# Team Notebook

El Rejunte - Universidad Tecnologica Nacional

October 1, 2025

## Contents

# 1 Algorithms

## 1.1 Divide And Conquer Dp

```
vector<vector<int>> dp; //maybe replace dim of K with only 2
int n;
// d&q DP: go down the range [l,r) like merge sort, but also
// making sure to iterate over [from,to) in each step, and
// spliting the [from,to) in 2 parts when goind down:
// [from, best] and [best, to)
void solve(int l, int r, int k, int from, int to) {
 if(l >= r) return;
 int cur = (l+r)/2;
 int bestpos = cur-1;
 int best = INF; // assumes we want to minimize cost
 forr(i,from,min(cur, to)) {
  // cost function that usually depends on dp[i][k-1]
  int c = fcost(i, k);
  if(c < best) best = c, bestpos = i;
 }
 dp[cur][k] = best;
 solve(l, cur, k, from, bestpos+1);
 solve(cur+1, r, k, bestpos, to);
}
```

## 1.2 Lis

```
// Change comparisons and binary search for non-increasing
// Given an array, paint it in the least number of colors so
      that each
// color turns to a non-increasing subsequence. Solution:
     Min number of
// colors=Length of the longest increasing subsequence
struct lis {
 T INF;
 int n; vector<T> a; // secuencia y su longitud
 vector<pair<T,int>> d; // d[i]=ultimo valor de la
      subsecuencia de tamanio i
 vector<int> p; // padres
 vector<T> ret; // respuesta

 lis(T INF_, vector<T> &a_) {
  n = sz(a_);
  INF = INF_;
  a = a_;
  d.resize(n+1);
  p.resize(n+1);
 }
 void rec(int i) {
  if(i == -1) return;
  ret.push_back(a[i]);
  rec(p[i]);
 }
 int run() {
  d[0] = {-INF,-1};
  forn(i,n) d[i+1] = {INF, -1};
  forn(i,n) {
   int j = int(upper_bound(d.begin(), d.end(), mp(a[i], n))-
        d.begin());
   if(d[j-1].fst<a[i] && a[i]<d[j].fst) {
    p[i] = d[j-1].snd;
    d[j] = {a[i], i};
   }
  }
  ret.clear();
  dforn(i, n+1) if(d[i].fst!=INF) {
   rec(d[i].snd); // reconstruir
   reverse(ret.begin(), ret.end());
   return i; // longitud
  }
  return 0;
 }
};
```

## 1.3 Mo

```
// Commented code should be used if updates are needed
int n, sq, nq; // array size, sqrt(array size), #queries
struct Qu {   //[l, r)
  int l, r, id;
  // int upds; // # of updates before this query
};
Qu qs[MAXN];
ll ans[MAXN]; // ans[i] = answer to ith query
// struct Upd{
//   int p, v, prev; // pos, new_val, prev_val
// };
// Upd vupd[MAXN];

// Without updates
bool qcomp(const Qu& a, const Qu& b) {
  if (a.l / sq != b.l / sq) return a.l < b.l;
  return (a.l / sq) & 1 ? a.r < b.r : a.r > b.r;
}

// With updates
// bool qcomp(const Qu &a, const Qu &b){
//   if(a.l/sq != b.l/sq) return a.l<b.l;
//   if(a.r/sq != b.r/sq) return a.r<b.r;
//   return a.upds < b.upds;
// }

// Without updates: O(n^2/sq + q*sq)
// with sq = sqrt(n): O(n*sqrt(n) + q*sqrt(n))
// with sq = n/sqrt(q): O(n*sqrt(q))
//
// With updates: O(sq*q + q*n^2/sq^2)
// with sq = n^(2/3): O(q*n^(2/3))
// with sq = (2*n^2)^(1/3) may improve a bit
void mos() {
  forn(i, nq) qs[i].id = i;
  sq = sqrt(n) + .5; // without updates
  // sq=pow(n, 2/3.0)+.5; // with updates
  sort(qs, qs + nq, qcomp);
  int l = 0, r = 0;
  init();
  forn(i, nq) {
    Qu q = qs[i];
    while (l > q.l) add(--l);
    while (r < q.r) add(r++);
    while (l < q.l) remove(l++);
    while (r > q.r) remove(--r);
    // while(upds<q.upds){
    //   if(vupd[upds].p >= l && vupd[upds].p < r) remove(
          vupd[upds].p);
    //   v[vupd[upds].p] = vupd[upds].v; // do update
    //   if(vupd[upds].p >= l && vupd[upds].p < r) add(vupd[
          upds].p);
    //   upds++;
    // }
    // while(upds>q.upds){
    //   upds--;
    //   if(vupd[upds].p >= l && vupd[upds].p < r) remove(
          vupd[upds].p);
    //   v[vupd[upds].p] = vupd[upds].prev; // undo update
    //   if(vupd[upds].p >= l && vupd[upds].p < r) add(vupd[
          upds].p);
    // }
    ans[q.id] = get_ans();
  }
}
```

# 2 Flow

## 2.1 Dinic

```
struct Edge {
  int u, v;
  ll cap, flow;
  Edge() {}
  Edge(int uu, int vv, ll c) : u(uu), v(vv), cap(c), flow(0)
      {}
};
struct Dinic {
  int N;
  vector<Edge> E;
  vector<vector<int>> g;
  vector<int> d, pt;
  Dinic(int n) : N(n), g(n), d(n), pt(n) {} // clear and
      init
  void addEdge(int u, int v, ll cap) {
    if (u != v) {
      g[u].pb(sz(E));
      E.pb({u, v, cap});
      g[v].pb(sz(E));
      E.pb({v, u, 0});
    }
  }
  bool BFS(int S, int T) {
    queue<int> q({S});
    fill(d.begin(), d.end(), N + 1);
    d[S] = 0;
    while (!q.empty()) {
      int u = q.front();
      q.pop();
      if (u == T) break;
      for (int k : g[u]) {
        Edge& e = E[k];
        if (e.flow < e.cap && d[e.v] > d[e.u] + 1) {
          d[e.v] = d[e.u] + 1;
          q.push(e.v);
        }
      }
    }
    return d[T] != N + 1;
  }
  ll DFS(int u, int T, ll flow = -1) {
    if (u == T || flow == 0) return flow;
    for (int& i = pt[u]; i < sz(g[u]); ++i) {
      Edge& e = E[g[u][i]];
      Edge& oe = E[g[u][i] ^ 1];
      if (d[e.v] == d[e.u] + 1) {
        ll amt = e.cap - e.flow;
        if (flow != -1 && amt > flow) amt = flow;
        if (ll pushed = DFS(e.v, T, amt)) {
          e.flow += pushed;
          oe.flow -= pushed;
          return pushed;
        }
      }
    }
    return 0;
  }
  ll maxFlow(int S, int T) { // O(V^2*E), unit nets: O(sqrt(
      V)*E)
    ll total = 0;
    while (BFS(S, T)) {
      fill(pt.begin(), pt.end(), 0);
      while (ll flow = DFS(S, T)) total += flow;
    }
    return total;
  }
};
// Dinic wrapper to allow setting demands of min flow on
    edges
// If an edge with a min flow demand is part of a cycle,
    then the result
// is not guaranteed to be correct, it could result in false
     positives
struct DinicWithDemands {
  int N;
  vector<pair<Edge, ll>> E; // (normal dinic edge, min flow)
  Dinic dinic;
  DinicWithDemands(int n) : N(n), E(0), dinic(n + 2) {}
  void addEdge(int u, int v, ll cap, ll minFlow) {
    assert(minFlow <= cap);
    if (u != v) E.pb(mp(Edge(u, v, cap), minFlow));
  }
  ll maxFlow(int S, int T) { // Same complexity as normal
      Dinic
    int SRC = N, SNK = N + 1;
    ll minFlowSum = 0;
    forall(e, E) { // force the min flow
      minFlowSum += e->snd;
      dinic.addEdge(SRC, e->fst.v, e->snd);
      dinic.addEdge(e->fst.u, SNK, e->snd);
      dinic.addEdge(e->fst.u, e->fst.v, e->fst.cap - e->snd);
    }
    dinic.addEdge(T, S, INF); // INF >= max possible flow
    ll flow = dinic.maxFlow(SRC, SNK);
    if (flow < minFlowSum) return -1; // no valid flow exists
    assert(flow == minFlowSum);
    // Now go back to the original network, to a valid
    // state where all min flow values are satisfied.
    forn(i, sz(E)) {
      forn(j, 4) {
        assert(j % 2 || dinic.E[6 * i + j].flow == E[i].snd);
        dinic.E[6 * i + j].cap = dinic.E[6 * i + j].flow = 0;
      }
      dinic.E[6 * i + 4].cap += E[i].snd;
      dinic.E[6 * i + 4].flow += E[i].snd;
      // don't change edge [6*i+5] to keep forcing the mins
    }
    forn(i, 2) dinic.E[6 * sz(E) + i].cap = dinic.E[6 * sz(E)
        + i].flow = 0;
    // Just finish the maxFlow now
    dinic.maxFlow(S, T);
    flow = 0; // get the result manually
    forall(e, dinic.g[S]) flow += dinic.E[*e].flow;
    return flow;
  }
};
```

## 2.2 Edmonds Karp

```
struct EdmondsKarp {
  int N;
  vector<unordered_map<int, ll>> g;
  vector<int> p;
  ll f;
  EdmondsKarp(int n) : N(n), g(n), p(n) {}
  void addEdge(int a, int b, int w) { g[a][b] = w; }
  void augment(int v, int SRC, ll minE) {
    if (v == SRC) f = minE;
    else if (p[v] != -1) {
      augment(p[v], SRC, min(minE, g[p[v]][v]));
      g[p[v]][v] -= f, g[v][p[v]] += f;
    }
  }
  ll maxflow(int SRC, int SNK) { // O(min(VE^2,Mf*E))
    ll ret = 0;
    do {
      queue<int> q;
      q.push(SRC);
      fill(p.begin(), p.end(), -1);
      f = 0;
      while (sz(q)) {
        int node = q.front();
        q.pop();
        if (node == SNK) break;
```

```
      forall(it, g[node]) if (it->snd > 0 && p[it->fst] ==
          -1) {
        q.push(it->fst), p[it->fst] = node;
      }
    }
    augment(SNK, SRC, INF); // INF > max possible flow
    ret += f;
  } while (f);
  return ret;
 }
};
```

## 2.3 Hopcroft Karp

```
struct HopcroftKarp { // [0,n)->[0,m) (ids independent in
     each side)
  int n, m;
  vector<vector<int>> g;
  vector<int> mt, mt2, ds;
  HopcroftKarp(int nn, int mm) : n(nn), m(mm), g(n) {}
  void add(int a, int b) { g[a].pb(b); }
  bool bfs() {
    queue<int> q;
    ds = vector<int>(n, -1);
    forn(i, n) if (mt2[i] < 0) ds[i] = 0, q.push(i);
    bool r = false;
    while (!q.empty()) {
      int x = q.front();
      q.pop();
      for (int y : g[x]) {
        if (mt[y] >= 0 && ds[mt[y]] < 0) {
          ds[mt[y]] = ds[x] + 1, q.push(mt[y]);
        } else if (mt[y] < 0) r = true;
      }
    }
    return r;
  }
  bool dfs(int x) {
    for (int y : g[x]) {
      if (mt[y] < 0 || ds[mt[y]] == ds[x] + 1 && dfs(mt[y]))
          {
        mt[y] = x, mt2[x] = y;
        return true;
      }
    }
    ds[x] = 1 << 30;
    return false;
  }
  int mm() { // O(sqrt(V)*E)
    int r = 0;
    mt = vector<int>(m, -1);
    mt2 = vector<int>(n, -1);
    while (bfs()) forn(i, n) if (mt2[i] < 0) r += dfs(i);
    return r;
  }
};
```

## 2.4 Hungarian

```
typedef long double td;
typedef vector<int> vi;
typedef vector<td> vd;
const td INF = 1e100; // for maximum set INF to 0, and
     negate costs
bool zz(td x) { return abs(x) < 1e-9; } // change to x==0,
     for ints/ll
struct Hungarian {
  int n;
  vector<vd> cs;
  vi L, R;
  Hungarian(int N, int M) : n(max(N, M)), cs(n, vd(n)), L(n)
       , R(n) {
    forn(x, N) forn(y, M) cs[x][y] = INF;
  }
  void set(int x, int y, td c) { cs[x][y] = c; }
  td assign() { // O(n^3)
    int mat = 0;
    vd ds(n), u(n), v(n);
    vi dad(n), sn(n);
    forn(i, n) u[i] = *min_element(cs[i].begin(), cs[i].end()
        );
    forn(j, n) {
      v[j] = cs[0][j] - u[0];
      forr(i, 1, n) v[j] = min(v[j], cs[i][j] - u[i]);
    }
    L = R = vi(n, -1);
    forn(i, n) forn(j, n) if (R[j] == -1 && zz(cs[i][j] - u[i
        ] - v[j])) {
      L[i] = j, R[j] = i, mat++;
      break;
    }
    for (; mat < n; mat++) {
      int s = 0, j = 0, i;
      while (L[s] != -1) s++;
      fill(dad.begin(), dad.end(), -1);
      fill(sn.begin(), sn.end(), 0);
      forn(k, n) ds[k] = cs[s][k] - u[s] - v[k];
      while (1) {
        j = -1;
        forn(k, n) if (!sn[k] && (j == -1 || ds[k] < ds[j]))
            j = k;
        sn[j] = 1, i = R[j];
        if (i == -1) break;
        forn(k, n) if (!sn[k]) {
          td new_ds = ds[j] + cs[i][k] - u[i] - v[k];
          if (ds[k] > new_ds) ds[k] = new_ds, dad[k] = j;
        }
      }
      forn(k, n) if (k != j && sn[k]) {
        td w = ds[k] - ds[j];
        v[k] += w, u[R[k]] -= w;
      }
      u[s] += ds[j];
      while (dad[j] >= 0) {
        int d = dad[j];
        R[j] = R[d], L[R[j]] = j, j = d;
      }
      R[j] = s, L[s] = j;
    }
    td ret = 0;
    forn(i, n) ret += cs[i][L[i]];
    return ret;
  }
};
```

## 2.5 Matching

```
vector<int> g[MAXN]; // [0,n)->[0,m)
int n, m;
int mat[MAXM];
bool vis[MAXN];
int match(int x) {
  if (vis[x]) return 0;
  vis[x] = true;
  for (int y : g[x])
    if (mat[y] < 0 || match(mat[y])) {
      mat[y] = x;
      return 1;
    }
  return 0;
}
vector<pair<int, int> > max_matching() { // O(V^2 * E)
  vector<pair<int, int> > r;
  memset(mat, -1, sizeof(mat));
  forn(i, n) memset(vis, false, sizeof(vis)), match(i);
  forn(i, m) if (mat[i] >= 0) r.pb({mat[i], i});
  return r;
```

```
}
```

## 2.6 Min Cost Max Flow

```
typedef ll tf;
typedef ll tc;
const tf INF_FLOW = 1e14;
const tc INF_COST = 1e14;
struct edge {
  int u, v;
  tf cap, flow;
  tc cost;
  tf rem() { return cap - flow; }
};
struct MCMF {
  vector<edge> e;
  vector<vector<int>> g;
  vector<tf> vcap;
  vector<tc> dist;
  vector<int> pre;
  tc minCost;
  tf maxFlow;
  // tf wantedFlow; // Use it for fixed flow instead of max
      flow
  MCMF(int n) : g(n), vcap(n), dist(n), pre(n) {}
  void addEdge(int u, int v, tf cap, tc cost) {
    g[u].pb(sz(e)), e.pb({u, v, cap, 0, cost});
    g[v].pb(sz(e)), e.pb({v, u, 0, 0, -cost});
  }
  // O(n*m * min(flow, n*m)), sometimes faster in practice
  void run(int s, int t) {
    vector<bool> inq(sz(g));
    maxFlow = minCost = 0; // result will be in these
        variables
    while (1) {
      fill(vcap.begin(), vcap.end(), 0), vcap[s] = INF_FLOW;
      fill(dist.begin(), dist.end(), INF_COST), dist[s] = 0;
      fill(pre.begin(), pre.end(), -1), pre[s] = 0;
      queue<int> q;
      q.push(s), inq[s] = true;
      while (sz(q)) { // Fast bellman-ford
        int u = q.front();
        q.pop(), inq[u] = false;
        for (auto eid : g[u]) {
          edge& E = e[eid];
          if (E.rem() && dist[E.v] > dist[u] + E.cost) {
            dist[E.v] = dist[u] + E.cost;
            pre[E.v] = eid;
            vcap[E.v] = min(vcap[u], E.rem());
            if (!inq[E.v]) q.push(E.v), inq[E.v] = true;
          }
        }
      }
      if (pre[t] == -1) break;
      tf flow = vcap[t];
      // flow = min(flow, wantedFlow - maxFlow); //For fixed
          flow
      maxFlow += flow;
      minCost += flow * dist[t];
      for (int v = t; v != s; v = e[pre[v]].u) {
        e[pre[v]].flow += flow;
        e[pre[v] ^ 1].flow -= flow;
      }
      // if(maxFlow == wantedFlow) break; //For fixed flow
    }
  }
};
```

## 2.7 Min Cut

```
// Suponemos un grafo con el formato definido en Push
    relabel
bitset<MAX_V> type, used; // reset this
void dfs1(int nodo) {
  type.set(nodo);
  forall(it, G[nodo]) if (!type[it->fst] && it->snd > 0)
       dfs1(it->fst);
}
void dfs2(int nodo) {
  used.set(nodo);
  forall(it, G[nodo]) {
    if (!type[it->fst]) {
      // edge nodo -> (it->fst) pertenece al min_cut
      // y su peso original era: it->snd + G[it->fst][nodo]
      // si no existia arista original al reves
    } else if (!used[it->fst]) dfs2(it->fst);
  }
}
void minCut() // antes correr algun maxflow()
{
  dfs1(SRC);
  dfs2(SRC);
  return;
}
```

## 2.8 Push Relabel

```
#define MAX_V 1000
int N; // valid nodes are [0...N-1]
#define INF 1e9
// special nodes
#define SRC 0
#define SNK 1
map<int, int> G[MAX_V]; // limpiar esto -- unordered_map
    mejora
// To add an edge use
#define add(a, b, w) G[a][b] = w
ll excess[MAX_V];
int height[MAX_V], active[MAX_V], cuenta[2 * MAX_V + 1];
queue<int> Q;

void enqueue(int v) {
  if (!active[v] && excess[v] > 0) active[v] = true, Q.push(
      v);
}
void push(int a, int b) {
  int amt = min(excess[a], ll(G[a][b]));
  if (height[a] <= height[b] || amt == 0) return;
  G[a][b] -= amt, G[b][a] += amt;
  excess[b] += amt, excess[a] -= amt;
  enqueue(b);
}
void gap(int k) {
  forn(v, N) {
    if (height[v] < k) continue;
    cuenta[height[v]]--;
    height[v] = max(height[v], N + 1);
    cuenta[height[v]]++;
    enqueue(v);
  }
}
void relabel(int v) {
  cuenta[height[v]]--;
  height[v] = 2 * N;
  forall(it, G[v]) if (it->snd) height[v] = min(height[v],
      height[it->fst] + 1);
  cuenta[height[v]]++;
  enqueue(v);
}
ll maxflow() // O(V^3)
{
  zero(height), zero(active), zero(cuenta), zero(excess);
  cuenta[0] = N - 1;
  cuenta[N] = 1;
  height[SRC] = N;
  active[SRC] = active[SNK] = true;
  forall(it, G[SRC]) {
```

```
    excess[SRC] += it->snd;
    push(SRC, it->fst);
  }
  while (sz(Q)) {
    int v = Q.front();
    Q.pop();
    active[v] = false;
    forall(it, G[v]) push(v, it->fst);
    if (excess[v] > 0) cuenta[height[v]] == 1 ? gap(height[v
        ]) : relabel(v);
  }
  ll mf = 0;
  forall(it, G[SRC]) mf += G[it->fst][SRC];
  return mf;
}
```

# 3 Game Theory

## 3.1 Green Hackenbush

```
// A two-player game played on an undirected graph where
     some nodes
// are connected to the ground. On each turn, a player
     removes an edge.
// If this removal splits the graph into two components, any
      component
// that is not connected to the ground is removed. A player
     loses the game
// if it's impossible to make a move.
struct green_hackenbush {
 vector<vector<int>> g;
 vector<int> tin, low, gr;
 int t, root, ans;
 green_hackenbush(int n) {
  t = 0, root = -1, ans = 0;
  g.resize(n); gr.resize(n);
  tin.resize(n); low.resize(n);
 }
 // make u a node in the ground
 void ground(int u) {
  gr[u] = 1; if(root == -1) root = u;
 }
 // call first ground() if u or v are in the ground
 void add_edge(int u, int v) {
  if(gr[u]) u = root;
  if(gr[v]) v = root;
  if(u == v) { ans ^= 1; return; }
  g[u].pb(v); g[v].pb(u);
 }
 int solve(int u, int d) {
  tin[u] = low[u] = ++t;
  int ret = 0;
  forn(i,sz(g[u])) {
   int v = g[u][i];
   if(v == d) continue;
   if(tin[v] == 0) {
    int retv = solve(v,u);
    low[u] = min(low[u],low[v]);
    if(low[v] > tin[u]) ret ^= (1+retv)^1;
    else ret ^= retv;
   }else low[u] = min(low[u], tin[v]);
  }
  forn(i,sz(g[u])) {
   int v = g[u][i];
   if(v != d && tin[u] <= tin[v]) ret ^= 1;
  }
  return ret;
 }
 int solve() {
  return root == -1? 0 : ans^solve(root,-1);
 }
};
```

# 4 Geometry

## 4.1 All Point Pairs

```
// after each step() execution pt is sorted by dot product
     of the event
struct all_point_pairs { // O(n*n*log(n*n)), must add id, u,
      v to pto
  vector<pto> pt, ev;
  vector<int> idx;
  int cur_step;
  all_point_pairs(vector<pto> pt_) : pt(pt_) {
    idx = vector<int>(sz(pt));
    forn(i, sz(pt)) forn(j, sz(pt)) if (i != j) {
      pto p = pt[j] - pt[i];
      p.u = pt[i].id, p.v = pt[j].id;
      ev.pb(p);
    }
    sort(ev.begin(), ev.end(), cmp(pto(0, 0), pto(1, 0)));
    pto start(ev[0].y, -ev[0].x);
    sort(pt.begin(), pt.end(),
         [&](pto& u, pto& v) { return u * start < v * start;
              });
    forn(i, sz(idx)) idx[pt[i].id] = i;
    cur_step = 0;
  }
  bool step() {
    if (cur_step >= sz(ev)) return false;
    int u = ev[cur_step].u, v = ev[cur_step].v;
    swap(pt[idx[u]], pt[idx[v]]);
    swap(idx[u], idx[v]);
    cur_step++;
    return true;
  }
};
```

## 4.2 Circle

```
#define sqr(a) ((a)*(a))
pto perp(pto a){return pto(-a.y, a.x);}
line bisector(pto a, pto b){
 line l = line(a, b); pto m = (a+b)/2;
 return line(-l.b, l.a, -l.b*m.x+l.a*m.y);
}
struct circle{
 pto o; T r;
 circle(){}
 circle(pto a, pto b, pto c) {
  o = bisector(a, b).inter(bisector(b, c));
  r = o.dist(a);
 }
 bool inside(pto p) { return (o-p).norm_sq() <= r*r+EPS; }
 bool inside(circle c) { // this inside of c
  T d = (o - c.o).norm_sq();
  return d <= (c.r-r) * (c.r-r) + EPS;
 }
 // circle containing p1 and p2 with radius r
 // swap p1, p2 to get snd solution
 circle* circle2PtoR(pto a, pto b, T r_) {
        ld d2 = (a-b).norm_sq(), det = r_*r_/d2 - ld(0.25);
        if(det < 0) return nullptr;
  circle *ret = new circle();
        ret->o = (a+b)/ld(2) + perp(b-a)*sqrt(det);
        ret->r = r_;
  return ret;
 }
 pair<pto, pto> tang(pto p) {
  pto m = (p+o)/2;
  ld d = o.dist(m);
  ld a = r*r/(2*d);
  ld h = sqrtl(r*r - a*a);
  pto m2 = o + (m-o)*a/d;
```

```
  pto per = perp(m-o)/d;
  return make_pair(m2 - per*h, m2 + per*h);
 }
 vector<pto> inter(line l) {
  ld a = l.a, b = l.b, c = l.c - l.a*o.x - l.b*o.y;
  pto xy0 = pto(a*c/(a*a + b*b), b*c/(a*a + b*b));
  if(c*c > r*r*(a*a + b*b) + EPS) {
   return {};
  }else if(abs(c*c - r*r*(a*a + b*b)) < EPS) {
   return { xy0 + o };
  }else{
   ld m = sqrtl((r*r - c*c/(a*a + b*b))/(a*a + b*b));
   pto p1 = xy0 + (pto(-b,a)*m);
   pto p2 = xy0 + (pto(b,-a)*m);
   return { p1 + o, p2 + o };
  }
 }
 vector<pto> inter(circle c) {
  line l;
  l.a = o.x - c.o.x;
  l.b = o.y - c.o.y;
  l.c = (sqr(c.r)-sqr(r)+sqr(o.x)-sqr(c.o.x)+sqr(o.y)-sqr(c.o.y))/2.0;
  return (*this).inter(l);
 }
 ld inter_triangle(pto a, pto b) { // area of intersection with oab
  if(abs((o-a)^(o-b)) <= EPS) return 0.;
  vector<pto> q = {a}, w = inter(line(a,b));
  if(sz(w) == 2) forn(i,sz(w)) if((a-w[i])*(b-w[i]) < -EPS) q.pb(w[i]);
  q.pb(b);
  if(sz(q) == 4 && (q[0]-q[1])*(q[2]-q[1]) > EPS) swap(q[1], q[2]);
  ld s = 0;
  forn(i, sz(q)-1){
   if(!inside(q[i]) || !inside(q[i+1])) {
    s += r*r*angle((q[i]-o),q[i+1]-o)/T(2);
   }
   else s += abs((q[i]-o)^(q[i+1]-o)/2);
  }
  return s;
 }
};
vector<ld> inter_circles(vector<circle> c){
 vector<ld> r(sz(c)+1); // r[k]: area covered by at least k circles
 forn(i, sz(c)) {    // O(n^2 log n) (high constant)
  int k = 1;
  cmp s(c[i].o, pto(1,0));
  vector<pair<pto,int>> p = {
   {c[i].o + pto(1,0)*c[i].r, 0},
   {c[i].o - pto(1,0)*c[i].r, 0}};
  forn(j, sz(c)) if(j != i) {
   bool b0 = c[i].inside(c[j]), b1 = c[j].inside(c[i]);
   if(b0 && (!b1 || i<j)) k++;
   else if(!b0 && !b1) {
    vector<pto> v = c[i].inter(c[j]);
    if(sz(v) == 2) {
     p.pb({v[0], 1}); p.pb({v[1], -1});
     if(s(v[1], v[0])) k++;
    }
   }
  }
  sort(p.begin(), p.end(), [&](pair<pto,int> a, pair<pto,int> b) {
   return s(a.fst,b.fst); });
  forn(j,sz(p)) {
   pto p0 = p[j? j-1: sz(p)-1].fst, p1 = p[j].fst;
   ld a = angle(p0 - c[i].o, p1 - c[i].o);
   r[k]+=(p0.x-p1.x)*(p0.y+p1.y)/ld(2)+c[i].r*c[i].r*(a-sinl(a))/ld(2);
   k += p[j].snd;
  }
 }
 return r;
}
```

## 4.3 Convex Hull Dyn

```
struct semi_chull {
  set<pto> pt; // maintains semi chull without collinears points
  // in case we want them on the set, make the changes commented below
  bool check(pto p) {
    if (pt.empty()) return false;
    if (*pt.rbegin() < p) return false;
    if (p < *pt.begin()) return false;
    auto it = pt.lower_bound(p);
    if (it->x == p.x) return p.y <= it->y; // change? for collinears
    pto b = *it;
    pto a = *prev(it);
    return ((b - p) ^ (a - p)) + EPS >= 0; // change? for collinears
  }
  void add(pto p) {
    if (check(p)) return;
    pt.erase(p);
    pt.insert(p);
    auto it = pt.find(p);
    while (true) {
      if (next(it) == pt.end() || next(next(it)) == pt.end())
        break;
      pto a = *next(it), b = *next(next(it));
      if (((b - a) ^ (p - a)) + EPS >= 0) { // change? for collinears
        pt.erase(next(it));
      } else break;
    }
    it = pt.find(p);
    while (true) {
      if (it == pt.begin() || prev(it) == pt.begin()) break;
      pto a = *prev(it), b = *prev(prev(it));
      if (((b - a) ^ (p - a)) - EPS <= 0) { // change? for collinears
        pt.erase(prev(it));
      } else break;
    }
  }
};
struct CHD {
  semi_chull sup, inf;
  void add(pto p) { sup.add(p), inf.add(p * (-1)); }
  bool check(pto p) { return sup.check(p) && inf.check(p * (-1)); }
};
```

## 4.4 Convex Hull Trick

```
struct CHT {
  deque<pto> h;
  T f = 1, pos;
  CHT(bool min_ = 0) : f(min_ ? 1 : -1), pos(0) {} // min_=1 for min queries
  void add(pto p) { // O(1), pto(m,b) <=> y = mx + b
    p = p * f;
    if (h.empty()) {
      h.pb(p);
      return;
    }
    // p.x should be the lower/greater hull x
    assert(p.x <= h[0].x || p.x >= h.back().x);
    if (p.x <= h[0].x) {
      while (sz(h) > 1 && h[0].left(p, h[1])) h.pop_front(), pos--;
      h.push_front(p), pos++;
```

```
    } else {
      while (sz(h) > 1 && h[sz(h) - 1].left(h[sz(h) - 2], p))
          h.pop_back();
      h.pb(p);
    }
    pos = min(max(T(0), pos), T(sz(h) - 1));
  }
  T get(T x) {
    pto q = {x, 1};
    // O(log) query for unordered x
    int L = 0, R = sz(h) - 1, M;
    while (L < R) {
      M = (L + R) / 2;
      if (h[M + 1] * q <= h[M] * q) L = M + 1;
      else R = M;
    }
    return h[L] * q * f;
    // O(1) query for ordered x
    while (pos > 0 && h[pos - 1] * q < h[pos] * q) pos--;
    while (pos < sz(h) - 1 && h[pos + 1] * q < h[pos] * q)
        pos++;
    return h[pos] * q * f;
  }
};
```

## 4.5 Convex Hull

```
// returns convex hull of p in CCW order
// left must return >=0 to delete collinear points
vector<pto> CH(vector<pto>& p) {
  if (sz(p) < 3) return p; // edge case, keep line or point
  vector<pto> ch;
  sort(p.begin(), p.end());
  forn(i, sz(p)) { // lower hull
    while (sz(ch) >= 2 && ch[sz(ch) - 1].left(ch[sz(ch) - 2],
        p[i]))
      ch.pop_back();
    ch.pb(p[i]);
  }
  ch.pop_back();
  int k = sz(ch);
  dforn(i, sz(p)) { // upper hull
    while (sz(ch) >= k + 2 && ch[sz(ch) - 1].left(ch[sz(ch) -
        2], p[i]))
      ch.pop_back();
    ch.pb(p[i]);
  }
  ch.pop_back();
  return ch;
}
```

## 4.6 Halfplane

```
struct halfplane { // left half plane
  pto u, uv;
  int id;
  ld angle;
  halfplane() {}
  halfplane(pto u_, pto v_) : u(u_), uv(v_ - u_), angle(
      atan2l(uv.y, uv.x)) {}
  bool operator<(halfplane h) const { return angle < h.angle
      ; }
  bool out(pto p) { return (uv ^ (p - u)) < -EPS; }
  pto inter(halfplane& h) {
    T alpha = ((h.u - u) ^ h.uv) / (uv ^ h.uv);
    return u + (uv * alpha);
  }
};
vector<pto> intersect(vector<halfplane> h) {
  pto box[4] = {{INF, INF}, {-INF, INF}, {-INF, -INF}, {INF,
      -INF}};
  forn(i, 4) h.pb(halfplane(box[i], box[(i + 1) % 4]));
  sort(h.begin(), h.end());
  deque<halfplane> dq;
  int len = 0;
  forn(i, sz(h)) {
    while (len > 1 && h[i].out(dq[len - 1].inter(dq[len - 2])
        )) {
      dq.pop_back(), len--;
    }
    while (len > 1 && h[i].out(dq[0].inter(dq[1]))) { dq.
        pop_front(), len--; }
    if (len > 0 && abs(h[i].uv ^ dq[len - 1].uv) <= EPS) {
      if (h[i].uv * dq[len - 1].uv < 0.) { return vector<pto
          >(); }
      if (h[i].out(dq[len - 1].u)) {
        dq.pop_back(), len--;
      } else continue;
    }
    dq.pb(h[i]);
    len++;
  }
  while (len > 2 && dq[0].out(dq[len - 1].inter(dq[len - 2])
      )) {
    dq.pop_back(), len--;
  }
  while (len > 2 && dq[len - 1].out(dq[0].inter(dq[1]))) {
    dq.pop_front(), len--;
  }
  if (len < 3) return vector<pto>();
  vector<pto> inter;
  forn(i, len) inter.pb(dq[i].inter(dq[(i + 1) % len]));
  return inter;
}
```

## 4.7 Kd Tree

```
bool cmpx(pto a, pto b) { return a.x + EPS < b.x; }
bool cmpy(pto a, pto b) { return a.y + EPS < b.y; }
struct kd_tree {
  pto p;
  T x0 = INF, x1 = -INF, y0 = INF, y1 = -INF;
  kd_tree *l, *r;
  T distance(pto q) {
    T x = min(max(x0, q.x), x1);
    T y = min(max(y0, q.y), y1);
    return (pto(x, y) - q).norm_sq();
  }
  kd_tree(vector<pto>&& pts) : p(pts[0]) {
    l = nullptr, r = nullptr;
    forn(i, sz(pts)) {
      x0 = min(x0, pts[i].x), x1 = max(x1, pts[i].x);
      y0 = min(y0, pts[i].y), y1 = max(y1, pts[i].y);
    }
    if (sz(pts) > 1) {
      sort(pts.begin(), pts.end(), x1 - x0 >= y1 - y0 ? cmpx
          : cmpy);
      int m = sz(pts) / 2;
      l = new kd_tree({pts.begin(), pts.begin() + m});
      r = new kd_tree({pts.begin() + m, pts.end()});
    }
  }
  void nearest(pto q, int k, priority_queue<pair<T, pto>>&
      ret) {
    if (l == nullptr) {
      // avoid query point as answer
      // if(p == q) return;
      ret.push({(q - p).norm_sq(), p});
      while (sz(ret) > k) ret.pop();
      return;
    }
    kd_tree *al = l, *ar = r;
    T bl = l->distance(q), br = r->distance(q);
    if (bl > br) swap(al, ar), swap(bl, br);
    al->nearest(q, k, ret);
    if (br < ret.top().fst) ar->nearest(q, k, ret);
    while (sz(ret) > k) ret.pop();
```

```
  }
  priority_queue<pair<T, pto>> nearest(pto q, int k) {
    priority_queue<pair<T, pto>> ret;
    forn(i, k) ret.push({INF * INF, pto(INF, INF)});
    nearest(q, k, ret);
    return ret;
  }
};
```

## 4.8 Li Chao Tree

```
typedef long long T;
const T INF = 1e18;
// Li-Chao works for any function such that any pair of the
    functions
// inserted intersect at most once with each other. Most
    problems are
// about lines, but you may want to adapt this struct to
    your function
struct line {
  T m, b;
  line() {}
  line(T m_, T b_) {
    m = m_;
    b = b_;
  }
  T f(T x) { return m * x + b; }
  line operator+(line l) { return line(m + l.m, b + l.b); }
  line operator*(T k) { return line(m * k, b * k); }
};
struct li_chao {
  vector<line> cur, add;
  vector<int> L, R;
  T f, minx, maxx;
  line identity;
  int cnt;
  void new_node(line cur_, int l = -1, int r = -1) {
    cur.pb(cur_);
    add.pb(line(0, 0));
    L.pb(l);
    R.pb(r);
    cnt++;
  }
  li_chao(bool min_, T minx_, T maxx_) { // for min: min_=1,
      for max: min_=0
    f = min_ ? 1 : -1;
    identity = line(0, INF);
    minx = minx_;
    maxx = maxx_;
    cnt = 0;
    new_node(identity); // root id is 0
  }
  // only needed when "adding" lines lazily
  void apply(int id, line to_add_) {
    add[id] = add[id] + to_add_;
    cur[id] = cur[id] + to_add_;
  }
  // this method is needed even when no lazy is used, to
      avoid
  // null pointers and other problems in the code
  void push_lazy(int id) {
    if (L[id] == -1) {
      new_node(identity);
      L[id] = cnt - 1;
    }
    if (R[id] == -1) {
      new_node(identity);
      R[id] = cnt - 1;
    }
    // code below only needed when lazy ops are needed
    apply(L[id], add[id]);
    apply(R[id], add[id]);
    add[id] = line(0, 0);
  }
  // only needed when "adding" lines lazily
  void push_line(int id, T tl, T tr) {
    T m = (tl + tr) / 2;
    insert_line(L[id], cur[id], tl, m);
    insert_line(R[id], cur[id], m, tr);
    cur[id] = identity;
  }
  // O(log), or persistent return int instead of void
  void insert_line(int id, line new_line, T l, T r) {
    T m = (l + r) / 2;
    bool lef = new_line.f(l) < cur[id].f(l);
    bool mid = new_line.f(m) < cur[id].f(m);
    // uncomment for persistent
    // line to_push = new_line, to_keep = cur[id];
    // if(mid) swap(to_push,to_keep);
    if (mid) swap(new_line, cur[id]);

    if (r - l == 1) {
      // uncomment for persistent
      // new_node(to_keep);
      // return cnt-1;
      return;
    }
    push_lazy(id);
    if (lef != mid) {
      // uncomment for persistent
      // int lid = insert_line(L[id],to_push, l, m);
      // new_node(to_keep, lid, R[id]);
      // return cnt-1;
      insert_line(L[id], new_line, l, m);
    } else {
      // uncomment for persistent
      // int rid = insert_line(R[id],to_push, m, r);
      // new_node(to_keep, L[id], rid);
      // return cnt-1;
      insert_line(R[id], new_line, m, r);
    }
  }
  // for persistent, return int instead of void
  void insert_line(int id, line new_line) {
    insert_line(id, new_line * f, minx, maxx);
  }
  // O(log^2) doesn't support persistence
  void insert_segm(int id, line new_line, T l, T r, T tl, T
      tr) {
    if (tr <= l || tl >= r || tl >= tr || l >= r) return;
    if (tl >= l && tr <= r) {
      insert_line(id, new_line, tl, tr);
      return;
    }
    push_lazy(id);
    T m = (tl + tr) / 2;
    insert_segm(L[id], new_line, l, r, tl, m);
    insert_segm(R[id], new_line, l, r, m, tr);
  }
  // [l,r)
  void insert_segm(int id, line new_line, T l, T r) {
    insert_segm(id, new_line * f, l, r, minx, maxx);
  }
  // O(log^2) doesn't support persistence
  void add_line(int id, line to_add_, T l, T r, T tl, T tr)
      {
    if (tr <= l || tl >= r || tl >= tr || l >= r) return;
    if (tl >= l && tr <= r) {
      apply(id, to_add_);
      return;
    }
    push_lazy(id);
    push_line(id, tl, tr); // comment if insert isn't used
    T m = (tl + tr) / 2;
    add_line(L[id], to_add_, l, r, tl, m);
    add_line(R[id], to_add_, l, r, m, tr);
  }
  void add_line(int id, line to_add_, T l, T r) {
    add_line(id, to_add_ * f, l, r, minx, maxx);
```

```
  }
  // O(log)
  T get(int id, T x, T tl, T tr) {
    if (tl + 1 == tr) return cur[id].f(x);
    push_lazy(id);
    T m = (tl + tr) / 2;
    if (x < m) return min(cur[id].f(x), get(L[id], x, tl, m))
        ;
    else return min(cur[id].f(x), get(R[id], x, m, tr));
  }
  T get(int id, T x) { return get(id, x, minx, maxx) * f; }
};
```

## 4.9 Line

```
int sgn(T x) { return x < 0 ? -1 : !!x; }
struct line {
  T a, b, c; // Ax+By=C
  line() {}
  line(T a_, T b_, T c_) : a(a_), b(b_), c(c_) {}
  // TO DO: check negative C (multiply everything by -1)
  line(pto u, pto v) : a(v.y - u.y), b(u.x - v.x), c(a * u.x
       + b * u.y) {}
  int side(pto v) { return sgn(a * v.x + b * v.y - c); }
  bool inside(pto v) { return abs(a * v.x + b * v.y - c) <=
      EPS; }
  bool parallel(line v) { return abs(a * v.b - v.a * b) <=
      EPS; }
  pto inter(line v) {
    T det = a * v.b - v.a * b;
    if (abs(det) <= EPS) return pto(INF, INF);
    return pto(v.b * c - b * v.c, a * v.c - v.a * c) / det;
  }
};
```

## 4.10 Point

```
typedef long double T; // double could be faster but less
     precise
typedef long double ld;
const T EPS = 1e-9; // if T is integer, set to 0
const T INF = 1e18;
struct pto{
 T x, y;
 pto() : x(0), y(0) {}
 pto(T _x, T _y) : x(_x), y(_y) {}
 pto operator+(pto b) { return pto(x+b.x, y+b.y); }
 pto operator-(pto b) { return pto(x-b.x, y-b.y); }
 pto operator+(T k) { return pto(x+k, y+k); }
 pto operator*(T k) { return pto(x*k, y*k); }
 pto operator/(T k) { return pto(x/k, y/k); }
 // dot product
 T operator*(pto b) { return x*b.x + y*b.y; }
 // module of cross product, a^b>0 if angle_cw(u,v)<180
 T operator^(pto b) { return x*b.y - y*b.x; }
 // vector projection of this above b
 pto proj(pto b) { return b*((*this)*b) / (b*b); }
 T norm_sq() { return x*x + y*y; }
 ld norm() { return sqrtl(x*x + y*y); }
 ld dist(pto b) { return (b - (*this)).norm(); }
 //rotate by theta rads CCW w.r.t. origin (0,0)
 pto rotate(T ang) {
  return pto(x*cosl(ang) - y*sinl(ang), x*sinl(ang) + y*cosl
      (ang));
 }
 // true if this is at the left side of line ab
 bool left(pto a, pto b) { return ((a-*this) ^ (b-*this)) >
      0; }
 bool operator<(const pto &b) const {
  return x < b.x-EPS || (abs(x - b.x) <= EPS && y < b.y-EPS)
      ;
 }
 bool operator==(pto b){ return abs(x-b.x)<=EPS && abs(y-b.y
     )<=EPS; }
};
ld angle(pto a, pto o, pto b) {
 pto oa = a-o, ob = b-o;
 return atan2l(oa^ob, oa*ob);
}
ld angle(pto a, pto b) { // smallest angle bewteen a and b
 ld cost = (a*b) / a.norm() / b.norm();
 return acosl(max(ld(-1.), min(ld(1.), cost)));
}
```

## 4.11 Poly

```
struct poly{
 vector<pto> pt;
 poly(){}
 poly(vector<pto> pt_) : pt(pt_) {}
 void delete_collinears() { // delete collinear points
  deque<pto> nxt; int len = 0;
  forn(i,sz(pt)) {
   if(len>1 && abs((pt[i]-nxt[len-2])^(nxt[len-1]-nxt[len
        -2])) <= EPS)
    nxt.pop_back(), len--;
    nxt.pb(pt[i]); len++;
  }
  if(len>2 && abs((nxt[1]-nxt[len-1])^(nxt[0]-nxt[len-1]))
       <= EPS)
   nxt.pop_front(), len--;
  if(len>2 && abs((nxt[len-1]-nxt[len-2])^(nxt[0]-nxt[len
       -2])) <= EPS)
   nxt.pop_back(), len--;
  pt.clear(); forn(i,sz(nxt)) pt.pb(nxt[i]);
 }
 void normalize() {
  delete_collinears();
  if(pt[2].left(pt[0], pt[1])) reverse(pt.begin(), pt.end())
       ; //make it CW
  int n = sz(pt), pi = 0;
  forn(i, n)
   if(pt[i].x<pt[pi].x || (pt[i].x==pt[pi].x && pt[i].y<pt[
        pi].y))
    pi = i;
  rotate(pt.begin(), pt.begin()+pi, pt.end());
 }
 bool is_convex() { // delete collinear points first
  int N = sz(pt);
  if(N < 3) return false;
  bool isLeft = pt[0].left(pt[1], pt[2]);
  forr(i, 1, sz(pt))
   if(pt[i].left(pt[(i+1)%N], pt[(i+2)%N]) != isLeft)
    return false;
  return true;
 }
 // for convex or concave polygons
 // excludes boundaries, check it manually
 bool inside(pto p) { // O(n)
  bool c = false;
  forn(i, sz(pt)) {
   int j = (i+1)%sz(pt);
   if((pt[j].y>p.y) != (pt[i].y > p.y) &&
   (p.x < (pt[i].x-pt[j].x)*(p.y-pt[j].y)/(pt[i].y-pt[j].y)+
        pt[j].x))
    c = !c;
  }
  return c;
 }
 bool inside_convex(pto p) { // O(lg(n)) normalize first
  if(p.left(pt[0], pt[1]) || p.left(pt[sz(pt)-1], pt[0]))
       return false;
  int a = 1, b = sz(pt)-1;
  while(b-a > 1){
   int c = (a+b)/2;
   if(!p.left(pt[0], pt[c])) a = c;
```

```
  else b = c;
 }
 return !p.left(pt[a], pt[a+1]);
}
// cuts this along line ab and return the left side
// (swap a, b for the right one)
poly cut(pto a, pto b) { // O(n)
 vector<pto> ret;
 forn(i, sz(pt)) {
  ld left1 = (b-a)^(pt[i]-a), left2 = (b-a)^(pt[(i+1)%sz(pt
      )]-a);
  if(left1 >= 0) ret.pb(pt[i]);
  if(left1*left2 < 0)
   ret.pb(line(pt[i], pt[(i+1)%sz(pt)]).inter(line(a, b)));
 }
 return poly(ret);
}
// cuts this with line ab and returns the range [from, to]
    that is
// strictly on the left side (note that indexes are
    circular)
ii cut(pto u, pto v) { // O(log(n)) for convex polygons
 int n = sz(pt); pto dir = v-u;
 int L = farthest(pto(dir.y,-dir.x));
 int R = farthest(pto(-dir.y,dir.x));
 if(!pt[L].left(u,v)) swap(L,R);
 if(!pt[L].left(u,v)) return mp(-1,-1); // line doesn't cut
      the poly

 ii ans;
 int l = L, r = L > R ? R+n : R;
 while(l<r) {
  int med = (l+r+1)/2;
  if(pt[med >= n ? med-n : med].left(u,v)) l = med;
  else r = med-1;
 }
 ans.snd = l >= n ? l-n : l;

 l = R, r = L < R ? L+n : L;
 while(l<r) {
  int med = (l+r)/2;
  if(!pt[med >= n ? med-n : med].left(u,v)) l = med+1;
  else r = med;
 }
 ans.fst = l >= n ? l-n : l;

 return ans;
}
// addition of convex polygons
poly minkowski(poly p) { // O(n+m) n=|this|,m=|p|
 this->normalize(); p.normalize();
 vector<pto> a = (*this).pt, b = p.pt;
 a.pb(a[0]); a.pb(a[1]);
 b.pb(b[0]); b.pb(b[1]);
 vector<pto> sum;
 int i = 0, j = 0;
 while(i < sz(a)-2 || j < sz(b)-2) {
  sum.pb(a[i]+b[j]);
  T cross = (a[i+1]-a[i])^(b[j+1]-b[j]);
  if(cross <= 0 && i < sz(a)-2) i++;
  if(cross >= 0 && j < sz(b)-2) j++;
 }
 return poly(sum);
}
pto farthest(pto v) { // O(log(n)) for convex polygons
 if(sz(pt)<10) {
  int k=0;
  forr(i,1,sz(pt)) if(v * (pt[i] - pt[k]) > EPS) k = i;
  return pt[k];
 }
 pt.pb(pt[0]);
 pto a=pt[1] - pt[0];
 int s = 0, e = sz(pt)-1, ua = v*a > EPS;
 if(!ua && v*(pt[sz(pt)-2]-pt[0]) <= EPS){ pt.pop_back();
     return pt[0];}
 while(1) {
  int m = (s+e)/2; pto c = pt[m+1]-pt[m];
  int uc = v*c > EPS;
  if(!uc && v*(pt[m-1]-pt[m]) <= EPS){ pt.pop_back();
      return pt[m];}
  if(ua && (!uc || v*(pt[s]-pt[m]) > EPS)) e = m;
  else if(ua || uc || v*(pt[s]-pt[m]) >= -EPS) s = m, a = c
      , ua = uc;
  else e = m;
  assert(e > s+1);
 }
}
ld inter_circle(circle c){ // area of intersection with
    circle
 ld r = 0.;
 forn(i,sz(pt)) {
  int j = (i+1)%sz(pt); ld w = c.inter_triangle(pt[i], pt[j
      ]);
  if(((pt[j]-c.o)^(pt[i]-c.o)) > 0) r += w;
  else r -= w;
 }
 return abs(r);
}
// area ellipse = M_PI*a*b where a and b are the semi axis
    lengths
// area triangle = sqrt(s*(s-a)(s-b)(s-c)) where s=(a+b+c)
    /2
ld area(){ // O(n)
 ld area = 0;
 forn(i, sz(pt)) area += pt[i]^pt[(i+1)%sz(pt)];
 return abs(area)/ld(2);
}
// returns one pair of most distant points
pair<pto,pto> callipers() { // O(n), for convex poly,
    normalize first
 int n = sz(pt);
 if(n <= 2) return {pt[0], pt[1%n]};
 pair<pto,pto> ret = {pt[0], pt[1]};
 T maxi = 0; int j = 1;
 forn(i,sz(pt)) {
  while(((pt[(i+1)%n]-pt[i])^(pt[(j+1)%n]-pt[j]))<-EPS)j=(j
      +1)%sz(pt);
  if(pt[i].dist(pt[j]) > maxi+EPS)
   ret = {pt[i], pt[j]}, maxi = pt[i].dist(pt[j]);
 }
 return ret;
}
pto centroid(){ // (barycenter, mass center, needs float
    points)
 int n = sz(pt);
 pto r(0,0); ld t=0;
 forn(i,n) {
  r = r + (pt[i] + pt[(i+1)%n]) * (pt[i] ^ pt[(i+1)%n]);
  t += pt[i] ^ pt[(i+1)%n];
 }
 return r/t/3;
}
};
// Dynamic convex hull trick (based on poly struct)
vector<poly> w;
void add(pto q) { // add(q), O(log^2(n))
 vector<pto> p = {q};
 while(!w.empty() && sz(w.back().pt) < 2*sz(p)){
  for(pto v : w.back().pt) p.pb(v);
  w.pop_back();
 }
 w.pb(poly(CH(p))); // CH = convex hull, must delete
     collinears
}
T query(pto v) { // max(q*v:q in w), O(log^2(n))
 T r = -INF;
 for(auto& p : w) r = max(r, p.farthest(v)*v);
 return r;
}
```

## 4.12 Radial Order

```
// radial sort around point O in CCW direction starting from
     vector v
struct cmp {
  pto o, v;
  cmp(pto no, pto nv) : o(no), v(nv) {}
  bool half(pto p) {
    assert(!(p.x == 0 && p.y == 0)); // (0,0) isn't well
         defined
    return (v ^ p) < 0 || ((v ^ p) == 0 && (v * p) < 0);
  }
  bool operator()(pto& p1, pto& p2) {
    return mp(half(p1 - o), T(0)) < mp(half(p2 - o), ((p1 - o
         ) ^ (p2 - o)));
  }
};
```

## 4.13 Segment

```
struct segm {
  pto s, e;
  segm(pto s_, pto e_) : s(s_), e(e_) {}
  pto closest(pto b) {
    pto bs = b - s, es = e - s;
    ld l = es * es;
    if (abs(l) <= EPS) return s;
    ld t = (bs * es) / l;
    if (t < 0.) return s;     // comment for lines
    else if (t > 1.) return e; // comment for lines
    return s + (es * t);
  }
  bool inside(pto b) { return abs(s.dist(b) + e.dist(b) - s.
       dist(e)) < EPS; }
  pto inter(segm b) { // if a and b are collinear, returns
       one point
    if ((*this).inside(b.s)) return b.s;
    if ((*this).inside(b.e)) return b.e;
    pto in = line(s, e).inter(line(b.s, b.e));
    if ((*this).inside(in) && b.inside(in)) return in;
    return pto(INF, INF);
  }
};
```

## 4.14 Voronoi

```
// Returns planar graph representing Delaunay's
     triangulation.
// Edges for each vertex are in ccw order.
// To use doubles replace __int128 for long double in line
     51
pto pinf = pto(INF, INF);
typedef struct QuadEdge* Q;
struct QuadEdge {
  int id, used;
  pto o;
  Q rot, nxt;
  QuadEdge(int id_ = -1, pto o_ = pinf)
      : id(id_), used(0), o(o_), rot(0), nxt(0) {}
  Q rev() { return rot->rot; }
  Q next() { return nxt; }
  Q prev() { return rot->next()->rot; }
  pto dest() { return rev()->o; }
};

Q edge(pto a, pto b, int ida, int idb) {
  Q e1 = new QuadEdge(ida, a);
  Q e2 = new QuadEdge(idb, b);
  Q e3 = new QuadEdge;
  Q e4 = new QuadEdge;
  tie(e1->rot, e2->rot, e3->rot, e4->rot) = {e3, e4, e2, e1
       };
  tie(e1->nxt, e2->nxt, e3->nxt, e4->nxt) = {e1, e2, e4, e3
       };
  return e1;
}

void splice(Q a, Q b) {
  swap(a->nxt->rot->nxt, b->nxt->rot->nxt);
  swap(a->nxt, b->nxt);
}

void del_edge(Q& e, Q ne) {
  splice(e, e->prev());
  splice(e->rev(), e->rev()->prev());
  delete e->rev()->rot;
  delete e->rev();
  delete e->rot;
  delete e;
  e = ne;
}

Q conn(Q a, Q b) {
  Q e = edge(a->dest(), b->o, a->rev()->id, b->id);
  splice(e, a->rev()->prev());
  splice(e->rev(), b);
  return e;
}

auto area(pto p, pto q, pto r) { return (q - p) ^ (r - q); }

// is p in circumference formed by (a,b,c)?
bool in_c(pto a, pto b, pto c, pto p) {
  // Warning: this number is O(max_coord^4).
  // Consider using doubles or an alternative method for
       this function
  __int128 p2 = p * p, A = a * a - p2, B = b * b - p2, C = c
       * c - p2;
  return area(p, a, b) * C + area(p, b, c) * A + area(p, c,
       a) * B > EPS;
}

pair<Q, Q> build_tr(vector<pto>& p, int l, int r) {
  if (r - l + 1 <= 3) {
    Q a = edge(p[l], p[l + 1], l, l + 1), b = edge(p[l + 1],
         p[r], l + 1, r);
    if (r - l + 1 == 2) return {a, a->rev()};
    splice(a->rev(), b);
    auto ar = area(p[l], p[l + 1], p[r]);
    Q c = abs(ar) > EPS ? conn(b, a) : 0;
    if (ar >= -EPS) return {a, b->rev()};
    return {c->rev(), c};
  }
  int m = (l + r) / 2;
  Q la, ra, lb, rb;
  tie(la, ra) = build_tr(p, l, m);
  tie(lb, rb) = build_tr(p, m + 1, r);
  while (1) {
    if (ra->dest().left(lb->o, ra->o)) ra = ra->rev()->prev()
         ;
    else if (lb->dest().left(lb->o, ra->o)) lb = lb->rev()->
         next();
    else break;
  }
  Q b = conn(lb->rev(), ra);
  auto valid = [&](Q e) { return b->o.left(e->dest(), b->
       dest()); };
  if (ra->o == la->o) la = b->rev();
  if (lb->o == rb->o) rb = b;
  while (1) {
    Q L = b->rev()->next();
    if (valid(L))
      while (in_c(b->dest(), b->o, L->dest(), L->next()->dest
           ()))
        del_edge(L, L->next());
    Q R = b->prev();
```

```
    if (valid(R))
      while (in_c(b->dest(), b->o, R->dest(), R->prev()->dest
          ()))
        del_edge(R, R->prev());
    if (!valid(L) && !valid(R)) break;
    if (!valid(L) || (valid(R) && in_c(L->dest(), L->o, R->o,
          R->dest())))
      b = conn(R, b->rev());
    else b = conn(b->rev(), L->rev());
  }
  return {la, rb};
}

vector<vector<int>> delaunay(vector<pto> v) {
  int n = sz(v);
  auto tmp = v;
  vector<int> id(n);
  iota(id.begin(), id.end(), 0);
  sort(id.begin(), id.end(), [&](int l, int r) { return v[l]
        < v[r]; });
  forn(i, n) v[i] = tmp[id[i]];
  assert(unique(v.begin(), v.end()) == v.end());
  vector<vector<int>> g(n);
  int col = 1;
  forr(i, 2, n) col &= abs(area(v[i], v[i - 1], v[i - 2]))
       <= EPS;
  if (col) {
    forr(i, 1, n) g[id[i - 1]].pb(id[i]), g[id[i]].pb(id[i -
         1]);
  } else {
    Q e = build_tr(v, 0, n - 1).fst;
    vector<Q> edg = {e};
    for (int i = 0; i < sz(edg); e = edg[i++]) {
      for (Q at = e; !at->used; at = at->next()) {
        at->used = 1;
        g[id[at->id]].pb(id[at->rev()->id]);
        edg.pb(at->rev());
      }
    }
  }
  return g;
}
```

# 5 Graphs

## 5.1 2sat

```
// Usage:
// 1. Create with n = number of variables (0-indexed)
// 2. Add restrictions through the existing methods, using ~
     X for
//   negating variable X for example.
// 3. Call satisf() to check whether there is a solution or
     not.
// 4. Find a valid assigment by looking at verdad[cmp[2*X]]
     for each
//   variable X
struct Sat2 {
  // We have a vertex representing a variable and other for
       its
  // negation. Every edge stored in G represents an
       implication.
  vector<vector<int>> G;
  // idx[i]=index assigned in the dfs
  // lw[i]=lowest index(closer from the root) reachable from
       i
  // verdad[cmp[2*i]]=valor de la variable i
  int N, qidx, qcmp;
  vector<int> lw, idx, cmp, verdad;
  stack<int> q;
  Sat2(int n) : G(2 * n), N(n) {}
  void tjn(int v) {
    lw[v] = idx[v] = ++qidx;
    q.push(v), cmp[v] = -2;
    forall(it, G[v]) if (!idx[*it] || cmp[*it] == -2) {
      if (!idx[*it]) tjn(*it);
      lw[v] = min(lw[v], lw[*it]);
    }
    if (lw[v] == idx[v]) {
      int x;
      do { x = q.top(), q.pop(), cmp[x] = qcmp; } while (x !=
           v);
      verdad[qcmp] = (cmp[v ^ 1] < 0);
      qcmp++;
    }
  }
  bool satisf() { // O(N)
    idx = lw = verdad = vector<int>(2 * N, 0);
    cmp = vector<int>(2 * N, -1);
    qidx = qcmp = 0;
    forn(i, N) {
      if (!idx[2 * i]) tjn(2 * i);
      if (!idx[2 * i + 1]) tjn(2 * i + 1);
    }
    forn(i, N) if (cmp[2 * i] == cmp[2 * i + 1]) return false
         ;
    return true;
  }
  // a -> b, here ids are transformed to avoid negative
       numbers
  void addimpl(int a, int b) {
    a = a >= 0 ? 2 * a : 2 * (~a) + 1;
    b = b >= 0 ? 2 * b : 2 * (~b) + 1;
    G[a].pb(b), G[b ^ 1].pb(a ^ 1);
  }
  void addor(int a, int b) { addimpl(~a, b); } // a | b = ~a
       -> b
  void addeq(int a, int b) {                 // a = b, a <-> b
       (iff)
    addimpl(a, b);
    addimpl(b, a);
  }
  void addxor(int a, int b) { addeq(a, ~b); } // a xor b
  void force(int x, bool val) {            // force x to take
       val
    if (val) addimpl(~x, x);
    else addimpl(x, ~x);
  }
  // At most 1 true in all v
  void atmost1(vector<int> v) {
    int auxid = N;
    N += sz(v);
    G.rsz(2 * N);
    forn(i, sz(v)) {
      addimpl(auxid, ~v[i]);
      if (i) {
        addimpl(auxid, auxid - 1);
        addimpl(v[i], auxid - 1);
      }
      auxid++;
    }
    assert(auxid == N);
  }
};
```

## 5.2 Articulation

```
int N;
vector<int> G[1000000];
// V[i]=node number(if visited), L[i]= lowest V[i] reachable
     from i
int qV, V[1000000], L[1000000], P[1000000];
void dfs(int v, int f) {
  L[v] = V[v] = ++qV;
  forall(it, G[v]) if (!V[*it]) {
    dfs(*it, v);
    L[v] = min(L[v], L[*it]);
```

```
    P[v] += L[*it] >= V[v];
  }
  else if (*it != f) L[v] = min(L[v], V[*it]); }
int cantart() { // O(n)
  qV = 0;
  zero(V), zero(P);
  dfs(1, 0);
  P[1]--;
  int q = 0;
  forn(i, N) if (P[i]) q++;
  return q;
}
```

## 5.3 Bellman Ford

```
// Mas lento que Dijsktra, pero maneja arcos con peso
    negativo
//
// Can solve systems of "difference inequalities":
// 1. for each inequality x_i - x_j <= k add an edge j->i
    with weight k
// 2. create an extra node Z and add an edge Z->i with
    weigth 0 for
//  each variable x_i in the inequalities
// 3. run(Z): if negcycle, no solution, otherwise "dist" is
    a solution
//
// Can transform a graph to get all edges of positive weight
//("Jhonson algorightm"):
// 1. Create an extra node Z and add edge Z->i with weight 0
     for all
//   nodes i
// 2. Run bellman ford from Z
// 3. For each original edge a->b (with weight w), change
    its weigt to
//   be w+dist[a]-dist[b] (where dist is the result of step
    2)
// 4. The shortest paths in the old and new graph are the
    same (their
//   weight result may differ, but the paths are the same)
// Note that this doesn't work well with negative cycles,
    but you can
// identify them before step 3 and then ignore all new
    weights that
// result in a negative value when executing step 3.
struct BellmanFord {
  vector<vector<ii>> G; // ady. list with pairs (weight, dst
      )
  vector<ll> dist;
  int N;
  BellmanFord(int n) : G(n), N(n) {}
  void addEdge(int a, int b, ll w) { G[a].pb(mp(w, b)); }
  void run(int src) { // O(VE)
    dist = vector<ll>(N, INF);
    dist[src] = 0;
    forn(i, N - 1) forn(j, N) if (dist[j] != INF) forall(it,
         G[j])
        dist[it->snd] = min(dist[it->snd], dist[j] + it->fst)
            ;
  }

  bool hasNegCycle() {
    forn(j, N) if (dist[j] != INF)
        forall(it, G[j]) if (dist[it->snd] > dist[j] + it->
            fst) return true;
    // inside if: all points reachable from it->snd will have
         -INF
    // distance. However this is not enough to identify which
         exact
    // nodes belong to a neg cycle, nor even which can reach
        a neg
    // cycle. To do so, you need to run SCC (kosaraju) and
        check
    // whether each SCC hasNegCycle independently. For those
        that
    // do hasNegCycle, then all of its nodes are part of a (
        not
    // necessarily simple) neg cycle.
    return false;
  }
};
```

## 5.4 Biconnected

```
struct Bicon {
  vector<vector<int>> G;
  struct edge {
    int u, v, comp;
    bool bridge;
  };
  vector<edge> ve;
  void addEdge(int u, int v) {
    G[u].pb(sz(ve)), G[v].pb(sz(ve));
    ve.pb({u, v, -1, false});
  }
  // d[i] = dfs id
  // b[i] = lowest id reachable from i
  // art[i]>0 iff i is an articulation point
  // nbc = total # of biconnected comps
  // nart = total # of articulation points
  vector<int> d, b, art;
  int n, t, nbc, nart;
  Bicon(int nn) {
    n = nn;
    t = nbc = nart = 0;
    b = d = vector<int>(n, -1);
    art = vector<int>(n, 0);
    G = vector<vector<int>>(n);
    ve.clear();
  }
  stack<int> st;
  void dfs(int u, int pe) { // O(n + m)
    b[u] = d[u] = t++;
    forall(eid, G[u]) if (*eid != pe) {
      int v = ve[*eid].u ^ ve[*eid].v ^ u;
      if (d[v] == -1) {
        st.push(*eid);
        dfs(v, *eid);
        if (b[v] > d[u]) ve[*eid].bridge = true; // bridge
        if (b[v] >= d[u]) {                    // art
          if (art[u]++ == 0) nart++;
          int last; // start biconnected
          do {
            last = st.top();
            st.pop();
            ve[last].comp = nbc;
          } while (last != *eid);
          nbc++; // end biconnected
        }
        b[u] = min(b[u], b[v]);
      } else if (d[v] < d[u]) { // back edge
        st.push(*eid);
        b[u] = min(b[u], d[v]);
      }
    }
  }
  void run() { forn(i, n) if (d[i] == -1) art[i]--, dfs(i,
       -1); }
  // block-cut tree (copy only if needed)
  vector<set<int>> bctree; // set to dedup
  vector<int> artid;       // art nodes to tree node (-1 for
        !arts)
  void buildBlockCutTree() { // call run first!!
    // node id: [0, nbc) -> bc, [nbc, nbc+nart) -> art
    int ntree = nbc + nart, auxid = nbc;
    bctree = vector<set<int>>(ntree);
    artid = vector<int>(n, -1);
    forn(i, n) if (art[i] > 0) {
```

```
      forall(eid, G[i]) { // edges always bc <-> art
        // depending on the problem, may want
        // to add more data on bctree edges
        bctree[auxid].insert(ve[*eid].comp);
        bctree[ve[*eid].comp].insert(auxid);
      }
      artid[i] = auxid++;
    }
  }
  int getTreeIdForGraphNode(int u) {
    if (artid[u] != -1) return artid[u];
    if (!G[u].empty()) return ve[G[u][0]].comp;
    return -1; // for nodes with no neighbours in G
  }
};
```

## 5.5 Centroid

```
// Usage: 1. Centroid(# nodes), 2. add tree edges, 3. build
     (), 4. use it
struct Centroid {
  vector<vector<int>> g;
  vector<int> vp, vsz;
  vector<bool> taken;
  Centroid(int n) : g(n), vp(n), vsz(n), taken(n) {}
  void addEdge(int a, int b) { g[a].pb(b), g[b].pb(a); }
  void build() { centroid(0, -1, -1); } // O(nlogn)
  int dfs(int node, int p) {
    vsz[node] = 1;
    forall(it, g[node]) if (*it != p && !taken[*it])
      vsz[node] += dfs(*it, node);
    return vsz[node];
  }
  void centroid(int node, int p, int cursz) {
    if (cursz == -1) cursz = dfs(node, -1);
    forall(it, g[node]) if (!taken[*it] && vsz[*it] > cursz /
        2) {
      vsz[node] = 0, centroid(*it, p, cursz);
      return;
    }
    taken[node] = true, vp[node] = p;
    // do something using node as centroid
    forall(it, g[node]) if (!taken[*it]) centroid(*it, node,
        -1);
  }
};
```

## 5.6 Diameter

```
vector<int> G[MAXN];
int n, m, p[MAXN], d[MAXN], d2[MAXN];
int bfs(int r, int* d) {
  queue<int> q;
  d[r] = 0, q.push(r);
  int v;
  while (sz(q)) {
    v = q.front();
    q.pop();
    forall(it, G[v]) if (d[*it] == -1) {
      d[*it] = d[v] + 1, p[*it] = v, q.push(*it);
    }
  }
  return v; // ultimo nodo visitado
}
vector<int> diams;
vector<ii> centros;
void diametros() {
  memset(d, -1, sizeof(d));
  memset(d2, -1, sizeof(d2));
  diams.clear(), centros.clear();
  forn(i, n) if (d[i] == -1) {
    int v, c;
    c = v = bfs(bfs(i, d2), d);
    forn(_, d[v] / 2) c = p[c];
    diams.pb(d[v]);
    if (d[v] & 1) centros.pb(ii(c, p[c]));
    else centros.pb(ii(c, c));
  }
}
```

## 5.7 Dijkstra

```
struct Dijkstra {     // WARNING: ii usually needs to be
     pair<ll, int>
  vector<vector<ii>> G; // ady. list with pairs (weight, dst
     )
  vector<ll> dist;
  // vector<int> vp; // for path reconstruction (parent of
       each node)
  int N;
  Dijkstra(int n) : G(n), N(n) {}
  void addEdge(int a, int b, ll w) { G[a].pb(mp(w, b)); }
  void run(int src) { // O(|E| log |V|)
    dist = vector<ll>(N, INF);
    // vp = vector<int>(N, -1);
    priority_queue<ii, vector<ii>, greater<ii>> Q;
    Q.push(make_pair(0, src)), dist[src] = 0;
    while (sz(Q)) {
      int node = Q.top().snd;
      ll d = Q.top().fst;
      Q.pop();
      if (d > dist[node]) continue;
      forall(it, G[node]) if (d + it->fst < dist[it->snd]) {
        dist[it->snd] = d + it->fst;
        // vp[it->snd] = node;
        Q.push(mp(dist[it->snd], it->snd));
      }
    }
  }
};
```

## 5.8 Dynamic Connectivity

```
struct UnionFind {
  int n, comp;
  vector<int> pre, si, c;
  UnionFind(int n = 0) : n(n), comp(n), pre(n), si(n, 1) {
    forn(i, n) pre[i] = i;
  }
  int find(int u) { return u == pre[u] ? u : find(pre[u]); }
  bool merge(int u, int v) {
    if ((u = find(u)) == (v = find(v))) return false;
    if (si[u] < si[v]) swap(u, v);
    si[u] += si[v], pre[v] = u, comp--, c.pb(v);
    return true;
  }
  int snap() { return sz(c); }
  void rollback(int snap) {
    while (sz(c) > snap) {
      int v = c.back();
      c.pop_back();
      si[pre[v]] -= si[v], pre[v] = v, comp++;
    }
  }
};
enum { ADD, DEL, QUERY };
struct Query {
  int type, u, v;
};
struct DynCon { // bidirectional graphs; create vble as
     DynCon name(cant_nodos)
  vector<Query> q;
  UnionFind dsu;
  vector<int> match, res;
```

```
  // se puede no usar cuando hay identificador para cada
      arista (mejora poco)
  map<ii, int> last;
  DynCon(int n = 0) : dsu(n) {}
  void add(int u, int v) // to add an edge
  {
    if (u > v) swap(u, v);
    q.pb((Query){ADD, u, v}), match.pb(-1);
    last[ii(u, v)] = sz(q) - 1;
  }
  void remove(int u, int v) // to remove an edge
  {
    if (u > v) swap(u, v);
    q.pb((Query){DEL, u, v});
    int prev = last[ii(u, v)];
    match[prev] = sz(q) - 1;
    match.pb(prev);
  }
  void query() // to add a question (query) type of query
  {
    q.pb((Query){QUERY, -1, -1}), match.pb(-1);
  }
  void process() // call this to process queries in the
      order of q
  {
    forn(i, sz(q)) if (q[i].type == ADD && match[i] == -1)
        match[i] = sz(q);
    go(0, sz(q));
  }
  void go(int l, int r) {
    if (l + 1 == r) {
      if (q[l].type == QUERY) // Aqui responder la query
          usando el dsu!
        res.pb(dsu.comp);    // aqui query=cantidad de
            componentes conexas
      return;
    }
    int s = dsu.snap(), m = (l + r) / 2;
    forr(i, m, r) if (match[i] != -1 && match[i] < l) dsu.
        merge(q[i].u, q[i].v);
    go(l, m);
    dsu.rollback(s);
    s = dsu.snap();
    forr(i, l, m) if (match[i] != -1 && match[i] >= r)
      dsu.merge(q[i].u, q[i].v);
    go(m, r);
    dsu.rollback(s);
  }
};
```

## 5.9 Euler Path

```
// Be careful with nodes with degree 0 when solving your
    problem, the
// comments below assume that there are no nodes with degree
    0.
//
// Euler [path/cycle] exists in a bidirectional graph iff
    the graph is
// connected and at most [2/0] nodes have odd degree. The
    path must
// start from an odd degree vertex when there are 2.
//
// Euler [path/cycle] exists in a directed graph iff the
    graph is
// [connected when making edges bidirectional / a single SCC
    ], and
// at most [1/0] node have indg - outdg = 1, at most [1/0]
    node have
// outdg - indg = 1, all the other nodes have indg = outdg.
    The path
// must start from the node with outdg - indg = 1, when
    there is one.
//
// Directed version (uncomment commented code for undirected
    )
struct edge {
  int y;
  // list<edge>::iterator rev;
  edge(int yy) : y(yy) {}
};
struct EulerPath {
  vector<list<edge>> g;
  EulerPath(int n) : g(n) {}
  void addEdge(int a, int b) {
    g[a].push_front(edge(b));
    // auto ia = g[a].begin();
    // g[b].push_front(edge(a));
    // auto ib = g[b].begin();
    // ia->rev=ib, ib->rev=ia;
  }
  vector<int> p;
  void go(int x) {
    while (sz(g[x])) {
      int y = g[x].front().y;
      // g[y].erase(g[x].front().rev);
      g[x].pop_front();
      go(y);
    }
    p.push_back(x);
  }
  vector<int> getPath(int x) { // get a path that starts
      from x
    // you must check that a path exists from x before
        calling get_path!
    p.clear(), go(x);
    reverse(p.begin(), p.end());
    return p;
  }
};
```

## 5.10 Floyd

```
// Min path between every pair of nodes in directed graph
// G[i][j] initially needs weight of edge (i, j) or INF
// be careful with multiedges and loops when assigning to G
int G[MAX_N][MAX_N];
void floyd() { // O(N^3)
  forn(k, N) forn(i, N) if (G[i][k] != INF) forn(j, N) if (G
      [k][j] != INF)
    G[i][j] = min(G[i][j], G[i][k] + G[k][j]);
}
bool inNegCycle(int v) { return G[v][v] < 0; }
// checks if there's a neg. cycle in path from a to b
bool hasNegCycle(int a, int b) {
  forn(i, N) if (G[a][i] != INF && G[i][i] < 0 && G[i][b] !=
       INF) return true;
  return false;
}
```

## 5.11 Heavy Light Decomposition

```
// Usage: 1. HLD(# nodes) 2. add tree edges 3. build() 4.
    use it
struct HLD {
  vector<int> w, p, dep; // weight,father,depth
  vector<vector<int>> g;
  HLD(int n) : w(n), p(n), dep(n), g(n), pos(n), head(n) {}
  void addEdge(int a, int b) { g[a].pb(b), g[b].pb(a); }
  void build() { p[0] = -1, dep[0] = 0, dfs1(0), curpos = 0,
       hld(0, -1); }
  void dfs1(int x) {
    w[x] = 1;
    for (int y : g[x]) if (y != p[x]) {
      p[y] = x, dep[y] = dep[x] + 1, dfs1(y);
      w[x] += w[y];
    }
```

```
  }
  int curpos;
  vector<int> pos, head;
  void hld(int x, int c) {
    if (c < 0) c = x;
    pos[x] = curpos++, head[x] = c;
    int mx = -1;
    for (int y : g[x]) if (y != p[x] && (mx < 0 || w[mx] < w[
        y])) mx = y;
    if (mx >= 0) hld(mx, c);
    for (int y : g[x]) if (y != mx && y != p[x]) hld(y, -1);
  }
  // Here ST is segtree static/dynamic/lazy or other DS
      according to problem
  tipo query(int x, int y, ST& st) { // ST tipo
    tipo r = neutro;
    while (head[x] != head[y]) {
      if (dep[head[x]] > dep[head[y]]) swap(x, y);
      r = oper(r, st.get(pos[head[y]], pos[y] + 1)); // ST
          oper
      y = p[head[y]];
    }
    if (dep[x] > dep[y]) swap(x, y);      // now x is lca
    r = oper(r, st.get(pos[x], pos[y] + 1)); // ST oper
    return r;
  }
};
// for point updates: st.set(pos[x], v) (x = node, v = new
    value)
// for lazy range updates: something similar to the query
    method
// for queries on edges: - assign values of edges to "child"
    node
//                      - change pos[x] to pos[x]+1 in query (
    line 34)
```

## 5.12 Kosaraju

```
struct Kosaraju {
  vector<vector<int>> G, gt;
  // nodos 0...N-1 ; componentes 0...cantcomp-1
  int N, cantcomp;
  vector<int> comp, used;
  stack<int> pila;
  Kosaraju(int n) : G(n), gt(n), N(n), comp(n) {}
  void addEdge(int a, int b) { G[a].pb(b), gt[b].pb(a); }
  void dfs1(int nodo) {
    used[nodo] = 1;
    forall(it, G[nodo]) if (!used[*it]) dfs1(*it);
    pila.push(nodo);
  }
  void dfs2(int nodo) {
    used[nodo] = 2;
    comp[nodo] = cantcomp - 1;
    forall(it, gt[nodo]) if (used[*it] != 2) dfs2(*it);
  }
  void run() {
    cantcomp = 0;
    used = vector<int>(N, 0);
    forn(i, N) if (!used[i]) dfs1(i);
    while (!pila.empty()) {
      if (used[pila.top()] != 2) {
        cantcomp++;
        dfs2(pila.top());
      }
      pila.pop();
    }
  }
};
```

## 5.13 Kruskal

```
struct Edge {
  int a, b, w;
};
bool operator<(const Edge& a, const Edge& b) { return a.w <
    b.w; }
// Minimun Spanning Tree in O(E log E)
ll kruskal(vector<Edge> &E, int n) {
  ll cost = 0; sort(E.begin(), E.end());
  UnionFind uf(n);
  forall(it, E) if(!uf.join(it->a, it->b))
    cost += it->w;
  return cost;
}
```

## 5.14 Lca

```
#define lg(x) (31 - __builtin_clz(x)) //=floor(log2(x))
// Usage: 1) Create 2) Add edges 3) Call build 4) Use
struct LCA {
  int N, LOGN, ROOT;
  // vp[k][node] holds the 2^k ancestor of node
  // L[v] holds the level of v
  vector<int> L;
  vector<vector<int>> vp, G;
  LCA(int n, int root) : N(n), LOGN(lg(n) + 1), ROOT(root),
      L(n), G(n) {
    // Here you may want to replace the default from root to
        other
    // value, like maybe -1.
    vp = vector<vector<int>>(LOGN, vector<int>(n, root));
  }
  void addEdge(int a, int b) { G[a].pb(b), G[b].pb(a); }
  void dfs(int node, int p, int lvl) {
    vp[0][node] = p, L[node] = lvl;
    forall(it, G[node]) if (*it != p) dfs(*it, node, lvl + 1)
        ;
  }
  void build() {
    // Here you may also want to change the 2nd param to -1
    dfs(ROOT, ROOT, 0);
    forn(k, LOGN - 1) forn(i, N) vp[k + 1][i] = vp[k][vp[k][i
        ]];
  }
  int climb(int a, int d) { // O(lgn)
    if (!d) return a;
    dforn(i, lg(L[a]) + 1) if (1 << i <= d) a = vp[i][a], d
        -= 1 << i;
    return a;
  }
  int lca(int a, int b) { // O(lgn)
    if (L[a] < L[b]) swap(a, b);
    a = climb(a, L[a] - L[b]);
    if (a == b) return a;
    dforn(i, lg(L[a]) + 1) if (vp[i][a] != vp[i][b]) a = vp[i
        ][a], b = vp[i][b];
    return vp[0][a];
  }
  int dist(int a, int b) { // returns distance between nodes
    return L[a] + L[b] - 2 * L[lca(a, b)];
  }
};
```

## 5.15 Prim

```
vector<ii> G[MAXN];
bool taken[MAXN];
priority_queue<ii, vector<ii>, greater<ii> > pq; // min heap
void process(int v) {
  taken[v] = true;
  forall(e, G[v]) if (!taken[e->second]) pq.push(*e);
}
// Minimun Spanning Tree in O(n^2)
ll prim() {
```

```
  zero(taken);
  process(0);
  ll cost = 0;
  while (sz(pq)) {
    ii e = pq.top();
    pq.pop();
    if (!taken[e.second]) cost += e.first, process(e.second);
  }
  return cost;
}
```

## 5.16 Tree Reroot

```
struct Edge {
  int u, v; // maybe add more data, depending on the problem
};
// USAGE:
// 1- define all the logic in SubtreeData
// 2- create a reroot and add all the edges
// 3- call Reroot.run()
struct SubtreeData {
  // Define here what data you need for each subtree
  SubtreeData() {} // just empty
  SubtreeData(int node) {
    // Initialize the data here as if this new subtree
    // has size 1, and its only node is 'node'
  }
  void merge(Edge* e, SubtreeData& s) {
    // Modify this subtree's data to reflect that 's' is
        being
    // merged into 'this' through the edge 'e'.
    // When e == NULL, then no edge is present, but then, '
        this'
    // and 's' have THE SAME ROOT (be CAREFUL with this).
    // These 2 subtrees don't have any other shared nodes nor
         edges.
  }
};
struct Reroot {
  int N; // # of nodes
  // vresult[i] = SubtreeData for the tree where i is the
      root
  // this should be what you need as result
  vector<SubtreeData> vresult, vs;
  vector<Edge> ve;
  vector<vector<int>> g; // the tree as a bidirectional
      graph
  Reroot(int n) : N(n), vresult(n), vs(n), ve(0), g(n) {}
  void addEdge(Edge e) { // will be added in both ways
    g[e.u].pb(sz(ve));
    g[e.v].pb(sz(ve));
    ve.pb(e);
  }
  void dfs1(int node, int p) {
    vs[node] = SubtreeData(node);
    forall(e, g[node]) {
      int nxt = node ^ ve[*e].u ^ ve[*e].v;
      if (nxt == p) continue;
      dfs1(nxt, node);
      vs[node].merge(&ve[*e], vs[nxt]);
    }
  }
  void dfs2(int node, int p, SubtreeData fromp) {
    vector<SubtreeData> vsuf(sz(g[node]) + 1);
    int pos = sz(g[node]);
    SubtreeData pref = vsuf[pos] = SubtreeData(node);
    vresult[node] = vs[node];
    dforall(e, g[node]) { // dforall = forall in reverse
      pos--;
      vsuf[pos] = vsuf[pos + 1];
      int nxt = node ^ ve[*e].u ^ ve[*e].v;
      if (nxt == p) {
        pref.merge(&ve[*e], fromp);
        vresult[node].merge(&ve[*e], fromp);
        continue;
      }
      vsuf[pos].merge(&ve[*e], vs[nxt]);
    }
    assert(pos == 0);
    forall(e, g[node]) {
      pos++;
      int nxt = node ^ ve[*e].u ^ ve[*e].v;
      if (nxt == p) continue;
      SubtreeData aux = pref;
      aux.merge(NULL, vsuf[pos]);
      dfs2(nxt, node, aux);
      pref.merge(&ve[*e], vs[nxt]);
    }
  }
  void run() {
    dfs1(0, 0);
    dfs2(0, 0, SubtreeData());
  }
};
```

## 5.17 Virtual Tree

```
// Usage: (VT = VirtualTree)
// 1- Build the LCA and use it for creating 1 VT instance
// 2- Call updateVT every time you want
// 3- Between calls of updateVT you probably want to use the
      tree, imp
// and VTroot variables from this struct to solve your
     problem
struct VirtualTree {
  // n = #nodes full tree
  // curt used for computing tin and tout
  int n, curt;
  LCA* lca;
  vector<int> tin, tout;
  vector<vector<ii>> tree; // {node, dist}, only parent ->
      child dire
  // imp[i] = true iff i was part of 'newv' from last time
      that
  // updateVT was called (note that LCAs are not imp)
  vector<bool> imp;
  void dfs(int node, int p) {
    tin[node] = curt++;
    forall(it, lca->G[node]) if (*it != p) dfs(*it, node);
    tout[node] = curt++;
  }
  VirtualTree(LCA* l) { // must call l.build() before
    lca = l, n = sz(l->G), lca = l, curt = 0;
    tin.rsz(n), tout.rsz(n), tree.rsz(n), imp.rsz(n);
    dfs(l->ROOT, l->ROOT);
  }
  bool isAncestor(int a, int b) { return tin[a] < tin[b] &&
       tout[a] > tout[b]; }
  int VTroot = -1; // root of the current VT
  vector<int> v;  // list of nodes of current VT (includes
      LCAs)
  void updateVT(vector<int>& newv) { // O(sz(newv)*log)
    assert(!newv.empty());         // this method assumes non
        -empty
    auto cmp = [this](int a, int b) { return tin[a] < tin[b];
         };
    forn(i, sz(v)) tree[v[i]].clear(), imp[v[i]] = false;
    v = newv;
    sort(v.begin(), v.end(), cmp);
    set<int> s;
    forn(i, sz(v)) s.insert(v[i]), imp[v[i]] = true;
    forn(i, sz(v) - 1) s.insert(lca->lca(v[i], v[i + 1]));
    v.clear();
    forall(it, s) v.pb(*it);
    sort(v.begin(), v.end(), cmp);
    stack<int> st;
    forn(i, sz(v)) {
```

```
    while (!st.empty() && !isAncestor(st.top(), v[i])) st.
         pop();
    assert(i == 0 || !st.empty());
    if (!st.empty()) tree[st.top()].pb(mp(v[i], lca->dist(
         st.top(), v[i])));
    st.push(v[i]);
  }
  VTroot = v[0];
 }
};
```

# 6 Math

## 6.1 Combinatorics

```
void cargarComb() { // O(MAXN^2)
 forn(i, MAXN) {  // comb[i][k]=i tomados de a k = i!/(k!*(
      i-k)!)
  comb[0][i] = 0;
  comb[i][0] = comb[i][i] = 1;
  forr(k, 1, i) comb[i][k] = (comb[i - 1][k - 1] + comb[i -
       1][k]) % MOD;
 }
}
ll lucas(ll n, ll k, int p) { // (n,k)%p, needs comb[p][p]
    precalculated
 ll aux = 1;
 while (n + k) {
  aux = (aux * comb[n % p][k % p]) % p;
  n /= p, k /= p;
 }
 return aux;
}
```

## 6.2 Crt Euclid

```
// ecuacion diofantica lineal
// sea d=gcd(a,b); la ecuacion a * x + b * y = c tiene
    soluciones enteras si
// d|c. La siguiente funcion nos sirve para esto. De forma
    general sera:
// x = x0 + (b/d)n   x0 = xx*c/d
// y = y0 - (a/d)n   y0 = yy*c/d
// la funcion devuelve d
ll euclid(ll a, ll b, ll &xx, ll &yy) {
 if (!b) return xx = 1, yy = 0, a;
 ll d = euclid(b, a % b, yy, xx);
 return yy -= a / b * xx, d;
}

// Chinese remainder theorem (special case): find z such
    that
// z % m = a, z % n = b. Here, z is unique modulo M = lcm(m,
     n)
// Return -1 when there is no solution
// CRT is associative and idempotent
ll crt(ll a, ll m, ll b, ll n) {
 if (n > m) swap(a, b), swap(m, n);
 ll x, y, g = euclid(m, n, x, y);
 if ((a - b) % g != 0) return -1; // comment to get RTE
      when there is no solution
 assert((a - b) % g == 0);
 x = (b - a) % n * x % n / g * m + a;
 return x < 0 ? x + m * n / g : x;
}

// Chinese remainder theorem: find z such that z % m[i] = r[
    i] for all i.
// Note that the solution is unique modulo M = lcm_i (m[i]).
// Return z. On failure, return -1.
// Note that we do not require the m[i]'s to be relatively
    prime.
ll crt(const vector<ll>& r, const vector<ll>& m) {
 assert(sz(r) == sz(m));
 ll ret = r[0], l = m[0];
 forr(i, 1, sz(m)) {
 ret = crt(ret, l, r[i], m[i]);
 l = lcm(r[i],m[i]);
 if (ret == -1) break;
 }
 return ret;
}
```

## 6.3 Discrete Log

```
// O(sqrt(m)*log(m))
// returns x such that a^x = b (mod m) or -1 if inexistent
ll discrete_log(ll a, ll b, ll m) {
 a %= m, b %= m;
 if (b == 1) return 0;
 int cnt = 0;
 ll tmp = 1;
 for (ll g = __gcd(a, m); g != 1; g = __gcd(a, m)) {
  if (b % g) return -1;
  m /= g, b /= g;
  tmp = tmp * a / g % m;
  ++cnt;
  if (b == tmp) return cnt;
 }
 map<ll, int> w;
 int s = (int)ceil(sqrt(m));
 ll base = b;
 forn(i, s) {
  w[base] = i;
  base = base * a % m;
 }
 base = expMod(a, s, m);
 ll key = tmp;
 forr(i, 1, s + 2) {
  key = base * key % m;
  if (w.count(key)) return i * s - w[key] + cnt;
 }
 return -1;
}
```

## 6.4 Fft

```
typedef __int128 T;
typedef double ld;
typedef vector<T> poly;
const T MAXN = (1 << 21); // MAXN must be power of 2,
// MOD-1 needs to be a multiple of MAXN, big mod and
    primitive root for NTT
const T MOD = 2305843009255636993LL, RT = 5;
// const T MOD = 998244353, RT = 3;

// NTT
struct CD {
 T x;
 CD(T x_) : x(x_) {}
 CD() {}
};
T mulmod(T a, T b) { return a * b % MOD; }
T addmod(T a, T b) {
 T r = a + b;
 if (r >= MOD) r -= MOD;
 return r;
}
T submod(T a, T b) {
 T r = a - b;
 if (r < 0) r += MOD;
 return r;
}
```

```
CD operator*(const CD& a, const CD& b) { return CD(mulmod(a.
    x, b.x)); }
CD operator+(const CD& a, const CD& b) { return CD(addmod(a.
    x, b.x)); }
CD operator-(const CD& a, const CD& b) { return CD(submod(a.
    x, b.x)); }
vector<T> rts(MAXN + 9, -1);
CD root(int n, bool inv) {
  T r = rts[n] < 0 ? rts[n] = expMod(RT, (MOD - 1) / n) :
      rts[n];
  return CD(inv ? expMod(r, MOD - 2) : r);
}

// FFT
// struct CD {
//   ld r, i;
//   CD(ld r_ = 0, ld i_ = 0) : r(r_), i(i_) {}
//   ld real() const { return r; }
//   void operator/=(const int c) { r /= c, i /= c; }
// };
// CD operator*(const CD& a, const CD& b) {
//   return CD(a.r * b.r - a.i * b.i, a.r * b.i + a.i * b.r);
// }
// CD operator+(const CD& a, const CD& b) { return CD(a.r +
    b.r, a.i + b.i); }
// CD operator-(const CD& a, const CD& b) { return CD(a.r -
    b.r, a.i - b.i); }
// const ld pi = acos(-1.0);

CD cp1[MAXN + 9], cp2[MAXN + 9];
int R[MAXN + 9];
void dft(CD* a, int n, bool inv) {
  forn(i, n) if (R[i] < i) swap(a[R[i]], a[i]);
  for (int m = 2; m <= n; m *= 2) {
    // ld z=2*pi/m*(inv?-1:1); // FFT
    // CD wi=CD(cos(z),sin(z)); // FFT
    CD wi = root(m, inv); // NTT
    for (int j = 0; j < n; j += m) {
      CD w(1);
      for (int k = j, k2 = j + m / 2; k2 < j + m; k++, k2++)
          {
        CD u = a[k];
        CD v = a[k2] * w;
        a[k] = u + v;
        a[k2] = u - v;
        w = w * wi;
      }
    }
  }
  // if(inv) forn(i,n) a[i]/=n; // FFT
  if (inv) { // NTT
    CD z(expMod(n, MOD - 2));
    forn(i, n) a[i] = a[i] * z;
  }
}
poly multiply(poly& p1, poly& p2) {
  int n = sz(p1) + sz(p2) + 1;
  int m = 1, cnt = 0;
  while (m <= n) m += m, cnt++;
  forn(i, m) {
    R[i] = 0;
    forn(j, cnt) R[i] = (R[i] << 1) | ((i >> j) & 1);
  }
  forn(i, m) cp1[i] = 0, cp2[i] = 0;
  forn(i, sz(p1)) cp1[i] = p1[i];
  forn(i, sz(p2)) cp2[i] = p2[i];
  dft(cp1, m, false);
  dft(cp2, m, false);
  forn(i, m) cp1[i] = cp1[i] * cp2[i];
  dft(cp1, m, true);
  poly res;
  n -= 2;
  // forn(i,n) res.pb((T) floor(cp1[i].real()+0.5)); // FFT
  forn(i, n) res.pb(cp1[i].x); // NTT
  return res;
}
```

## 6.5 Fraction

```
struct frac {
  int p, q;
  frac(int p = 0, int q = 1) : p(p), q(q) { norm(); }
  void norm() {
    int a = gcd(q, p);
    if (a) p /= a, q /= a;
    else q = 1;
    if (q < 0) q = -q, p = -p;
  }
  frac operator+(const frac& o) {
    int a = gcd(o.q, q);
    return frac(p * (o.q / a) + o.p * (q / a), q * (o.q / a))
        ;
  }
  frac operator-(const frac& o) {
    int a = gcd(o.q, q);
    return frac(p * (o.q / a) - o.p * (q / a), q * (o.q / a))
        ;
  }
  frac operator*(frac o) {
    int a = gcd(o.p, q), b = gcd(p, o.q);
    return frac((p / b) * (o.p / a), (q / a) * (o.q / b));
  }
  frac operator/(frac o) {
    int a = gcd(o.q, q), b = gcd(p, o.p);
    return frac((p / b) * (o.q / a), (q / a) * (o.p / b));
  }
  bool operator<(const frac& o) const { return p * o.q < o.p
      * q; }
  bool operator==(frac o) { return p == o.p && q == o.q; }
};
```

## 6.6 Gauss Jordan Bitset

```
// https://cp-algorithms.com/linear_algebra/linear-system-
    gauss.html
// special case of gauss_jordan_mod with mod=2, bitset for
    efficiency
// finds lexicograhically minimal solution (0 < 1, False <
    True)
// for lexicographically maximal change your solution model
    accordingly
int gauss(vector<bitset<N> > a, int n, int m, bitset<N>& ans
    ) {
  vector<int> where(m, -1);
  for (int col = m - 1, row = 0; col >= 0 && row < n; --col)
      {
    for (int i = row; i < n; ++i)
      if (a[i][col]) {
        swap(a[i], a[row]);
        break;
      }
    if (!a[row][col]) continue;
    where[col] = row;

    for (int i = 0; i < n; ++i)
      if (i != row && a[i][col]) a[i] ^= a[row];
    ++row;
  }
  ans.reset();
  forn(i, m) if (where[i] != -1) { ans[i] = a[where[i]][m] &
      a[where[i]][i]; }
  forn(i, n) if ((ans & a[i]).count() % 2 != a[i][m]) return
      0;
  forn(i, m) if (where[i] == -1) return INF;
  return 1;
}
```

## 6.7 Gauss Jordan Mod

```
// inv -> modular inverse function
// disclaimer: not very well tested, but got AC on a problem
    with this
int gauss(vector<vector<int> > a, vector<int>& ans) {
  int n = (int)a.size();
  int m = (int)a[0].size() - 1;

  vector<int> where(m, -1);
  for (int col = 0, row = 0; col < m && row < n; ++col) {
    int sel = row;
    for (int i = row; i < n; ++i)
      if (a[i][col] > a[sel][col]) sel = i;
    if (a[sel][col] == 0) continue;
    for (int i = col; i <= m; ++i) swap(a[sel][i], a[row][i])
        ;
    where[col] = row;

    for (int i = 0; i < n; ++i)
      if (i != row) {
        int c = (a[i][col] * inv(a[row][col])) % MOD;
        for (int j = col; j <= m; ++j)
          a[i][j] = (a[i][j] - a[row][j] * c % MOD + MOD) %
              MOD;
      }
    ++row;
  }
  ans.clear();
  ans.rsz(m, 0);
  for (int i = 0; i < m; ++i)
    if (where[i] != -1) ans[i] = (a[where[i]][m] * inv(a[
        where[i]][i])) % MOD;
  for (int i = 0; i < n; ++i) {
    int sum = 0;
    for (int j = 0; j < m; ++j) sum = (sum + ans[j] * a[i][j
        ]) % MOD;
    if ((sum - a[i][m] + MOD) % MOD != 0) return 0;
  }

  for (int i = 0; i < m; ++i)
    if (where[i] == -1) return INF;
  return 1;
}
```

## 6.8 Gauss Jordan

```
// https://cp-algorithms.com/linear_algebra/linear-system-
    gauss.html
const double EPS = 1e-9;
const int INF = 2; // a value to indicate infinite solutions

int gauss(vector<vector<double> > a, vector<double>& ans) {
  int n = (int)a.size();
  int m = (int)a[0].size() - 1;

  vector<int> where(m, -1);
  for (int col = 0, row = 0; col < m && row < n; ++col) {
    int sel = row;
    for (int i = row; i < n; ++i)
      if (abs(a[i][col]) > abs(a[sel][col])) sel = i;
    if (abs(a[sel][col]) < EPS) continue;
    for (int i = col; i <= m; ++i) swap(a[sel][i], a[row][i])
        ;
    where[col] = row;

    for (int i = 0; i < n; ++i)
      if (i != row) {
        double c = a[i][col] / a[row][col];
        for (int j = col; j <= m; ++j) a[i][j] -= a[row][j] *
            c;
      }
    ++row;
  }

  ans.assign(m, 0);
  for (int i = 0; i < m; ++i)
    if (where[i] != -1) ans[i] = a[where[i]][m] / a[where[i
        ]][i];
  for (int i = 0; i < n; ++i) {
    double sum = 0;
    for (int j = 0; j < m; ++j) sum += ans[j] * a[i][j];
    if (abs(sum - a[i][m]) > EPS) return 0;
  }

  for (int i = 0; i < m; ++i)
    if (where[i] == -1) return INF;
  return 1;
}
```

## 6.9 Karatsuba

```
template<typename T> void rec_kara(T* a, int one, T* b, int
    two, T* r) {
 if(min(one, two) <= 20) { // must be at least "<= 1"
  forn(i, one) forn(j, two) r[i+j] += a[i] * b[j];
  return;
 }
 const int x = min(one, two);
 if(one < two) rec_kara(a, x, b + x, two - x, r + x);
 if(two < one) rec_kara(a + x, one - x, b, x, r + x);
 const int n = (x + 1) / 2, right = x / 2;
 vector<T> tu(2 * n);
 rec_kara(a, n, b, n, tu.data());
 forn(i, 2*n-1) {
  r[i] += tu[i];
  r[i+n] -= tu[i];
  tu[i] = 0;
 }
 rec_kara(a + n, right, b + n, right, tu.data());
 forn(i, 2*right-1) r[i+n] -= tu[i], r[i+2*n] += tu[i];
 tu[n-1] = a[n-1]; tu[2*n-1] = b[n-1];
 forn(i, right) tu[i] = a[i]+a[i+n], tu[i+n] = b[i]+b[i+n];
 rec_kara(tu.data(), n, tu.data() + n, n, r + n);
}
template<typename T> vector<T> multiply(vector<T> a, vector<
    T> b) {
 if(a.empty() || b.empty()) return {};
 vector<T> r(a.size() + b.size() - 1);
 rec_kara(a.data(), a.size(), b.data(), b.size(), r.data());
 return r;
}
```

## 6.10 Matrix Exp

```
typedef ll tipo; // maybe use double or other depending on
    the problem
struct Mat {
  int N; // square matrix
  vector<vector<tipo>> m;
  Mat(int n) : N(n), m(n, vector<tipo>(n, 0)) {}
  vector<tipo>& operator[](int p) { return m[p]; }
  Mat operator*(Mat& b) { // O(N^3), multiplication
    assert(N == b.N);
    Mat res(N);
    forn(i, N) forn(j, N) forn(k, N) // remove MOD if not
        needed
      res[i][j] = (res[i][j] + m[i][k] * b[k][j]) % MOD;
    return res;
  }
  Mat operator^(int k) { // O(N^3 * logk), exponentiation
    Mat res(N), aux = *this;
    forn(i, N) res[i][i] = 1;
    while (k)
      if (k & 1) res = res * aux, k--;
      else aux = aux * aux, k /= 2;
    return res;
```

```
  }
};
```

## 6.11 Modular Inverse

```
#define MAXMOD 15485867
ll inv[MAXMOD];   // inv[i]*i=1 mod MOD
void calc(int p) { // O(p)
  inv[1] = 1;
  forr(i, 2, p) inv[i] = p - ((p / i) * inv[p % i]) % p;
}
int inverso(int x) {                    // O(log MOD)
  return expMod(x, eulerPhi(MOD) - 1); // si mod no es primo
      (sacar a mano)
  return expMod(x, MOD - 2);          // si mod es primo
}

// fact[i] = i!%MOD and ifact[i] = 1/(i!)%MOD
// inv is modular inverse function
ll fact[MAXN], ifact[MAXN];
void build_facts() { // O(MAXN)
  fact[0] = 1;
  forr(i, 1, MAXN) fact[i] = fact[i - 1] * i % MOD;
  ifact[MAXN - 1] = inverso(fact[MAXN - 1]);
  dforn(i, MAXN - 1) ifact[i] = ifact[i + 1] * (i + 1) % MOD
      ;
  return;
}
// n! / k!*(n-k)!
// assumes 0 <= n < MAXN
ll comb(ll n, ll k) {
  if (k < 0 || n < k) return 0;
  return fact[n] * ifact[k] % MOD * ifact[n - k] % MOD;
}
```

## 6.12 Modular Operations

```
const ll MOD = 1000000007; // Change according to problem
// Only needed for MOD > 2^31
// Actually, for 2^31 < MOD < 2^63 it's usually better to
     use __int128
// and normal multiplication (* operator) instead of mulMod
// returns (a*b) %c, and minimize overfloor
ll mulMod(ll a, ll b, ll m = MOD) { // O(log b)
  ll x = 0, y = a % m;
  while (b > 0) {
    if (b % 2 == 1) x = (x + y) % m;
    y = (y * 2) % m;
    b /= 2;
  }
  return x % m;
}
ll expMod(ll b, ll e, ll m = MOD) { // O(log e)
  if (e < 0) return 0;
  ll ret = 1;
  while (e) {
    if (e & 1) ret = ret * b % m; // ret = mulMod(ret,b,m);
        //if needed
    b = b * b % m;              // b = mulMod(b,b,m);
    e >>= 1;
  }
  return ret;
}
ll sumMod(ll a, ll b, ll m = MOD) {
  a %= m;
  b %= m;
  if (a < 0) a += m;
  if (b < 0) b += m;
  return (a + b) % m;
}
ll difMod(ll a, ll b, ll m = MOD) {
  a %= m;
  b %= m;
  if (a < 0) a += m;
  if (b < 0) b += m;
  ll ret = a - b;
  if (ret < 0) ret += m;
  return ret;
}
ll divMod(ll a, ll b, ll m = MOD) { return mulMod(a, inverso
     (b), m); }
```

## 6.13 Phollard Rho

```
bool es_primo_prob(ll n, int a) {
  if (n == a) return true;
  ll s = 0, d = n - 1;
  while (d % 2 == 0) s++, d /= 2;
  ll x = expMod(a, d, n);
  if ((x == 1) || (x + 1 == n)) return true;
  forn(i, s - 1) {
    x = (x * x) % n; // mulMod(x, x, n); In most cases, it is
         necessary to use mulMod to avoid TLE
    if (x == 1) return false;
    if (x + 1 == n) return true;
  }
  return false;
}
bool rabin(ll n) { // devuelve true si n es primo
  if (n == 1) return false;
  const int ar[] = {2, 3, 5, 7, 11, 13, 17, 19, 23};
  forn(j, 9) if (!es_primo_prob(n, ar[j])) return false;
  return true;
}
ll rho(ll n) {
  if ((n & 1) == 0) return 2;
  ll x = 2, y = 2, d = 1;
  ll c = rand() % n + 1;
  while (d == 1) {
    // may want to avoid mulMod if possible
    // maybe replace with * operator using __int128?
    x = (mulMod(x, x, n) + c) % n;
    y = (mulMod(y, y, n) + c) % n;
    y = (mulMod(y, y, n) + c) % n;
    if (x - y >= 0) d = gcd(n, x - y);
    else d = gcd(n, y - x);
  }
  return d == n ? rho(n) : d;
}
void factRho(ll n, map<ll, ll>& f) { // O ((n ^ 1/4) * logn)
  if (n == 1) return;
  if (rabin(n)) {
    f[n]++;
    return;
  }
  ll factor = rho(n);
  factRho(factor, f);
  factRho(n / factor, f);
}
```

## 6.14 Prime Functions

```
#define MAXP 100000 // no necesariamente primo
int criba[MAXP + 1];
void crearCriba() {
  int w[] = {4, 2, 4, 2, 4, 6, 2, 6};
  for (int p = 25; p <= MAXP; p += 10) criba[p] = 5;
  for (int p = 9; p <= MAXP; p += 6) criba[p] = 3;
  for (int p = 4; p <= MAXP; p += 2) criba[p] = 2;
  for (int p = 7, cur = 0; p * p <= MAXP; p += w[cur++ & 7])
    if (!criba[p])
      for (int j = p * p; j <= MAXP; j += (p << 1))
        if (!criba[j]) criba[j] = p;
}
vector<int> primos;
```

```
void buscarPrimos() {
  crearCriba();
  forr(i, 2, MAXP + 1) if (!criba[i]) primos.push_back(i);
}

// factoriza bien numeros hasta MAXP^2, llamar a
     buscarPrimos antes
void fact(ll n, map<ll, ll>& f) { // O (cant primos)
  forall(p, primos) {
    while (!(n % *p)) {
      f[*p]++; // divisor found
      n /= *p;
    }
  }
  if (n > 1) f[n]++;
}

// factoriza bien numeros hasta MAXP, llamar crearCriba
     antes
void fact2(ll n, map<ll, ll>& f) { // O (lg n)
  while (criba[n]) {
    f[criba[n]]++;
    n /= criba[n];
  }
  if (n > 1) f[n]++;
}

// Usar asi: divisores(fac, divs, fac.begin()); NO ESTA
     ORDENADO
void divisores(map<ll, ll>& f, vector<ll>& divs, map<ll, ll
     >::iterator it,
          ll n = 1) {
  if (it == f.begin()) divs.clear();
  if (it == f.end()) {
    divs.pb(n);
    return;
  }
  ll p = it->fst, k = it->snd;
  ++it;
  forn(_, k + 1) divisores(f, divs, it, n), n *= p;
}
ll cantDivs(map<ll, ll>& f) {
  ll ret = 1;
  forall(it, f) ret *= (it->second + 1);
  return ret;
}
ll sumDivs(map<ll, ll>& f) {
  ll ret = 1;
  forall(it, f) {
    ll pot = 1, aux = 0;
    forn(i, it->snd + 1) aux += pot, pot *= it->fst;
    ret *= aux;
  }
  return ret;
}

ll eulerPhi(ll n) { // con criba: O(lg n)
  map<ll, ll> f;
  fact(n, f);
  ll ret = n;
  forall(it, f) ret -= ret / it->first;
  return ret;
}
ll eulerPhi2(ll n) { // O (sqrt n)
  ll r = n;
  forr(i, 2, n + 1) {
    if ((ll)i * i > n) break;
    if (n % i == 0) {
      while (n % i == 0) n /= i;
      r -= r / i;
    }
  }
  if (n != 1) r -= r / n;
  return r;
}
```

## 6.15 Simplex

```
typedef double tipo;
typedef vector<tipo> vt;
// maximize c^T x s.t. Ax<=b, x>=0, returns pair (max val,
     solution vector)
pair<tipo, vt> simplex(vector<vt> A, vt b, vt c) {
  int n = sz(b), m = sz(c);
  tipo z = 0.;
  vector<int> X(m), Y(n);
  forn(i, m) X[i] = i;
  forn(i, n) Y[i] = i + m;
  auto pivot = [&](int x, int y) {
    swap(X[y], Y[x]);
    b[x] /= A[x][y];
    forn(i, m) if (i != y) A[x][i] /= A[x][y];
    A[x][y] = 1 / A[x][y];
    forn(i, n) if (i != x && abs(A[i][y]) > EPS) {
      b[i] -= A[i][y] * b[x];
      forn(j, m) if (j != y) A[i][j] -= A[i][y] * A[x][j];
      A[i][y] *= -A[x][y];
    }
    z += c[y] * b[x];
    forn(i, m) if (i != y) c[i] -= c[y] * A[x][i];
    c[y] *= -A[x][y];
  };
  while (1) {
    int x = -1, y = -1;
    tipo mn = -EPS;
    forn(i, n) if (b[i] < mn) mn = b[i], x = i;
    if (x < 0) break;
    forn(i, m) if (A[x][i] < -EPS) {
      y = i;
      break;
    }
    assert(y >= 0); // no solution to Ax<=b
    pivot(x, y);
  }
  while (1) {
    tipo mx = EPS;
    int x = -1, y = -1;
    forn(i, m) if (c[i] > mx) mx = c[i], y = i;
    if (y < 0) break;
    tipo mn = 1e200;
    forn(i, n) if (A[i][y] > EPS && b[i] / A[i][y] < mn) {
      mn = b[i] / A[i][y], x = i;
    }
    assert(x >= 0); // c^T x is unbounded
    pivot(x, y);
  }
  vt r(m);
  forn(i, n) if (Y[i] < m) r[Y[i]] = b[i];
  return {z, r};
}
```

## 6.16 Simpson

```
typedef long double T;
// polar coordinates: x=r*cos(theta), y=r*sin(theta), f=(r*r
     )/2
T simpson(std::function<T(T)> f, T a, T b, int n = 10000) {
     // O(n)
  T area = 0, h = (b - a) / T(n), fa = f(a), fb;
  forn(i, n) {
    fb = f(a + h * T(i + 1));
    area += fa + T(4) * f(a + h * T(i + 0.5)) + fb;
    fa = fb;
  }
  return area * h / T(6.);
}
```

# 7 Strings

## 7.1 Aho Corasick

```
struct Node {
  map<char, int> next, go;
  int p, link, leafLink;
  char pch;
  vector<int> leaf;
  Node(int pp, char c) : p(pp), link(-1), leafLink(-1), pch(
      c) {}
};
struct AhoCorasick {
  vector<Node> t = {Node(-1, -1)};
  void add_string(string s, int id) {
    int v = 0;
    for (char c : s) {
      if (!t[v].next.count(c)) {
        t[v].next[c] = sz(t);
        t.pb(Node(v, c));
      }
      v = t[v].next[c];
    }
    t[v].leaf.pb(id);
  }
  int go(int v, char c) {
    if (!t[v].go.count(c)) {
      if (t[v].next.count(c)) t[v].go[c] = t[v].next[c];
      else t[v].go[c] = v == 0 ? 0 : go(get_link(v), c);
    }
    return t[v].go[c];
  }
  int get_link(int v) { // suffix link
    if (t[v].link < 0) {
      if (!v || !t[v].p) t[v].link = 0;
      else t[v].link = go(get_link(t[v].p), t[v].pch);
    }
    return t[v].link;
  }
  // like suffix link, but only going to the root or to a
      node with
  // a non-emtpy "leaf" list. Copy only if needed
  int get_leaf_link(int v) {
    if (t[v].leafLink < 0) {
      if (!v || !t[v].p) t[v].leafLink = 0;
      else if (!t[get_link(v)].leaf.empty()) t[v].leafLink =
          t[v].link;
      else t[v].leafLink = get_leaf_link(t[v].link);
    }
    return t[v].leafLink;
  }
};
```

## 7.2 Booth

```
// Booth's algorithm
// Find lexicographically minimal string rotation in O(|S|)
int booth(string S) {
  S += S; // Concatenate string to it self to avoid modular
      arithmetic
  int n = sz(S);
  vector<int> f(n, -1);
  int k = 0; // Least rotation of string found so far
  forr(j, 1, n) {
    char sj = S[j];
    int i = f[j - k - 1];
    while (i != -1 and sj != S[k + i + 1]) {
      if (sj < S[k + i + 1]) k = j - i - 1;
      i = f[i];
    }
    if (sj != S[k + i + 1]) {
      if (sj < S[k]) k = j;
      f[j - k] = -1;
    } else {
      f[j - k] = i + 1;
    }
  }
  return k; // Lexicographically minimal string rotation's
      position
}
```

## 7.3 Hash Simple

```
// P should be a prime number, could be randomly generated,
// sometimes is good to make it close to alphabet size
// MOD[i] must be a prime of this order, could be randomly
     generated
const int P = 1777771, MOD[2] = {999727999, 1070777777};
const int PI[2] = {325255434, 10018302}; // PI[i] = P^-1 %
     MOD[i]
struct Hash {
  ll h[2];
  vector<ll> vp[2];
  deque<int> x;
  Hash(vector<int>& s) {
    forn(i, sz(s)) x.pb(s[i]);
    forn(k, 2) vp[k].rsz(s.size() + 1);
    forn(k, 2) {
      h[k] = 0;
      vp[k][0] = 1;
      ll p = 1;
      forr(i, 1, sz(s) + 1) {
        h[k] = (h[k] + p * s[i - 1]) % MOD[k];
        vp[k][i] = p = (p * P) % MOD[k];
      }
    }
  }
  // Put the value val in position pos and update the hash
      value
  void change(int pos, int val) {
    forn(i, 2) h[i] = (h[i] + vp[i][pos] * (val - x[pos] +
        MOD[i])) % MOD[i];
    x[pos] = val;
  }
  // Add val to the end of the current string
  void push_back(int val) {
    int pos = sz(x);
    x.pb(val);
    forn(k, 2) {
      assert(pos <= sz(vp[k]));
      if (pos == sz(vp[k])) vp[k].pb(vp[k].back() * P % MOD[k
          ]);
      ll p = vp[k][pos];
      h[k] = (h[k] + p * val) % MOD[k];
    }
  }
  // Delete the first element of the current string
  void pop_front() {
    assert(sz(x) > 0);
    forn(k, 2) {
      h[k] = (h[k] - x[0] + MOD[k]) % MOD[k];
      h[k] = h[k] * PI[k] % MOD[k];
    }
    x.pop_front();
  }
  ll getHashVal() { return (h[0] << 32) | h[1]; }
};
```

## 7.4 Hash

```
// P should be a prime number, could be randomly generated,
// sometimes is good to make it close to alphabet size
// MOD[i] must be a prime of this order, could be randomly
     generated
const int P = 1777771, MOD[2] = {999727999, 1070777777};
```

```
const int PI[2] = {325255434, 10018302}; // PI[i] = P^-1 %
    MOD[i]
struct Hash {
  vector<int> h[2], pi[2];
  vector<ll> vp[2]; // Only used if getChanged is used (
      delete it if not)
  Hash(string& s) {
    forn(k, 2) h[k].rsz(s.size() + 1), pi[k].rsz(s.size() +
        1),
      vp[k].rsz(s.size() + 1);
    forn(k, 2) {
      h[k][0] = 0;
      vp[k][0] = pi[k][0] = 1;
      ll p = 1;
      forr(i, 1, sz(s) + 1) {
        h[k][i] = (h[k][i - 1] + p * s[i - 1]) % MOD[k];
        pi[k][i] = (1LL * pi[k][i - 1] * PI[k]) % MOD[k];
        vp[k][i] = p = (p * P) % MOD[k];
      }
    }
  }
  ll get(int s, int e) { // get hash value of the substring
      [s, e)
    ll H[2];
    forn(i, 2) {
      H[i] = (h[i][e] - h[i][s] + MOD[i]) % MOD[i];
      H[i] = (1LL * H[i] * pi[i][s]) % MOD[i];
    }
    return (H[0] << 32) | H[1];
  }
  // get hash value of [s, e) if origVal in pos is changed
      to val
  // Assumes s <= pos < e. If multiple changes are needed,
  // do what is done in the for loop for every change
  ll getChanged(int s, int e, int pos, int val, int origVal)
      {
    ll hv = get(s, e), hh[2];
    hh[1] = hv & ((1LL << 32) - 1);
    hh[0] = hv >> 32;
    forn(i, 2) hh[i] = (hh[i] + vp[i][pos] * (val - origVal +
        MOD[i])) % MOD[i];
    return (hh[0] << 32) | hh[1];
  }
};
```

## 7.5 Hash128

```
typedef __int128 bint; // needs gcc compiler?
const bint MOD = 212345678987654321LL, P = 1777771, PI =
    106955741089659571LL;
struct Hash {
  vector<bint> h, pi;
  Hash(string& s) {
    assert((P * PI) % MOD == 1);
    h.resize(s.size() + 1), pi.resize(s.size() + 1);
    h[0] = 0, pi[0] = 1;
    bint p = 1;
    forr(i, 1, sz(s) + 1) {
      h[i] = (h[i - 1] + p * s[i - 1]) % MOD;
      pi[i] = (pi[i - 1] * PI) % MOD;
      p = (p * P) % MOD;
    }
  }
  ll get(int s, int e) { // get hash value of the substring
      [s, e)
    return (((h[e] - h[s] + MOD) % MOD) * pi[s]) % MOD;
  }
};
```

## 7.6 Kmp

```
// b[i] = longest border of t[0,i) = length of the longest
    prefix of
// the substring P[0..i-1) that is also suffix of the
    substring P[0..i)
// For "AABAACAABAA", b[i] = {-1, 0, 1, 0, 1, 2, 0, 1, 2, 3,
     4, 5}
vector<int> kmppre(string& P) { //
  vector<int> b(sz(P) + 1);
  b[0] = -1;
  int j = -1;
  forn(i, sz(P)) {
    while (j >= 0 && P[i] != P[j]) j = b[j];
    b[i + 1] = ++j;
  }
  return b;
}
void kmp(string& T, string& P) { // Text, Pattern -- O(|T|+|
    P|)
  int j = 0;
  vector<int> b = kmppre(P);
  forn(i, sz(T)) {
    while (j >= 0 && T[i] != P[j]) j = b[j];
    if (++j == sz(P)) {
      // Match at i-j+1, do something
      j = b[j];
    }
  }
}
```

## 7.7 Lcp

```
// LCP(sa[i], sa[j]) = min(lcp[i+1], lcp[i+2], ..., lcp[j])
// example: "banana", sa = {5,3,1,0,4,2}, lcp =
    {0,1,3,0,0,2}
// Num of dif substrings: (n*n+n)/2 - (sum over lcp array)
// Build suffix array (sa) before calling
vector<int> computeLCP(string& s, vector<int>& sa) {
  int n = s.size(), L = 0;
  vector<int> lcp(n), plcp(n), phi(n);
  phi[sa[0]] = -1;
  forr(i, 1, n) phi[sa[i]] = sa[i - 1];
  forn(i, n) {
    if (phi[i] < 0) {
      plcp[i] = 0;
      continue;
    }
    while (s[i + L] == s[phi[i] + L]) L++;
    plcp[i] = L;
    L = max(L - 1, 0);
  }
  forn(i, n) lcp[i] = plcp[sa[i]];
  return lcp; // lcp[i]=LCP(sa[i-1],sa[i])
}
```

## 7.8 Manacher

```
int d1[MAXN]; // d1[i] = max odd palindrome centered on i
int d2[MAXN]; // d2[i] = max even palindrome centered on i
// s  aabbaacaabbaa
// d1 1111117111111
// d2 0103010010301
void manacher(string& s) { // O(|S|) - find longest
    palindromic substring
  int l = 0, r = -1, n = s.size();
  forn(i, n) { // build d1
    int k = i > r ? 1 : min(d1[l + r - i], r - i);
    while (i + k < n && i - k >= 0 && s[i + k] == s[i - k]) k
        ++;
    d1[i] = k--;
    if (i + k > r) l = i - k, r = i + k;
  }
  l = 0, r = -1;
  forn(i, n) { // build d2
```

```
    int k = (i > r ? 0 : min(d2[l + r - i + 1], r - i + 1)) +
        1;
    while (i + k <= n && i - k >= 0 && s[i + k - 1] == s[i -
        k]) k++;
    d2[i] = --k;
    if (i + k - 1 > r) l = i - k, r = i + k - 1;
  }
}
```

## 7.9 Suffix Array Slow

```
pair<int, int> sf[MAXN];
bool sacomp(int lhs, int rhs) { return sf[lhs] < sf[rhs]; }
vector<int> constructSA(string& s) { // O(n log^2(n))
  int n = s.size();                  // (sometimes fast enough)
  vector<int> sa(n), r(n);
  forn(i, n) r[i] = s[i]; // r[i]: equivalence class of s[i
      ..i+m)
  for (int m = 1; m < n; m *= 2) {
    // sf[i] = {r[i], r[i+m]}, used to sort for next
        equivalence classes
    forn(i, n) sa[i] = i, sf[i] = {r[i], i + m < n ? r[i + m]
        : -1};
    stable_sort(sa.begin(), sa.end(), sacomp); // O(n log(n))
    r[sa[0]] = 0;
    // if sf[sa[i]] == sf[sa[i-1]] then same equivalence
        class
    forr(i, 1, n) r[sa[i]] = sf[sa[i]] != sf[sa[i - 1]] ? i :
        r[sa[i - 1]];
  }
  return sa;
}
```

## 7.10 Suffix Array

```
#define RB(x) (x < n ? r[x] : 0)
void csort(vector<int>& sa, vector<int>& r, int k) { //
    counting sort O(n)
  int n = sa.size();
  vector<int> f(max(255, n), 0), t(n);
  forn(i, n) f[RB(i + k)]++;
  int sum = 0;
  forn(i, max(255, n)) f[i] = (sum += f[i]) - f[i];
  forn(i, n) t[f[RB(sa[i] + k)]++] = sa[i];
  sa = t;
}
vector<int> constructSA(string& s) { // O(n logn)
  int n = s.size(), rank;
  vector<int> sa(n), r(n), t(n);
  forn(i, n) sa[i] = i, r[i] = s[i]; // r[i]: equivalence
      class of s[i..i+k)
  for (int k = 1; k < n; k *= 2) {
    csort(sa, r, k);
    csort(sa, r, 0);    // counting sort, O(n)
    t[sa[0]] = rank = 0; // t : equivalence classes array for
         next size
    forr(i, 1, n) {
      // check if sa[i] and sa[i-1] are in te same
          equivalence class
      if (r[sa[i]] != r[sa[i - 1]] || RB(sa[i] + k) != RB(sa[
          i - 1] + k))
        rank++;
      t[sa[i]] = rank;
    }
    r = t;
    if (r[sa[n - 1]] == n - 1) break;
  }
  return sa;
}
```

## 7.11 Suffix Automaton

```
// The substrings of S can be decomposed into equivalence
    classes
// 2 substr are of the same class if they have the same set
    of endpos
// Example: endpos("bc") = {2, 4, 6} in "abcbcbc"
// Each class is a node of the automaton.
// Len is the longest substring of each class
// Link in state X is the state where the longest suffix of
    the longest
// substring in X, with a different endpos set, belongs
// The links form a tree rooted at 0
// last is the state of the whole string after each
    extention
struct state {
  int len, link;
  map<char, int> next;
};  // clear next!!
state st[MAXN];
int sz, last;
void sa_init() {
  last = st[0].len = 0;
  sz = 1;
  st[0].link = -1;
}
void sa_extend(char c) {
  int k = sz++, p; // k = new state
  st[k].len = st[last].len + 1;
  // while c is not present in p assign it as edge to the
      new state and
  // move through link (note that p always corresponds to a
      suffix state)
  for (p = last; p != -1 && !st[p].next.count(c); p = st[p].
      link)
    st[p].next[c] = k;
  if (p == -1) st[k].link = 0;
  else {
    // state p already goes to state q through char c. Then,
        link of k
    // should go to a state with len = st[p].len + 1 (because
         of c)
    int q = st[p].next[c];
    if (st[p].len + 1 == st[q].len) st[k].link = q;
    else {
      // q is not the state we are looking for. Then, we
      // create a clone of q (w) with the desired length
      int w = sz++;
      st[w].len = st[p].len + 1;
      st[w].next = st[q].next;
      st[w].link = st[q].link;
      // go through links from p and while next[c] is q,
          change it to w
      for (; p != -1 && st[p].next[c] == q; p = st[p].link)
        st[p].next[c] = w;
      // change link of q from p to w, and finally set link
          of k to w
      st[q].link = st[k].link = w;
    }
  }
  last = k;
}
//  input: abcbcbc
//  i,link,len,next
//  0 -1 0 (a,1) (b,5) (c,7)
//  1 0 1 (b,2)
//  2 5 2 (c,3)
//  3 7 3 (b,4)
//  4 9 4 (c,6)
//  5 0 1 (c,7)
//  6 11 5 (b,8)
//  7 0 2 (b,9)
//  8 9 6 (c,10)
//  9 5 3 (c,11)
//  10 11 7
//  11 7 4 (b,8)
```

## 7.12 Suffix Tree

```
const int INF = 1e6 + 10; // INF > n
const int MAXLOG = 20;  // 2^MAXLOG > 2*n
// The SuffixTree of S is the compressed trie that would
     result after
// inserting every suffix of S.
// As it is a COMPRESSED trie, some edges may correspond to
     strings,
// instead of chars, and the compression is done in a way
     that every
// vertex that doesn't correspond to a suffix and has only
     one
// descendent, is omitted.
struct SuffixTree {
  vector<char> s;
  vector<map<int, int>> to; // fst char of substring on edge
      -> node
  // s[fpos[i], fpos[i]+len[i]) is the substring on the edge
      between
  // i's father and i.
  // link[i] goes to the node that corresponds to the
      substring that
  // result after "removing" the first character of the
      substring that
  // i represents. Only defined for internal (non-leaf)
      nodes.
  vector<int> len, fpos, link;
  SuffixTree(int nn = 0) { // O(nn). nn should be the
      expected size
    s.reserve(nn), to.reserve(2 * nn), len.reserve(2 * nn);
    fpos.reserve(2 * nn), link.reserve(2 * nn);
    make_node(0, INF);
  }
  int node = 0, pos = 0, n = 0;
  int make_node(int p, int l) {
    fpos.pb(p), len.pb(l), to.pb({}), link.pb(0);
    return sz(to) - 1;
  }
  void go_edge() {
    while (pos > len[to[node][s[n - pos]]]) {
      node = to[node][s[n - pos]];
      pos -= len[node];
    }
  }
  void add(char c) {
    s.pb(c), n++, pos++;
    int last = 0;
    while (pos > 0) {
      go_edge();
      int edge = s[n - pos];
      int& v = to[node][edge];
      int t = s[fpos[v] + pos - 1];
      if (v == 0) {
        v = make_node(n - pos, INF);
        link[last] = node;
        last = 0;
      } else if (t == c) {
        link[last] = node;
        return;
      } else {
        int u = make_node(fpos[v], pos - 1);
        to[u][c] = make_node(n - 1, INF);
        to[u][t] = v;
        fpos[v] += pos - 1, len[v] -= pos - 1;
        v = u, link[last] = u, last = u;
      }
      if (node == 0) pos--;
      else node = link[node];
    }
  }
// Call this after you finished building the SuffixTree to
     correctly
// set some values of the vector 'len' that currently have
     a big
// value (because of INF usage). Note that you shouldn't
    call 'add'
// anymore after calling this method.
void finishedAdding() {
  forn(i, sz(len)) if (len[i] + fpos[i] > n) len[i] = n -
      fpos[i];
}
// From here, copy only if needed!!
// Map each suffix with it corresponding leaf node
// vleaf[i] = node id of leaf of suffix s[i..n)
// Note that the last character of the string must be
    unique
// Use 'buildLeaf' not 'dfs' directly. Also '
    finishedAdding' must
// be called before calling 'buildLeaf'.
// When this is needed, usually binary lifting (vp) and
    depths are
// also needed.
// Usually you also need to compute extra information in
    the dfs.
vector<int> vleaf, vdepth;
vector<vector<int>> vp;
void dfs(int cur, int p, int curlen) {
  if (cur > 0) curlen += len[cur];
  vdepth[cur] = curlen;
  vp[cur][0] = p;
  if (to[cur].empty()) {
    assert(0 < curlen && curlen <= n);
    assert(vleaf[n - curlen] == -1);
    vleaf[n - curlen] = cur;
    // here maybe compute some extra info
  } else forall(it, to[cur]) {
      dfs(it->snd, cur, curlen);
      // maybe change return type and here compute extra
          info
    }
  // maybe return something here related to extra info
}
void buildLeaf() {
  vdepth.rsz(sz(to), 0);             // tree size
  vleaf.rsz(n, -1);                  // string size
  vp.rsz(sz(to), vector<int>(MAXLOG)); // tree size * log
  dfs(0, 0, 0);
  forr(k, 1, MAXLOG) forn(i, sz(to)) vp[i][k] = vp[vp[i][k
       - 1]][k - 1];
  forn(i, n) assert(vleaf[i] != -1);
}
};
```

## 7.13 Trie

```
struct Trie {
  map<char, Trie> m; // Trie* when using persistence
  // For persistent trie only. Call "clone" probably from
  // "add" and/or other methods, to implement persistence.
  void clone(int pos) {
    Trie* prev = NULL;
    if (m.count(pos)) prev = m[pos];
    m[pos] = new Trie();
    if (prev != NULL) {
      m[pos]->m = prev->m;
      // copy other relevant data
    }
  }
  void add(const string& s, int p = 0) {
    if (s[p]) m[s[p]].add(s, p + 1);
  }
  void dfs() {
    // Do stuff
    forall(it, m) it->second.dfs();
  }
};
```

### 7.14 Zfunction

```
// z[i] = length of longest substring starting from s[i]
    that is prefix of s
// z[i] = max k: s[0,k) == s[i,i+k)
vector<int> zFunction(string& s) {
  int l = 0, r = 0, n = sz(s);
  vector<int> z(n, 0);
  forr(i, 1, n) {
    if (i <= r) z[i] = min(r - i + 1, z[i - l]);
    while (i + z[i] < n && s[z[i]] == s[i + z[i]]) z[i]++;
    if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
  }
  return z;
}
void match(string& T, string& P) { // Text, Pattern -- O(|T
    |+|P|)
  string s = P + '$' + T; //'$' should be a character that
      is not present in T
  vector<int> z = zFunction(s);
  forr(i, sz(P) + 1, sz(s)) if (z[i] == sz(P)); // match
      found, idx = i-sz(P)-1
}
```

# 8 Structures

## 8.1 Bigint

```
#define BASEXP 6
#define BASE 1000000
#define LMAX 1000
struct bint {
  int l;
  ll n[LMAX];
  bint(ll x = 0) {
    l = 1;
    forn(i, LMAX) {
      if (x) l = i + 1;
      n[i] = x % BASE;
      x /= BASE;
    }
  }
  bint(string x) {
    l = (x.size() - 1) / BASEXP + 1;
    fill(n, n + LMAX, 0);
    ll r = 1;
    forn(i, sz(x)) {
      n[i / BASEXP] += r * (x[x.size() - 1 - i] - '0');
      r *= 10;
      if (r == BASE) r = 1;
    }
  }
  void out() {
    cout << n[l - 1];
    dforn(i, l - 1) printf("%6.6llu", n[i]); // 6=BASEXP!
  }
  void invar() {
    fill(n + l, n + LMAX, 0);
    while (l > 1 && !n[l - 1]) l--;
  }
};
bint operator+(const bint& a, const bint& b) {
  bint c;
  c.l = max(a.l, b.l);
  ll q = 0;
  forn(i, c.l) q += a.n[i] + b.n[i], c.n[i] = q % BASE, q /=
      BASE;
  if (q) c.n[c.l++] = q;
  c.invar();
  return c;
}
pair<bint, bool> lresta(const bint& a, const bint& b) // c =
    a - b
{
  bint c;
  c.l = max(a.l, b.l);
  ll q = 0;
  forn(i, c.l) q += a.n[i] - b.n[i], c.n[i] = (q + BASE) %
      BASE,
                          q = (q + BASE) / BASE - 1;
  c.invar();
  return make_pair(c, !q);
}
bint& operator-=(bint& a, const bint& b) { return a = lresta
    (a, b).first; }
bint operator-(const bint& a, const bint& b) { return lresta
    (a, b).first; }
bool operator<(const bint& a, const bint& b) { return !
    lresta(a, b).second; }
bool operator<=(const bint& a, const bint& b) { return
    lresta(b, a).second; }
bool operator==(const bint& a, const bint& b) { return a <=
    b && b <= a; }
bint operator*(const bint& a, ll b) {
  bint c;
  ll q = 0;
  forn(i, a.l) q += a.n[i] * b, c.n[i] = q % BASE, q /= BASE
      ;
  c.l = a.l;
  while (q) c.n[c.l++] = q % BASE, q /= BASE;
  c.invar();
  return c;
}
bint operator*(const bint& a, const bint& b) {
  bint c;
  c.l = a.l + b.l;
  fill(c.n, c.n + b.l, 0);
  forn(i, a.l) {
    ll q = 0;
    forn(j, b.l) q += a.n[i] * b.n[j] + c.n[i + j], c.n[i + j
        ] = q % BASE,
                                                         q /=
                                                           BASE
                                                           ;
    c.n[i + b.l] = q;
  }
  c.invar();
  return c;
}
pair<bint, ll> ldiv(const bint& a, ll b) { // c = a / b ; rm
      = a % b
  bint c;
  ll rm = 0;
  dforn(i, a.l) {
    rm = rm * BASE + a.n[i];
    c.n[i] = rm / b;
    rm %= b;
  }
  c.l = a.l;
  c.invar();
  return make_pair(c, rm);
}
bint operator/(const bint& a, ll b) { return ldiv(a, b).
    first; }
ll operator%(const bint& a, ll b) { return ldiv(a, b).second
    ; }
pair<bint, bint> ldiv(const bint& a, const bint& b) {
  bint c;
  bint rm = 0;
  dforn(i, a.l) {
    if (rm.l == 1 && !rm.n[0]) rm.n[0] = a.n[i];
    else {
      dforn(j, rm.l) rm.n[j + 1] = rm.n[j];
      rm.n[0] = a.n[i];
      rm.l++;
    }
    ll q = rm.n[b.l] * BASE + rm.n[b.l - 1];
    ll u = q / (b.n[b.l - 1] + 1);
```

```
    ll v = q / b.n[b.l - 1] + 1;
    while (u < v - 1) {
      ll m = (u + v) / 2;
      if (b * m <= rm) u = m;
      else v = m;
    }
    c.n[i] = u;
    rm -= b * u;
  }
  c.l = a.l;
  c.invar();
  return make_pair(c, rm);
}
bint operator/(const bint& a, const bint& b) { return ldiv(a
    , b).first; }
bint operator%(const bint& a, const bint& b) { return ldiv(a
    , b).second; }
```

## 8.2 Disjoint Intervals

```
// stores disjoint intervals as [first, second)
// the final result is the union of the inserted intervals
// [1, 5), [2, 4), [10, 13), [11, 15) -> [1, 5), [10, 15)
struct disjoint_intervals {
  set<ii> s;
  void insert(ii v) {
    if (v.fst >= v.snd) return;
    auto at = s.lower_bound(v);
    auto it = at;
    if (at != s.begin() && (--at)->snd >= v.fst) v.fst = at->
        fst, --it;
    for (; it != s.end() && it->fst <= v.snd; s.erase(it++))
      v.snd = max(v.snd, it->snd);
    s.insert(v);
  }
};
```

## 8.3 Fenwick Tree

```
struct FenwickTree {
  int N;          // maybe replace vector with unordered_map
      when "many 0s"
  vector<tipo> ft; // for more dimensions, make ft multi-
      dimensional
  FenwickTree(int n) : N(n), ft(n + 1) {}
  void upd(int i0, tipo v) { // add v to i0th element (0-
      based)
    // add extra fors for more dimensions
    for (int i = i0 + 1; i <= N; i += i & -i) ft[i] += v;
  }
  tipo get(int i0) { // get sum of range [0,i0)
    tipo r = 0;      // add extra fors for more dimensions
    for (int i = i0; i; i -= i & -i) r += ft[i];
    return r;
  }
  tipo get_sum(int i0, int i1) { // get sum of range [i0,i1)
      (0-based)
    return get(i1) - get(i0);
  }
};
```

## 8.4 Gain Cost Set

```
// stores pairs (benefit,cost) (erases non-optimal pairs)
// Note that these pairs will be increasing by g and
    increasing by c
// If we insert a pair that is included in other, the big
    one will be deleted
// For lis 2d, create a GCS por each posible length, use as
    (-g, c) and
// binary search looking for the longest length where (-g, c
    ) could be added
struct GCS {
  set<ii> s;
  void add(int g, int c) {
    ii x = {g, c};
    auto p = s.lower_bound(x);
    if (p != s.end() && p->snd <= x.snd) return;
    if (p != s.begin()) { // erase pairs with less or eq
        benefit and more cost
      --p;
      while (p->snd >= x.snd) {
        if (p == s.begin()) {
          s.erase(p);
          break;
        }
        s.erase(p--);
      }
    }
    s.insert(x);
  }
  int get(int gain) { // min cost for the benefit greater or
      equal to gain
    auto p = s.lower_bound((ii){gain, -INF});
    int r = p == s.end() ? INF : p->snd;
    return r;
  }
};
```

## 8.5 Hash Table

```
struct Hash { // similar logic for any other data type
  size_t operator()(const vector<int>& v) const {
    size_t s = 0;
    for (auto& e : v) s ^= hash<int>()(e) + 0x9e3779b9 + (s
        << 6) + (s >> 2);
    return s;
  }
};
unordered_set<vector<int>, Hash> s; // unordered_map<key,
    value, hasher>
```

## 8.6 Indexed Set

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
//<key,mapped type,comparator,...>
typedef tree<int,null_type,less<int>,rb_tree_tag,
 tree_order_statistics_node_update> indexed_set;
// find_by_order(i) returns iterator to the i-th elemnt
// order_of_key(k): returns position of the lower bound of k
    (0-indexed)
// Ej: 12, 100, 505, 1000, 10000.
// order_of_key(10) == 0, order_of_key(100) == 1,
// order_of_key(707) == 3, order_of_key(9999999) == 5
```

## 8.7 Link Cut Tree

```
const int N_DEL = 0, N_VAL = 0; // neutral elements for
    delta & values
inline int u_oper(int x, int y){ return x + y; } // update
    operation
inline int q_oper(int lval, int rval){ return lval + rval; }
     // query operation
inline int u_segm(int d, int len){return d==N_DEL?N_DEL:d*
    len;} // upd segment
inline int u_delta(int d1, int d2){ // update delta
  if(d1==N_DEL) return d2;
  if(d2==N_DEL) return d1;
  return u_oper(d1, d2);
```

```
}
inline int a_delta(int v, int d){ // apply delta
 return d==N_DEL ? v : u_oper(d, v);
}

// Splay tree
struct node_t{
  int szi, n_val, t_val, d;
  bool rev;
  node_t *c[2], *p;
  node_t(int v) : szi(1), n_val(v), t_val(v), d(N_DEL), rev
       (0), p(0){
    c[0]=c[1]=0;
  }
  bool is_root(){return !p || (p->c[0] != this && p->c[1] !=
        this);}
  void push(){
    if(rev){
      rev=0; swap(c[0], c[1]);
      forr(x,0,2) if(c[x]) c[x]->rev^=1;
    }
    n_val = a_delta(n_val, d); t_val=a_delta(t_val, u_segm(d,
          szi));
    forr(x,0,2) if(c[x])
  c[x]->d = u_delta(d, c[x]->d);
    d=N_DEL;
  }
  void upd();
};
typedef node_t* node;
int get_sz(node r){return r ? r->szi : 0;}
int get_tree_val(node r){
  return r ? a_delta(r->t_val, u_segm(r->d,r->szi)) : N_VAL;
}
void node_t::upd() {
  t_val=q_oper(q_oper(get_tree_val(c[0]),a_delta(n_val,d)),
       get_tree_val(c[1]));
  szi = 1 + get_sz(c[0]) + get_sz(c[1]);
}
void conn(node c, node p, int is_left){
 if(c) c->p = p;
 if(is_left>=0) p->c[!is_left] = c;
}
void rotate(node x){
  node p = x->p, g = p->p;
  bool gCh=p->is_root(), is_left = x==p->c[0];
  conn(x->c[is_left],p,is_left);
  conn(p,x,!is_left);
  conn(x,g,gCh?-1:(p==g->c[0]));
  p->upd();
}
void splay(node x){
  while(!x->is_root()){
    node p = x->p, g = p->p;
    if(!p->is_root()) g->push();
    p->push(); x->push();
    if(!p->is_root()) rotate((x==p->c[0])==(p==g->c[0])? p :
         x);
    rotate(x);
  }
  x->push(); x->upd();
}

// Link-cut Tree
// Keep information of a tree (or forest) and allow to make
     many types of
// operations (see them below) in an efficient way.
     Internally, each node of
// the tree will have at most 1 "preferred" child, and as a
     consequence, the
// tree can be seen as a set of independent "preferred"
     paths. Each of this
// paths is basically a list, represented with a splay tree,
      where the
// "implicit key" (for the BST) of each element is the depth
      of the
// corresponding node in the original tree (or forest). Also
     , each of these
// preferred paths (except one of them), will know who its "
     father path" is,
// i.e. will know the preferred path of the father of the
     top-most node.

// Make the path from the root to 'x' to be a "preferred
     path", and also make
// 'x' to be the root of its splay tree (not the root of the
      original tree).
node expose(node x){
  node last = 0;
  for(node y=x; y; y=y->p)
 splay(y), y->c[0] = last, y->upd(), last = y;
  splay(x);
  return last;
}
void make_root(node x){expose(x);x->rev^=1;}
node get_root(node x){
 expose(x);
 while(x->c[1]) x = x->c[1];
 splay(x);
 return x;
}
node lca(node x, node y){expose(x); return expose(y);}
bool connected(node x, node y){
 expose(x); expose(y);
 return x==y ? 1 : x->p!=0;
}
// makes x son of y
void link(node x, node y){ make_root(x); x->p=y; }
void cut(node x, node y){ make_root(x); expose(y); y->c[1]->
     p=0; y->c[1]=0; }
node father(node x){
 expose(x);
 node r = x->c[1];
 if(!r) return 0;
 while(r->c[0]) r = r->c[0];
 return r;
}
// cuts x from its father keeping tree root
void cut(node x){ expose(father(x)); x->p = 0; }
int query(node x, node y){
 make_root(x); expose(y);
 return get_tree_val(y);
}
void update(node x, node y, int d){
 make_root(x); expose(y); y->d=u_delta(y->d,d);
}
node lift_rec(node x, int k){
 if(!x) return 0;
 if(k == get_sz(x->c[0])){ splay(x); return x; }
 if(k < get_sz(x->c[0])) return lift_rec(x->c[0],k);
 return lift_rec(x->c[1], k-get_sz(x->c[0])-1);
}
// k-th ancestor of x (lift(x,1) is x's father)
node lift(node x, int k){ expose(x);return lift_rec(x,k); }
// distance from x to its tree root
int depth(node x){ expose(x);return get_sz(x)-1; }
```

## 8.8 Merge Sort Tree

```
typedef ii datain;   // data that goes into the DS
typedef int query;   // info related to a query
typedef bool dataout; // data that results from a query
struct DS {
  set<datain> s; // replace set with what's needed for the
       problem
  void insert(const datain& x) {
    // modify this method according to problem
    // the example below is "disjoint intervals" (i.e. union
         of ranges)
```

```
    datain xx = x; // copy to avoid changing original
    if (xx.fst >= xx.snd) return;
    auto at = s.lower_bound(xx);
    auto it = at;
    if (at != s.begin() && (--at)->snd >= xx.fst) xx.fst = at
        ->fst, --it;
    for (; it != s.end() && it->fst <= xx.snd; s.erase(it++))
      xx.snd = max(xx.snd, it->snd);
    s.insert(xx);
  }
  void get(const query& q, dataout& ans) {
    // modify this method according to problem
    // the example below is "is there any range covering q?"
    set<datain>::iterator ite = s.ub(mp(q + 1, 0));
    if (ite != s.begin() && prev(ite)->snd > q) ans = true;
  }
};
struct MST {
  int sz;
  vector<DS> t;
  MST(int n) {
    sz = 1 << (32 - __builtin_clz(n));
    t = vector<DS>(2 * sz);
  }
  void insert(int i, int j, datain& x) { insert(i, j, x, 1,
      0, sz); }
  void insert(int i, int j, datain& x, int n, int a, int b)
      {
    if (j <= a || b <= i) return;
    if (i <= a && b <= j) {
      t[n].insert(x);
      return;
    }
    // needed when want to update ranges that intersec with [
        i,j)
    // usually only needed on range-query + point-update
        problem
    // t[n].insert(x);
    int c = (a + b) / 2;
    insert(i, j, x, 2 * n, a, c);
    insert(i, j, x, 2 * n + 1, c, b);
  }
  void get(int i, int j, query& q, dataout& ans) {
    return get(i, j, q, ans, 1, 0, sz);
  }
  void get(int i, int j, query& q, dataout& ans, int n, int
      a, int b) {
    if (j <= a || b <= i) return;
    if (i <= a && b <= j) {
      t[n].get(q, ans);
      return;
    }
    // needed when want to get from ranges that intersec with
        [i,j)
    // usually only needed on point-query + range-update
        problem
    // t[n].get(q, ans);
    int c = (a + b) / 2;
    get(i, j, q, ans, 2 * n, a, c);
    get(i, j, q, ans, 2 * n + 1, c, b);
  }
};  // Use: 1- definir todo lo necesario en DS, 2- usar
```

## 8.9 Rope

```
#include <ext/rope>
using namespace __gnu_cxx;
rope<int> s;
// Sequence with O(log(n)) random access, insert, erase at
    any position
// s.push_back(x)
// s.append(other_rope)
// s.insert(i,x)
// s.insert(i,other_rope) // insert rope r at position i
// s.erase(i,k) // erase subsequence [i,i+k)
// s.substr(i,k) // return new rope corresponding to
    subsequence [i,i+k)
// s[i] // access ith element (cannot modify)
// s.mutable_reference_at(i) // acces ith element (allows
    modification)
// s.begin() and s.end() are const iterators (use
    mutable_begin(), mutable_end()
// to allow modification)
```

## 8.10 Segtree 2d

```
#define operacion(x, y) max(x, y)
int n, m;
int a[MAXN][MAXN], st[2 * MAXN][2 * MAXN];
void build() { // O(n*m)
  forn(i, n) forn(j, m) st[i + n][j + m] = a[i][j];
  forn(i, n) dforn(j, m) // build st of row i+n (each row
      independently)
    st[i + n][j] = operacion(st[i + n][j << 1], st[i + n][j
        << 1 | 1]);
  dforn(i, n) forn(j, 2 * m) // build st of ranges of rows
    st[i][j] = operacion(st[i << 1][j], st[i << 1 | 1][j]);
}
void upd(int x, int y, int v) { // O(logn * logm)
  st[x + n][y + m] = v;
  for (int j = y + m; j > 1; j >>= 1) // update ranges
      containing y+m
    st[x + n][j >> 1] = operacion(st[x + n][j], st[x + n][j ^
        1]);
  for (int i = x + n; i > 1; i >>= 1) // in each range that
      contains row x+n
    for (int j = y + m; j; j >>= 1) // update the ranges that
        contains y+m
      st[i >> 1][j] = operacion(st[i][j], st[i ^ 1][j]);
}
int query(int x0, int x1, int y0, int y1) { // O(logn * logm
    )
  int r = NEUT;
  // start at the bottom and move up each time
  for (int i0 = x0 + n, i1 = x1 + n; i0 < i1; i0 >>= 1, i1
      >>= 1) {
    int t[4], q = 0;
    // if the whole segment of row node i0 is included, then
        move right
    if (i0 & 1) t[q++] = i0++;
    // if the whole segment of row node i1-1 is included,
        then move left
    if (i1 & 1) t[q++] = --i1;
    forn(k, q) for (int j0 = y0 + m, j1 = y1 + m; j0 < j1; j0
        >>= 1, j1 >>= 1) {
      if (j0 & 1) r = operacion(r, st[t[k]][j0++]);
      if (j1 & 1) r = operacion(r, st[t[k]][--j1]);
    }
  }
  return r;
}
```

## 8.11 Segtree Dynamic

```
typedef ll tipo;
const tipo neutro = 0;
tipo oper(const tipo& a, const tipo& b) { return a + b; }
struct ST {
  int sz;
  vector<tipo> t;
  ST(int n) {
    sz = 1 << (32 - __builtin_clz(n));
    t = vector<tipo>(2 * sz, neutro);
  }
  tipo& operator[](int p) { return t[sz + p]; }
```

```
  void updall() { dforn(i, sz) t[i] = oper(t[2 * i], t[2 * i + 1]); }
  tipo get(int i, int j) { return get(i, j, 1, 0, sz); }
  tipo get(int i, int j, int n, int a, int b) { // O(log n), [i, j)
    if (j <= a || b <= i) return neutro;
    if (i <= a && b <= j) return t[n]; // n = node of range [a,b)
    int c = (a + b) / 2;
    return oper(get(i, j, 2 * n, a, c), get(i, j, 2 * n + 1, c, b));
  }
  void set(int p, tipo val) { // O(log n)
    p += sz;
    while (p > 0 && t[p] != val) {
      t[p] = val;
      p /= 2;
      val = oper(t[p * 2], t[p * 2 + 1]);
    }
  }
}; // Use: definir oper tipo neutro,
// cin >> n; ST st(n); forn(i, n) cin >> st[i]; st.updall();
```

## 8.12 Segtree Implicit

```
typedef int tipo;
const tipo neutro = 0;
tipo oper(const tipo& a, const tipo& b) { return a + b; }
// Compressed segtree, it works for any range (even negative indexes)
struct ST {
  ST *lc, *rc;
  tipo val;
  int L, R;
  ST(int l, int r, tipo x = neutro) {
    lc = rc = nullptr;
    L = l, R = r, val = x;
  }
  ST(int l, int r, ST* lp, ST* rp) {
    if (lp != nullptr && rp != nullptr && lp->L > rp->L) swap(lp, rp);
    lc = lp, rc = rp;
    L = l, R = r;
    val = oper(lp == nullptr ? neutro : lp->val,
           rp == nullptr ? neutro : rp->val);
  }
  // O(log(R-L)), parameter 'isnew' only needed when persistent
  // This operation inserts at most 2 nodes to the tree, i.e. the
  // total memory used by the tree is O(N), where N is the number
  // of times this 'set' function is called. (2*log nodes when persistent)
  void set(int p, tipo x, bool isnew = false) { // return ST * for persistent
    // uncomment for persistent
    // if(!isnew) {
    //   ST* newnode = new ST(L, R, lc, rc);
    //   return newnode->set(p, x, true);
    // }
    if (L + 1 == R) {
      val = x;
      return; // 'return this;' for persistent
    }
    int m = (L + R) / 2;
    ST** c = p < m ? &lc : &rc;
    if (!*c) *c = new ST(p, p + 1, x);
    else if ((*c)->L <= p && p < (*c)->R) {
      // replace by comment for persistent
      (*c)->set(p, x);
      // *c = (*c)->set(p,x);
    } else {
      int l = L, r = R;
      while ((p < m) == ((*c)->L < m)) {
        if (p < m) r = m;
        else l = m;
        m = (l + r) / 2;
      }
      *c = new ST(l, r, *c, new ST(p, p + 1, x));
      // The code above, inside this else block, could be
      // replaced by the following 2 lines when the complete
      // range has the form [0, 2^k)
      // int rm = (1<<(32-__builtin_clz(p^(*c)->L)))-1;
      // *c = new ST(p & ~rm, (p | rm)+1, *c, new ST(p, p+1, x));
    }
    val = oper(lc ? lc->val : neutro, rc ? rc->val : neutro);
    // return this; // uncomment for persistent
  }
  tipo get(int ql, int qr) { // O(log(R-L))
    if (qr <= L || R <= ql) return neutro;
    if (ql <= L && R <= qr) return val;
    return oper(lc ? lc->get(ql, qr) : neutro, rc ? rc->get(ql, qr) : neutro);
  }
}; // Usage: 1- RMQ st(MIN_INDEX, MAX_INDEX) 2- normally use set/get
```

## 8.13 Segtree Lazy

```
typedef ll Elem;
typedef ll Alt;
const Elem neutro = 0;
const Alt neutro2 = 0;
Elem oper(const Elem& a, const Elem& b) { return a + b; }
struct ST {
  int sz;
  vector<Elem> t;
  vector<Alt> dirty; // Alt and Elem could be different types
  ST(int n) {
    sz = 1 << (32 - __builtin_clz(n));
    t = vector<Elem>(2 * sz, neutro);
    dirty = vector<Alt>(2 * sz, neutro2);
  }
  Elem& operator[](int p) { return t[sz + p]; }
  void updall() { dforn(i, sz) t[i] = oper(t[2 * i], t[2 * i + 1]); }
  void push(int n, int a, int b) { // push dirt to n's child nodes
    if (dirty[n] != neutro2) {   // n = node of range [a,b)
      t[n] += dirty[n] * (b - a); // CHANGE for your problem
      if (n < sz) {
        dirty[2 * n] += dirty[n];    // CHANGE for your problem
        dirty[2 * n + 1] += dirty[n]; // CHANGE for your problem
      }
      dirty[n] = neutro2;
    }
  }
  Elem get(int i, int j, int n, int a, int b) { // O(lgn)
    if (j <= a || b <= i) return neutro;
    push(n, a, b);                 // adjust value before using it
    if (i <= a && b <= j) return t[n]; // n = node of range [a,b)
    int c = (a + b) / 2;
    return oper(get(i, j, 2 * n, a, c), get(i, j, 2 * n + 1, c, b));
  }
  Elem get(int i, int j) { return get(i, j, 1, 0, sz); }
  // altera los valores en [i, j) con una alteracion de val
  void update(Alt val, int i, int j, int n, int a, int b) { // O(lgn)
    push(n, a, b);
    if (j <= a || b <= i) return;
    if (i <= a && b <= j) {
```

```
      dirty[n] += val; // CHANGE for your problem
      push(n, a, b);
      return;
    }
    int c = (a + b) / 2;
    update(val, i, j, 2 * n, a, c), update(val, i, j, 2 * n +
        1, c, b);
    t[n] = oper(t[2 * n], t[2 * n + 1]);
  }
  void update(Alt val, int i, int j) { update(val, i, j, 1,
      0, sz); }
}; // Use: definir operacion, neutros, Alt, Elem, uso de
    dirty
// cin >> n; ST st(n); forn(i,n) cin >> st[i]; st.updall()
```

## 8.14 Segtree Persistent

```
typedef int tipo;
const tipo neutro = 0;
tipo oper(const tipo& a, const tipo& b) { return a + b; }
struct ST {
  int n;
  vector<tipo> st;
  vector<int> L, R;
  ST(int nn) : n(nn), st(1, neutro), L(1, 0), R(1, 0) {}
  int new_node(tipo v, int l = 0, int r = 0) {
    int id = sz(st);
    st.pb(v), L.pb(l), R.pb(r);
    return id;
  }
  int init(vector<tipo>& v, int l, int r) {
    if (l + 1 == r) return new_node(v[l]);
    int m = (l + r) / 2, a = init(v, l, m), b = init(v, m, r)
        ;
    return new_node(oper(st[a], st[b]), a, b);
  }
  int update(int cur, int pos, tipo val, int l, int r) {
    int id = new_node(st[cur], L[cur], R[cur]);
    if (l + 1 == r) {
      st[id] = val;
      return id;
    }
    int m = (l + r) / 2, ASD; // MUST USE THE ASD!!!
    if (pos < m) ASD = update(L[id], pos, val, l, m), L[id] =
        ASD;
    else ASD = update(R[id], pos, val, m, r), R[id] = ASD;
    st[id] = oper(st[L[id]], st[R[id]]);
    return id;
  }
  tipo get(int cur, int from, int to, int l, int r) {
    if (to <= l || r <= from) return neutro;
    if (from <= l && r <= to) return st[cur];
    int m = (l + r) / 2;
    return oper(get(L[cur], from, to, l, m), get(R[cur], from
        , to, m, r));
  }
  int init(vector<tipo>& v) { return init(v, 0, n); }
  int update(int root, int pos, tipo val) {
    return update(root, pos, val, 0, n);
  }
  tipo get(int root, int from, int to) { return get(root,
      from, to, 0, n); }
}; // usage: ST st(n); root = st.init(v) (or root = 0);
// new_root = st.update(root,i,x); st.get(root,l,r);
```

## 8.15 Segtree Static

```
// Solo para funciones idempotentes (como min y max, pero no
    sum)
// Usar la version dynamic si la funcion no es idempotente
struct RMQ {
#define LVL 10 // LVL such that 2^LVL>n
  tipo vec[LVL][1 << (LVL + 1)];
  tipo& operator[](int p) { return vec[0][p]; }
  tipo get(int i, int j) { // intervalo [i,j) - O(1)
    int p = 31 - __builtin_clz(j - i);
    return min(vec[p][i], vec[p][j - (1 << p)]);
  }
  void build(int n) { // O(nlogn)
    int mp = 31 - __builtin_clz(n);
    forn(p, mp) forn(x, n - (1 << p)) vec[p + 1][x] =
        min(vec[p][x], vec[p][x + (1 << p)]);
  }
}; // Use: define LVL y tipo; insert data with []; call
    build; answer queries
```

## 8.16 Treap Implicit

```
// An array represented as a treap, where the "key" is the
    index.
// However, the key is not stored explicitly, but can be
    calculated as
// the sum of the sizes of the left child of the ancestors
    where the node
// is in the right subtree of it.
// (commented parts are specific to range sum queries and
    other problems)
// rng = random number generator, works better than rand in
    some cases
mt19937 rng(chrono::steady_clock::now().time_since_epoch().
    count());
typedef struct item* pitem;
struct item {
  int pr, cnt, val;
  bool rev; // for reverse operation
  int sum;  // for range query
  int add;  // for lazy prop
  pitem l, r;
  pitem p; // ptr to parent, for getRoot
  item(int val) : pr(rng()), cnt(1), val(val), rev(false),
      sum(val), add(0) {
    l = r = p = NULL;
  }
};
void push(pitem node) {
  if (node) {
    // for reverse operation
    if (node->rev) {
      swap(node->l, node->r);
      if (node->l) node->l->rev ^= true;
      if (node->r) node->r->rev ^= true;
      node->rev = false;
    }
    // for lazy prop
    node->val += node->add, node->sum += node->cnt * node->
        add;
    if (node->l) node->l->add += node->add;
    if (node->r) node->r->add += node->add;
    node->add = 0;
  }
}
int cnt(pitem t) { return t ? t->cnt : 0; }
int sum(pitem t) { return t ? push(t), t->sum : 0; } // for
    range query
void upd_cnt(pitem t) {
  if (t) {
    t->cnt = cnt(t->l) + cnt(t->r) + 1;
    t->sum = t->val + sum(t->l) + sum(t->r); // for range sum
    if (t->l) t->l->p = t;                  // for getRoot
    if (t->r) t->r->p = t;                  // for getRoot
    t->p = NULL;                            // for getRoot
  }
}
void split(pitem node, pitem& L, pitem& R, int sz) { // sz:
    wanted size for L
```

```
  if (!node) {
    L = R = 0;
    return;
  }
  push(node);
  // If node's left child has at least sz nodes, go left
  if (sz <= cnt(node->l)) split(node->l, L, node->l, sz), R
      = node;
  // Else, go right changing wanted sz
  else split(node->r, node->r, R, sz - 1 - cnt(node->l)), L
      = node;
  upd_cnt(node);
}
void merge(pitem& result, pitem L, pitem R) { // O(log)
  push(L), push(R);
  if (!L || !R) result = L ? L : R;
  else if (L->pr > R->pr) merge(L->r, L->r, R), result = L;
  else merge(R->l, L, R->l), result = R;
  upd_cnt(result);
}
void insert(pitem& node, pitem x, int pos) { // 0-index O(
    log)
  pitem l, r;
  split(node, l, r, pos);
  merge(l, l, x);
  merge(node, l, r);
}
void erase(pitem& node, int pos) { // 0-index O(log)
  if (!node) return;
  push(node);
  if (pos == cnt(node->l)) merge(node, node->l, node->r);
  else if (pos < cnt(node->l)) erase(node->l, pos);
  else erase(node->r, pos - 1 - cnt(node->l));
  upd_cnt(node);
}
// reverse operation
void reverse(pitem& node, int L, int R) { //[L, R) O(log)
  pitem t1, t2, t3;
  split(node, t1, t2, L);
  split(t2, t2, t3, R - L);
  t2->rev ^= true;
  merge(node, t1, t2);
  merge(node, node, t3);
}
// lazy add
void add(pitem& node, int L, int R, int x) { //[L, R) O(log)
  pitem t1, t2, t3;
  split(node, t1, t2, L);
  split(t2, t2, t3, R - L);
  t2->add += x;
  merge(node, t1, t2);
  merge(node, node, t3);
}
// range query get
int get(pitem& node, int L, int R) { //[L, R) O(log)
  pitem t1, t2, t3;
  split(node, t1, t2, L);
  split(t2, t2, t3, R - L);
  push(t2);
  int ret = t2->sum;
  merge(node, t1, t2);
  merge(node, node, t3);
  return ret;
}
void push_all(pitem t) { // for getRoot
  if (t->p) push_all(t->p);
  push(t);
}
pitem getRoot(pitem t, int& pos) { // get root and position
    for node t
  push_all(t);
  pos = cnt(t->l);
  while (t->p) {
    pitem p = t->p;
    if (t == p->r) pos += cnt(p->l) + 1;
    t = p;
  }
  return t;
}
void output(pitem t) { // useful for debugging
  if (!t) return;
  push(t);
  output(t->l);
  cout << ' ' << t->val;
  output(t->r);
}
```

## 8.17 Treap

```
typedef struct item* pitem;
struct item {
  // pr = randomized priority, key = BST value, cnt = size
      of subtree
  int pr, key, cnt;
  pitem l, r;
  item(int key) : key(key), pr(rand()), cnt(1), l(NULL), r(
      NULL) {}
};
int cnt(pitem node) { return node ? node->cnt : 0; }
void upd_cnt(pitem node) {
  if (node) node->cnt = cnt(node->l) + cnt(node->r) + 1;
}
// splits t in l and r - l: <= key, r: > key
void split(pitem node, int key, pitem& L, pitem& R) { // O(
    log)
  if (!node) L = R = 0;
  // if cur > key, go left to split and cur is part of R
  else if (key < node->key) split(node->l, key, L, node->l),
       R = node;
  // if cur <= key, go right to split and cur is part of L
  else split(node->r, key, node->r, R), L = node;
  upd_cnt(node);
}
// 1) go down the BST following the key of the new node (x),
     until
//  you reach NULL or a node with lower pr than the new one.
// 2.1) if you reach NULL, put the new node there
// 2.2) if you reach a node with lower pr, split the subtree
     rooted at that
// node, put the new one there and put the split result as
    children of it
void insert(pitem& node, pitem x) { // O(log)
  if (!node) node = x;
  else if (x->pr > node->pr) split(node, x->key, x->l, x->r)
      , node = x;
  else insert(x->key <= node->key ? node->l : node->r, x);
  upd_cnt(node);
}
// Assumes that the key of every element in L <= to the keys
     in R
void merge(pitem& result, pitem L, pitem R) { // O(log)
  // If one of the nodes is NULL, the merge result is the
      other node
  if (!L || !R) result = L ? L : R;
  // if L has higher priority than R, put L and update it's
      right child
  // with the merge result of L->r and R
  else if (L->pr > R->pr) merge(L->r, L->r, R), result = L;
  // if R has higher priority than L, put R and update it's
      left child
  // with the merge result of L and R->l
  else merge(R->l, L, R->l), result = R;
  upd_cnt(result);
}
// go down the BST following the key to erase. When the key
    is found,
// replace that node with the result of merging it children
void erase(pitem& node, int key) { // O(log), (erases only 1
     repetition)
```

```
  if (node->key == key) merge(node, node->l, node->r);
  else erase(key < node->key ? node->l : node->r, key);
  upd_cnt(node);
}
// union of two treaps
void unite(pitem& t, pitem L, pitem R) { // O(M*log(N/M))
  if (!L || !R) {
    t = L ? L : R;
    return;
  }
  if (L->pr < R->pr) swap(L, R);
  pitem p1, p2;
  split(R, L->key, p1, p2);
  unite(L->l, L->l, p1);
  unite(L->r, L->r, p2);
  t = L;
  upd_cnt(t);
}
pitem kth(pitem t, int k) { // element at "position" k
  if (!t) return 0;
  if (k == cnt(t->l)) return t;
  return k < cnt(t->l) ? kth(t->l, k) : kth(t->r, k - cnt(t
      ->l) - 1);
}
pair<int, int> lb(pitem t, int key) { // position and value
     of lower_bound
  if (!t) return {0, 1 << 30};      // (special value)
  if (key > t->key) {
    auto w = lb(t->r, key);
    w.fst += cnt(t->l) + 1;
    return w;
  }
  auto w = lb(t->l, key);
  if (w.fst == cnt(t->l)) w.snd = t->key;
  return w;
}
```

## 8.18 Union Find Rollbacks

```
struct dsu_save {
  int v, rnkv, u, rnku;
  dsu_save() {}
  dsu_save(int _v, int _rnkv, int _u, int _rnku)
      : v(_v), rnkv(_rnkv), u(_u), rnku(_rnku) {}
};
struct dsu_with_rollbacks {
  vector<int> p, rnk;
  int comps;
  stack<dsu_save> op;
  dsu_with_rollbacks() {}
  dsu_with_rollbacks(int n) {
    p.rsz(n), rnk.rsz(n);
    forn(i, n) { p[i] = i, rnk[i] = 0; }
    comps = n;
  }
  int find_set(int v) { return (v == p[v]) ? v : find_set(p[
      v]); }
  bool unite(int v, int u) {
    v = find_set(v), u = find_set(u);
    if (v == u) return false;
    comps--;
    if (rnk[v] > rnk[u]) swap(v, u);
    op.push(dsu_save(v, rnk[v], u, rnk[u]));
    p[v] = u;
    if (rnk[u] == rnk[v]) rnk[u]++;
    return true;
  }
  void rollback() {
    if (op.empty()) return;
    dsu_save x = op.top();
    op.pop(), comps++;
    p[x.v] = x.v, rnk[x.v] = x.rnkv;
    p[x.u] = x.u, rnk[x.u] = x.rnku;
  }
};
```

## 8.19 Union Find

```
struct UnionFind {
  int nsets;
  vector<int> f, setsz; // f[i] = parent of node i
  UnionFind(int n) : nsets(n), f(n, -1), setsz(n, 1) {}
  int comp(int x) { return (f[x] == -1 ? x : f[x] = comp(f[x
      ])); } // O(1)
  bool join(int i, int j) { // returns true if already in
      the same set
    int a = comp(i), b = comp(j);
    if (a != b) {
      if (setsz[a] > setsz[b]) swap(a, b);
      f[a] = b; // the bigger group (b) now represents the
          smaller (a)
      nsets--, setsz[b] += setsz[a];
    }
    return a == b;
  }
};
```

# 9 Template

## 9.1 Template

```
#include <bits/stdc++.h>
#define forr(i, a, b) for (int i = (a); i < (b); i++)
#define forn(i, n) forr(i, 0, n)
#define dforn(i, n) for (int i = (n) - 1; i >= 0; i--)
#define forall(it, v) for (auto it = v.begin(); it != v.end
    (); it++)
#define sz(c) ((int)c.size())
#define rsz resize
#define pb push_back
#define mp make_pair
#define lb lower_bound
#define ub upper_bound
#define fst first
#define snd second

#ifdef ANARAP
// local
#else
// judge
#endif

using namespace std;

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

int main() {
// agregar g++ -DANARAP en compilacion
#ifdef ANARAP
  freopen("input.in", "r", stdin);
  // freopen("output.out","w", stdout);
#endif
  ios::sync_with_stdio(false);
  cin.tie(NULL);
  cout.tie(NULL);
  return 0;
}
```

# 10 Utils

## 10.1 C++ Utils

```
// 1- (mt19937_64 for 64-bits version)
```

```
mt19937 rng(
    chrono::steady_clock::now().time_since_epoch().count());
shuffle(v.begin(), v.end(), rng); // vector random shuffle
// 2- Pragma
#pragma GCC optimize("O3,unroll-loops")
#pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
// 3- Custom comparator for set/map
struct comp {
  bool operator()(const double& a, const double& b) const {
    return a + EPS < b;
  }
};
set<double, comp> w; // or map<double,int,comp>
// 4- Iterate over non empty subsets of bitmask
for (int s = m; s; s = (s - 1) & m) // Decreasing order
for (int s = 0; s = s - m & m;)  // Increasing order
// 5- Other bits operations
int __builtin_popcount(unsigned int x) // # of bits on in x
int __builtin_popcountll(unsigned long long x) // ll version
int __builtin_ctz(unsigned int x) //# of trailing 0 (x != 0)
int __builtin_clz(unsigned int x) // # of leading 0 (x != 0)
v = (x & (-x)) // Find the least significant bit that is on
// 6- Input
inline void Scanf(int& a) { // only positive ints
  char c = 0;
  while (c < 33) c = getc(stdin);
  a = 0;
  while (c > 33) a = a * 10 + c - '0', c = getc(stdin);
}
```

## 10.2 Compile Commands

g++ -std=c++20 file -o filename
Para Geany:
compile: g++ -DANARAP -std=c++20 -g -O2 -Wconversion -Wshadow -Wall -Wextra -c "%f"
build: g++ -DANARAP -std=c++20 -g -O2 -Wconversion -Wshadow -Wall -Wextra -o "%e" "%f"

## 10.3 Python Example

```
import sys, math
input = sys.stdin.readline
############ ---- Input Functions ---- ############
def inp():
    return(int(input()))
def inlt():
    return(list(map(int,input().split())))
def insr():
    s = input()
    return(list(s[:len(s) - 1]))
def invr():
    return(map(float,input().split()))

n, k = inlt() # read two numbers in a line
```

## 10.4 Test Generator

```
# usage: (note that test_generator.py refers to this file)
# 1. Modify the code below to generate the tests you want to
     use
# 2. Compile the 2 solutions to compare (e.g. A.cpp B.cpp
     into A B)
# 3. run: python3 test_generator.py A B
# Note that 'test_generator.py', 'A' and 'B' must be in the
    SAME FOLDER
# Note that A and B must READ FROM STANDARD INPUT, not from
    file,
# be careful with the usual freopen("input.in", "r", stdin)
    in them
import sys, subprocess
from datetime import datetime
from random import randint, seed

def buildTestCase(): # example of trivial "a+b" problem
 a = randint(1,100)
 b = randint(1,100)
 return f"{a} {b}\n"

seed(datetime.now().timestamp())
ntests = 100 # change as wanted
sol1 = sys.argv[1]
sol2 = sys.argv[2]
# Sometimes it's a good idea to use extra arguments that
    could then be
# passed to 'buildTestCase' and help you "shape" your tests
for curtest in range(ntests):
 test_case = buildTestCase()
 # Here the test is executed and outputs are compared
 print("running... ", end='')
 ans1 = subprocess.check_output(f"./{sol1}",
  input=test_case.encode('utf-8')).decode('utf-8')
 ans2 = subprocess.check_output(f"./{sol2}",
  input=test_case.encode('utf-8')).decode('utf-8')
 if ans1 == ans2:
  assert ans1 != "", 'ERROR?? ans1 = ans2 = empty string
     ("")'
  print("OK")
 else:
  print("FAILED!")
  print(test_case)
  print(f"ans from {sol1}:\n{ans1}")
  print(f"ans from {sol2}:\n{ans2}")
  break
```

## 10.5 Theory

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

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

**Burnside's lemma:** Given a group $G$ of symmetries and a set $X$, the number of elements of $X$ *up to symmetry* equals

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

where $X^g$ are the elements fixed by $g$ ($g.x = x$). If $f(n)$ counts "configurations" (of some sort) of length $n$, we can ignore rotational symmetry using $G = Z_n$ to get

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

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

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

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

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

- sub-diagonal monotone paths in an $n \times n$ grid.
- strings with $n$ pairs of parenthesis, correctly nested.

- binary trees with with $n+1$ leaves (0 or 2 children).
- binary search trees with $n$ vertices.
- binary trees with $n$ nodes is $C_n * n!$.
- ordered trees with $n+1$ vertices.
- ways a convex polygon with $n+2$ sides can be cut into triangles by connecting vertices with straight lines.
- permutations of $[n]$ with no 3-term increasing subseq.

**Stars and bars:** Count number of ways to partition a set of $n$ unlabeled objects into $k$ (possibly empty) labeled subsets:

$$\binom{n+k-1}{n}$$

**Stirling numbers:** Count number of ways to partition a set of $n$ labeled objects into $k$ nonempty unlabeled subsets:

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

**Bell numbers:** Count number of partitions of a set with $n$ members:

$$B_n = \sum_{k=0}^{n} S_{n,k}$$

**Binomial formula:**

$$(x+y)^n = \sum_{k=0}^{n}\binom{n}{k}x^{n-k}y^k = \sum_{k=0}^{n}\binom{n}{k}x^k y^{n-k}$$

**Number of Spanning Trees:** Create an $N \times N$ matrix `mat`, and for each edge $a \to b \in G$, do `mat[a][b]--`, `mat[b][b]++` (and `mat[b][a]--`, `mat[a][a]++` if $G$ is undirected). Remove the $i$th row and column and take the determinant; this yields the number of directed spanning trees rooted at $i$ (if $G$ is undirected, remove any row/column).

**Erdős–Gallai theorem:** A simple graph with node degrees $d_1 \geq \cdots \geq d_n$ exists iff $d_1 + \cdots + d_n$ is even and for every $k = 1 \dots n$,

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

**Equations:**

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

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

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

**Triangles:** Side lengths: $a, b, c$
Semiperimeter: $p = \dfrac{a+b+c}{2}$
Area: $A = \sqrt{p(p-a)(p-b)(p-c)}$
Circumradius: $R = \dfrac{abc}{4A}$
Inradius: $r = \dfrac{A}{p}$
Length of median (divides triangle into two equal-area triangles): $m_a = \frac{1}{2}\sqrt{2b^2+2c^2-a^2}$

**Spherical coordinates:**

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

**Sums:**

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

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

**Series:**

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

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

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

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

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

**Lagrange interpolation:** $n+1$ points determine the following polynomial of degree $n$:

$$f(x) = \sum_{i=1}^{n} y_i \prod_{i \neq j}\frac{x-x_j}{x_i-x_j}$$

Note: denominator can be get efficiently if x coordinates are all from 1 to $n+1$ using suffix and prefix arrays.
**Expectation is linear:**

$$E(aX+bY) = aE(X) + bE(Y)$$

**Pick's theorem:** $A = I + \frac{B}{2} - 1$
**Konig's Theorem:** In a bipartite graph, max matching = min vertex cover (cover edges using nodes).
Also, min edge cover (cover nodes using edges) = max independent set = N - min vertex cover = N - max matching

**Dilworth's Theorem:** An antichain in a partially ordered set is a set of elements no two of which are comparable to each other, and a chain is a set of elements every two of which are comparable. A chain decomposition is a partition of the elements of the order into disjoint chains. Dilworth's theorem states that for any partially ordered set, the sizes of the max antichain and of the min chain decomposition are equal. Equivalent to Konig's theorem on the bipartite graph $(U, V, E)$ where $U = V = S$ and $(u, v)$ is an edge when $u < v$. Those vertices outside the min vertex cover in both $U$ and $V$ form a max antichain.