2025

Template Book 3.0

Contents

Comb 2
nCr MO 2
Link Cut Tree 3
Tree Sack 5
Virtual Tree 6
De Bruijn 6
Bellman 6
Euler Path (Undirected) 7
Euler Path (Directed) 7
HopCroft 7
Hungarian 8
Dominator Tree 9
Fast Dinic 10
BipartiteDSU 10
MO Tree 11
Arethmetic Segment Tree 12
Offline 2D BIT 13
PST 13
Treap 14
Implicit Treap 15
SQRT 16
Minimum Adjacent Swaps 16
GP Hashmap 16
Wavelet Tree 17
Manacher 18
Suffix Array 19
Suffix Autometa 20
Hashing 22
Arethmitics on Strings 23
Misc 25
Discrete Log 25
2D Hash 26
Gamal Notes 27
Geometry 28
Transformations 28
Angles 28
Lines 29
Segments 29
Rays 30
Polygons 30
Circles 31
Minimum Enclosing Circle 31
Maximum Circle Cover 32
Biggest Circle in convex 32
Frequency Lines 32
Polygon Clipping 33
Centroid of polygon 33
Convex Hull 33
Minkowski Sum 34
Rotating Calipers 34
Half Plane 35
Circle Union 36
Integer Angles 37
3D-Geometry 37
Transformation 3D 41
3D-Misc 42
2D – MISC 45

Comb

```
int fact[N], invr[N];
void build() {
  fact[0] = 1;
  for(int i = 1; i < N; ++i) fact[i]=1ll*fact[i-1]*i%mod;
  invr[N - 1] = inverse(fact[N - 1]);
  for(int i = N - 2; ~i; --i) invr[i] = 1ll*invr[i + 1]*(i + 1)%mod;
}

ll power(ll b, ll e, ll m) {
  ll ans = 1;
  for (; e; b = b * b % m, e /= 2)
    if (e & 1) ans = ans * b % m;
  return ans;
}

ll inverse(ll b) { return power(b, mod - 2, mod); }

int nPr(int n, int r) {return n<r?0:1ll*fact[n]*invr[n - r]%mod;}
int nCr(int n, int r) {return 1ll*nPr(n, r)*invr[r]%mod;}
int SAndBars(int n, int k) {return min(n, k)<0?0:nCr(n+k-1, k-1);}
int catalan(int n) {return 1ll*nCr(n * 2, n)*inverse(n + 1)%mod;}

int nCr(int n, int r){
  __int128 sum = 1;
  for (int i = 1; i <= r; i++) sum = sum * (n - r + i) / i;
  return (int)sum;
}

// build pascal triangle
void build() {
  nCr[0][0] = 1;
  for(int row = 1; row < N; ++row) {
    nCr[row][0] = 1;
    for(int i = 0; i <= row / 2; ++i) {
      int curr = nCr[row - 1][i];
      if(i) curr += nCr[row - 1][i - 1];
      if(curr >= mod) curr -= mod;
      nCr[row][i] = nCr[row][row - i] = curr % mod;
    }
  }
}
```

nCr MO

```
// each query is sum of nCi [0, r]
struct Query{ int n, r, idx, t; };

void addM(int n, int m){
  curr = curr + nCr(n, m);
  if(curr >= mod) curr -= mod;
}

void remM(int n, int m) {
  curr = (curr - nCr(n, m) + mod) % mod;
}

void addN(int n, int r){
  curr = 2LL * curr - nCr(n, r);
  while(curr >= mod) curr -= mod;
  while(curr < 0) curr += mod;
}

void remN(int n, int r) {
  curr = 1ll * (nCr(n - 1, r) + curr) * inv2 % mod;
}

vector<int> MO(vector<Query> &Q, int t) {
  const int sq = sqrt(N);
  sort(Q.begin(), Q.end(),
    [&](Query &a, Query &b) {
      if(a.n/sq == b.n/sq) return a.n / sq & 1? a.r > b.r : a.r <
b.r;
      return a.n/sq < b.n/sq;
    });

  int nmo = Q[0].n, rmo = 0;
  addM(nmo, 0);

  vector<int> res(t);
  for(auto &[nq, rq, iq, type]: Q){
    while(nmo < nq) addN(nmo++, rmo);
    while(nmo > nq) remN(nmo--, rmo);
    while(rmo < rq) addM(nmo, ++rmo);
    while(rmo > rq) remM(nmo, rmo--);

    res[iq] += type * curr;
    if(res[iq] < 0) res[iq] += mod;
    if(res[iq] >= mod) res[iq] -= mod;
  }

  return res;
}
```

Link Cut Tree

```
struct Node {
  int p = 0, c[2] = {0, 0}, pp = 0;
  bool flip = 0;
  // sz -> aux tree size, ssz = subtree size in rep tree, vsz =
virtual tree size
  int sz = 0, ssz = 0, vsz = 0;
  ll val = 0, sum = 0, lazy = 0, subsum = 0, vsum = 0;

  Node() {}

  Node(int x) {
    val = x, sum = x, sz = 1;
    lazy = 0, ssz = 1, vsz = 0;
    subsum = x, vsum = 0;
  }
};

struct LCT {
  vector<Node> t;
  LCT() {}
  LCT(int n) : t(n + 1) {}

  LCT(vector<int>& vals) {
    t.assign(vals.size(), {});
    for (int i = 1; i < vals.size(); ++i) {
      t[i] = Node(vals[i]);
    }
  }

  // <independant splay tree code>
  int dir(int x, int y) { return t[x].c[1] == y; }

  void set(int x, int d, int y) {
    if (x) t[x].c[d] = y, pull(x);
    if (y) t[y].p = x;
  }

  void pull(int x) {
    // merge
    if (!x) return;
    int &l = t[x].c[0], &r = t[x].c[1];
    push(l);
    push(r);
    t[x].sum = t[l].sum + t[r].sum + t[x].val;
    t[x].sz = t[l].sz + t[r].sz + 1;
    t[x].ssz = t[l].ssz + t[r].ssz + t[x].vsz + 1;
    t[x].subsum = t[l].subsum + t[r].subsum + t[x].vsum +
t[x].val;
  }

void push(int x) {
  if (!x) return;
  int &l = t[x].c[0], &r = t[x].c[1];
  if (t[x].flip) {
    swap(l, r);
    if (l) t[l].flip ^= 1;
    if (r) t[r].flip ^= 1;
    t[x].flip = 0;
  }
  if (t[x].lazy) {
    t[x].val += t[x].lazy;
    t[x].sum += t[x].lazy * t[x].sz;
    t[x].subsum += t[x].lazy * t[x].ssz;
    t[x].vsum += t[x].lazy * t[x].vsz;
    if (l) t[l].lazy += t[x].lazy;
    if (r) t[r].lazy += t[x].lazy;
    t[x].lazy = 0;
  }
}

void rotate(int x, int d) {
  int y = t[x].p, z = t[y].p, w = t[x].c[d];
  swap(t[x].pp, t[y].pp);
  set(y, !d, w);
  set(x, d, y);
  set(z, dir(z, y), x);
}

void splay(int x) {
  for (push(x); t[x].p;) {
    int y = t[x].p, z = t[y].p;
    push(z);
    push(y);
    push(x);
    int dx = dir(y, x), dy = dir(z, y);
    if (!z) rotate(x, !dx);
    else if (dx == dy) rotate(y, !dx), rotate(x, !dx);
    else rotate(x, dy), rotate(x, dx);
  }
}

// </independant splay tree code>

// making it a root in the rep. tree
void make_root(int u) {
  access(u);
  int l = t[u].c[0];
  t[l].flip ^= 1;
  swap(t[l].p, t[l].pp);
  t[u].vsz += t[l].ssz;
  t[u].vsum += t[l].subsum;
  set(u, 0, 0);
}
```

```
  // make the path from root to u a preferred path
  // returns last path-parent of a node as it moves up the
tree
  int access(int _u) {
    int last = _u;
    for (int v = 0, u = _u; u; u = t[v = u].pp) {
      splay(u), splay(v);
      t[u].vsz -= t[v].ssz;
      t[u].vsum -= t[v].subsum;
      int r = t[u].c[1];
      t[u].vsz += t[r].ssz;
      t[u].vsum += t[r].subsum;
      t[v].pp = 0;
      swap(t[r].p, t[r].pp);
      set(u, 1, v);
      last = u;
    }
    splay(_u);
    return last;
  }

  // link v as a child to u
  void link(int u, int v) {
    // assert(!connected(u, v));
    make_root(v);
    access(u), splay(u);
    t[v].pp = u;
    t[u].vsz += t[v].ssz;
    t[u].vsum += t[v].subsum;
  }

  void cut(int u) {
    // cut par[u] -> u, u is non root vertex
    access(u);
    assert(t[u].c[0] != 0);
    t[t[u].c[0]].p = 0;
    t[u].c[0] = 0;
    pull(u);
  }

  void cut(int u, int v) {
    if (depth(u) < depth(v)) swap(u, v);
    cut(u);
  }

  // parent of u in the rep. tree
  int get_parent(int u) {
    access(u);
    splay(u);
    push(u);
    u = t[u].c[0];
    push(u);
    while (t[u].c[1]) {
      u = t[u].c[1];
      push(u);
    }
    splay(u);
    return u;
  }

  // root of the rep. tree containing this node
  int find_root(int u) {
    access(u);
    splay(u);
    push(u);
    while (t[u].c[0]) {
      u = t[u].c[0];
      push(u);
    }
    splay(u);
    return u;
  }

  bool connected(int u, int v) {
    return find_root(u) == find_root(v);
  }

  // depth in the rep. tree
  int depth(int u) {
    access(u);
    splay(u);
    return t[u].sz;
  }

  int lca(int u, int v) {
    // assert(connected(u, v));
    if (u == v) return u;
    if (depth(u) > depth(v)) swap(u, v);
    access(v);
    return access(u);
  }

  int is_root(int u) {
    return get_parent(u) == 0;
  }

  int component_size(int u) {
    return t[find_root(u)].ssz;
  }

  int subtree_size(int u) {
    int p = get_parent(u);
    if (p == 0) {
      return component_size(u);
    }
    cut(u);
    int ans = component_size(u);
```

```
    link(p, u);
    return ans;
  }

  ll component_sum(int u) {
    return t[find_root(u)].subsum;
  }

  ll subtree_sum(int u) {
    int p = get_parent(u);
    if (p == 0) {
      return component_sum(u);
    }
    cut(u);
    ll ans = component_sum(u);
    link(p, u);
    return ans;
  }

  // sum of the subtree of u when root is specified
  ll subtree_query(int u, int root) {
    int cur = find_root(u);
    make_root(root);
    ll ans = subtree_sum(u);
    make_root(cur);
    return ans;
  }

  // path sum
  ll query(int u, int v) {
    int cur = find_root(u);
    make_root(u);
    access(v);
    ll ans = t[v].sum;
    make_root(cur);
    return ans;
  }

  void upd(int u, int x) {
    access(u);
    splay(u);
    t[u].val += x;
  }

  // add x to the nodes on the path from u to v
  void upd(int u, int v, int x) {
    int cur = find_root(u);
    make_root(u);
    access(v);
    t[v].lazy += x;
    make_root(cur);
  }
};
```

Tree Sack

```
void getSz(int u, int p = 0) {
  sz[u] = 1;
  for(auto v: adj[u])
    if(v != p) getSz(v, u), sz[u] += sz[v];
}

void upd(int c, int t) {}

void add(int u, int t, int p) {
  upd(u, t);
  for(auto v: adj[u])
    if(v != p) add(v, t, u);
}

void go(int u, int p = 0, int keep = 0) {
  array<int, 2> bg{-1, -1};
  for(auto v: adj[u])
    if(v != p) bg = max(bg, {sz[v], v});

  for(auto v: adj[u]) {
    if(v == p || v == bg[1]) continue;
    go(v, u);
  }

  if(~bg[1]) go(bg[1], u, 1);

  upd(u, 1);
  for(auto v: adj[u])
    if(v != bg[1] && v != p) add(v, 1, u);

  if(!keep) add(u, 0, p);
}
```

Virtual Tree

```
void VTree(vector<int> &v) {
   sort(v.begin(), v.end(), [](int x, int y) { return in[x] < in[y]; });

   int s = v.size();
   for (int i = 1; i < s; i++)
      v.push_back(getLca(v[i], v[i - 1]));

   sort(v.begin(), v.end());
   v.erase(unique(v.begin(), v.end()), v.end());
   sort(v.begin(), v.end(), [](int x, int y) { return in[x] < in[y]; });


   int n = v.size();
   stack<int> st;
   vector<vector<int>> t(n);
   st.push(0);

   // T is the adj of the virtual tree rooted at 0
   // the real id of node u is v[u]
   for (int i = 1; i < n; i++) {
      while(!isAncestor(v[st.top()], v[i])) st.pop();
      t[st.top()].push_back(i);
      st.push(i);
   }
}
```

De Bruijn

```
string A = "01"; // Character set
unordered_set<string> seen;
vector<int> edges;
void dfs(string node) {
   for (int i = 0; i < A.size(); ++i) {
      string str = node + A[i];
      if (!seen.count(str)) {
         seen.insert(str);
         dfs(str.substr(1));
         edges.push_back(i);
      }
   }
}

string deBruijn(int n) {
   seen.clear(), edges.clear();
   auto st = string(n - 1, A[0]);
   dfs(st);
   string res;
   int l = pow(A.size(), n);
   for (int i = 0; i < l; ++i) res += A[edges[i]];
   res += st;
   return res;
}
```

Bellman

```
const int oo = 2e18;
vector<array<int, 3>> edg;

// to find any negative cycle set all d = 0
vector<int> bellman(int n, int s) {
   vector d(n + 1, oo), p(n + 1, 0ll);
   d[s] = 0;

   int lst = -1;
   for(int i = 0; i <= n; ++i) {
      lst = -1;
      for(auto &[u, v, w]: edg)
         if(d[u] + w < d[v])
            d[v] = d[u] + w, lst = v, p[v] = u;
      if(lst == -1) break;
   }

   // no negative cycle
   if (lst == -1) return d;

   for(int i = 0; i < n; ++i)
      for(auto &[u, v, w]: edg)
         if(d[u] < oo && d[u] + w < d[v]) d[v] = -oo;

   int y = lst;
   for (int i = 0; i < n; ++i) y = p[y];

   vector<int> cyc;
   for (int v = y;; v = p[v]) {
      cyc.push_back(v);
      if (v == y && cyc.size() > 1) break;
   }
   reverse(cyc.begin(), cyc.end());

   return d;
}
```

Euler Path (Undirected)

```
bool EulerPathUndir() {
  int st = -1, en = -1, cand = 1;
  for(int u = 1; u <= n; ++u) {
    if(!adj[u].empty()) cand = u;
    if(deg[u] & 1) {
      if(st == -1) st = u;
      else if(en == -1) en = u;
      else return false;
    }
  }

  if(st == -1) st = cand;
  else if(en == -1) return false;

  vector<int> edges, nodes, vis(m);
  auto dfs = [&](auto &&dfs, int u) -> void {
    while(!adj[u].empty()) {
      auto [v, i] = adj[u].back();
      adj[u].pop_back();
      if(vis[i]) continue;
      vis[i] = 1, dfs(dfs, v);
      edges.push_back(i);
      nodes.push_back(v);
    }
  };

  dfs(dfs, st);
  if(size(edges) != m) return false;
  nodes.push_back(st);
  reverse(nodes.begin(), nodes.end());
  reverse(edges.begin(), edges.end());
  return true;
}
```

Euler Path (Directed)

```
bool EulerPathUndir() {
  int st = -1, en = -1, cand = 1;
  for(int u = 1; u <= n; ++u) {
    if(!adj[u].empty()) cand = u;
    if(deg[u] & 1) {
      if(st == -1) st = u;
      else if(en == -1) en = u;
      else return false;
    }
  }

  if(st == -1) st = cand;
  else if(en == -1) return false;

  vector<int> edges, nodes, vis(m);
  auto dfs = [&](auto &&dfs, int u) -> void {
    while(!adj[u].empty()) {
      auto [v, i] = adj[u].back();
      adj[u].pop_back();
      if(vis[i]) continue;
      vis[i] = 1, dfs(dfs, v);
      edges.push_back(i);
      nodes.push_back(v);
    }
  };

  dfs(dfs, st);
  if(size(edges) != m) return false;
  nodes.push_back(st);
  reverse(nodes.begin(), nodes.end());
  reverse(edges.begin(), edges.end());
  return true;
}
```

HopCroft

```
struct HK {
  int n, m;
  vector<vector<int>> g;
  vector<int> l, r, d, p; int ans;
  HK(int n, int m) : n(n), m(m), g(n), l(n, -1), r(m, -1), ans(0) {}

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

  int match(){
    while (true) {
      queue<int> q; d.assign(n, -1);
      for (int i = 0; i < n; i++)
        if (l[i] == -1) q.push(i), d[i] = 0;
      while (!q.empty()) {
        int x = q.front(); q.pop();
        for (int y : g[x])
          if (r[y] != -1 && d[r[y]] == -1)
            d[r[y]] = d[x] + 1, q.push(r[y]);
      }
      bool match = false;
      for (int i = 0; i < n; i++)
        if (l[i] == -1 && dfs(i)) ++ans, match = true;
      if (!match) break;
    }
    return ans;
  }

  bool dfs(int x) {
    for (int y : g[x])
      if (r[y] == -1 || (d[r[y]] == d[x] + 1 && dfs(r[y])))
        return l[x] = y, r[y] = x, d[x] = -1, true;
    return d[x] = -1, false;
  }
};
```

Hungarian

```
// one indexed, weighted (minimum cost from maximum matching), O(N ^ 3)
// for maximum cost make edges negative
struct Hungarian {
  static const int oo = 2e18;
  int n;
  vector<int> fx, fy, d, l, r, arg, trace;
  vector<vector<int>> c;
  queue<int> q;
  int start = 0, finish = 0;

  Hungarian(int _n, int _m): n(max(_n, _m)) {
    fx.assign(n + 1, 0);
    fy.assign(n + 1, 0);
    d.assign(n + 1, 0);
    l.assign(n + 1, 0);
    r.assign(n + 1, 0);
    arg.assign(n + 1, 0);
    trace.assign(n + 1, 0);
    c.assign(n + 1, vector<int>(n + 1));
    for (int i = 1; i <= n; ++i) {
      fy[i] = l[i] = r[i] = 0;
      for (int j = 1; j <= n; ++j) c[i][j] = oo;
      // make it 0 for maximum cost matching
      // (not necessarily with maximum matching)
    }
  }

  void add_edge(int u, int v, int cost) { c[u][v] = min(c[u][v], cost); }
  int getC(int u, int v) { return c[u][v] - fx[u] - fy[v]; }

  void initBFS() {
    while (!q.empty()) q.pop();
    q.push(start);
    for (int i = 0; i <= n; ++i) trace[i] = 0;
    for (int v = 1; v <= n; ++v)
      d[v] = getC(start, v), arg[v] = start;
    finish = 0;
  }

  void findAugPath() {
    while (!q.empty()) {
      int u = q.front();
      q.pop();
      for (int v = 1; v <= n; ++v) {
        if (!trace[v]) {
          int w = getC(u, v);
          if (not w) {
            trace[v] = u;
            if (!r[v]) {
              finish = v;
              return;
            }
            q.push(r[v]);
          }
          if (d[v] > w) {
            d[v] = w;
            arg[v] = u;
          }
        }
      }
    }
  }

  void subX_addY() {
    int delta = oo;
    for (int v = 1; v <= n; ++v) {
      if (trace[v] == 0 && d[v] < delta) {
        delta = d[v];
      }
    }

    fx[start] += delta;
    for (int v = 1; v <= n; ++v) {
      if (trace[v]) {
        int u = r[v];
        fy[v] -= delta;
        fx[u] += delta;
      } else d[v] -= delta;
    }

    for (int v = 1; v <= n; ++v) {
      if (!trace[v] && !d[v]) {
        trace[v] = arg[v];
        if (not r[v]) {
          finish = v;
          return;
        }
        q.push(r[v]);
      }
    }
  }

  void Enlarge() {
    do {
      int u = trace[finish];
      int nxt = l[u];
      l[u] = finish;
      r[finish] = u;
      finish = nxt;
    } while (finish);
  }

  pair<int, int> maximum_matching() {
```

```
    for (int u = 1; u <= n; ++u) {
      fx[u] = c[u][1];
      for (int v = 1; v <= n; ++v)
        fx[u] = min(fx[u], c[u][v]);
    }
    for (int v = 1; v <= n; ++v) {
      fy[v] = c[1][v] - fx[1];
      for (int u = 1; u <= n; ++u)
        fy[v] = min(fy[v], c[u][v] - fx[u]);
    }
    for (int u = 1; u <= n; ++u) {
      start = u;
      initBFS();
      while (!finish) {
        findAugPath();
        if (!finish) subX_addY();
      }
      Enlarge();
    }
    int ans = 0, cnt = 0;
    for (int i = 1; i <= n; ++i) {
      if (c[i][l[i]] != oo) ans += c[i][l[i]], ++cnt;
      else l[i] = 0;
    }
    return { ans, cnt };
  }
};
```

Dominator Tree

```
// for a root r
// dom[u] is the lowest node that exists on all paths from r to u
struct DominatorTree {
  int T = 0, n;
  vector<vector<int>> rg, bucket, adj;
  vector<int> dsu, par, sdom, idom, dom, label, id, rev;

  DominatorTree(int n, int r, auto& adj):
    n(n++), rg(n), bucket(n), adj(adj), dsu(n),
    par(n), sdom(n), idom(n), dom(n), label(n), id(n), rev(n) {
dfs(r), build(); }

  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];
  }

  void dfs(int u) {
    id[u] = ++T, rev[T] = u;
    label[T] = sdom[T] = dsu[T] = T;
    for (int& w : adj[u]) {
      if (!id[w]) dfs(w), par[id[w]] = id[u];
      rg[id[w]].push_back(id[u]);
    }
  }

  void build() {
    for (int i = n; i; i--) {
      for (int& u : rg[i]) sdom[i] = min(sdom[i], sdom[find(u)]);
      if (i > 1) bucket[sdom[i]].push_back(i);
      for (int& w : bucket[i]) {
        int v = find(w);
        idom[w] = sdom[v] == sdom[w] ? sdom[w] : v;
      }
      if (i > 1) dsu[i] = par[i];
    }

    for (int i = 2; i <= n; i++)
      if (idom[i] != sdom[i]) idom[i] = idom[idom[i]];

    for (int u = 1; u <= n; ++u) dom[rev[u]] = rev[idom[u]];
  }
};
```

Fast Dinic

```
static const int oo = 2e15;

struct Edge {
   int u, v, flow = 0, cap = 0; // keep the order
   Edge(int u, int v): u(u), v(v) {}
   Edge(int u, int v, int c): u(u), v(v), cap(c) {}
   int rem() { return cap - flow; }
};

struct Dinic {
   int n, s, t, flow = 0;
   vector<int> lvl, ptr, q;
   vector<vector<int>> adj;
   vector<Edge> edges;

   Dinic(int n, int s, int t):
      n(++n), s(s), t(t), lvl(n), ptr(n), q(n), adj(n) {}

   void addEdge(int u, int v, int w = oo, int undir = 0) {
      adj[u].push_back(edges.size());
      edges.push_back(Edge(u, v, w));
      adj[v].push_back(edges.size());
      edges.push_back(Edge(v, u, w * undir));
   }

   int dfs(int u, int cf = oo) {
      if(u == t || !cf) return cf;

      for(; ptr[u] < adj[u].size(); ++ptr[u]) {
         int i = adj[u][ptr[u]];
         auto &[_, v, f, c] = edges[i];
         if(f == c || lvl[v] != lvl[u] + 1) continue;

         int p = dfs(v, min(cf, c - f));
         if(!p) continue;

         edges[i].flow += p;
         edges[i ^ 1].flow -= p;

         return p;
      }

      return 0;
   }
  void move() {
      q[0] = s;
      for (int L = 0; L <= 30; L++) do {
         lvl = ptr = vector<int>(n);
         int qi = 0, qe = lvl[s] = 1;
         while (qi < qe && !lvl[t]) {
            int u = q[qi++];
            for (int &i: adj[u]) {
               auto &[_, v, f, c] = edges[i];
               if (!lvl[v] && (c - f) >> (30 - L))
                  q[qe++] = v, lvl[v] = lvl[u] + 1;
            }
         }
         while (int f = dfs(s, oo)) flow += f;
      } while (lvl[t]);
   }
};
```

BipartiteDSU

```
struct BipartiteDSU {
   vector<int> sz,bipartite;
   vector<pair<int, int>>par;

   BipartiteDSU(int n) : par(n), sz(n, 1),bipartite(n) {
      for (int i = 0; i < n; ++i) {
         par[i] = {i,0};
      }
   }

   pair<int, int> find(int u) {
      if (u == par[u].fi)return {u, 0};
      int parity = par[u].se;
      par[u] = find(par[u].first);
      par[u].se ^= parity;
      return par[u];
   }

   bool same(int x, int y) { return find(x).first == find(y).first; }

   bool join(int u, int v) {
      pair<int,int>pu = find(u);
      pair<int,int>pv = find(v);
      u = pu.first;
      v = pv.first;
      int x = pu.second,y = pv.second;
      if (u == v) {
         if(x == y)
            bipartite[u] = false;
         return false;
      }
      if (sz[u] < sz[v])
         swap(u, v);
      par[v] = {u, x ^ y ^ 1};
      bipartite[u] &= bipartite[v];
      sz[u] += sz[v];
      return true;
   }

   int size(int x) { return sz[find(x).first]; }
};
```

MO Tree

```
/*
 * lc == -1 ==> means lc == u
 * if you need to handle weights on edges
 * put the weight on the child node
 * if(u == lc) del(u) before answering if vis[u]
 * and then add it again
 */

struct Query { int l, r, lc, ind; };

struct MOTree {
   static const int B = 21;
   int n, sq, l = 0, r = 0, timer = 0;
   ll curr = 0;
   vector<Query> qu;
   vector<int> v, in, out, vis, lvl;
   vector<array<int, B>> up;

   MOTree(vector<vector<int>> &adj) {
      n = adj.size();
      sq = sqrt(n) + 1;
      in.assign(n, {});
      out.assign(n, {});
      vis.assign(n, {});
      lvl.assign(n, {});
      up.assign(n, {});
      v.assign(n * 2, {});
      dfs(1, 0, 0, adj);
   }

   void dfs(int u, int p, int d, auto &adj) {
      lvl[u] = d;
      v[timer] = u;
      in[u] = timer++;
      up[u][0] = p;
      for(int i = 1; i < B; ++i)
         up[u][i] = up[up[u][i - 1]][i - 1];

      for(int &v: adj[u])
         if(v != p) dfs(v, u, d + 1, adj);

      v[timer] = u;
      out[u] = timer++;
   }

   int getLca(int u, int v) {
      if(lvl[u] > lvl[v]) swap(u, v);

      int k = lvl[v] - lvl[u];
      for(int i = B - 1; ~i; --i)
         if(k >> i & 1) v = up[v][i];

      for(int i = B - 1; ~i; --i)
         if(up[u][i] != up[v][i])
            u = up[u][i], v = up[v][i];

      return u == v? u : up[v][0];
   }

   void add(int u) {}
   void del(int u) {}

   void upd(int u) {
      vis[u]? del(u) : add(u);
      vis[u] ^= 1;
   }

   void move(int &lq, int &rq) {
      while (r < rq) upd(v[++r]);
      while (l < lq) upd(v[l++]);
      while (l > lq) upd(v[--l]);
      while (r > rq) upd(v[r--]);
   }

   void solve() {
      sort(qu.begin(), qu.end(),
         [&](auto &lf, auto &ri) {
            if (lf.l / sq == ri.l / sq) return lf.r < ri.r;
            return lf.l / sq < ri.l / sq;
         });

      l = qu[0].l, r = qu[0].l - 1;

      vector<ll> res(qu.size());
      for (auto &[lq, rq, lc, iq]: qu) {
         move(lq, rq);
         if(~lc) upd(lc);
         res[iq] = curr;
         if(~lc) upd(lc);
      }

      for (ll &i: res) cout << i << endl;
   }

   void addQuery(int u, int v) {
      if(in[u] > in[v]) swap(u, v);
      int lc = getLca(u, v);
      Query q; q.ind = qu.size(), q.lc = lc;
      if(lc == u) q.l = in[u], q.r = in[v], q.lc = -1;
      else q.l = out[u], q.r = in[v];
      qu.push_back(q);
   }
};
```

Arethmetic Segment Tree

```
struct Node {
   int val = 0, st = 0, dist = 0;
   bool isLazy = 0;

   Node() {}
   Node(int x): val(x) {}
   void add(int a, int d, int sz) {
      val += a * sz + sz * (sz - 1) / 2 * d;
      st += a, dist += d;
      isLazy = 1;
   }
};

#define lNode (x * 2 + 1)
#define rNode (x * 2 + 2)
#define md (lx + (rx - lx) / 2)

struct Sagara {
   int n;
   vector<Node> node;

   Sagara(int sz) {
      n = 1;
      while (n < sz) n *= 2;
      node.assign(n * 2, Node());
   }

   Node merge(Node &l, Node &r) {
      Node res;
      res.val = l.val + r.val;
      return res;
   }

   void propagate(int x, int lx, int rx) {
      if (rx - lx == 1 || !node[x].isLazy) return;
      node[lNode].add(node[x].st, node[x].dist, md - lx);
      node[rNode].add(node[x].st + (md - lx) * node[x].dist,
node[x].dist, rx - md);
      node[x].st = node[x].dist = node[x].isLazy = 0;
   }

   void update(int l, int r, int s, int d, int x, int lx, int rx) {
      propagate(x, lx, rx);

      if (lx >= r || rx <= l) return;
      if (lx >= l && rx <= r)
         return node[x].add(s + d * (lx - l), d, rx - lx);

      update(l, r, s, d, lNode, lx, md);
      update(l, r, s, d, rNode, md, rx);
      node[x] = merge(node[lNode], node[rNode]);
   }

   void update(int l, int r, int s, int d) { update(l, r, s, d, 0, 0,
n); }

   Node query(int l, int r, int x, int lx, int rx) {
      propagate(x, lx, rx);

      if (lx >= l && rx <= r) return node[x];
      if (lx >= r || rx <= l) return Node();

      Node L = query(l, r, lNode, lx, md);
      Node R = query(l, r, rNode, md, rx);

      return merge(L, R);
   }

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

Offline 2D BIT

```
template <typename T>
struct BIT2D {
  int n;
  vector<vector<int>> vals;
  vector<vector<T>> bit;

  int ind(const vector<int> &v, int x) {
    return upper_bound(begin(v), end(v), x) - begin(v) - 1;
  }

  // n: the limit of the first dimension
  // todo: all update operations you will make
  BIT2D(int n, vector<array<int, 2>> &todo): n(n+1),
vals(n+1), bit(n+1) {
    sort(begin(todo), end(todo),
      [](auto &a, auto &b) { return a[1] < b[1]; });

    for (int i = 0; i < n; i++) vals[i].push_back(-oo);
    for (auto [r, c] : todo)
      for (; r < n; r |= r + 1)
        if (vals[r].back() != c) vals[r].push_back(c);

    for (int i = 0; i < n; i++) bit[i].resize(vals[i].size());
  }

  void add(int r, int c, T val) {
    for (; r < n; r |= r + 1) {
      int i = ind(vals[r], c);
      for (; i < bit[r].size(); i |= i + 1) bit[r][i] += val;
    }
  }

  T query(int r, int c) {
    T ans = 0;
    for (; r >= 0; r = (r & r + 1) - 1) {
      int i = ind(vals[r], c);
      for (; i >= 0; i = (i & i + 1) - 1) ans += bit[r][i];
    }
    return ans;
  }

  T query(int r1, int c1, int r2, int c2) {
    return query(r2, c2) - query(r2, c1 - 1) - query(r1 - 1, c2)
+ query(r1 - 1, c1 - 1);
  }

  void reset() {
    for(auto &v: bit) for(auto &i: v) i = 0;
  }
};
```

PST

```
struct Node {
  Node *l, *r;
  int val = 0;

  Node(int val): l(NULL), r(NULL), val(val) {}
  Node(): l(NULL), r(NULL) {}
  Node(Node *l, Node *r): l(l), r(r) {
    if (l != NULL) val += l->val;
    if (r != NULL) val += r->val;
  }

  void addChild() {
    l = new Node(), r = new Node();
  }
};

struct PST {
  int n;
  PST(int n): n(n + 1) {}

  Node merge(Node x, Node y) {
    Node ret;
    ret.val = x.val + y.val;
    return ret;
  }

  Node *set(Node *v, int i, int val, int lx, int rx) {
    if (lx == rx) return new Node(val);
    int mid = (lx + rx) / 2;
    if(!v->l) v->addChild();
    if (i <= mid) return new Node(set(v->l, i, val, lx, mid), v->r);
    return new Node(v->l, set(v->r, i, val, mid + 1, rx));
  }
  Node *set(Node *v, int i, int val) { return set(v, i, val, 0, n - 1); }

  // [l, r] r is included
  Node query(Node *v, int l, int r, int lx, int rx) {
    if (l > rx || r < lx) return {};
    if (l <= lx && r >= rx) return *v;
    if(!v->l) v->addChild();
    int mid = (lx + rx) / 2;
    return merge(query(v->l, l, r, lx, mid), query(v->r, l, r, mid + 1,
rx));
  }

  Node query(Node *v, int l, int r) { return query(v, l, r, 0, n - 1); }
  int getKth(Node *a, Node *b, int k, int lx = 0, int rx = n - 1) {
    if (lx == rx) return lx;
    if(!a->l) a->addChild();
    if(!b->l) b->addChild();
    int rem = b->l->val - a->l->val;
    int mid = (lx + rx) / 2;
    if (rem >= k) return getKth(a->l, b->l, k, lx, mid);
    return getKth(a->r, b->r, k - rem, mid + 1, rx);
  }
};
```

Treap

```
mt19937
rng(chrono::steady_clock::now().time_since_epoch().cou
nt());
#define getrand(l, r) uniform_int_distribution<long long>(l,
r)(rng)

struct TreapNode {
  int sz = 1, rev = 0, key;
  ll p = getrand(1, 2e18);
  TreapNode *l = NULL, *r = NULL;
  TreapNode(int k): key(k) {};
};

using Treap = TreapNode*;

int size(Treap t) { return t? t->sz : 0; }

void prop(Treap t) {
  if(!t || !t->rev) return;
  swap(t->l, t->r);
  if(t->l) t->l->rev ^= 1;
  if(t->r) t->r->rev ^= 1;
  t->rev = 0;
}

Treap recalc(Treap t) {
  prop(t->l), prop(t->r);
  t->sz = size(t->l) + 1 + size(t->r);
  return t;
}

Treap merge(Treap l, Treap r) {
  if(!l || !r) return r ? r : l;
  prop(l), prop(r);

  if(l->p < r->p) {
    l->r = merge(l->r, r);
    return recalc(l);
  }

  r->l = merge(l, r->l);
  return recalc(r);
}
```

```
array<Treap, 2> split(Treap t, int pivot) {
  if(!t) return {NULL, NULL};
  prop(t);

  if(t->key > pivot) {
    auto [left, right] = split(t->l, pivot);
    t->l = right;
    return {left, recalc(t)};
  }

  auto [left, right] = split(t->r, pivot);
  t->r = left;
  return {recalc(t), right};
}

void swap(Treap &s, Treap &t, int l, int r) {
  auto [a, b] = split(s, r);
  auto [c, d] = split(a, l - 1);

  auto [e, f] = split(t, r);
  auto [i, j] = split(e, l - 1);

  s = merge(c, merge(j, b));
  t = merge(i, merge(d, f));
}

Treap answer(Treap t, int l, int r) {
  auto [a, b] = split(t, r);
  auto [c, d] = split(a, l - 1);
  cout << size(d) << endl;
  return merge(merge(c, d), b);
}

void print(Treap t) {
  if(!t) return;
  prop(t);
  print(t->l);
  // cout << t->val;
  print(t->r);
}
```

Implicit Treap

```
mt19937
rng(chrono::steady_clock::now().time_since_epoch().cou
nt());
#define getrand(l, r) uniform_int_distribution<long long>(l,
r)(rng)

struct TreapNode {
  int sz = 1, val, rev = 0;
  ll p = getrand(1, 2e18), sum;
  TreapNode *l = NULL, *r = NULL;
  TreapNode(int a): val(a), sum(a) {}
};

using Treap = TreapNode*;

int size(Treap t) { return t? t->sz : 0; }
ll sum(Treap t) { return t? t->sum : 0; }

void prop(Treap t) {
  if(!t || !t->rev) return;
  swap(t->l, t->r);
  if(t->l) t->l->rev ^= 1;
  if(t->r) t->r->rev ^= 1;
  t->rev = 0;
}

Treap recalc(Treap t) {
  prop(t->l), prop(t->r);
  t->sz = size(t->l) + 1 + size(t->r);
  t->sum = sum(t->l) + t->val + sum(t->r);
  return t;
}

Treap merge(Treap l, Treap r) {
  if(!l || !r) return r ? r : l;
  prop(l), prop(r);

  if(l->p < r->p) {
    l->r = merge(l->r, r);
    return recalc(l);
  }

  r->l = merge(l, r->l);
  return recalc(r);
}

array<Treap, 2> split(Treap t, int sz) {
  if(!t) return {NULL, NULL};
  prop(t);

  if(size(t->l) >= sz) {
    auto [left, right] = split(t->l, sz);
    t->l = right;
    return {left, recalc(t)};
  }

  auto [left, right] = split(t->r, sz - size(t->l) - 1);
  t->r = left;
  return {recalc(t), right};
}

Treap apply(Treap t, int l, int r) {
  auto [a, b] = split(t, r);
  auto [c, d] = split(a, l - 1);
  d->rev ^= 1;
  return merge(merge(c, d), b);
}

Treap answer(Treap t, int l, int r) {
  auto [a, b] = split(t, r);
  auto [c, d] = split(a, l - 1);
  cout << d->sum << endl;
  return merge(merge(c, d), b);
}

void print(Treap t) {
  if(!t) return;
  prop(t);
  print(t->l);
  cout << t->val;
  print(t->r);

}
```

SQRT

```
const int N = 5e5 + 10, SQ = 314, B = N / SQ + 1;
int blk[B], a[N];

void update(int i, int x) {
  int b = i / SQ;
  blk[b] -= a[i];
  a[i] = x;
  blk[b] += x;
}

int answer(int l, int r) {
  int res = 0;

  int L = l / SQ + 1, R = r / SQ;
  for (int b = L; b < R; ++b) res += blk[b];

  for (int i = l; i < min(r + 1, L * SQ); ++i) res += a[i];
  if (L <= R) for (int i = R * SQ; i <= r; ++i) res += a[i];

  return res;
}
```

Minimum Adjacent Swaps

```
// minimum adjacent swaps
// to convert a to b
int cost(vector<int> &a, vector<int> &b) {
  int n = a.size();
  map<int, deque<int>> pos;
  ordered_set<int> st;

  int res = 0;
  for (int i = 0; i < n; ++i) {
    pos[a[i]].push_back(i);
    st.insert(i);
  }

  for (int i = 0; i < n; ++i) {
    int idx = pos[b[i]].front();
    pos[b[i]].pop_front();
    res += st.order_of_key(idx);
    st.erase(idx);
  }

  return res;
}
```

GP Hashmap

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;

struct custom_hash {
  static uint64_t splitmix64(uint64_t x) {
    x += 0x9e3779b97f4a7c15;
    x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
    x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
    return x ^ (x >> 31);
  }

  size_t operator()(uint64_t x) const {
    static const uint64_t FIXED_RANDOM =
chrono::steady_clock::now().time_since_epoch().count();
    return splitmix64(x + FIXED_RANDOM);
  }
};

using GPMap = gp_hash_table<int, int, custom_hash>;
```

Wavelet Tree

```
const int MAXN = (int)3e5 + 9;
const int MAXV = (int)1e9 + 9; //maximum value of any
element in array

//array values can be negative too, use appropriate
minimum and maximum value
struct wavelet_tree {
 int lo, hi;
 wavelet_tree *l, *r;
 int *b, *c, bsz, csz; // c holds the prefix sum of elements

 wavelet_tree() {
  lo = 1;
  hi = 0;
  bsz = 0;
  csz = 0, l = NULL;
  r = NULL;
 }

 void init(int *from, int *to, int x, int y) {
  lo = x, hi = y;
  if(from >= to) return;
  int mid = (lo + hi) >> 1;
  auto f = [mid](int x) {
   return x <= mid;
  };
  b = (int*)malloc((to - from + 2) * sizeof(int));
  bsz = 0;
  b[bsz++] = 0;
  c = (int*)malloc((to - from + 2) * sizeof(int));
  csz = 0;
  c[csz++] = 0;
  for(auto it = from; it != to; it++) {
   b[bsz] = (b[bsz - 1] + f(*it));
   c[csz] = (c[csz - 1] + (*it));
   bsz++;
   csz++;
  }
  if(hi == lo) return;
  auto pivot = stable_partition(from, to, f);
  l = new wavelet_tree();
  l->init(from, pivot, lo, mid);
  r = new wavelet_tree();
  r->init(pivot, to, mid + 1, hi);
 }
 //kth smallest element in [l, r]
 //for array [1,2,1,3,5] 2nd smallest is 1 and 3rd smallest is
2
 int kth(int l, int r, int k) {
  if(l > r) return 0;
  if(lo == hi) return lo;
  int inLeft = b[r] - b[l - 1], lb = b[l - 1], rb = b[r];
  if(k <= inLeft) return this->l->kth(lb + 1, rb, k);
  return this->r->kth(l - lb, r - rb, k - inLeft);
 }
 //count of numbers in [l, r] Less than or equal to k
 int LTE(int l, int r, int k) {
  if(l > r || k < lo) return 0;
  if(hi <= k) return r - l + 1;
  int lb = b[l - 1], rb = b[r];
  return this->l->LTE(lb + 1, rb, k) + this->r->LTE(l - lb, r - rb,
k);
 }
 //count of numbers in [l, r] equal to k
 int count(int l, int r, int k) {
  if(l > r || k < lo || k > hi) return 0;
  if(lo == hi) return r - l + 1;
  int lb = b[l - 1], rb = b[r];
  int mid = (lo + hi) >> 1;
  if(k <= mid) return this->l->count(lb + 1, rb, k);
  return this->r->count(l - lb, r - rb, k);
 }
 //sum of numbers in [l ,r] less than or equal to k
 int sum(int l, int r, int k) {
  if(l > r or k < lo) return 0;
  if(hi <= k) return c[r] - c[l - 1];
  int lb = b[l - 1], rb = b[r];
  return this->l->sum(lb + 1, rb, k) + this->r->sum(l - lb, r -
rb, k);
 }
 ~wavelet_tree() {
  delete l;
  delete r;
 }
};
```

Manacher

```
struct Manacher {
  int n;
  vector<int> odd, even; // even[i]: center = (i - 1, i)
  Manacher(string& s): n(size(s)) {
    odd.resize(n), even.resize(n);

    for (int i = 0, l = 0, r = -1; i < n; i++) {
      odd[i] = i <= r ? min(odd[r - i + l], r - i + 1) : 1;
      while (i + odd[i] < n && i - odd[i] >= 0 && s[i + odd[i]] == s[i - odd[i]])
        odd[i]++;
      if (i + odd[i] - 1 > r) {
        r = i + odd[i] - 1;
        l = i - odd[i] + 1;
      }
    }

    for (int i = 0, l = 0, r = -1; i < n; i++) {
      even[i] = i <= r ? min(even[r - i + l + 1], r - i + 1) : 0;
      while (i + even[i] < n && i - even[i] - 1 >= 0 && s[i + even[i]] == s[i - even[i] - 1])
        even[i]++;
      if (i + even[i] - 1 > r) {
        r = i + even[i] - 1;
        l = i - even[i];
      }
    }
  }

  bool isPal(int l, int r) {
    if (l > r || r >= n) return false;
    int sz = r - l + 1, m = l + sz / 2;
    return (sz & 1 ? odd[m] * 2 - 1 : even[m] * 2) >= sz;
  }
};
```

Suffix Array

```
struct SuffixArray {
  int n;
  vector<int> suff, lcp, pos, lg;
  vector<array<int, 21>> table;
  SuffixArray(string& s, int lim = 256) {
    n = s.size() + 1;
    int k = 0, a, b;
    vector<int> c(s.begin(), s.end() + 1), tmp(n), frq(max(n, lim));
    c.back() = 0;
    suff = lcp = pos = tmp, iota(suff.begin(), suff.end(), 0);
    for (int j = 0, p = 0; p < n; j = max(1ll, j * 2), lim = p) {
      p = j, iota(tmp.begin(), tmp.end(), n - j);
      for (int i = 0; i < n; i++)
        if (suff[i] >= j) tmp[p++] = suff[i] - j;

      fill(frq.begin(), frq.end(), 0);
      for (int i = 0; i < n; i++) frq[c[i]]++;
      for (int i = 1; i < lim; i++) frq[i] += frq[i - 1];
      for (int i = n; i--;) suff[--frq[c[tmp[i]]]] = tmp[i];

      swap(c, tmp), p = 1, c[suff[0]] = 0;
      for (int i = 1; i < n; i++) {
        a = suff[i - 1], b = suff[i];
        c[b] = tmp[a] == tmp[b] && tmp[a + j] == tmp[b + j] ? p - 1 : p++;
      }
    }

    for (int i = 1; i < n; i++) pos[suff[i]] = i;
    for (int i = 0, j; i < n - 1; lcp[pos[i++]] = k)
      for (k && k--, j = suff[pos[i] - 1]; s[i + k] == s[j + k]; k++) {}
  }

  void preLcp() {
    lg.resize(n + 5);
    table.resize(n + 5);
    for (int i = 2; i < n + 5; ++i) lg[i] = lg[i / 2] + 1;
    for (int i = 0; i < n; ++i) table[i][0] = lcp[i];
    for (int j = 1; j <= lg[n]; ++j)
      for (int i = 0; i <= n - (1 << j); ++i)
        table[i][j] = min(table[i][j - 1], table[i + (1 << (j - 1))][j - 1]);
  }

  // pass the pos of the suffixes
  int queryLcp(int i, int j) {
    if (i == j) return n - suff[i] - 1;
    if (i > j) swap(i, j);
    i++;
    int len = lg[j - i + 1];
    return min(table[i][len], table[j - (1 << len) + 1][len]);
  }
};
```

Suffix Autometa

```
struct SuffixAutomaton {
  static const int A = 26;

  struct State {
    int len = 0, lnk = -1, cnt = 0, d = 1;
    int firstPos = 0, sum = 0; // if you need
    bool isClone = 0;
    array<int, A> nxt;
    State() { nxt.fill(-1); }
  };

  vector<State> t{{}};
  int lst = 0;

  SuffixAutomaton(string &s) {
    for(char &ch: s) insert(ch);
  }

  void insert(int ch) {
    int c = ch - 'a', me = t.size(), p = lst;
    t.push_back({});
    t[me].len = t[p].len + 1;
    t[me].firstPos = t[me].len - 1;
    t[me].cnt = 1;
    t[me].lnk = 0;
    lst = me;

    while(~p && t[p].nxt[c] == -1) {
      t[p].nxt[c] = me;
      p = t[p].lnk;
    }

    if(p == -1) return;

    int q = t[p].nxt[c];
    if(t[q].len == t[p].len + 1) {
      t[me].lnk = q;
      return;
    }

    int clone = t.size();
    t.push_back(t[q]);

    t[clone].len = t[p].len + 1;
    t[clone].isClone = 1;
    t[clone].cnt = 0;

    while (~p && t[p].nxt[c] == q) {
      t[p].nxt[c] = clone;
      p = t[p].lnk;
    }

    t[q].lnk = t[me].lnk = clone;
  }

  int move(int v, char &c) { return ~v? t[v].nxt[c - 'a'] : -1; }

  void move(int &v, int &len, char &c) {
    while(move(v, c) == -1) {
      v = t[v].lnk;
      if(v == -1) break;
      len = t[v].len;
    }

    if(~v) v = move(v, c), ++len;
    else v = 0;
  }

  int distinctSubstrings(){
    int ans = 0;
    for(int i = 1; i < t.size(); i++)
      ans += t[i].len - t[t[i].lnk].len;
    return ans;
  }

  int lenOfDistinctSubstrings() {
    int ans = 0;
    for(int i = 1; i < t.size(); i++) {
      int mn = t[t[i].lnk].len + 1, mx = t[i].len;
      ans += (mx - mn + 1) * (mx + mn) / 2;
    }
    return ans;
  }

  void preCount() {
    vector<array<int, 2>> v;
    for(int i = 0; i < t.size(); ++i)
      v.push_back({t[i].len, i});

    sort(v.rbegin(), v.rend());
    for(auto &[_, i]: v) {
      t[i].sum = t[i].cnt;
      if(i) t[t[i].lnk].cnt += t[i].cnt;
      for(int &to: t[i].nxt) if(~to) {
        t[i].d += t[to].d;
        t[i].sum += t[to].sum;
      }
    }
  }

  int countOcc(string &p) {
    int v = 0;
    for(int i = 0; i < p.size(); ++i) {
      v = move(v, p[i]);
      if(v == -1) return 0;
    }
    return t[v].cnt;
  }

  string kthDistinct(int k) {
    int v = 0;
    string ans;

    while(~v && k) {
      k--;
      int nxt = -1, add = 0;
      for(int i = 0; i < A; ++i) {
```

```
      int to = t[v].nxt[i];
      if(to == -1) continue;

      nxt = to, add = i;
      if(k - t[to].d < 0) break;
      k -= t[to].d;
    }

    if(nxt == -1) break;
    ans += char(add + 'a');
    v = nxt;
  }

  return k? "" : ans;
}

string kth(int k) {
  int v = 0;
  string ans;

  while(k > 0) {
    int nxt = -1, add = 0;
    for(int i = 0; i < A; ++i) {
      int to = t[v].nxt[i];
      if(to == -1) continue;

      nxt = to, add = i;
      if(k - t[to].sum <= 0) break;
      k -= t[to].sum;
    }

    if(nxt == -1) break;

    ans += char(add + 'a');
    v = nxt;
    k -= t[v].cnt;
  }

  return k > 0? "" : ans;
}

int LCS(string &s) {
  int v = 0, ans = 0, len = 0;
  for(char &ch: s) {
    move(v, len, ch);
    ans = max(ans, len);
  }
  return ans;
}
};
```

- Suffix Automaton can be represented as a DAG
- each node of this DAG is a state
- the edge between states (transition) means adding a character
- every path in the graph starting from node 0 represents a distinct substring in the string
- you can build it online by inserting every character one by one
- the state (node) with its Suffix Links (other type of edges) represents a tree
- each state corresponds to a set of **distinct** substrings whose lengths form a contiguous range.
- each substring is a suffix of the longest string (the string with length **len**
- the range of lengths can be calculated as follows:
    - // let st be the state you are calculating for:
    - // this state represent a set of distinct substrings of lengths [L, R]
    - int L = tree[st.lnk].len + 1;
    - int R = st.len
- **len:** length of the longest substring represented by this state
- **cnt:** the number of occurrences of this set of substrings
- **d:** total number of distinct substrings in the subgraph rooted at me

Hashing

```
mt19937
rng(chrono::steady_clock::now().time_since_epoch().count());
#define getrand(l, r) uniform_int_distribution<long long>(l, r)(rng)

const int c = 2;
typedef array<int, c> H;

struct Hash {
  int mod[c], base[c];
  vector<H> pw, inv;
  int N, st;

  Hash(int _n, int _s) {
    N = _n + 1;
    st = _s - 1;
    pw.resize(N);
    inv.resize(N);
    pre();
  }

  void gen(int k) {
    auto check = [](int x) {
      for (int i = 2; i * i <= x; ++i)
        if (!(x % i)) return false;
      return true;
    };
    mod[k] = getrand(1e8, 2e9);
    base[k] = getrand(30, 120);
    while (!check(mod[k]))--mod[k];
  }

  void pre() {
    for (int k = 0; k < c; ++k) {
      gen(k);
      pw[0][k] = inv[0][k] = 1;
      int invB = power(base[k], mod[k] - 2, k);
      for (int i = 1; i < N; ++i) {
        pw[i][k] = mul(pw[i - 1][k], base[k], k);
        inv[i][k] = mul(inv[i - 1][k], invB, k);
      }
    }
  }

  vector<H> build(string &s) {
    int n = s.size();
    vector<H> hash(n);
    for (int k = 0; k < c; ++k) {
      hash[0][k] = mul(s[0] - st, pw[0][k], k);
      for (int i = 1; i < n; ++i)
        hash[i][k] = add(hash[i - 1][k], mul((s[i] - st), pw[i][k], k), k);
    }
    return hash;
  }

  H getFullHash(string &s) {
    int n = s.size();
    H res;
    for (int k = 0; k < c; ++k) {
      res[k] = mul(s[0] - st, pw[0][k], k);
      for (int i = 1; i < n; ++i)
        res[k] = add(res[k], mul((s[i] - st), pw[i][k], k), k);
    }
    return res;
  }

  // hash value of the string s[l, r] both l, r included
  H getHash(int l, int r, vector<H> &hash) {
    H res;
    for (int k = 0; k < c; ++k) {
      res[k] = hash[r][k];
      if (l) {
        res[k] = add(res[k], -hash[l - 1][k], k);
        res[k] = mul(res[k], inv[l][k], k);
      }
    }
    return res;
  }

  void concat(H &l, H &r, int lSize) {
    for (int k = 0; k < c; ++k)
      l[k] = add(l[k], mul(r[k], pw[lSize][k], k), k);
  }

  int add(int a, int b, int &k) {
    ll ans = (ll) a + b;
    if (ans >= mod[k]) ans -= mod[k];
    if (ans < 0) ans += mod[k];
    return ans;
  }

  int mul(int a, int b, int &k) {
    return (1ll * a * b) % mod[k];
  }

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

Arethmitics on Strings

```
string add(string a, string b) {
   int len = max(a.size(), b.size());

   reverse(a.begin(), a.end());
   reverse(b.begin(), b.end());
   while (a.size() < len) a += '0';
   while (b.size() < len) b += '0';

   string ans;
   int carry = 0;
   for (int i = 0; i < len; ++i) {
      int current = carry + (a[i] - '0') + (b[i] - '0');
      ans.push_back(char(current % 10 + '0'));
      carry = current / 10;
   }

   if (carry > 0) ans += char(carry + '0');
   reverse(ans.begin(), ans.end());
   return ans;
}

string divideWithBase(const string& num, int base, int&
remain) {
   string quotient;
   quotient.reserve(num.size());
   int carry = 0;
   for (char c : num) {
      int d = c - '0';
      int current = carry * 10 + d;
      int q = current / base;
      carry = current % base;
      if (not quotient.empty() or q != 0) {
         quotient.push_back(char('0' + q));
      }
   }
   remain = carry;
   return quotient.empty() ? "0" : quotient;
}

string convertToBase(const string& decimal, int base) {
   if (decimal == "0") return "0";
   static auto digits = "0123456789";
   string n = decimal, result;
   while (n != "0") {
      int remain;
      n = divideWithBase(n, base, remain);
      result.push_back(digits[remain]);
   }
   reverse(result.begin(), result.end());
   return result;
}

string toDecimal(string& x, int base) {
   int f = 1;
   string ans = "0";
   reverse(x.begin(), x.end());
   for (int i = 0; i < x.size(); ++i, f *= base) {
      int t = 1LL * (x[i] - '0') * f;
      string y = to_string(t);
      ans = add(ans, y);
   }
   return ans;
}

string removeLeadingZeros(const string& num) {
   int i = 0, n = num.length();
   while (i < n && num[i] == '0') i++;
   if (i == n) return "0";
   return num.substr(i);
}

int compare(const string& a, const string& b) {
   string aClean = removeLeadingZeros(a);
   string bClean = removeLeadingZeros(b);
   if (aClean.length() > bClean.length()) return 1;
   if (aClean.length() < bClean.length()) return -1;
   for (int i = 0; i < aClean.length(); ++i) {
      if (aClean[i] > bClean[i]) return 1;
      if (aClean[i] < bClean[i]) return -1;
   }
   return 0;
}

string subtract(string a, string b) {
   a = removeLeadingZeros(a);
   b = removeLeadingZeros(b);
   bool minus = compare(a, b) < 0;
   if (minus) swap(a, b);

   int i = a.size() - 1;
   int j = b.size() - 1;
   int borrow = 0;
   string res;

   while (i >= 0 or j >= 0) {
      int digit1 = i >= 0? a[i--] - '0' : 0;
      int digit2 = j >= 0? b[j--] - '0' : 0;
      int diff = digit1 - digit2 - borrow;
      if (diff < 0) diff += 10, borrow = 1;
      else borrow = 0;
      res.push_back(diff + '0');
   }

   while (!res.empty() && res.back() == '0')
      res.pop_back();
```

```
    reverse(res.begin(), res.end());
    res = removeLeadingZeros(res);
    if(minus) res = '-' + res;
    return res.empty() ? "0" : res;
}

// works on decimals
string multiply(string& a, string& b) {
    if (a == "0" || b == "0") return "0";
    vector<int> c(a.size() + b.size());
    for (int i = a.size() - 1; i >= 0; --i) {
        for (int j = b.size() - 1; j >= 0; j--) {
            c[i + j + 1] += (a[i] - '0') * (b[j] - '0');
            c[i + j] += c[i + j + 1] / 10;
            c[i + j + 1] %= 10;
        }
    }

    int i = 0;
    string ans = "";
    while (c[i] == 0) i++;
    while (i < c.size()) ans += to_string(c[i++]);
    return ans;
}

string multiply(string num, char digit) {
    if (digit == '0') return "0";
    reverse(num.begin(), num.end());
    int carry = 0;
    string ans;
    for (char c : num) {
        int product = (c - '0') * (digit - '0') + carry;
        carry = product / 10;
        ans.push_back((product % 10) + '0');
    }
    if (carry > 0) ans.push_back(carry + '0');
    reverse(ans.begin(), ans.end());
    ans = removeLeadingZeros(ans);
    return ans.empty() ? "0" : ans;
}
```

```
pair<string, string> divide(string dividend, string divisor) {
    // divisor == "0"

    dividend = removeLeadingZeros(dividend);
    divisor = removeLeadingZeros(divisor);

    if (dividend == "0") return {"0", "0"};

    int cmp = compare(dividend, divisor);
    if (cmp < 0) return {"0", dividend};
    if (cmp == 0) return {"1", "0"};

    string quotient;
    string seg;
    int pos = 0;
    int len = dividend.size();

    while (pos < len) {
        seg += dividend[pos];
        ++pos;
        seg = removeLeadingZeros(seg);
        while (pos < len and compare(seg, divisor) < 0) {
            seg += dividend[pos];
            pos++;
            quotient.push_back('0');
            seg = removeLeadingZeros(seg);
        }

        if (compare(seg, divisor) < 0) {
            quotient.push_back('0');
            continue;
        }

        char q_digit = '0';
        for (char trial = '9'; trial >= '1'; --trial) {
            string product = multiply(divisor, trial);
            if (compare(product, seg) <= 0) {
                q_digit = trial;
                break;
            }
        }

        quotient.push_back(q_digit);
        string product = multiply(divisor, q_digit);
        seg = subtract(seg, product);
        seg = removeLeadingZeros(seg);
    }

    quotient = removeLeadingZeros(quotient);
    if (quotient.empty()) quotient = "0";

    return {quotient, seg.empty() or seg == "0" ? "0" : seg};
}
```

Misc

```
// Fractions up to N

vector<int> s;
for (int i = 0; i <= n; i++) {
  unsigned long long k = a / b;
  a -= b * k;
  a *= 10;
  s.push_back(k);
}

// kth bracket sequece O(n^2)
string kth_balanced(int n, int k) {
  vector d(2*n+1, vector<int>(n+1, 0));
  d[0][0] = 1;
  for (int i = 1; i <= 2*n; i++) {
    d[i][0] = d[i-1][1];
    for (int j = 1; j < n; j++)
      d[i][j] = d[i-1][j-1] + d[i-1][j+1];
    d[i][n] = d[i-1][n-1];
  }

  string ans;
  int depth = 0;
  for (int i = 0; i < 2*n; i++) {
    if (depth + 1 <= n && d[2*n-i-1][depth+1] >= k) {
      ans += '(';
      depth++;
    } else {
      ans += ')';
      if (depth + 1 <= n)
        k -= d[2*n-i-1][depth+1];
      depth--;
    }
  }
  return ans;
}

// O(n)
bool next_balanced_sequence(string & s) {
  int n = s.size();
  int depth = 0;
  for (int i = n - 1; i >= 0; i--) {
    if (s[i] == '(') depth--;
    else  depth++;
    if (s[i] == '(' && depth > 0) {
      depth--;
      int open = (n - i - 1 - depth) / 2;
      int close = n - i - 1 - open;
      string next = s.substr(0, i) + ')' + string(open, '(') +
string(close, ')');
      s.swap(next);
      return true;
    }
  }
  return false;
}
```

Discrete Log

```
//// a ^ x = b (% mod) --- O( sqrt(mod) )
bool DiscreteLog(int a, int b, int M, int &x) {
  a %= M; //// b %= mod ???
  int alpha = 1, add = 0, g;
  while ((g = __gcd(a, M)) > 1) {
    if (b == alpha)
      return x = add, true;
    if (b % g)return false;
    b /= g;
    M /= g;
    ++add;
    alpha = alpha * (a / g % M) % M;
  }

  int ph = phi(M);
  int inv = power(a, ph - 1, M);
  unordered_map<int, int> mp;
  int m = ceil(sqrt(M));
  int k = 1;
  for (int i = 0, j = alpha; i < m; ++i) {
    mp.emplace(j, i);
    j = j * a % M;
    k = k * inv % M;
  }
  x = -1;
  for (int i = 0, j = b; i < m; ++i) {
    auto it = mp.find(j);
    if (it != mp.end()) {
      int cur = it->second + i * m + add;
      if (x == -1 || cur < x)
        x = cur;
    }
    j = j * k % M;
  }
  return x != -1;
}
```

2D Hash

```
struct Hashing {
   vector<vector<int>> hs;
   vector<int> PWX, PWY;
   int n, m;
   static const int PX = 3731,  PY = 2999, mod = 998244353;
   Hashing() {}
   Hashing(vector<string>& s) {
      n = (int)s.size(), m = (int)s[0].size();
      hs.assign(n + 1, vector<int>(m + 1, 0));
      PWX.assign(n + 1, 1);
      PWY.assign(m + 1, 1);
      for (int i = 0; i < n; i++) PWX[i + 1] = 1LL * PWX[i] * PX % mod;
      for (int i = 0; i < m; i++) PWY[i + 1] = 1LL * PWY[i] * PY % mod;
      for (int i = 0; i < n; i++) {
         for (int j = 0; j < m; j++) {
            hs[i + 1][j + 1] = s[i][j] - 'a' + 1;
         }
      }
      for (int i = 0; i <= n; i++) {
         for (int j = 0; j < m; j++) {
            hs[i][j + 1] = (hs[i][j + 1] + 1LL * hs[i][j] * PY % mod) % mod;
         }
      }
      for (int i = 0; i < n; i++) {
         for (int j = 0; j <= m; j++) {
            hs[i + 1][j] = (hs[i + 1][j] + 1LL * hs[i][j] * PX % mod) % mod;
         }
      }
   }
   int get_hash(int x1, int y1, int x2, int y2) { // 1-indexed
      x1--;
      y1--;
      int dx = x2 - x1, dy = y2 - y1;
      return (1LL * (hs[x2][y2] - 1LL * hs[x2][y1] * PWY[dy] % mod +
mod) % mod -
            1LL * (hs[x1][y2] - 1LL * hs[x1][y1] * PWY[dy] % mod +
mod) % mod * PWX[dx] % mod + mod) % mod;
   }
};
```

Gamal Notes

Number of labelled rooted forests (n + 1)^(n-1)

Number of labeled trees with given degree sequence with size n
(n - 2)! / ((d1 - 1)! * (d2 - 1)! * (dn - 1)!)

Number of labeled graphs Gn = 2^(n*(n-1)/2)
Number of connected labeled graphs
Cn = Gn - 1/n * Sum(k * nCk * Ck * Gn-k) k = [1,n-1]

Number of labeled graphs with k components
D[n][k] = Sum(n-1Cs-1 * Cs * D[n-s][k-1]) s = [1,n]

Misere Nim where The player who removes the last stone loses the game if the piles are equal to 1 then decide with parity otherwise normal nim.

To play nim on a tree rooted at node 1 where you can remove any subtree other than 1:

```
int dfs(int u,int par){
   int XOR = 0;
   for(auto v:adj[u]){
      if(v == par)continue;
      int ret = dfs(v,u);
      XOR ^= ret + 1;
   }
   return XOR;
}
```

Number of moves to get back to where you started in a circle with size n and jump with size k is n / gcd(n,k).

a(x, y) = x + y, b(x, y) = x - y.
Manhattan Distance(x1, y1, x2, y2) = max(abs(a1 - a2), abs(b1 - b2))

In Alien's trick
if we will take answer when picked >= k : maximize picking
if we will take answer when picked <= k : minimize picking

Geometry

```
typedef ld T;
typedef complex<T> pt;
#define x real()
#define y imag()

const ld EPS = 1E-9;

int sgn(T val) {
   if(val > EPS) return 1;
   if(val < -EPS) return -1;
   return 0;
}

bool operator==(pt a, pt b) {return !sgn(a.x - b.x) && !sgn(a.y -
b.y);}
bool operator!=(pt a, pt b) {return !(a == b);}

T sq(pt p) {return p.x*p.x + p.y*p.y;}

ld abs(pt p) {return sqrtl(sq(p));}

pt perp(pt p) {return {-p.y, p.x};}

T dot(pt v, pt w) {return v.x*w.x + v.y*w.y;}

T cross(pt v, pt w) {return v.x*w.y - v.y*w.x;}

bool isPerp(pt v, pt w) {return dot(v,w) == 0;}
```

Transformations

```
// scale point p by a factor around c
pt scale(pt c, T factor, pt p) {
   return c + (p-c)*factor;
}

//To rotate point p by a certain angle ϕ around center c
pt rot(pt p, pt c, ld a) {
   pt v = p - c;
   return {c.x +v.x*cos(a) - v.y*sin(a), c.y + v.x*sin(a) +
v.y*cos(a)};
}

//point p has image fp, point q has image fq then what is
image of point r
pt linearTransfo(pt p, pt q, pt r, pt fp, pt fq) {
   pt pq = q-p, num{cross(pq, fq-fp), dot(pq, fq-fp)};
   return fp + pt{cross(r-p, num), dot(r-p, num)} / sq(pq);
}
```

Angles

```
//(AB X AC) --> relative to AB: if(C right) ret neg else if (C
left) pos
T orient(pt a, pt b, pt c) {return cross(b-a,c-a);}

//check p in between angle(bac) counter clockwise
bool inAngle(pt a, pt b, pt c, pt p) {
   if(a == b && b == c){ //point angle
      return a == p;
   }

   int abp = sgn(orient(a, b, p)), acp = sgn(orient(a, c, p)),
abc = sgn(orient(a, b, c));

   auto cmpProj = [&](pt p0, pt p1, pt v) {
      return dot(v,p0) <= dot(v,p1) + EPS;
   };

   if(!abc){ //ray angle
      pt v = b - a;
      if(cmpProj(c, b, v)) swap(b, c);
      //ray a -> b
      if(cmpProj(a, b, v)) return !abp && cmpProj(a, p, v);
   }

   if (abc < 0) swap(abp, acp);
   return (abp >= 0 && acp <= 0) ^ (abc < 0);
}

//Get angle between V, W
ld angle(pt v, pt w) {
   ld val = dot(v,w) / abs(v) / abs(w);
   val = min(val, 1.0L);
   val = max(val, -1.0L);
   return acosl(val);
}

//calc BAC angle
ld orientedAngle(pt a, pt b, pt c) {
   if (orient(a,b,c) >= 0)
      return angle(b-a, c-a);
   else
      return 2*M_PI - angle(b-a, c-a);
}


// amplitude travelled around point A, from P to Q
ld angleTravelled(pt a, pt p, pt q) {
   ld ampli = angle(p-a, q-a);
   if (orient(a,p,q) > 0) return ampli;
   else return -ampli;
}
```

Lines

```
struct line {
  pt v; T c;

// From direction vector v and offset c
  line(pt v, T c) : v(v), c(c) {}

// From equation ax+by=c
  line(T a, T b, T _c){
    v = {b,-a};
    c = _c;
  }
// From points P and Q
  line(pt p, pt q){
    v = q-p, c = cross(v,p);
  }

// - these work with T = int
  T side(pt p) {return cross(v,p)-c;}
  ld dist(pt p) {return abs(side(p)) / abs(v);}
  ld sqDist(pt p) {return side(p)*side(p) / (T)sq(v);}
  line perpThrough(pt p) {return {p, p + perp(v)};}
  bool cmpProj(pt p, pt q) {
    return dot(v,p) < dot(v,q);
  }
  line translate(pt t) {return {v, c + cross(v,t)};}

// - these require T = double
  line shiftLeft(double dist) {return {v, c + dist*abs(v)};}
  pt proj(pt p) {return p - perp(v)*side(p)/sq(v);}
  pt refl(pt p) {return p - perp(v) * (T)2.0 * side(p)/sq(v);}

  void print(){
    cout << -v.y << " x + " << v.x << " y = " << c << endl;
  }

  pt pointOnLine(){
    if(!sgn(v.x)) return {-c /v.y, 0};
    return {0, c / v.x};
  }
};

//Two lines Intersection 0 : no, -1 : infinite
int inter(line l1, line l2, pt &out) {
  T d = cross(l1.v, l2.v);
  if (!sgn(d)) { // parallel
    pt p = l1.pointOnLine();
    return sgn(l2.side(p)) ? 0 : -1;
  }
  out = (l2.v*l1.c - l1.v*l2.c) / d; // requires floating-point
coordinates
  return true;
}

//Bisector of Two lines (interior -> between v1 line, v2 line)
line bisector(line l1, line l2, bool interior) {
  assert(cross(l1.v, l2.v) != 0); // l1 and l2 cannot be parallel!
  T sign = interior ? 1 : -1;
  return {l2.v/(T)abs(l2.v) + l1.v/(T)abs(l1.v) * sign,
      l2.c/abs(l2.v) + l1.c/abs(l1.v) * sign};
}
```

Segments

```
bool inDisk(pt a, pt b, pt p) {
  return sgn(dot(a-p, b-p)) <= 0;
}

bool onSegment(pt a, pt b, pt p) {
  if(p == a || p == b) return 1;
  return !sgn(orient(a,b,p)) && inDisk(a,b,p);
}

bool properInter(pt a, pt b, pt c, pt d, pt &out) {
  T oa = orient(c,d,a),
      ob = orient(c,d,b),
      oc = orient(a,b,c),
      od = orient(a,b,d);
// Proper intersection exists iff opposite signs
  if (sgn(oa)*sgn(ob) < 0 && sgn(oc)*sgn(od) < 0) {
    out = (a*ob - b*oa) / (ob-oa);
    return true;
  }
  return false;
}

set<pair<ld,ld>> inters(pt a, pt b, pt c, pt d) {
  set<pair<ld,ld>> s;
  pt out;
  if (properInter(a,b,c,d,out)) return {make_pair(out.x, out.y)};
  if (onSegment(c,d,a)) s.insert(make_pair(a.x, a.y));
  if (onSegment(c,d,b)) s.insert(make_pair(b.x, b.y));
  if (onSegment(a,b,c)) s.insert(make_pair(c.x, c.y));
  if (onSegment(a,b,d)) s.insert(make_pair(d.x, d.y));
  return s;
}

ld segPoint(pt a, pt b, pt p) {
  if (a != b) {
    line l(a,b);
    if (l.cmpProj(a,p) && l.cmpProj(p,b)) // if closest to projection
      return l.dist(p); // output distance to line
  }
  return min(abs(p-a), abs(p-b)); // otherwise distance to A or B
}

ld segSeg(pt a, pt b, pt c, pt d) {
  pt dummy;
  if (properInter(a,b,c,d,dummy))
    return 0;
  return min({segPoint(a,b,c), segPoint(a,b,d),
       segPoint(c,d,a), segPoint(c,d,b)})
```

Rays

```
// does p lie on ray starting from s in direction of e
bool onRay(pt s, pt e, pt p) {
  if(p == s || p == e) return true;
  return !sgn(orient(p, s, e)) && sgn(dot(p - s, e - s)) >= 0;
}

//dist between p and ray starting from s in direction of e
ld rayPoint(pt s, pt e, pt p) {
  line l(s,e);
  if (l.cmpProj(s, p)) {
    return l.dist(p); // output distance to line
  }
  return abs(s - p);
}

pair<int, pt> rayRayInter(pt s1, pt e1, pt s2, pt e2) {
  // first = 0 no intersection
  // first = 1 intersection
  // first = -1 infinite intersection
  pt p;
  int cnt = inter(line(s1, e1), line(s2, e2), p);
  if (cnt == 0) return {cnt, p};
  else if (cnt == 1) {
    if (onRay(s1, e1, p) && onRay(s2, e2, p))
      return {cnt, p};
    else
      return {0, {0, 0}};
  } else {
    if (onRay(s2, e2, s1) || onRay(s1, e1, s2))
      return {-1, onRay(s2, e2, s1) ? s1 : s2};
    else
      return {0, {0, 0}};
  }
}

ld rayRayDist(pt s1, pt e1, pt s2, pt e2) {
  if (rayRayInter(s1, e1, s2, e2).first != 0)
    return 0;
  ld ret = min(rayPoint(s2, e2, s1), rayPoint(s1, e1, s2));
  return ret;
}
```

Polygons

```
bool isConvex(vector<pt> p) {
  bool hasPos=false, hasNeg=false;
  for (int i=0, n=p.size(); i<n; i++) {
    int o = sgn(orient(p[i], p[(i+1)%n], p[(i+2)%n]));
    if (o > 0) hasPos = true;
    if (o < 0) hasNeg = true;
  }
  return !(hasPos && hasNeg);
}

ld areaTriangle(pt a, pt b, pt c) {
  return abs(cross(b-a, c-a)) / 2.0;
}

ld areaPolygon(vector<pt> p) {
  ld area = 0.0;
  for (int i = 0, n = p.size(); i < n; i++) {
    area += cross(p[i], p[(i+1)%n]);
  }
  return abs(area) / 2.0;
}

// true if P at least as high as A
bool above(pt a, pt p) {
  return p.y >= a.y;
}

// check if [PQ] crosses ray from A
bool crossesRay(pt a, pt p, pt q) {
  return (above(a,q) - above(a,p)) * sgn(orient(a,p,q)) > 0;
}

// if strict, returns false when A is on the boundary
bool inPolygon(vector<pt> p, pt a, bool strict = true) {
  int numCrossings = 0;
  for (int i = 0, n = p.size(); i < n; i++) {
    if (onSegment(p[i], p[(i+1)%n], a))
      return !strict;
    numCrossings += crossesRay(a, p[i], p[(i+1)%n]);
  }
  return numCrossings & 1; // inside if odd number of
crossings
}
```

Circles

```
pair<pt, T> circumCircle(pt a, pt b, pt c) {
   b = b-a, c = c-a; // consider coordinates relative to A
   assert(cross(b,c) != 0); // no circumcircle if A,B,C aligned
   return {a + perp(b*sq(c) - c*sq(b))/cross(b,c)/(T)2,
abs(perp(b*sq(c) - c*sq(b))/cross(b,c)/(T)2)};
}

int circleLine(pt o, ld r, line l, pair<pt,pt> &out) {
   ld h2 = r*r - l.sqDist(o);
   if (h2 >= 0) { // the line touches the circle
      pt p = l.proj(o); // point P
      pt h = l.v* (T)(sqrtl(h2)/abs(l.v)); // vector parallel to l, of
length h
      out = {p-h, p+h};
   }
   return 1 + sgn(h2);
}

int circleCircle(pt o1, T r1, pt o2, T r2, pair<pt,pt> &out) {
   pt d=o2-o1; T d2=sq(d);
   if (d2 == 0) {assert(r1 != r2); return 0;} // concentric
   T pd = (d2 + r1*r1 - r2*r2)/2; // = |O_1P| * d
   T h2 = r1*r1 - pd*pd/d2; // = h^2
   if (h2 >= 0) {
      pt p = o1 + d*pd/d2, h = perp(d)*sqrtl(h2/d2);
      out = {p-h, p+h};
   }
   return 1 + sgn(h2);
}

int tangents(pt o1, T r1, pt o2, T r2, bool inner,
vector<pair<pt,pt>> &out) {
   if (inner) r2 = -r2;
   pt d = o2-o1;
   T dr = r1-r2, d2 = sq(d), h2 = d2-dr*dr;
   if (!sgn(d2) || h2 < 0) {assert(h2 != 0); return 0;} //inside
each other
   for (T sign : {-1,1}) {
      pt v = (d*dr + perp(d)*sqrtl(h2)*sign)/d2;
      out.push_back({o1 + v*r1, o2 + v*r2});
   }
   return 1 + (h2 > 0);
}

ld area2Circles(pt o1, T r1, pt o2, T r2){
   int d2 = sq(o1 - o2);

   // no intersection or tangent from outside
   if((r1 + r2) * (r1 + r2) <= d2) return 0;

   // concentric
   if(abs(r1 - r2) * abs(r1 - r2) >= d2){
      return min(r1, r2) * min(r1, r2) * M_PI;
   }

   ld theta1 = 2 * acosl((d2 + r1 * r1 - r2 * r2) / (2.0 * sqrtl(d2) * r1));
   ld theta2 = 2 * acosl((d2 + r2 * r2 - r1 * r1) / (2.0 * sqrtl(d2) * r2));

   return (r1 * r1 * (theta1 - sinl(theta1)) + r2 * r2 * (theta2 -
sinl(theta2))) / 2.0 ;
}
```

Minimum Enclosing Circle

```
// given n points, find the minimum enclosing circle of the
points
// call convex_hull() before this for faster solution
// expected O(n)
pair<pt, ld> minimum_enclosing_circle(vector<pt> &p) {
   random_shuffle(p.begin(), p.end());
   int n = p.size();
   pt c = p[0];
   ld r = 0;
   for (int i = 1; i < n; i++) {
      if (sgn(abs(c - p[i]) - r) > 0) {
         c = p[i], r = 0;
         for (int j = 0; j < i; j++) {
            if (sgn(abs(c - p[j]) - r) > 0) {
               c = (p[i] + p[j]) / (T)2.0, r = abs(p[i] - p[j]) / 2;
               for (int k = 0; k < j; k++) {
                  if (sgn(abs(c - p[k]) - r) > 0) {
                     auto [curC, curR] = circumCircle(p[i], p[j],
p[k]);
                     c = curC, r = curR;
                  }
               }
            }
         }
      }
   }
   return {c, r};
}
```

Maximum Circle Cover

```
// find a circle of radius r that contains as many points as possible
// O(n^2 log n);
pair<int, pt> maximum_circle_cover(vector<pt> p, ld r) {
  int n = p.size();
  int ans = 0;
  int id = 0; ld th = 0;
  for (int i = 0; i < n; ++i) {
    // maximum circle cover when the circle goes through this point
    vector<pair<ld, int>> events = {{-M_PI, +1}, {M_PI, -1}};
    for (int j = 0; j < n; ++j) {
      if (j == i) continue;
      ld d = abs(p[i] - p[j]);
      if (d > r * 2) continue;
      ld dir = arg(p[j] - p[i]);
      ld ang = acos(d / 2 / r);
      ld st = dir - ang, ed = dir + ang;
      if (st > M_PI) st -= M_PI * 2;
      if (st <= -M_PI) st += M_PI * 2;
      if (ed > M_PI) ed -= M_PI * 2;
      if (ed <= -M_PI) ed += M_PI * 2;
      events.push_back({st - EPS, +1}); // take care of precisions!
      events.push_back({ed, -1});
      if (st > ed) {
        events.push_back({-M_PI, +1});
        events.push_back({+M_PI, -1});
      }
    }
    sort(events.begin(), events.end());
    int cnt = 0;
    for (auto &&e: events) {
      cnt += e.second;
      if (cnt > ans) {
        ans = cnt;
        id = i; th = e.first;
      }
    }
  }
  pt w = pt(p[id].x + r * cosl(th), p[id].y + r * sinl(th));
  return {ans, w};
}
```

Biggest Circle in convex

```
pair<pt, ld> biggestCircleInConvex(vector<pt> & p){

  auto get = [&](ld r){
    vector<Halfplane> tot;
    for (int i = 0, n = p.size(); i < n; ++i) {
      pt v = perp(p[(i + 1) % n] - p[i]);
      v /= abs(v);
      v *= r;
      tot.push_back(Halfplane(p[i] + v, p[(i + 1)%n] + v));
    }
    return hp_intersect(tot);
  };

  ld l = 0, r = 1e9, mid;
  for (int i = 0; i < 200; ++i) {
    mid = (l + r)/2;
    if(get(mid).size()) l = mid;
    else r = mid;
  }
  return {get(l)[0], mid};
}
```

Frequency Lines

```
// normalize (a,b,c) so that gcd(a,b,c)=1 and
// (a>0) or (a==0 && b>0) or (a==0 && b==0 && c>0)
  tuple<ll,ll,ll> normalize_line(ll a, ll b, ll c) {
    // make sign of (a,b) consistent
    if (a < 0 || (a == 0 && b < 0) || (a == 0 && b == 0 && c < 0)) {
      a = -a; b = -b; c = -c;
    }
    ll g = gcd(abs(a), gcd(abs(b), abs(c)));
    if (g > 1) { a /= g; b /= g; c /= g; }
    return {a,b,c};
  }
  map<tuple<ll,ll,ll>, set<int>> lines;
  for(int i = 0; i < n; i++){
    for(int j = i+1; j < n; j++){
      auto [x1, y1] = p[i];
      auto [x2, y2] = p[j];
      ll a =  y2 - y1;
      ll b = -(x2 - x1);
      ll c =  x2 * y1 - x1 * y2;
      auto key = normalize_line(a,b,c);
      lines[key].insert(i);
      lines[key].insert(j);
    }
  }
  for (auto [line, points] : lines) {
    int f = points.size();
  }
```

Polygon Clipping

```
// Checks if a point p is inside the edge (a, b)
bool inside(pt p, pt a, pt b) {
   return sgn(orient(a, b, p)) >= 0; // Change to > for strict
inside
}

// Computes intersection of line passing by a, b with the
line passing by c, d
pt intersection(pt a, pt b, pt c, pt d) {
   T oa = orient(c,d,a), ob = orient(c,d,b);
   return (a*ob - b*oa) / (ob-oa);
}

// Sutherland–Hodgman algorithm for polygon clipping
vector<pt> clipPolygon(vector<pt> poly, pt a, pt b) {
   vector<pt> newPoly;
   int n = poly.size();
   for (int i = 0; i < n; i++) {
      pt cur = poly[i], prev = poly[(i + n - 1) % n];
      bool curInside = inside(cur, a, b);
      bool prevInside = inside(prev, a, b);
      if (curInside) {
         if (!prevInside)
newPoly.push_back(intersection(prev, cur, a, b));
         newPoly.push_back(cur);
      } else if (prevInside) {
         newPoly.push_back(intersection(prev, cur, a, b));
      }
   }
   return newPoly;
}
```

Centroid of polygon

```
// centroid of a (possibly non-convex) polygon,
// coordinates are listed in a clockwise or
// counterclockwise fashion.
// the "center of gravity" or "center of mass".
pt centroid(vector<pt> &p) {
   int n = p.size(); pt c(0, 0);
   ld sum = 0;
   for (int i = 0; i < n; i++) sum += cross(p[i], p[(i + 1) % n]);
   ld scale = 3.0 * sum;
   for (int i = 0; i < n; i++) {
      int j = (i + 1) % n;
      c = c + (p[i] + p[j]) * cross(p[i], p[j]);
   }
   return c / scale;
}
```

Convex Hull

```
bool cw(pt a, pt b, pt c, bool include_collinear) {
   int o = sgn(orient(a, b, c));
   return o < 0 || (include_collinear && o == 0);
}

bool collinear(pt a, pt b, pt c) { return sgn(orient(a, b, c)) ==
0; }

void convex_hull(vector<pt>& a, bool include_collinear =
false) {
   pt p0 = *min_element(a.begin(), a.end(), [](pt a, pt b) {
      return make_pair(a.y, a.x) < make_pair(b.y, b.x);
   });
   sort(a.begin(), a.end(), [&p0](const pt& a, const pt& b) {
      int o = sgn(orient(p0, a, b));
      if (o == 0)
         return (p0.x-a.x)*(p0.x-a.x) + (p0.y-a.y)*(p0.y-a.y)
            < (p0.x-b.x)*(p0.x-b.x) + (p0.y-b.y)*(p0.y-b.y);
      return o < 0;
   });
   if (include_collinear) {
      int i = (int)a.size()-1;
      while (i >= 0 && collinear(p0, a[i], a.back())) i--;
      reverse(a.begin()+i+1, a.end());
   }

   vector<pt> st;
   for (int i = 0; i < (int)a.size(); i++) {
      while (st.size() > 1 && !cw(st[st.size()-2], st.back(), a[i],
include_collinear))
         st.pop_back();
      if(st.empty() || a[i] != st.back())
         st.push_back(a[i]);
   }

   if (include_collinear == false && st.size() == 2 && st[0] ==
st[1])
      st.pop_back();

   a = st;
}
```

Minkowski Sum

```
void reorder_polygon(vector<pt> & P){
  size_t pos = 0;
  for(size_t i = 1; i < P.size(); i++){
    if(P[i].y < P[pos].y || (P[i].y == P[pos].y && P[i].x <
P[pos].x))
      pos = i;
  }
  rotate(P.begin(), P.begin() + pos, P.end());
}

//p must be counter clockwise

vector<pt> minkowski(vector<pt> P, vector<pt> Q){
  // the first vertex must be the lowest
  reorder_polygon(P);
  reorder_polygon(Q);
  // we must ensure cyclic indexing
  P.push_back(P[0]);
  P.push_back(P[1]);
  Q.push_back(Q[0]);
  Q.push_back(Q[1]);
  // main part
  vector<pt> result;
  size_t i = 0, j = 0;
  while(i < P.size() - 2 || j < Q.size() - 2){
    result.push_back(P[i] + Q[j]);
    auto c = cross(P[i + 1] - P[i], Q[j + 1] - Q[j]);
    if(c >= 0 && i < P.size() - 2)
      ++i;
    if(c <= 0 && j < Q.size() - 2)
      ++j;
  }
  return result;
}
```

Rotating Calipers

```
vector<pair<int, int>> all_anti_podal(int n, vector<pt> &p) {
  vector<pair<int, int>> result;

  auto nx = [&](int i){return (i+1)%n;};
  auto pv = [&](int i){return (i-1+n)%n;};

  // parallel edges should't be visited twice
  vector<bool> vis(n, false);

  for (int p1 = 0, p2 = 0; p1 < n; ++p1) {
    // the edge that we are going to consider in this
iteration
    // the datatype is Point, but it acts as a vector
    pt base = p[nx(p1)] - p[p1];
    // the last condition makes sure that the cross
products don't have the same sign
    while (p2 == p1 || p2 == nx(p1) || sgn(cross(base,
p[nx(p2)] - p[p2])) == sgn(cross(base, p[p2] - p[pv(p2)]))) {
      p2 = nx(p2);
    }

    if (vis[p1]) continue;
    vis[p1] = true;

    // seg p1 nx(p1) -> p2
    result.push_back({p1, p2});
    result.push_back({nx(p1), p2});

    // if both edges from p1 and p2 are parallel to each
other
    if (sgn(cross(base, p[nx(p2)] - p[p2])) == 0) {
      //seg(p1, nx(p1)) -> seg(p2, nx(p2))
      result.push_back({p1, nx(p2)});
      result.push_back({nx(p1), nx(p2)});
      vis[p2] = true;
    }
  }

  return result;
}

// maximum distance from a convex polygon to another
convex polygon
ld maximum_dist_from_polygon_to_polygon(vector<pt>
&u, vector<pt> &v){ //O(n)
  int n = (int)u.size(), m = (int)v.size();
  ld ans = 0;
  if (n < 3 || m < 3) {
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) ans = max(ans, sq(u[i] - v[j]));
    }
    return sqrtl(ans);
  }
  if (u[0].x > v[0].x) swap(n, m), swap(u, v);
  int i = 0, j = 0, step = n + m + 10;
  while (j + 1 < m && v[j].x < v[j + 1].x) j++ ;
  while (step--) {
    if (cross(u[(i + 1)%n] - u[i], v[(j + 1)%m] - v[j]) >= 0) j = (j +
1) % m;
    else i = (i + 1) % n;
    ans = max(ans, sq(u[i] - v[j]));
  }
  return sqrtl(ans);
}
```

Half Plane

```
// Basic half-plane struct.
struct Halfplane {
  // 'p' is a passing point of the line and 'pq' is the direction vector of the line.
  pt p, pq;
  long double angle;

  Halfplane() {}
  Halfplane(const pt& a, const pt& b) : p(a), pq(b - a) {
    angle = atan2l(pq.y, pq.x);
  }

  // Check if point 'r' is outside this half-plane.
  // Every half-plane allows the region to the LEFT of its line.
  bool out(const pt& r) {
    return cross(pq, r - p) < -EPS;
  }

  // Comparator for sorting.
  bool operator < (const Halfplane& e) const {
    return angle < e.angle;
  }

  // Intersection point of the lines of two half-planes. It is assumed they're never parallel.
  friend pt inter(const Halfplane& s, const Halfplane& t) {
    long double alpha = cross((t.p - s.p), t.pq) / cross(s.pq, t.pq);
    return s.p + (s.pq * alpha);
  }
};


vector<pt> hp_intersect(vector<Halfplane>& H) {
  const int inf = 1e9;
  pt box[4] = {  // Bounding box in CCW order
      pt(inf, inf),
      pt(-inf, inf),
      pt(-inf, -inf),
      pt(inf, -inf)
  };

  for(int i = 0; i<4; i++) { // Add bounding box half-planes.
    Halfplane aux(box[i], box[(i+1) % 4]);
    H.push_back(aux);
  }

  // Sort by angle and start algorithm
  sort(H.begin(), H.end());
  deque<Halfplane> dq;
  int len = 0;
  for(int i = 0; i < int(H.size()); i++) {

    // Remove from the back
    while (len > 1 && H[i].out(inter(dq[len-1], dq[len-2]))) {
      dq.pop_back();
      --len;
    }

    // Remove from the front
    while (len > 1 && H[i].out(inter(dq[0], dq[1]))) {
      dq.pop_front();
      --len;
    }

    // Special case check: Parallel half-planes
    if (len > 0 && fabsl(cross(H[i].pq, dq[len-1].pq)) < EPS) {
      // Opposite parallel half-planes that ended up checked against each other.
      if (dot(H[i].pq, dq[len-1].pq) < 0.0)
        return vector<pt>();

      // Same direction half-plane: keep only the leftmost half-plane.
      if (H[i].out(dq[len-1].p)) {
        dq.pop_back();
        --len;
      }
      else continue;
    }

    // Add new half-plane
    dq.push_back(H[i]);
    ++len;
  }

  // Final cleanup: Check half-planes at the front against the back and vice-versa
  while (len > 2 && dq[0].out(inter(dq[len-1], dq[len-2]))) {
    dq.pop_back();
    --len;
  }

  while (len > 2 && dq[len-1].out(inter(dq[0], dq[1]))) {
    dq.pop_front();
    --len;
  }

  // Report empty intersection if necessary
  if (len < 3) return vector<pt>();

  // Reconstruct the convex polygon from the remaining half-planes.
  vector<pt> ret(len);
  for(int i = 0; i+1 < len; i++) {
    ret[i] = inter(dq[i], dq[i+1]);
  }
  ret.back() = inter(dq[len-1], dq[0]);
  return ret;
}
```

Circle Union

```
// O(n^2 log n)
typedef struct CircleUnion {
  int n;
  ld xs[2020], ys[2020], r[2020];
  int covered[2020];
  vector<pair<ld, ld> > seg, cover;
  ld arc, pol;
  inline ld sq(const ld val) {return val * val;}
  inline ld dist(ld x1, ld y1, ld x2, ld y2) {return sqrt(sq(x1 - x2) +
sq(y1 - y2));}
  inline ld angle(ld A, ld B, ld C) {
    ld val = (sq(A) + sq(B) - sq(C)) / (2 * A * B);
    if (val < -1) val = -1;
    if (val > +1) val = +1;
    return acos(val);
  }
  CircleUnion() {
    n = 0;
    seg.clear(), cover.clear();
    arc = pol = 0;
  }
  void init() {
    n = 0;
    seg.clear(), cover.clear();
    arc = pol = 0;
  }
  void add(ld xx, ld yy, ld rr) {
    xs[n] = xx, ys[n] = yy, r[n] = rr, covered[n] = 0, n++;
  }
  void getarea(int i, ld lef, ld rig) {
    arc += 0.5 * r[i] * r[i] * (rig - lef - sin(rig - lef));
    ld x1 = xs[i] + r[i] * cos(lef), y1 = ys[i] + r[i] * sin(lef);
    ld x2 = xs[i] + r[i] * cos(rig), y2 = ys[i] + r[i] * sin(rig);
    pol += x1 * y2 - x2 * y1;
  }
  ld solve() {
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < i; j++) {
        if (!sgn(xs[i] - xs[j]) && !sgn(ys[i] - ys[j]) && !sgn(r[i] - r[j])) {
          r[i] = 0.0;
          break;
        }
      }
    }
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        if (i != j && sgn(r[j] - r[i]) >= 0 && sgn(dist(xs[i], ys[i], xs[j],
ys[j]) - (r[j] - r[i])) <= 0) {
          covered[i] = 1;
          break;
        }
      }
    }
    for (int i = 0; i < n; i++) {
      if (sgn(r[i]) && !covered[i]) {
        seg.clear();
        for (int j = 0; j < n; j++) {
          if (i != j) {
            ld d = dist(xs[i], ys[i], xs[j], ys[j]);
            if (sgn(d - (r[j] + r[i])) >= 0 || sgn(d - abs(r[j] - r[i])) <= 0) {
              continue;
            }
            ld alpha = atan2(ys[j] - ys[i], xs[j] - xs[i]);
            ld beta = angle(r[i], d, r[j]);
            pair<ld, ld> tmp(alpha - beta, alpha + beta);
            if (sgn(tmp.first) <= 0 && sgn(tmp.second) <= 0) {
              seg.push_back(pair<ld, ld>(2 * M_PI + tmp.first, 2 *
M_PI + tmp.second));
            }
            else if (sgn(tmp.first) < 0) {
              seg.push_back(pair<ld, ld>(2 * M_PI + tmp.first, 2 *
M_PI));
              seg.push_back(pair<ld, ld>(0, tmp.second));
            }
            else {
              seg.push_back(tmp);
            }
          }
        }
        sort(seg.begin(), seg.end());
        ld rig = 0;
        for (vector<pair<ld, ld> >::iterator iter = seg.begin(); iter !=
seg.end(); iter++) {
          if (sgn(rig - iter->first) >= 0) {
            rig = max(rig, iter->second);
          }
          else {
            getarea(i, rig, iter->first);
            rig = iter->second;
          }
        }
        if (!sgn(rig)) {
          arc += r[i] * r[i] * M_PI;
        }
        else {
          getarea(i, rig, 2 * M_PI);
        }
      }
    }
    return pol / 2.0 + arc;
  }
} CU;

void solve() {
  int n;
  while(cin >> n && n){
    CU c;
    c.init();
    for (int i = 0; i < n; ++i) {
      ld xx, yy, r; cin >> xx >> yy >> r;
      c.add(xx, yy, r);
    }

    cout << c.solve() << endl;
  }
};
```

Integer Angles

```
bool half(pt p) {
   return p.y > 0 || (p.y == 0 && p.x < 0);
}

bool half_pos(pt p) {
   return p.y > 0 || (p.y == 0 && p.x > 0);
}

struct angle_t {
   pt d;
   angle_t(){
      d = {0,0};
   }
   angle_t(pt _p){
      d = _p;
   }
};

bool operator <(angle_t a, angle_t b) {
   return make_tuple(!half_pos(a.d), 0) <
         make_tuple(!half_pos(b.d), cross(a.d,b.d));
}

angle_t operator -(angle_t a, angle_t b){
   angle_t ret = angle_t({a.d.x*b.d.x + a.d.y*b.d.y, a.d.y *
b.d.x - b.d.y*a.d.x});
   return ret;
}

angle_t minimise(angle_t ang){
   if(!half(ang.d)){
      ang = angle_t({1 ,0}) - ang;
   }
   return ang;
}

bool operator ==(angle_t a, angle_t b) {
   return sgn(a.d.x) == sgn(b.d.x) && sgn(a.d.y) ==
sgn(b.d.y) && a.d.x * b.d.y == a.d.y * b.d.x;
}
```

3D-Geometry

```
struct p3 {
   T x, y, z;

   p3 operator+(p3 p) { return {x + p.x, y + p.y, z + p.z}; }
   p3 operator-(p3 p) { return {x - p.x, y - p.y, z - p.z}; }
   p3 operator*(T d) { return {x * d, y * d, z * d}; }
   p3 operator/(T d) { return {x / d, y / d, z / d}; }

   bool operator==(p3 p) { return !sgn(x - p.x) && !sgn(y - p.y)
&& !sgn(z - p.z); }
   bool operator!=(p3 p) { return !operator==(p); }
};

p3 zero = {0, 0, 0};

// Dot product: returns v · w
T operator|(p3 v, p3 w) { return v.x * w.x + v.y * w.y + v.z *
w.z; }

// Squared length: returns |v|^2
T sq(p3 v) { return v | v; }

// Euclidean norm: returns |v|
ld abs(p3 v) { return sqrtl(sq(v)); }

// Unit vector: returns v normalized
p3 unit(p3 v) { return v / abs(v); }

// Angle between vectors v and w (in radians)
ld angle(p3 v, p3 w) {
   ld cosTheta = (v | w) / abs(v) / abs(w);
   return acosl(max(-1.0L, min(1.0L, cosTheta)));
}

// Cross product: returns v × w
p3 operator*(p3 v, p3 w) {
   return {v.y * w.z - v.z * w.y,
         v.z * w.x - v.x * w.z,
         v.x * w.y - v.y * w.x};
}

// Signed orientation test in 3D: positive if s is above plane
defined by p,q,r
T orient(p3 p, p3 q, p3 r, p3 s) { return (q - p) * (r - p) | (s - p);
}

// Orientation by given normal n: orientation of triangle pqr
along n
T orientByNormal(p3 p, p3 q, p3 r, p3 n) { return (q - p) * (r -
p) | n; }
```

```cpp
// Plane defined by normal n and offset d
struct plane {
  p3 n; // normal vector
  T d;  // plane offset
  // n · p = d

  // Construct from normal n and offset d
  plane(p3 n, T d) : n(n), d(d) {}

  // Construct from normal n and point p on plane
  plane(p3 n, p3 p) : n(n), d(n | p) {}

  // Construct from three points P,Q,R
  plane(p3 p, p3 q, p3 r) : plane((q - p) * (r - p), p) {}

  // Signed distance to plane: >0 above, <0 below, 0 on
  T side(p3 p) { return (n | p) - d; }

  // Perpendicular distance from point to plane
  ld dist(p3 p) { return abs(side(p)) / abs(n); }

  // Translate plane by vector t
  plane translate(p3 t) { return {n, d + (n | t)}; }

  // Shift plane up by distance along normal (double)
  plane shiftUp(ld dist) { return {n, d + dist * abs(n)}; }

  // Project point onto plane
  p3 proj(p3 p) { return p - n * side(p) / sq(n); }

  // Reflect point across plane
  p3 refl(p3 p) { return p - n * 2 * side(p) / sq(n); }

  p3 pickPoint() const {

    std::uniform_real_distribution<double> dist(0.0, 1.0);

    double v1 = dist(rng);

    double v2 = dist(rng);



    auto [a, b, c] = n;

    // solve a*x + b*y + c*z = d

    if (std::abs(a) > EPS) {

      double x = (d - b * v1 - c * v2) / a;

      return {(T)x, (T)v1, (T)v2};

    }

    if (std::abs(b) > EPS) {

      double y = (d - a * v1 - c * v2) / b;

      return {(T)v1, (T)y, (T)v2};

    }

    if (std::abs(c) > EPS) {

      double z = (d - a * v1 - b * v2) / c;

      return {(T)v1, (T)v2, (T)z};

    }

    // plane is degenerate (n == 0), return origin

    return {0, 0, 0};

  }


};

// Coordinate frame: origin and basis vectors
struct coords {
  p3 o, dx, dy, dz;

  // Build orthonormal basis from points P,Q,R on plane
  coords(p3 p, p3 q, p3 r) : o(p) {
    dx = unit(q - p);
    dz = unit(dx * (r - p));
    dy = dz * dx;
  }

  // Build basis from four points P,Q,R,S using raw
differences
  coords(p3 p, p3 q, p3 r, p3 s) :
      o(p), dx(q - p), dy(r - p), dz(s - p) {}

  // Convert 3D point to local 3D coordinates
  p3 pos3d(p3 p) { return {(p - o) | dx, (p - o) | dy, (p - o) | dz};
}
};
```

```
// 3D line: direction and origin
struct line3d {
  p3 d, o;

  // Construct line from points P,Q
  line3d(p3 p, p3 q) : d(q - p), o(p) {}

  // Construct intersection of two planes p1,p2
  line3d(plane p1, plane p2) {
    d = p1.n * p2.n;
    o = (p2.n * p1.d - p1.n * p2.d) * d / sq(d);
  }

  // Squared distance from point p to this line
  ld sqDist(p3 p) { return sq(d * (p - o)) / sq(d); }

  // Distance from point p to this line
  ld dist(p3 p) { return sqrtl(sqDist(p)); }

  // Compare projections of points p,q onto this line
  bool cmpProj(p3 p, p3 q) { return (d | p) < (d | q); }

  // Project point onto line
  p3 proj(p3 p) { return o + d * (d | (p - o)) / sq(d); }

  // Reflect point across line
  p3 refl(p3 p) { return proj(p) * 2 - p; }

  // Intersection of this line with plane p
  p3 inter(plane p) { return o - d * p.side(o) / (p.n | d); }
};

// Distance between two lines in 3D
ld dist(line3d l1, line3d l2) {
  p3 n = l1.d * l2.d;
  if (n == zero) // parallel
    return l1.dist(l2.o);
  return abs((l2.o - l1.o) | n) / abs(n);
}

// Closest point on l1 to l2
p3 closestOnL1(line3d l1, line3d l2) {
  p3 n2 = l2.d * (l1.d * l2.d);
  return l1.o + l1.d * ((l2.o - l1.o) | n2) / (l1.d | n2);
}

// Smallest angle between vectors v and w (0..pi/2)
ld smallAngle(p3 v, p3 w) {
  return acosl(min(abs(v | w) / abs(v) / abs(w), 1.0L));
}

// Angle between planes p1 and p2
ld angle(plane p1, plane p2) {
  return smallAngle(p1.n, p2.n);
}

// Angle between lines l1 and l2
ld angle(line3d l1, line3d l2) {
  return smallAngle(l1.d, l2.d);
}

// Angle between plane and line
ld angle(plane p, line3d l) {
  return PI / 2 - smallAngle(p.n, l.d);
}

// Check if two planes are parallel
bool isParallel(plane p1, plane p2) {
  return p1.n * p2.n == zero;
}

// Check if two lines are parallel
bool isParallel(line3d l1, line3d l2) {
  return l1.d * l2.d == zero;
}

// Check if plane and line are parallel
bool isParallel(plane p, line3d l) {
  return (p.n | l.d) == 0;
}

// Check if two planes are perpendicular
bool isPerpendicular(plane p1, plane p2) {
  return (p1.n | p2.n) == 0;
}

// Check if two lines are perpendicular
bool isPerpendicular(line3d l1, line3d l2) {
  return (l1.d | l2.d) == 0;
}

// Check if plane and line are perpendicular
bool isPerpendicular(plane p, line3d l) {
  return p.n * l.d == zero;
}

// Line perpendicular to plane p through point o
line3d perpThrough(plane p, p3 o) { return line3d(o, o +
p.n); }

// Plane perpendicular to line l through point o
plane perpThrough(line3d l, p3 o) { return plane(l.d, o); }
```

```
//area of one face
p3 vectorArea2(vector<p3> p) { // vector area * 2 (to avoid
divisions)
  p3 S = zero;
  for (int i = 0, n = p.size(); i < n; i++)
    S = S + p[i] * p[(i + 1) % n];
  return S;
}

ld area(vector<p3> p) {
  return abs(vectorArea2(p)) / 2.0L;
}
// Create arbitrary comparator for map<>
bool operator<(p3 p, p3 q) {
  return tie(p.x, p.y, p.z) < tie(q.x, q.y, q.z);
}
struct edge {
  int v;
  bool same; // = is the common edge in the same order?
};

// Given a series of faces (lists of points), reverse some of
them
// so that their orientations are consistent
void reorient(vector<vector<p3>> &fs) {
  int n = fs.size();
  // Find the common edges and create the resulting graph
  vector<vector<edge>> g(n);
  map<pair<p3, p3>, int> es;

  for (int u = 0; u < n; u++) {
    for (int i = 0, m = fs[u].size(); i < m; i++) {
      p3 a = fs[u][i], b = fs[u][(i + 1) % m];
      // Let's look at edge [AB]
      if (es.count({a, b})) { // seen in same order
        int v = es[{a, b}];
        g[u].push_back({v, true});
        g[v].push_back({u, true});
      } else if (es.count({b, a})) {// seen in different order
        int v = es[{b, a}];
        g[u].push_back({v, false});
        g[v].push_back({u, false});
      } else { // not seen yet
        es[{a, b}] = u;
      }
    }
  }
  // Perform BFS to find which faces should be flipped
  vector<bool> vis(n, false), flip(n);
  flip[0] = false;
  queue<int> q;
  q.push(0);
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    for (edge e: g[u]) {
      if (!vis[e.v]) {
        vis[e.v] = true;
        // If the edge was in the same order,
        // exactly one of the two should be flipped
        flip[e.v] = flip[u] ^ e.same;
        q.push(e.v);
      }
    }
  }
  // Actually perform the flips
  for (int u = 0; u < n; u++)
    if (flip[u])
      reverse(fs[u].begin(), fs[u].end());
}

//faces must be oriented
ld volume(vector<vector<p3>> fs) {
  ld vol6 = 0.0;
  for (vector<p3> f: fs)
    vol6 += (vectorArea2(f) | f[0]);
  return abs(vol6) / 6.0L;
}

// Spherical Coordinates to cartisian coordinates, given
angles in degrees
p3 sph(ld r, ld lat, ld lon) {
  lat *= PI / 180, lon *= PI / 180;
  return {r * cosl(lat) * cosl(lon), r * cosl(lat) * sinl(lon), r *
sinl(lat)};
}

//sphere line intersection
int sphereLine(p3 o, ld r, line3d l, pair<p3, p3> &out) {
  ld h2 = r * r - l.sqDist(o);
  if (h2 < 0) return 0; // the line doesn't touch the sphere
  p3 p = l.proj(o); // point P
  p3 h = l.d * sqrt(h2) / abs(l.d); // vector parallel to l, of
length h
  out = {p - h, p + h};
  return 1 + (h2 > 0);
}

//distance from a to b in sphere centered at o
ld greatCircleDist(p3 o, ld r, p3 a, p3 b) {
  return r * angle(a - o, b - o);
}

//spherical segment is valid if points are not directly
opposite
bool validSegment(p3 a, p3 b) {
  return a * b != zero || (a | b) > 0;
}
```

```
//proper intersection between spherical segments
bool properInter(p3 a, p3 b, p3 c, p3 d, p3 &out) {
  p3 ab = a * b, cd = c * d; // normals of planes OAB and
OCD
  int oa = sgn(cd | a),
      ob = sgn(cd | b),
      oc = sgn(ab | c),
      od = sgn(ab | d);
  out = ab * cd * od; // four multiplications => careful with
overflow!
  return (oa != ob && oc != od && oa != oc);
}

// check if p is on a spherical segment ab
bool onSphSegment(p3 a, p3 b, p3 p) {
  p3 n = a * b;
  if (n == zero)
    return a * p == zero && (a | p) > 0;
  return (n | p) == 0 && (n | a * p) >= 0 && (n | b * p) <= 0;
}

struct directionSet : vector<p3> {
  using vector::vector; // import constructors
  void insert(p3 p) {
    for (p3 q: *this) if (p * q == zero) return;
    push_back(p);
  }
};

directionSet intersSph(p3 a, p3 b, p3 c, p3 d) {
  assert(validSegment(a, b) && validSegment(c, d));
  p3 out;
  if (properInter(a, b, c, d, out)) return {out};
  directionSet s;
  if (onSphSegment(c, d, a)) s.insert(a);
  if (onSphSegment(c, d, b)) s.insert(b);
  if (onSphSegment(a, b, c)) s.insert(c);
  if (onSphSegment(a, b, d)) s.insert(d);
  return s;
}

//spherical angle bac
ld angleSph(p3 a, p3 b, p3 c) {
  return angle(a * b, a * c);
}

//spherical angle bac oriented (counter clock-wise)
ld orientedAngleSph(p3 a, p3 b, p3 c) {
  if ((a * b | c) >= 0)
    return angleSph(a, b, c);
  else
    return 2 * PI - angleSph(a, b, c);
}

//area of spherical polygon
ld areaOnSphere(ld r, vector<p3> p) {
  int n = p.size();
  ld sum = -(n - 2) * PI;
  for (int i = 0; i < n; i++)
    sum += orientedAngleSph(p[(i + 1) % n], p[(i + 2) % n],
p[i]);
  return r * r * sum;
}

// 0 -> o is outside polyhedron
// +1 -> is inside the polyhedron, and the vector areas # S
of the faces are oriented towards the outside;
// -1 -> is inside the polyhedron, and the vector areas # S of
the faces are oriented towards the inside;
int windingNumber3D(vector<vector<p3>> fs) {
  ld sum = 0;
  for (vector<p3> f: fs)
    sum += remainder(areaOnSphere(1, f), 4 * PI);
  return roundl(sum / (4 * PI));
}
```

Transformation 3D

// 4x4 matrix for homogeneous transformations

```
struct mat4 {
  T m[4][4];
  // Construct identity
  mat4() {
    for (int i = 0; i < 4; ++i)
      for (int j = 0; j < 4; ++j)
        m[i][j] = (i == j ? 1 : 0);
  }
  // Matrix multiplication
  mat4 operator*(const mat4 &o) const {
    mat4 r;
    for (int i = 0; i < 4; ++i) {
      for (int j = 0; j < 4; ++j) {
        r.m[i][j] = 0;
        for (int k = 0; k < 4; ++k)
          r.m[i][j] += m[i][k] * o.m[k][j];
      }
    }
    return r;
  }
  // Apply to homogeneous vector [x,y,z,1]
  p3 transform(p3 p) const {
    ld x = m[0][0]*p.x + m[0][1]*p.y + m[0][2]*p.z + m[0][3];
    ld y = m[1][0]*p.x + m[1][1]*p.y + m[1][2]*p.z + m[1][3];
    ld z = m[2][0]*p.x + m[2][1]*p.y + m[2][2]*p.z + m[2][3];
    ld w = m[3][0]*p.x + m[3][1]*p.y + m[3][2]*p.z + m[3][3];
    // assume w==1 or normalize
    if (w != 1 && fabs((double)w) > EPS) {
```

```
      x /= w; y /= w; z /= w;
    }
    return { (T)x, (T)y, (T)z };
  }

  // Create translation matrix
  static mat4 translate(T tx, T ty, T tz) {
    mat4 r;
    r.m[0][3] = tx;
    r.m[1][3] = ty;
    r.m[2][3] = tz;
    return r;
  }
  // Create scaling matrix
  static mat4 scale(T sx, T sy, T sz) {
    mat4 r;
    r.m[0][0] = sx;
    r.m[1][1] = sy;
    r.m[2][2] = sz;
    r.m[3][3] = 1;
    return r;
  }
  // Create rotation about arbitrary axis through origin by
angle (radians)
  static mat4 rotateAxis(p3 axis, ld angle) {
    axis = unit(axis);
    ld x = axis.x, y = axis.y, z = axis.z;
    ld c = cosl(angle), s = sinl(angle), t = 1 - c;
    mat4 r;
    r.m[0][0] = t*x*x + c;
    r.m[0][1] = t*x*y - s*z;
    r.m[0][2] = t*x*z + s*y;
    r.m[1][0] = t*x*y + s*z;
    r.m[1][1] = t*y*y + c;
    r.m[1][2] = t*y*z - s*x;
    r.m[2][0] = t*x*z - s*y;
    r.m[2][1] = t*y*z + s*x;
    r.m[2][2] = t*z*z + c;
    return r;
  }
};


// Example usage:

// p3 p = {1,2,3};

// Rotate p around axis (dx,dy,dz) by angle theta

// mat4 R = mat4::rotateAxis({dx,dy,dz}, theta);

// p3 p_rot = R.transform(p);

// Scale by (a,b,c): mat4 S = mat4::scale(a,b,c);

// Translate by (tx,ty,tz): mat4 T = mat4::translate(tx,ty,tz);

// Combine: mat4 M = T * R * S; // scale, then rotate, then
translate

// p3 p_tr = M.transform(p)
```

3D-Misc

```
// returns center of circle passing through three
// non-colinear and co-planer points a, b and c
p3 circle_center(p3 a, p3 b, p3 c) {
  p3 v1 = b - a, v2 = c - a;
  double v1v1 = v1 | v1, v2v2 = v2 | v2, v1v2 = v1 | v2;
  double base = 0.5 / (v1v1 * v2v2 - v1v2 * v1v2);
  double k1 = base * v2v2 * (v1v1 - v1v2);
  double k2 = base * v1v1 * (v2v2 - v1v2);
  return a + v1 * k1 + v2 * k2;
}

// segment ab to point c
double distance_from_segment_to_point(p3 a, p3 b, p3 c)
{
  auto dist = [&](p3 a, p3 b) {
    return sqrt((a - b) | (a - b));
  };

  if (sgn((b - a) | (c - a)) < 0) return dist(a, c);
  if (sgn((a - b) | (c - b)) < 0) return dist(b, c);
  return fabs(abs((unit(b - a) * (c - a))));
}

double distance_from_triangle_to_point(p3 a, p3 b, p3 c,
p3 d) {
  plane P(a, b, c);
  p3 proj = P.proj(d);
  double dis = min(distance_from_segment_to_point(a, b, d),
           min(distance_from_segment_to_point(b, c, d),
distance_from_segment_to_point(c, a, d)));
  int o = sgn(orientByNormal(a, b, proj, P.n));
  int inside = o == sgn(orientByNormal(b, c, proj, P.n));
  inside &= o == sgn(orientByNormal(c, a, proj, P.n));
  if (inside) return abs(d - proj);
  return dis;
}

double distance_from_triangle_to_segment(p3 a, p3 b, p3
c, p3 d, p3 e) {
  double l = 0.0, r = 1.0;
  int cnt = 100;
  double ret = inf;
  while (cnt--) {
    double mid1 = l + (r - l) / 3.0, mid2 = r - (r - l) / 3.0;
    double x = distance_from_triangle_to_point(a, b, c, d +
(e - d) * mid1);
    double y = distance_from_triangle_to_point(a, b, c, d +
(e - d) * mid2);
```

```
        if (x < y) {
            r = mid2;
            ret = x;
        } else {
            ret = y;
            l = mid1;
        }
    }
    return ret;
}

// triangles are solid
double distance_from_triangle_to_triangle(p3 a, p3 b, p3
c, p3 d, p3 e, p3 f) {
    double ret = inf;
    ret = min(ret, distance_from_triangle_to_segment(a, b,
c, d, e));
    ret = min(ret, distance_from_triangle_to_segment(a, b,
c, e, f));
    ret = min(ret, distance_from_triangle_to_segment(a, b,
c, f, d));
    ret = min(ret, distance_from_triangle_to_segment(d, e, f,
a, b));
    ret = min(ret, distance_from_triangle_to_segment(d, e, f,
b, c));
    ret = min(ret, distance_from_triangle_to_segment(d, e, f,
c, a));
    return ret;
}

struct sphere {
    p3 c;
    double r;
    sphere() {}
    sphere(p3 c, double r) : c(c), r(r) {}
};

// spherical cap is a portion of a sphere cut off by a plane
// https://en.wikipedia.org/wiki/Spherical_cap
struct spherical_cap {
    p3 c;
    double r;
    spherical_cap() {}
    spherical_cap(p3 c, double r) : c(c), r(r) {}

    // angle th is the polar angle between the rays from the
center of the sphere to one edge of the cap
    // and orthogonal line from the center of the sphere to
the plane of the cap

    // height of the cap (just like real world cap)
    double height(double th)     {
        return r * (1 - cos(th));
    }
    // radius of the base of the cap
    double base_radius(double th)  {
        return r * sin(th);
    }
    // volume of the cap
    double volume(double th)     {
        double h = height(th);
        return PI * h * h * (3 * r - h) / 3.0;
    }
    // surface area of the cap
    double surface_area(double th) {
        double h = height(th);
        return 2 * PI * r * h;
    }
};

// returns the sphere passing through four points
sphere circumscribed_sphere(p3 a, p3 b, p3 c, p3 d) {
    assert( sign(plane(a, b, c).side(d)) != 0);

    plane u = plane(a - b, (a + b) / 2);
    plane v = plane(b - c, (b + c) / 2);
    plane w = plane(c - d, (c + d) / 2);

    assert(!is_parallel(u, v));
    assert(!is_parallel(v, w));
    line3d l1(u, v), l2(v, w);
    assert( sign(dist(l1, l2)) == 0);

    p3 C = closest_on_l1(l1, l2);
    return sphere(C, abs(C - a));
}

// https://mathworld.wolfram.com/Sphere-
SphereIntersection.html
// it won't work if one sphere is totally inside the other
sphere
// handle that case separately
// returns the surface area and volume of the intersection
pair<double, double> sphere_sphere_intersection(sphere
s1, sphere s2) {
    double d = abs(s1.c - s2.c);
    if(sign(d - s1.r - s2.r) >= 0) return {0, 0}; // not intersecting
    // only the distance matters, so we will now consider the
centers
    // of the big sphere to be (0, 0, 0) and (d, 0, 0) for the
small sphere
    // we can transform the results back to w.r.t the real
centers if we want

    double R = max(s1.r, s2.r);
    double r = min(s1.r, s2.r);
    double y = R + r - d;
    double x = (R * R - r * r + d * d) / (2 * d);
```

```
  // the intersecting plane is parallel to the yz plane
  // with the above x value as its x coordinate
  double w = d * d - r * r  + R * R;
  double a = sqrt(4 * d * d * R * R - w * w) / (2.0 * d);
  // a is the radius of the intersecting circle on the intersecting plane
  // with center (x, 0)
  double h1 = R - x;
  double h2 = y - h1;
  // h1 is the height of the intersecting spherical cap of the big sphere
  // h2 is for the small sphere

  // total volume of the whole intersection = sum of the volumes of the spherical caps
  double volume      = PI * h1 * h1 * (3 * R - h1) / 3.0 + PI * h2 * h2 * (3 * r - h2) / 3.0;
  // total surface area of the intersecting spherical caps
  double surface_area = 2 * PI * R * h1 + 2 * PI * r * h2;
  return make_pair(surface_area, volume);
}

sphere smallest_enclosing_sphere(vector<p3> p) {
  int n = p.size();
  p3 c(0, 0, 0);
  for(int i = 0; i < n; i++) c = c + p[i];
  c = c / n;

  double ratio = 0.1;
  int pos = 0;
  int it = 100000;
  while (it--) {
    pos = 0;
    for (int i = 1; i < n; i++) {
      if(sq(c - p[i]) > sq(c - p[pos])) pos = i;
    }
    c = c + (p[pos] - c) * ratio;
    ratio *= 0.998;
  }
  return sphere(c, abs(c - p[pos]));
}

// it returns the angle of the spherical cap that is formed by the intersection of all tangents
double tangent_from_point_to_sphere(p3 p, sphere s) {
  double d = abs(p - s.c);
  if (sign(d - s.r) < 0) return -1; // inside the sphere, so no tangent
  if (sign(d - s.r) == 0) return -2; // on the sphere, handle separately
  double tangent_length = sqrt(d * d - s.r * s.r);
  double th = acos(s.r / d);
  return th;
}

struct pyramid {
  int n;     // number of sides of the pyramid
  double l;  // length of each side
  double ang;
  pyramid(int _n, double _l) {
    n = _n;
    l = _l;
    ang = PI / n;
  }
  double base_area() {
    return l * l * n / (4 * tan(ang));
  }
  double height() {
    return l * sqrt(1 - 1 / (4 * sin(ang) * sin(ang)) );
  }
  double volume() {
    return base_area() * height() / 3;
  }
};

struct cylinder {
  double r, h; // radius and height
  cylinder(double _r, double _h) {
    r = _r;
    h = _h;
  }
  double volume() {
    return PI * r * r * h;
  }
  double surface_area() {
    return 2 * PI * r * h + 2 * PI * r * r;
  }
};

struct cone {
  double r, h; // radius and height
  cone(double _r, double _h) {
    r = _r;
    h = _h;
  }
  double volume() {
    return PI * r * r * h / 3.0;
  }
  double surface_area() {
    return PI * r * (r + sqrt(h * h + r * r));
  }
};
```

2D – MISC

```
//5 - outside and do not intersect
//4 - intersect outside in one point
//3 - intersect in 2 points
//2 - intersect inside in one point
//1 - inside and do not intersect
int circle_circle_relation(pt a, double r, pt b, double R) {
  double d = dist(a, b);
  if (sign(d - r - R) > 0)  return 5;
  if (sign(d - r - R) == 0) return 4;
  double l = fabs(r - R);
  if (sign(d - r - R) < 0 && sign(d - l) > 0) return 3;
  if (sign(d - l) == 0) return 2;
  if (sign(d - l) < 0) return 1;
  assert(0); return -1;
}

// returns the circle such that for all points w on the circumference of the circle
// dist(w, a) : dist(w, b) = rp : rq
// rp != rq
// https://en.wikipedia.org/wiki/Circles_of_Apollonius
circle get_apollonius_circle(pt p, pt q, double rp, double rq ){
  rq *= rq ;
  rp *= rp ;
  double a = rq - rp ;
  assert(sign(a));
  double g = rq * p.x - rp * q.x ; g /= a ;
  double h = rq * p.y - rp * q.y ; h /= a ;
  double c = rq * p.x * p.x - rp * q.x * q.x + rq * p.y * p.y - rp * q.y * q.y ;
  c /= a ;
  pt o(g, h);
  double r = g * g + h * h - c ;
  r = sqrt(r);
  return circle(o,r);
}

// given a convex polygon p, and a line ab and the top vertex of the polygon
// returns the intersection of the line with the polygon
// it returns the indices of the edges of the polygon that are intersected by the line
// so if it returns i, then the line intersects the edge (p[i], p[(i + 1) % n])
array<int, 2> convex_line_intersection(vector<pt> &p, pt a, pt b, int top) {
  int end_a = extreme_vertex(p, (a - b).perp(), top);
  int end_b = extreme_vertex(p, (b - a).perp(), top);
  auto cmp_l = [&](int i) { return orientation(a, p[i], b); };
  if (cmp_l(end_a) < 0 || cmp_l(end_b) > 0)
    return {-1, -1}; // no intersection
  array<int, 2> res;
  for (int i = 0; i < 2; i++) {
    int lo = end_b, hi = end_a, n = p.size();
    while ((lo + 1) % n != hi) {
      int m = ((lo + hi + (lo < hi ? 0 : n)) / 2) % n;
      (cmp_l(m) == cmp_l(end_b) ? lo : hi) = m;
    }
    res[i] = (lo + !cmp_l(hi)) % n;
    swap(end_a, end_b);
  }
  if (res[0] == res[1]) return {res[0], -1}; // touches the vertex res[0]
  if (!cmp_l(res[0]) && !cmp_l(res[1]))
    switch ((res[0] - res[1] + (int)p.size() + 1) % p.size()) {
      case 0: return {res[0], res[0]}; // touches the edge (res[0], res[0] + 1)
      case 2: return {res[1], res[1]}; // touches the edge (res[1], res[1] + 1)
    }
  return res; // intersects the edges (res[0], res[0] + 1) and (res[1], res[1] + 1)
}

pair<pt, int> point_poly_tangent(vector<pt> &p, pt Q, int dir, int l, int r) {
  while (r - l > 1) {
    int mid = (l + r) >> 1;
    bool pvs = orientation(Q, p[mid], p[mid - 1]) != -dir;
    bool nxt = orientation(Q, p[mid], p[mid + 1]) != -dir;
    if (pvs && nxt) return {p[mid], mid};
    if (!(pvs || nxt)) {
      auto p1 = point_poly_tangent(p, Q, dir, mid + 1, r);
      auto p2 = point_poly_tangent(p, Q, dir, l, mid - 1);
      return orientation(Q, p1.first, p2.first) == dir ? p1 : p2;
    }
    if (!pvs) {
      if (orientation(Q, p[mid], p[l]) == dir)  r = mid - 1;
      else if (orientation(Q, p[l], p[r]) == dir) r = mid - 1;
      else l = mid + 1;
    }
    if (!nxt) {
      if (orientation(Q, p[mid], p[l]) == dir)  l = mid + 1;
      else if (orientation(Q, p[l], p[r]) == dir) r = mid - 1;
      else l = mid + 1;
    }
  }
  pair<pt, int> ret = {p[l], l};
  for (int i = l + 1; i <= r; i++) ret = orientation(Q, ret.first, p[i]) != dir ? make_pair(p[i], i) : ret;
  return ret;
}
// (ccw, cw) tangents from a point that is outside this convex polygon
// returns indexes of the points
```

```
// ccw means the tangent from Q to that point is in the
same direction as the polygon ccw direction
pair<int, int> tangents_from_point_to_polygon(vector<pt>
&p, pt Q){
  int ccw = point_poly_tangent(p, Q, 1, 0, (int)p.size() -
1).second;
  int cw = point_poly_tangent(p, Q, -1, 0, (int)p.size() -
1).second;
  return make_pair(ccw, cw);
}

// minimum distance from a point to a convex polygon
// it assumes point lie strictly outside the polygon
double dist_from_point_to_polygon(vector<pt> &p, pt z) {
  double ans = inf;
  int n = p.size();
  if (n <= 3) {
    for(int i = 0; i < n; i++) ans = min(ans,
dist_from_point_to_seg(p[i], p[(i + 1) % n], z));
    return ans;
  }
  auto [r, l] = tangents_from_point_to_polygon(p, z);
  if(l > r) r += n;
  while (l < r) {
    int mid = (l + r) >> 1;
    double left = dist2(p[mid % n], z), right= dist2(p[(mid +
1) % n], z);
    ans = min({ans, left, right});
    if(left < right) r = mid;
    else l = mid + 1;
  }
  ans = sqrt(ans);
  ans = min(ans, dist_from_point_to_seg(p[l % n], p[(l + 1)
% n], z));
  ans = min(ans, dist_from_point_to_seg(p[l % n], p[(l - 1 +
n) % n], z));
  return ans;
}
// minimum distance from convex polygon p to line ab
// returns 0 is it intersects with the polygon
// top - upper right vertex
double dist_from_polygon_to_line(vector<pt> &p, pt a, pt
b, int top) { //O(log n)
  pt orth = (b - a).perp();
  if (orientation(a, b, p[0]) > 0) orth = (a - b).perp();
  int id = extreme_vertex(p, orth, top);
  if (dot(p[id] - a, orth) > 0) return 0.0; //if orth and a are in
the same half of the line, then poly and line intersects
  return dist_from_point_to_line(a, b, p[id]); //does not
intersect
}
// minimum distance from a convex polygon to another
convex polygon
// the polygon doesnot overlap or touch
// tested in https://toph.co/p/the-wall
double dist_from_polygon_to_polygon(vector<pt> &p1,
vector<pt> &p2) { // O(n log n)
  double ans = inf;
  for (int i = 0; i < p1.size(); i++) {
    ans = min(ans, dist_from_point_to_polygon(p2, p1[i]));
  }
  for (int i = 0; i < p2.size(); i++) {
    ans = min(ans, dist_from_point_to_polygon(p1, p2[i]));
  }
  return ans;
}
// returns the area of the intersection of the circle with
center c and radius r
// and the triangle formed by the points c, a, b
double _triangle_circle_intersection(pt c, double r, pt a, pt
b) {
  double sd1 = sq(c - a), sd2 = sq(c - b);
  if(sd1 > sd2) swap(a, b), swap(sd1, sd2);
  double sd = sq(a - b);
  double d1 = sqrtl(sd1), d2 = sqrtl(sd2), d = sqrt(sd);
  double valx = abs(sd2 - sd - sd1) / (2 * d);
  double h = sqrtl(sd1 - valx * valx);
  if(r >= d2) return h * d / 2;
  double area = 0;
  if(sd + sd1 < sd2) {
    if(r < d1) area = r * r * (acos(h / d2) - acos(h / d1)) / 2;
    else {
      area = r * r * ( acos(h / d2) - acos(h / r)) / 2;
      double valy = sqrtl(r * r - h * h);
      area += h * (valy - valx) / 2;
    }
  }
  else {
    if(r < h) area = r * r * (acos(h / d2) + acos(h / d1)) / 2;
    else {
      area += r * r * (acos(h / d2) - acos(h / r)) / 2;
      double valy = sqrtl(r * r - h * h);
      area += h * valy / 2;
      if(r < d1) {
        area += r * r * (acos(h / d1) - acos(h / r)) / 2;
        area += h * valy / 2;
      }
      else area += h * valx / 2;
    }
  }
  return area;
}
// intersection between a simple polygon and a circle
double polygon_circle_intersection(vector<pt> &v, pt p,
double r) {
  int n = v.size();
  double ans = 0.00;
  pt org = {0, 0};
```

```
  for(int i = 0; i < n; i++) {
    int x = orientation(p, v[i], v[(i + 1) % n]);
    if(x == 0) continue;
    double area = _triangle_circle_intersection(org, r, v[i] -
p, v[(i + 1) % n] - p);
    if (x < 0) ans -= area;
    else ans += area;
  }
  return abs(ans);
}
```