s[i:] and s,
 * except z[0] = 0. (abacaba -> 0010301)
 * Time: O(n)
 * Status: stress-tested
 */
#pragma once

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

various

DivideAndConquerDP.cpp

```
/**
 * Author: Simon Lindholm
 * License: CC0
 * Source: Codeforces
 * Description: Given $a[i] = \min_{lo(i) \le k <
hi(i)}(f(i, k))$ where the (minimal)
 * optimal $k$ increases with $i$, computes $a[i]$ for $i
= L..R-1$.
 * Time: O((N + (hi-lo)) \log N)
 * Status: tested on
http://codeforces.com/contest/321/problem/E
 */
#pragma once

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

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

FastKnapsack.cpp

```
/**
 * Author: Mårten Wiman
 * License: CC0
 * Source: Pisinger 1999, "Linear Time Algorithms for
Knapsack Problems with Bounded Weights"
 * Description: Given N non-negative integer weights w
and a non-negative target t,
 * computes the maximum S <= t such that S is the sum of
some subset of the weights.
 * Time: O(N \max(w_i))
 * Status: Tested on kattis:eavesdropperevasion,
stress-tested
 */
#pragma once

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

KnuthDP.cpp

```
/**
 * Efficiently solves interval DP of the form:
 * dp[i][j] = min(dp[i][k] + dp[k][j] + cost[i][j])
 * where the optimal `k` for dp[i][j] lies between
opt[i][j - 1] and opt[i + 1][j].
 * Requirements:
 * - cost(i, j) must satisfy the quadrangle inequality
(monotonicity).
 * - opt[i][j] increases with i and j.
 * Time Complexity: O(N^2)
 * Works best when transitions are expensive (like O(1)
cost + nested loop).
 */
 const int INF = 1e9;
 int dp[500][500], opt[500][500]; // adjust sizes as
needed
 int cost(int i, int j); // define your cost function
 void knuthDP(int n) {
     for (int i = 0; i < n; ++i) {
         dp[i][i] = 0;
         opt[i][i] = i;
     }
     for (int len = 1; len < n; ++len) {
         for (int i = 0; i + len < n; ++i) {
             int j = i + len;
             dp[i][j] = INF;
             int l = opt[i][j - 1];
             int r = opt[i + 1][j];
```

```
            if (l > r) swap(l, r);
            for (int k = l; k <= r; ++k) {
                int val = dp[i][k] + dp[k][j] + cost(i,
j);
                if (val < dp[i][j]) {
                    dp[i][j] = val;
                    opt[i][j] = k;
                }
            }
        }
    }
}
```

trie.cpp

```
const int MAXN = 1000005;

struct trie {
    vector<vi> nxt;
    vi fin;
    int used = 0;

    trie() {
        nxt = vector<vi>(MAXN, vi(26, -1));
        fin = vi(MAXN);
    }

    void insert(string s) {
        int cur = 0;
        for (int i = 0; i < s.size(); i++) {
            int c = s[i] - 'a';
            if (nxt[cur][c] == -1) {
                used++;
                nxt[cur][c] = used;
            }
            cur = nxt[cur][c];
        }
        fin[cur]++;
    }

    bool search(string s) {
        int cur = 0;
        for (int i = 0; i < s.size(); i++) {
            int c = s[i] - 'a';
            if (nxt[cur][c] == -1) {
                return false;
            }
            cur = nxt[cur][c];
        }
        return fin[cur] > 0;
    }

    int ans = 0;

    void traverse(int node, int level) {
        for (int i = 0; i < 26; i++) {
            if (nxt[node][i] == -1) continue;
            for (int j = i + 1; j < 26; j++) {
                if (nxt[node][j] == -1) continue;
            }
        }
        for (int i = 0; i < 26; i++) if (nxt[node][i] !=
-1) traverse(nxt[node][i], level + 1);
    }
};
```