**DSU:**

```
int parent[N];
int sz[N];
void make_set(int n){
    for (int i = 0; i <= n; i++){
        parent[i] = i;
        sz[i] = 1;
    }
}
int find_set(int v){
    if (v == parent[v]) return v;
    return parent[v] = find_set(parent[v]);
}
void union_sets(int a, int b){
    a = find_set(a);
    b = find_set(b);
    if (a != b){
        if (sz[a] < sz[b])
            swap(a, b);
        parent[b] = a;
        sz[a] += sz[b];
    }
}
```

**Centroid Decomposition:**

```
vector<int> g[N];
int del[N], sz[N], par[N], curSize;
void dfs(int u, int p){
    sz[u] = 1;
    for(int i = 0; i < g[u].size(); i++){
        int nd = g[u][i];
        if(nd == p || del[nd]) continue;
        dfs(nd, u);
        sz[u] += sz[nd];
    }
}
int findCentroid(int u, int p){
    for(int i = 0; i < g[u].size(); i++){
        int nd = g[u][i];
        if(nd == p || del[nd] || sz[nd] <= curSize
/ 2) continue;
        return findCentroid(nd, u);
    }
    return u;
}
void decompose(int u, int p){
    dfs(u, -1);
    curSize = sz[u];
    int cen = findCentroid(u, -1);
    // call solve function here
    if(p == -1) p = cen;
    par[cen] = p, del[cen] = 1;
    for(int i = 0; i < g[cen].size(); i++){
        int nd = g[cen][i];
        if(!del[nd]) decompose(nd, cen);
    }
}
```

**BIT:**

```
ll bit[N];
int n = N-1;
ll get(int i){
    ll sum = 0;
    while (i){
        sum += bit[i];
        i -= (i & -i);
    }
    return sum;
}
void update(int i, ll x){
    while (i <= n){
        bit[i] += x;
        i += (i & -i);
    }
}
void range_update(int l, int r, ll x){
    update(l, x);
    update(r + 1, -x);
}
```

**2D BIT:**

```
ll bit2d[N][N];
ll n = N-1, m = N-1;

ll get(ll x, ll y) {
    ll ret = 0;
    for(ll i=x; i>=0; i=(i&(i+1))-1)
        for(int j=y; j>=0; j=(j&(j+1))-1)
            ret += bit2d[i][j];
    return ret;
}

void update(ll x, ll y, ll delta) {
    for(int i=x; i<n; i=i|(i+1))
        for(int j=y; j<m; j=j|(j+1))
            bit2d[i][j] += delta;
}

ll query(ll x1, ll y1, ll x2, ll y2){
    return get(x2, y2)-get(x1-1, y2)-
        get(x2, y1-1)+get(x1-1, y1-1);
}
```

**Extended Euclidian:**

```
int gcd(int a, int b, int &x, int &y){
    if(b == 0){
        x = 1;
        y = 0;
        return a;
    }
    int x1, y1;
    int d = gcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a / b);
    return d;
}
```

**Random Number Generator:**

```
mt19937_64
rng(chrono::steady_clock::now().time_since_epoch()
.count());
inline ll gen_random(ll l, ll r) {
    return uniform_int_distribution<ll>(l, r)(rng);
}
inline double gen_random_real(double l, double r)
{
    return uniform_real_distribution<double>(l,
r)(rng);
}

mt19937_64 gen (random_device{}());
```

**Pragma:**

```
#pragma GCC optimize("Ofast,unroll-loops")
#pragma GCC target("avx,avx2,fma")
```

**Convex Hull Trick (CHT):**

```
struct line {
   ll m, c;
   line(ll _m, ll _c){
       m = _m, c = _c;
   }
};

struct CHT {
   vector<line> v;
   int t;
   void init(int type){
       t = type;
       v.clear();
   }

   bool bad(line l1, line l2, line l3){
       __int128 x3 = __int128(l3.c - l1.c)*(l1.m -
l2.m);
       __int128 x2 = __int128(l2.c - l1.c)*(l1.m -
l3.m);
       if(t==1) return x3 <= x2;
              // m decreasing - min query
       if(t==2) return x2 <= x3;
              // m decreasing - max query
       if(t==3) return x2 <= x3;
              // m increasing - min query
       if(t==4) return x3 <= x2;
              // m increasing - max query
   }

   void add(line l){
       int sz = v.size();
       while(sz>=2 && bad(v[sz-2], v[sz-1], l)){
           v.pop_back();
           sz--;
       }
       v.push_back(l);
   }
   // ternary search
   ll query(ll x){
       if(v.empty()) return 0;
       ll lo = 0, hi = v.size()-1;
       ll ret = (t&1) ? LL_INF : -LL_INF;
       while(lo <= hi){
           ll m1 = lo + (hi - lo)/3;
           ll m2 = hi - (hi - lo)/3;
           ll r1 = v[m1].m*x + v[m1].c;
           ll r2 = v[m2].m*x + v[m2].c;
           if(r1 > r2){
               if(t&1){
                   ret = min(ret, r2);
                   lo = m1+1;
               }
               else {
                   ret = max(ret, r1);
                   hi = m2-1;
               }
           }
           else {
               if(t&1){
                   ret = min(ret, r1), hi = m2-1;
               }
               else {
                   ret = max(ret, r2), lo = m1+1;
               }
           }
       }
       return ret;
   }
};
```

**Dynamic CHT:**

```
// Add lines of the form mx+c, and query max
values at points x.
// For min query, add line in (-m, -c) format.
Returns -ans.
struct Line {
   mutable ll m, c, p;
   bool isQuery;
   bool operator<(const Line& o) const {
       if(o.isQuery)
           return p < o.p;
       return m < o.m;
   }
};
struct LineContainer : multiset<Line> {
// (for doubles, use inf=1/.0, div(a,b)=a/b)
   const ll inf = LLONG_MAX;
   ll div(ll a, ll b){ // floor division
       return a/b - ((a^b)<0 && a%b);
   }
   bool isect(iterator x, iterator y){
       if (y == end()){x->p = inf; return false;}
       if(x->m == y->m) x->p = x->c > y->c ? inf :
-inf;
       else x->p = div(y->c - x->c, x->m - y->m);
       return x->p >= y->p;
   }
   void add(ll m, ll c){
       auto z = insert({m, c, 0, 0}), y = z++, x =
y;
       while(isect(y, z)) z = erase(z);
       if (x != begin() && isect(--x, y)) isect(x,
y = erase(y));
       while((y = x) != begin() && (--x)->p >=
y->p)
           isect(x, erase(y));
   }
   ll query(ll x){
       if(empty()) return inf;
       Line q; q.p = x, q.isQuery = 1;
       auto l = *lower_bound(q);
       return l.m * x + l.c;
   }
};
```

**Convex Hull (Monotone Chain):**

```
#define siz 100009
struct point {
   ll x, y;
};
point p[siz], hull[2 * siz];
ll sz = 0;
bool cmp(point a, point b){
   if(a.x != b.x)
       return a.x < b.x;
   return a.y < b.y;
}
ll cross (point a, point b, point c){
   return (b.x - a.x) * (c.y - a.y) - (b.y - a.y)
* (c.x - a.x);
}
void ConvexHull(ll n){
   // handle the case of n <= 2 manually
   sz = 0;
   sort(p, p + n, cmp);
   // Building lower hull
   for(ll i = 0; i < n; i++){
       while (sz > 1 && cross(hull[sz - 2],
hull[sz - 1], p[i]) <= 0) --sz;
       // use < 0 for taking co-linear points
```

```
        hull[sz++] = p[i];
    }
    // Building upper hull
    for(int i = n-2, j = sz + 1; i >= 0; i--){
        while (sz >= j && cross(hull[sz - 2],
hull[sz - 1], p[i]) <= 0) --sz;
        // use < 0 for taking co-linear points
        hull[sz++] = p[i];
    }
    sz-;
    //last point same as first point. so, sz--
    //sz is the size of convex hull
}
```

**Custom Hash for Unordered Map:**

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

    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM =
chrono::steady_clock::now().time_since_epoch().cou
nt();
        return splitmix64(x + FIXED_RANDOM);
    }
};
// Declaration: unordered_map <int, int,
custom_hash> numbers;
// Usage: same as normal unordered_map
// Ex: numbers[5] = 2;

// *** To use gp_hash_table (faster than
unordered_map) ****  //
// Add these extra two lines:
// #include <ext/pb_ds/assoc_container.hpp>
// using namespace __gnu_pbds;

// Declaration: gp_hash_table<int, int,
custom_hash> numbers;
// Usage: Same as unordered_map
```

**Max Flow (Dinic):**

```
const ll maxnodes = 10005;
ll nodes = maxnodes, src, dest;
ll dist[maxnodes], q[maxnodes], work[maxnodes];
struct Edge {
    ll to, rev;
    ll f, cap;
};
vector<Edge> g[maxnodes];
void addEdge(ll s, ll t, ll cap){
    Edge a = {t, (ll) g[t].size(), 0, cap};
    Edge b = {s, (ll) g[s].size(), 0, 0};
    g[s].push_back(a);
    g[t].push_back(b);
}
bool dinic_bfs(){
    fill(dist, dist + nodes, -1);
    dist[src] = 0;
    ll index = 0;
    q[index++] = src;
    for(ll i = 0; i < index; i++){
        ll u = q[i];
        for(ll j=0; j<(ll) g[u].size(); j++){
            Edge &e = g[u][j];
            if(dist[e.to] < 0 && e.f < e.cap){
                dist[e.to] = dist[u] + 1;
                q[index++] = e.to;
            }
        }
    }
    return dist[dest] >= 0;
}
ll dinic_dfs(ll u, ll f){
    if(u == dest) return f;
    for(ll &i=work[u]; i<(ll)g[u].size();i++){
        Edge &e = g[u][i];
        if (e.cap <= e.f) continue;
        if (dist[e.to] == dist[u] + 1){
            ll flow = dinic_dfs(e.to, min(f, e.cap
- e.f));
            if(flow > 0){
                e.f += flow;
                g[e.to][e.rev].f -= flow;
                return flow;
            }
        }
    }
    return 0;
}
ll maxFlow(ll _src, ll _dest){
    src = _src;
    dest = _dest;
    ll result = 0;
    while(dinic_bfs()){
        fill(work, work + nodes, 0);
        while(ll delta = dinic_dfs(src, INF))
            result += delta;
    }
    return result;
}
```

**Linear Diophantine Equation:**

```
// Diophantine Eqn: a*x + b*y = gcd(a, b)
// egcd computes one solution (x, y) for gcd(a,b)
= g
// Note: computed value g can be negative.
// Given one solution (x0, y0), other solutions
have form:
// xk = x0 + k*b/g and yk = y0 - k*a/g
ll egcd(ll a,ll b,ll &x,ll &y){
    if(a == 0) {x = 0; y = 1; return b;}
    ll x1,y1;
    ll gcd = egcd(b%a, a, x1, y1);
    x = y1 - (b / a) * x1; y = x1;
    return gcd;
}
inline ll Floor(ll p, ll q) {return p>0 ? p/q :
p/q - !!(p%q);}
inline ll Ceil(ll p, ll q)  {return p<0 ? p/q :
p/q + !!(p%q);}

// Number of solutions of Diophantine Eqn: Ax + By
= C
// A,B,C,x,y integers and X1 <= x <= X2 and Y1 <=
y <= Y2
inline ll solve(ll A,ll B,ll C,ll X1,ll X2,ll
Y1,ll Y2){
    if(A<0) {A = -A; B = -B; C = -C;}
    ll G = __gcd(A,B);
    if(G && C%G) return 0;
    if(A==0 && B==0) return (C==0) ? (X2 - X1 + 1)
* (Y2 - Y1 + 1) : 0;
    if(A==0) return (Y1 <= C/B && C/B <= Y2) ? (X2
- X1 + 1) : 0;
    if(B==0) return (X1 <= C/A && C/A <= X2) ? (Y2
- Y1 + 1) : 0;
```

```
    ll x,y;
    egcd(A,B,x,y);
    x = x * (C/G);  y = y * (C/G);
    ll newX1 = C - B*Y1, newX2 = C - B*Y2;
    if(newX1> newX2) swap(newX1, newX2);
    newX2 = Floor(newX2, A); newX1 = Ceil(newX1,
A);
    if(X1 > newX2) return 0;
    if(X2 < newX1) return 0;
    X1 = max(X1, newX1); X2 = min(X2, newX2);
    ll div = abs(B/G);
    if(x < X1) return (X2 - x) / div - (X1 - 1 - x)
/ div;
    if(x > X2) return (x - X1) / div - (x - X2 - 1)
/ div;
    return 1 + (x - X1) / div + (X2 - x) / div;
}
```

**Miller Rabin:**

```
ll mulmod(ll a, ll b, ll c){
    ll x = 0, y = a % c;
    while (b){
        if (b & 1) x = (x + y) % c;
        y = (y << 1) % c;
        b >>= 1;
    }
    return x % c;
}
ll fastPow(ll x, ll n, ll MOD){
    ll ret = 1;
    while (n){
        if(n&1) ret = mulmod(ret, x, MOD);
        x = mulmod(x, x, MOD);
        n >>= 1;
    }
    return ret % MOD;
}
const int a[9] = {2,3,5,7,11,13,17,19,23};
bool isPrime(ll n){
    if(n == 2 || n == 3) return true;
    if(n == 1 || !(n & 1)) return false;
    ll d = n - 1;
    int s = 0;
    while (d % 2 == 0) s++, d /= 2;
    for(int i = 0; i < 9; i++){
        if(n == a[i]) return true;
        bool comp = fastPow(a[i], d, n)!=1;
        if(comp)
            for(int j = 0; j < s; j++){
                ll fp = fastPow(a[i], (1LL <<
(ll)j)*d, n);
                if (fp == n - 1){
                    comp = false;
                    break;
                }
            }
        if(comp) return false;
    }
    return true;
}
```

**Geometry Basic:**

```
//rotate p by theta degrees CCW w.r.t point c
point rotate(point p, point c, double theta){
    double rad = DEG_to_RAD(theta);
        // multiply theta with PI / 180.0
    p.x -= c.x, p.y -= c.y;
    return point(p.x*cos(rad) - p.y*sin(rad) + c.x,
        p.x*sin(rad) + p.y*cos(rad) + c.y);
}
```

```
// in 3D, to rotate a vector v by θ angle around a
unit vector k describing the axis/line,
// v_rot=vcosθ+(k×v)sinθ+k(k.v)(1-cosθ)

/* Line Template Starts */
struct line { double a, b, c; }; // ax + by + c =
0
// the answer is stored in the third parameter
(pass by reference)
void pointsToLine(point p1, point p2, line &l){
    if (fabs(p1.x - p2.x) < EPS) {    // vertical
line is fine
        l.a = 1.0, l.b = 0.0, l.c = -p1.x; //
default values
    }
    else {
        l.a = -(double)(p1.y - p2.y) / (p1.x -
p2.x);
        l.b = 1.0; // IMPORTANT: we fix the value
of b to 1.0
        l.c = -(double)(l.a * p1.x) - p1.y;
    }
}
bool areParallel(line l1, line l2){ // check
coefficients a & b
    return (fabs(l1.a-l2.a) < EPS) &&
(fabs(l1.b-l2.b) < EPS);
}
bool areSame(line l1, line l2){ // also check
coefficient c
    return areParallel(l1 ,l2) && (fabs(l1.c -
l2.c) < EPS);
}
// returns true (+ intersection point) if two
lines are intersect
bool areIntersect(line l1, line l2, point &p){
    if (areParallel(l1, l2))
        return false; // no intersection
    // solve system of 2 linear algebraic equations
with 2 unknowns
    p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a *
l1.b - l1.a * l2.b);
    // special case: test for vertical line to
avoid division by zero
    if (fabs(l1.b) > EPS)
        p.y = -(l1.a * p.x + l1.c);
    else
        p.y = -(l2.a * p.x + l2.c);
    return true;
}
/* Line Template Ends */

/* Line Segment (Vector) Starts */
struct vec {
    double x, y; // name: 'vec' is different from
STL vector
    vec(double _x, double _y) : x(_x), y(_y) {}
};
vec toVec(point a, point b){  // convert 2 points
to vector a->b
    return vec(b.x - a.x, b.y - a.y);
}
vec scale(vec v, double s){  // nonnegative s =
[<1 .. 1 .. >1]
    return vec(v.x * s, v.y * s);   //
shorter.same.longer
}
point translate(point p, vec v){  // translate p
according to v
    return point(p.x + v.x, p.y + v.y);
}
```

```
double dot(vec a, vec b) { return (a.x * b.x + a.y
* b.y); }
double norm_sq(vec v) { return v.x * v.x + v.y *
v.y; }

// returns the distance from p to the line defined
by
// two points a and b (a and b must be different)
// the closest point is stored in the 4th
parameter (byref)
double distToLine(point p, point a, point b, point
&c){
   // formula: c = a + u * ab
   vec ap = toVec(a, p), ab = toVec(a, b);
   double u = dot(ap, ab) / norm_sq(ab);
   c = translate(a, scale(ab, u)); // translate a
to c
   return dist(p, c);  // Euclidean distance
between p and c
}

// returns the distance from p to the line segment
ab defined by
// two points a and b (still OK if a == b)
// the closest point is stored in the 4th
parameter (byref)
double distToLineSegment(point p, point a, point
b, point &c){
   vec ap = toVec(a, p), ab = toVec(a, b);
   double u = dot(ap, ab) / norm_sq(ab);
   if (u < 0.0) {
       c = point(a.x, a.y); // closer to a
       return dist(p, a); // Euclidean distance
between p and a
   }
   if (u > 1.0) {
       c = point(b.x, b.y); // closer to b
       return dist(p, b);  // Euclidean distance
between p and a
   }
   return distToLine(p, a, b, c);
}
double angle(point a, point o, point b){  //
returns angle aob in rad
   vec oa = toVec(o, a), ob = toVec(o, b);
   return acos(dot(oa, ob) / sqrt(norm_sq(oa) *
norm_sq(ob)));
}
/* Line Segment (Vector) Ends */

/* Circle Template Starts */
int insideCircle(point p, point c, int r){  // all
integer version
   int dx = p.x - c.x, dy = p.y - c.y;
   int Euc = dx * dx + dy * dy, rSq = r * r; //
all integer
   return Euc < rSq ? 0 : Euc == rSq ? 1 : 2;
//inside/border/outside
}

// Two circles intersecting in two points p1 and
p2
bool circle2PtsRad(point p1, point p2, double r,
point &c){
   double d2 = (p1.x - p2.x) * (p1.x - p2.x)+
               (p1.y - p2.y) * (p1.y - p2.y);
   double det = r * r / d2 - 0.25;
   if (det < 0.0)
       return false;
   double h = sqrt(det);
   c.x = (p1.x + p2.x) * 0.5 + (p1.y - p2.y) * h;
   c.y = (p1.y + p2.y) * 0.5 + (p2.x - p1.x) * h;
   return true;
} // to get the other center, reverse p1 and p2

// Line-Circle Intersection
int getLineCircleIntersection (Point p, Point q,
Circle O, double& t1, double& t2, vector<Point>&
sol){
   vec v = q - p;
   //sol.clear();
   double a = v.x, b = p.x - O.o.x, c = v.y, d =
p.y - O.o.y;
   double e = a*a+c*c, f = 2*(a*b+c*d), g =
b*b+d*d-O.r*O.r;
   double delta = f*f - 4*e*g;
   if (delta < -EPS) return 0;
   if (abs(delta)<=EPS) {
       t1 = t2 = -f / (2 * e);
       sol.push_back(p + v * t1);
       return 1;
   }

   t1 = (-f - sqrt(delta)) / (2 * e);
sol.push_back(p + v * t1);
   t2 = (-f + sqrt(delta)) / (2 * e);
sol.push_back(p + v * t2);
   return 2;
}

// Circle-Circle Intersection
double getLength (vec a) { return sqrt(dot(a, a));
}
vec ccw(vec a, double co, double si) {return
vec(a.x*co-a.y*si, a.y*co+a.x*si);}
vec cw (vec a, double co, double si) {return
vec(a.x*co+a.y*si, a.y*co-a.x*si);}
int getCircleCircleIntersection (Circle o1, Circle
o2, vector<Point>& sol){
   double d = getLength(o1.o - o2.o);
   if (abs(d)<=EPS) {
       if (abs(o1.r - o2.r)<=EPS) return -1;
       return 0;
   }
   if ((o1.r + o2.r - d) < -EPS) return 0;
   if ((fabs(o1.r-o2.r) - d) > EPS) return 0;

   vec v = o2.o - o1.o; // obj.o is the center
point
   double co = (o1.r*o1.r + getPLength(v) -
o2.r*o2.r) / (2 * o1.r * getLength(v));
   double si = sqrt(fabs(1.0 - co*co));
   Point p1 = scale(cw(v,co, si), o1.r) + o1.o;
   Point p2 = scale(ccw(v,co, si), o1.r) + o1.o;

   sol.push_back(p1);
   if (p1 == p2) return 1;
   sol.push_back(p2);
   return 2;
}
/* Circle Template Ends */

/* Triangle Template Starts */
// Radius of Inscribed Circle (incircle) &
Circumcircle
double rInCircle(double ab, double bc, double ca){
   return area(ab, bc, ca) / (0.5 * perimeter(ab,
bc, ca));
}
double rCircumCircle(double ab, double bc, double
ca){
   return ab * bc * ca / (4.0 * area(ab, bc, ca));
}
```

```
// if this function returns 1, ctr will be the
inCircle center
// and r is the same as rInCircle. returns 0 for
no incircle.
int inCircle(point p1, point p2, point p3, point
&ctr, double &r){
   r = rInCircle(dist(p1,p2), dist(p2,p3),
dist(p3,p1));
   if (fabs(r) < EPS)
       return 0; // no inCircle center
   line l1, l2; // compute these two angle
bisectors
   double ratio = dist(p1, p2) / dist(p1, p3);
   point p = translate(p2, scale(toVec(p2, p3),
ratio / (1 + ratio)));
   pointsToLine(p1, p, l1);
   ratio = dist(p2, p1) / dist(p2, p3);
   p = translate(p1, scale(toVec(p1, p3), ratio /
(1 + ratio)));
   pointsToLine(p2, p, l2);
   areIntersect(l1, l2, ctr); // get their
intersection point
   return 1;
}
bool pointInTriangle(Point a, Point b, Point c,
Point p){
    double s1 = getArea(a,b,c);
    double s2 = getArea(p,b,c) + getArea(p,a,b) +
getArea(p,c,a);
    return abs(s1-s2)<EPS;
}
/* Triangle Template Ends */
```

**Gaussian Elimination:**

```
/* Gaussian Elimination. Complexity: O(n^3)
* Equation System:
  a0*x0 + a1*x1 + ...+ an*xn = val0
  b0*x0 + b1*x1 + ...+ bn*xn = val1
  ...
* The Matrix Passes to Gauss() as matrix a:
  |a0 a1 ... an val0|
  |b0 b1 ... bn val1|
  |  ...         ..  |
  |  ...         valn|
* ans vector will contain the value of xi*/
const double eps = 1e-9;
int Gauss(vector<vector<double>> a, vector<double>
&ans){
   int n = (int)a.size(), m = (int)a[0].size() -
1;
   vector<int> pos(m, -1);
   double det = 1; int rank = 0;
   for(int col = 0, row = 0; col < m && row < n;
++col) {
       int mx = row;
       for(int i = row; i < n; i++)
if(fabs(a[i][col]) > fabs(a[mx][col])) mx = i;
       if(fabs(a[mx][col]) < eps) {det = 0;
continue;}
       for(int i = col; i <= m; i++)
swap(a[row][i], a[mx][i]);
       if (row != mx) det = -det;
       det *= a[row][col];
       pos[col] = row;
       for(int i = 0; i < n; i++) {
           if(i != row && fabs(a[i][col]) > eps) {
               double c = a[i][col] / a[row][col];
               for(int j = col; j <= m; j++)
a[i][j] -= a[row][j] * c;
           }
       }
       ++row; ++rank;
   }
   ans.assign(m, 0);
   for(int i = 0; i < m; i++) {
       if(pos[i] != -1) ans[i] = a[pos[i]][m] /
a[pos[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(fabs(sum - a[i][m]) > eps) return -1;
//no solution
   }
   for(int i = 0; i < m; i++) if(pos[i] == -1)
return 2; //infente solutions
   return 1; //unique solution
}
```

**FWHT:**

```
/* Fast Walsh Hadamard Transforms for XOR, AND, OR
Convolution
* Let, A = {1,2,2}, B = (3,4,5)
* All possible XOR between A, B will be:
   C = {1,1,2,4,5,6,6,7,7}
* Let, p1 is the frequency array of A,
       p2 is the frequency array of B,
       res will be the frequency array of C.
* FWHT calculates the res array in O(mlogm),
  where m is the size of the res array.
*/
// Define which is needed
#define bitwiseXOR
//#define bitwiseAND
//#define bitwiseOR

void FWHT(vector <ll> &p, bool inverse){
   int n = p.size();
   while(n&(n-1)){
       p.pb(0);
       n++;
   }
   for(int len = 1; 2*len<=n; len <<= 1){
       for(int i = 0; i<n; i += len+len){
           for(int j = 0; j < len; j++){
               ll u = p[i+j];
               ll v = p[i+len+j];
               #ifdef bitwiseXOR
               p[i+j] = u+v;
               p[i+len+j] = u-v;
               #endif // bitwiseXOR

               #ifdef bitwiseAND
               if(!inverse) {
                   p[i+j] = v;
                   p[i+len+j] = u+v;
               }
               else {
                   p[i+j] = v-u;
                   p[i+len+j] = u;
               }
               #endif // bitwiseAND

               #ifdef bitwiseOR
               if(!inverse) {
                   p[i+j] = u+v;
                   p[i+len+j] = u;
               }
               else {
                   p[i+j] = v;
                   p[i+len+j] = u-v;
               }
```

```
            #endif // bitwiseOR
        }
    }
  }

  #ifdef bitwiseXOR
  if(inverse){
      for(int i = 0; i < n; i++)
          p[i] /= n;
  }
  #endif // bitwiseXOR
}
vector <ll> calc(vector <ll> &p1, vector <ll>
&p2){
  FWHT(p1, 0), FWHT(p2, 0);
  vector <ll> res(p1.size());
  for(int i = 0; i < res.size(); ++i)
      res[i] = p1[i] * p2[i];
  FWHT(res, 1);
  return res;
}
// Just call calc(p1, p2);
// Note: p1, p2, res all are frequency arrays.
```

**L-R Flow:**

```
// LRFlow -> Max Flow with [l_i, r_i] range's flow
in edges
struct LRFlow {
  struct edge {
      int u, v; ll lo,hi; int id;
  };
  vector <edge> E;
  int N, superSource, superSink;
  void init(int n){
      N = n+10;
      superSource = n+1, superSink = n+2;
      E.clear();
  }
  void addEdgeLR(int u, int v, ll lo, ll hi, int
id=-1){
      E.pb({u, v, lo, hi, id});
  }
  bool feasible(int s, int t, ll lo=-1, ll
hi=-1){
      if(lo != -1) E.pb({t, s, lo, hi, -1});
      for0(i, N) g[i].clear(); // Clear the flow
graph
      ll target = 0;
      for(edge &e : E){
          if(e.lo>0) {
              addEdge(superSource,e.v,e.lo);
              addEdge(e.u, superSink, e.lo);
              target += e.lo;
          }
          addEdge(e.u, e.v, e.hi-e.lo);
      }
      ll flow = maxFlow(superSource, superSink);
      if(lo != -1) E.pop_back();
      if(flow < target) return false;
      return true;
  }
  ll maxFlowLR(ll s, ll t) {
      if(!feasible(s, t, 0, inf)) return -1;
      return maxFlow(s, t);
  }
};
// LRFlow lrf = LRFlow();
// lrf.init(n); to initialize with n nodes
// lrf.addEdgeLR(u, v, l, r, i); edge_i from u to
v. Capacity = [L,R]
// lrf.maxFlowLR(s, t); max flow from s to t
satisfying [l_i, r_i] range flows of edges
// g[], maxFlow(s, t), addEdge(u, v, c) are from
typical max flow algo.
```

**Suffix Array:**

```
// O(n log n) Suffix Array
#define MAX_N 1000020
int n, t;
char s[MAX_N];
int SA[MAX_N], LCP[MAX_N];
int RA[MAX_N], tempRA[MAX_N];
int tempSA[MAX_N];
int c[MAX_N];
int Phi[MAX_N], PLCP[MAX_N];

void countingSort(int k){    // O(n)
  int i, sum, maxi = max(300, n);
  // up to 255 ASCII chars or length of n
  memset(c, 0, sizeof c);
  // clear frequency table
  for(i = 0; i < n; i++)
  // count the frequency of each integer rank
  c[i + k < n ? RA[i + k] : 0]++;
  for(i = sum = 0; i < maxi; i++) {
      int t = c[i]; c[i] = sum; sum += t;
  }
  for(i = 0; i < n; i++)
      // shuffle the suffix array if necessary
      tempSA[c[SA[i] + k < n ? RA[SA[i] + k] :
0]++] = SA[i];
  for(i = 0; i < n; i++)
      // update the suffix array SA
      SA[i] = tempSA[i];
}
void buildSA(){
  int i, k, r;
  for(i = 0; i < n; i++) RA[i] = s[i];
  // initial rankings
  for(i = 0; i < n; i++) SA[i] = i;
  // initial SA: {0, 1, 2, ..., n-1}
  for(k = 1; k < n; k <<= 1) {
      // repeat sorting process log n times
      countingSort(k); // actually radix sort:
sort based on the second item
      countingSort(0);
      // then (stable) sort based on the first
item
      tempRA[SA[0]] = r = 0;
      // re-ranking; start from rank r = 0
      for(i = 1; i < n; i++)
          // compare adjacent suffixes
          tempRA[SA[i]] = // if same pair => same
rank r; otherwise, increase r
              (RA[SA[i]] == RA[SA[i - 1]] &&
RA[SA[i] + k] == RA[SA[i-1] + k]) ? r : ++r;
      for(i = 0; i < n; i++)
          // update the rank array RA
          RA[i] = tempRA[i];
      if(RA[SA[n - 1]] == n - 1) break;
      // nice optimization trick
  }
}
void buildLCP(){
  int i, L;
  Phi[SA[0]] = -1;
  // default value
  for(i = 1; i < n; i++)
      // compute Phi in O(n)
      Phi[SA[i]] = SA[i - 1];
  // remember which suffix is behind this suffix
```

```
    for(i = L = 0; i < n; i++) {
        // compute Permuted LCP in O(n)
        if(Phi[i] == -1) { PLCP[i] = 0; continue; }
        // special case
        while(s[i + L] == s[Phi[i] + L]) L++;
        // L increased max n times
        PLCP[i] = L;
        L = max(L - 1, 0);
        // L decreased max n times
    }
    for(i = 0; i < n; i++)
        // compute LCP in O(n)
        LCP[i] = PLCP[SA[i]];
        // put the permuted LCP to the correct
position
}
// n = string length + 1
// s = the string
// memset(LCP, 0, sizeof(LCP));  setting all index
of LCP to zero
// buildSA(); for building suffix array
// buildLCP(); for building LCP array
// LCP is the longest common prefix with the
previous suffix here
// SA[0] holds the empty suffix "\0".
```

**Pollard Rho:**

```
/// find any divisor of (n) in ~O(n^(1/4))
namespace PollardRho {
    mt19937
rnd(chrono::steady_clock::now().time_since_epoch()
.count());
    const int P = 1e6 + 9;
    ll seq[P];
    inline ll add_mod(ll x, ll y, ll m){
        return (x += y) < m ? x : x - m;
    }
    inline ll mul_mod(ll x, ll y, ll m){
        return (__int128)x*y % m;
    }
    ll pollard_rho(ll n){
        if(n<=1) return 1;
        if(isPrime(n)) return n;
        while(1){
            ll x = rnd() % n, y = x, c = rnd() % n,
u = 1, v, t = 0;
            ll *px = seq, *py = seq;
            while (1) {
                *py++ = y = add_mod(mul_mod(y, y,
n), c, n);
                *py++ = y = add_mod(mul_mod(y, y,
n), c, n);
                if((x = *px++) == y) break;
                v = u;
                u = mul_mod(u, abs(y - x), n);
                if(!u) __gcd(v, n);
                if(++t == 32){
                    t = 0;
                    if ((u = __gcd(u, n)) > 1 && u
< n) return u;
                }
            }
            if(t && (u = __gcd(u, n)) > 1 && u < n)
return u;
        }
    }
}
// isPrime(n) -> use Miller-Rabin or similar
efficient primality test
// long long divisor = PollardRho::pollard_rho(n);
// to find one (any) divisor of n
```

**MO's on Tree:**

```
//find distinct "weights in nodes" of the path
(from node u to node v)
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 40005;
const int MAXM = 100005;
const int LN = 19;
int N, M, K, cur, A[MAXN], LVL[MAXN],
pa[LN][MAXN];
int BL[MAXN << 1], ID[MAXN << 1], FREQ[MAXN],
ANS[MAXM];
int d[MAXN], in[MAXN], out[MAXN];
bool VIS[MAXN];
vector < int > graph[MAXN];
struct query {
    int id, l, r, lc;
    bool operator < (const query& rhs){
        if(BL[l] != BL[rhs.l])
            return BL[l] < BL[rhs.l];
        if(BL[l] & 1)
            return r < rhs.r;
        return r > rhs.r;
    }
} Q[MAXM];
// Set up Stuff
void dfs(int u, int par, int d){
    in[u] = ++cur, ID[cur] = u;
    pa[0][u] = par, LVL[u] = d;
    for(auto &v : graph[u]){
        if (v == par)
            continue;
        dfs(v, u, d+1);
    }
    out[u] = ++cur, ID[cur] = u;
}

// Function returns lca of u and v
int LCA(int u, int v){
    if(LVL[u] < LVL[v]) swap(u,v);
    int diff = LVL[u] - LVL[v];
    for(int i = 0; i < LN; i++) if( (diff>>i)&1 ) u
= pa[i][u];
    if(u == v) return u;
    for(int i = LN-1; i >= 0; i--){
        if(pa[i][u] != pa[i][v]) {
            u = pa[i][u];
            v = pa[i][v];
        }
    }
    return pa[0][u];
}
inline void check(int x, int& res){
    // If (x) occurs twice, then don't consider
it's value
    if(VIS[x]){
        if(--FREQ[A[x]] == 0)
            res--;
    }
    else if(!VIS[x]){
        if(FREQ[A[x]]++ == 0)
            res++;
    }
    VIS[x] ^= 1;
}

void compute(){
    // Perform standard Mo's Algorithm
    int curL = Q[0].l, curR = Q[0].l - 1, res = 0;
    for(int i = 0; i < M; i++){
```

```
        while(curL < Q[i].l)
            check(ID[curL++], res);
        while(curL > Q[i].l)
            check(ID[--curL], res);
        while(curR < Q[i].r)
            check(ID[++curR], res);
        while(curR > Q[i].r)
            check(ID[curR--], res);
        int u = ID[curL], v = ID[curR];
        // Case 2
        if(Q[i].lc != u and Q[i].lc != v)
            check(Q[i].lc, res);
        ANS[Q[i].id] = res;
        if(Q[i].lc != u and Q[i].lc != v)
            check(Q[i].lc, res);
    }
    for(int i = 0; i < M; i++)
        printf("%d\n", ANS[i]);
}
void init(int N) {
    cur = 0;
    for(int i = 1; i <= N; i++) {
        graph[i].clear();
        VIS[i] = FREQ[i] = 0;
        for(int j=0; j<LN; j++) pa[j][i] = -1;
    }
}
int main(){
    int u, v, x;
    scanf("%d %d", &N, &M);
    init(N);
    // Inputting Values
    for (int i = 1; i <= N; i++) {
        scanf("%d", &A[i]);
        d[i] = A[i];
    }
    // Compressing Coordinates
    sort(d + 1, d + N + 1);
    K = unique(d + 1, d + N + 1) - d - 1;
    for (int i = 1; i <= N; i++)
        A[i] = upper_bound(d + 1, d + K + 1, A[i])
- d;
    // Inputting Tree
    for (int i = 1; i < N; i++){
        scanf("%d %d", &u, &v);
        graph[u].push_back(v);
        graph[v].push_back(u);
    }
    dfs(1, -1, 0);
    // Build Sparse Table
    for(int i=1; i<LN; i++)
        for(int j=1; j<=N; j++)
            if(pa[i-1][j] != -1)
                pa[i][j] = pa[i-1][pa[i-1][j]];
    int size = sqrt(cur);
    for(int i = 1; i <= cur; i++)
        BL[i] = (i - 1) / size + 1;
    for(int i = 0; i < M; i++){
        scanf("%d %d", &u, &v);
        Q[i].lc = LCA(u, v);
        if(in[u] > in[v])
            swap(u, v);
        if(Q[i].lc == u)
            Q[i].l = in[u], Q[i].r = in[v];
        else
            Q[i].l = out[u], Q[i].r = in[v];
        Q[i].id = i;
    }
    sort(Q, Q + M);
    compute();
}
```

**Palindromic Tree:**

```
int tree[N][26], idx;
ll len[N], link[N], cnt[N], t;
char s[N]; // 1-indexed
void add(ll p){
   while(s[p - len[t] - 1] != s[p]) t = link[t];
   // searching node for creating pTp type
palindrome.
   ll x = link[t], c = s[p] - 'a';
   while(s[p - len[x] - 1] != s[p]) x = link[x];
   // searching node to link pXp type palindrome,
where pXp is a proper suffix.
   if(!tree[t][c]){
       tree[t][c] = ++idx;
       len[idx] = len[t] + 2;
       link[idx] = len[idx] == 1 ? 2 : tree[x][c];
   }
   t = tree[t][c];
   cnt[t]++;
}
/* node 1 and node 2 are the two roots.
* idx-2 is the number of total distinct
palindromes in the string s.
* Let, a node is i,
* len[i] represents the length of the palindrome
represented by node i.
* link[i] represents the node containing the
palindrome which is the largest proper suffix
* of the palindrome of node i.
*/
int main(){
   len[1] = -1, link[1] = 1;
   len[2] = 0, link[2] = 1;
   idx = t = 2;
   memset(tree, 0, sizeof(tree));
   memset(cnt, 0, sizeof(cnt));
   scanf("%s", s+1);
   ll len = strlen(s+1);
   for(ll i = 1; i <= len; i++) add(i);
   // adding each index in pal tree one by one.
O(len).
   for(ll i = idx; i > 2; i--) cnt[ link[i] ] +=
cnt[i];
   // adding count to the suffix link.
   // cnt[i] now holds the count of the palindrome
represented by node i in the string s.
   return 0;
}
```

**Heavy Light Decomposition:**

```
/* Operations:
* 1 u x : set val[u] = x
* 2 u v : sum of val[i] in (u,v) path */
#define ll long long
#define pb push_back
const ll sz = 3e4 + 10;
vector <ll> g[sz];
ll sub[sz], in[sz], out[sz], head[sz], tim;
ll par[sz], tr[4*sz];
void dfs_siz(ll u, ll p){
   sub[u] = 1, par[u] = p;
   for(ll &v : g[u]){
       if(v == p) continue;
       dfs_siz(v, u);
       sub[u] += sub[v];
       if(sub[v] > sub[ g[u][0] ])
           swap(v, g[u][0]);
   }
}
/* DFS for Heavy-Light Decomposition
```

```
* head[u] = Head of the chain of node u
* Operations can be performed in
[in[head[u]],in[u]] range.
* As DFS-time is used, [in[u],out[u]] range can be
used
  for performing operations on the subtree of node
u. */
void dfs_hld(ll u, ll p){
   if(p == -1) head[u] = u; // root
   in[u] = ++tim;
   for(ll &v : g[u]) {
       if(v == p) continue;
       head[v] = (v==g[u][0]? head[u] : v);
       dfs_hld(v, u);
   }
   out[u] = tim;
}
// Typical Segment Tree on [1, tim] range
void build(ll lo, ll hi, ll node){
   if(lo == hi) {
       tr[node] = 0;
       return;
   }
   ll mid = lo+hi>>1, lft=node<<1, rgt=lft|1;
   build(lo, mid, lft);
   build(mid+1, hi, rgt);
   tr[node] = 0;
}
void update(ll lo, ll hi, ll idx, ll v, ll node){
   if(lo > idx || hi < idx) return;
   if(lo == hi){
       tr[node] = v;
       return;
   }
   ll mid = lo+hi>>1, lft=node<<1, rgt=lft|1;
   update(lo, mid, idx, v, lft);
   update(mid+1, hi, idx, v, rgt);
   tr[node] = tr[lft] + tr[rgt];
}
ll query(ll lo, ll hi, ll l, ll r, ll node){
   if(lo > r || hi < l)
       return 0;
   if(lo >= l && hi <= r)
       return tr[node];
   ll mid = lo+hi>>1, lft=node<<1, rgt=lft|1;
   return query(lo, mid, l, r, lft)
       +  query(mid+1, hi, l, r, rgt);
}
// Segment Tree Ends
inline bool isAncestor(int p,int u){
   return in[p]<=in[u]&&out[p]>=out[u];
}
void upd_val(ll u, ll val) {
   update(1, tim, in[u], val, 1);
}
ll query_path(ll u, ll v){
   ll ret = 0;
   while(!isAncestor(head[u],v)) {
       ret += query(1,tim, in[head[u]], in[u], 1);
       u=par[head[u]];
   }
   swap(u,v);
   while(!isAncestor(head[u],v)) {
       ret += query(1,tim, in[head[u]], in[u], 1);
       u=par[head[u]];
   }
   if(in[v]<in[u])swap(u,v);
   ret += query(1,tim, in[u], in[v], 1); // u is
LCA
   return ret;
}
void clr(ll n){
   tim = 0;
   for(ll i=1; i <=n; i++) g[i].clear();
}
int main(){
   ll t, n, q, u, v, op, root=1;
   cin >> t;
   while(t--) {
       scanf("%lld", &n); clr(n);
       for(ll i=1; i<n; i++){
           scanf("%lld %lld", &u, &v);
           g[u].pb(v), g[v].pb(u);
       }
       dfs_siz(root, -1);
       dfs_hld(root, -1);
       build(1, tim, 1);
       scanf("%lld", &q);
       while(q--){
           scanf("%lld %lld %lld",&op,&u,&v);
           if(op == 1) upd_val(u, v);
           else printf("%lld\n", query_path(u,
v));
       }
   }
   return 0;
}
```

**FFT:**

```
/* Multiply (7x^2 + 8x^1 + 9x^0) with (6x^1 +
5x^0)
  * ans = 42x^3 + 83x^2 + 94x^1 + 45x^0
  * A = {9, 8, 7}
  * B = {5, 6}
  * V = multiply(A,B)
  * V = {45, 94, 83, 42} ***/
/*** Tricks
  * Use vector < bool > if you need to check only
the status of the sum
  * Use bigmod if the power is over same
polynomial && power is big
  * Use long double if you need more precision
  * Use long long for overflow
***/
typedef vector <int> vi;
const double PI = 2.0 * acos(0.0);
using cd = complex<double>;
void fft(vector<cd> & a, bool invert = 0){
   int n = a.size();
   for(int i = 1, j = 0; i < n; i++){
       int bit = n >> 1;
       for (; j & bit; bit >>= 1) j ^= bit;
       j ^= bit;
       if (i < j) swap(a[i], a[j]);
   }
   for(int len = 2; len <= n; len <<= 1){
       double ang = 2*PI/len*(invert?-1:1);
       cd wlen(cos(ang), sin(ang));
       for(int i = 0; i < n; i += len){
           cd w(1);
           for (int j = 0; j < len/2; j++){
               cd u = a[i+j], v = a[i+j+len/2] *
w;
               a[i+j] = u + v;
               a[i+j+len/2] = u - v;
               w *= wlen;
           }
       }
   }
   if(invert){
       for(cd & x : a) x /= n;
   }
}
```

```
void ifft(vector <cd> & p){
   fft(p, 1);
}
vi multiply(vi const& a, vi const& b){
   vector<cd> fa(a.begin(), a.end()),
fb(b.begin(), b.end());
   int n = 1;
   while (n < a.size() + b.size())
       n <<= 1;
   fa.resize(n);
   fb.resize(n);
   fft(fa);
   fft(fb);
   for (int i = 0; i < n; i++)
       fa[i] *= fb[i];
   ifft(fa);
   vi result(n);
   for (int i = 0; i < n; i++)
       result[i] = round(fa[i].real());
   return result;
}
```

**Chinese Remainder Theorem (CRT):**

```
/** A CRT solver which works even when moduli are
not pairwise coprime
   1. Add equations using addEquation() method
   2. Call solve() to get {x, N} pair, where x is
the unique solution modulo N.
   Assumptions:
       1. LCM of all mods will fit into long long.
*/
class ChineseRemainderTheorem {
   typedef long long vlong;
   typedef pair<vlong,vlong> pll;
   /** CRT Equations stored as pairs of vector.
See addEqation()*/
   vector<pll> equations;
public:
   void clear(){
       equations.clear();
   }
  /**Add equation of the form x = r (mod m)*/
   void addEquation( vlong r, vlong m ){
       equations.push_back({r, m});
   }
   pll solve(){
       if (equations.size() == 0) return {-1,-1};
/// No equations to solve
       vlong a1 = equations[0].first;
       vlong m1 = equations[0].second;
       a1 %= m1;
       /** Initially x = a_0 (mod m_0)*/
       /** Merge the solution with remaining
equations */
       for(int i = 1; i < equations.size(); i++ ){
           vlong a2 = equations[i].first;
           vlong m2 = equations[i].second;
           vlong g = __gcd(m1, m2);
           if(a1%g != a2%g) return {-1,-1}; ///
Conflict in equations
           /** Merge the two equations*/
           vlong p, q;
           ext_gcd(m1/g, m2/g, &p, &q);
           vlong mod = m1 / g * m2;
           vlong x = ((__int128)a1 * (m2/g) % mod
*q % mod + (__int128)a2 * (m1/g) % mod * p % mod)
% mod;
           /** Merged equation*/
           a1 = x;
           if ( a1 < 0 ) a1 += mod;
           m1 = mod;
       }
       return {a1, m1};
   }
};
```

Euler's Totient of N

```
ll phi(ll n){
   ll res = n;
   for(ll i=0; p[i]*p[i] <= n; i++){
       if (n % p[i]== 0){
       // subtract multiples of p[i] from r
           res -= (res / p[i]);
           while (n % p[i]== 0)
               n /= p[i];
       }
   }
   if (n > 1) res -= (res / n);
   return res;
}
```

**Matrix Exponentiation:**

```
#define ll long long
const ll MOD = 1e9 + 7;
const ll MOD2 = MOD * MOD;   /// Only when (MOD *
MOD) fits into long long
#define row 2
#define col 2
struct MatExpo {
   ll exponents[64][row][col];
   ll ident[row][col] = { {1, 0}, {0, 1} };  ///
Identity Matrix
   ll result[row][col], mat[row][col];
   MatExpo(){
       /// Creating Base Matrix
       ll base[row][col] = {{1, 1}, {1, 0}};
       memcpy(exponents[0], base, sizeof(base));
       /// Calculating all exponents
       for(ll p = 1; p < 62; p++) {
           for(ll i = 0; i < row; i++) {
               for(ll j = 0; j < col; j++) {
                   ll tmp = 0;
                   for(ll k = 0; k < col; k++){
                       tmp += exponents[p -
1][i][k] * exponents[p - 1][k][j];
                       while(tmp >= MOD2) ///
faster because we can do it by subtraction
                           tmp -= MOD2;
                   }
                   exponents[p][i][j] = tmp % MOD;
               }
           }
       }
   }
   ll ans(ll m) {
       /// Return from base case
       if(m == 0 || m == 1)
           return 1;
       memcpy(mat, ident, sizeof(ident));
       ll n = m - 1;   /// Here, (n - 1)th power
of base matrix represents the nth term
       for(ll p = 60; p >= 0; p--) {
           if((n >> p) & 1) {
               for(ll i = 0; i < row; i++) {
                   for(ll j = 0; j < col; j++) {
                       ll tmp = 0;
                       for(ll k = 0; k < col; k++)
{
                           tmp += mat[i][k] *
exponents[p][k][j];
```

```
                        while(tmp >= MOD2) ///
Taking modulo MOD2 is easy, because we can do it
by subtraction
                            tmp -= MOD2;
                    }
                    result[i][j] = tmp % MOD;
                }
            }
            memcpy(mat, result,
sizeof(result));
        }
    }
    return (result[0][0]+result[0][1])%MOD;
  }
};
// MatExpo ex = MatExpo();
// ans = ex.ans(n);      nth term, term count
starts from 0
```

**Matrix Expo Optimized:**

```
/// O(n^2logn) per query if matrix is fixed for
all queries
#define ll long long
const ll MOD = 1e9 + 7;
const ll MOD2 = MOD * MOD;   /// Only when (MOD *
MOD) fits into long long
#define row 2
#define col 2
ll exponents[64][row][col];
ll result[row], vect[row];
ll base[row][col] = { {1, 1}, {1, 0} }; /// Base
Matrix
ll baseVect[row] = {1, 1};  /// fibonacci sequence
{1, 1, 2.....} here. So,f(1)=f(0)=1.
struct MatExpo{
   MatExpo(){
      memcpy(exponents[0], base, sizeof(base));
      /// Calculating all exponents
      for(ll p = 1; p < 62; p++){
         for(ll i = 0; i < row; i++){
            for(ll j = 0; j < col; j++){
               ll tmp = 0;
               for(ll k = 0; k < col; k++){
                  tmp += exponents[p -
1][i][k] * exponents[p - 1][k][j];
                  while(tmp >= MOD2) ///
Taking modulo MOD2 is easy, because we can do it
by subtraction
                     tmp -= MOD2;
               }
               exponents[p][i][j]=tmp%MOD;
            }
         }
      }
   }

   ll ans(ll m){
      /// Return from base case
      if(m == 0 || m == 1) return 1;
      memcpy(vect, baseVect, sizeof(baseVect));
      ll n = m - 1, tmp; // Here, (n - 1)th power
of base matrix represents the nth term
      for(ll p = 60; p >= 0; p--){
         if((n >> p) & 1){
            for(ll i = 0; i < row; i++){
               tmp = 0;
               for(ll j = 0; j < col; j++){
                  tmp += exponents[p][i][j] *
vect[j];
                  while(tmp >= MOD2) ///
faster, because we can do it by subtraction
                     tmp -= MOD2;
               }
               result[i] = tmp % MOD;
            }
            memcpy(vect, result,
sizeof(result));
         }
      }
      return result[0] % MOD;
   }
};
// MatExpo ex = MatExpo();
// ans = ex.ans(n);     nth term, term count
starts from 0
```

**SOS DP:**

```
/// Suboptimal Bruteforce Method O(3^n):
// iterate over all the masks
for (int mask = 0; mask < (1<<n); mask++) {
   F[mask] = A[0];
   //iterate over all the subsets of the mask
   for(int i = mask; i > 0; i = (i-1) & mask){
      F[mask] += A[i];
   }
}
```

**Some Macros:**

```
#include <bits/stdc++.h>
using namespace std;
#define fastio
std::ios_base::sync_with_stdio(false);cin.tie(NULL
);cout.tie(NULL);
#ifdef DEBUG
#define dbg(x)
{cerr<<__func__<<":"<<__LINE__<<"\t"<<#x<<" =
"<<x<<endl;}
#endif
```

**Aho Corasick:**

```
// Aho-Corasick
// Complexity : |Text| + Sum of all |Pattern| +
O(number of Occurrences)
// if occurrence positions needed, Worst Case
Complexity : (SumLen) Root (SumLen)
#include <bits/stdc++.h>
using namespace std;
const int MAXT = 1000005;  // Length of Text
const int MAXP = 1000005;  // Sun of all |Pattern|
const int MAXQ = 1000005;  // Number of Patterns
int n;
map<char,int> Next[MAXP];
int Root;                   // AC automaton Root
int Nnode;                  // Total node count
int Link[MAXP];             // failure links
int Len[MAXP];              // Len[i] = length of
i-th pattern
vector<int> End[MAXP];      // End[i] = indices of
patterns those end in node i
// vector<int> Occ[MAXQ];  // Occ[i] = occurrences
of i-th pattern
vector<int> edgeLink[MAXP];
vector<int> perNodeText[MAXP];
int in[MAXQ], out[MAXQ];
int euler[MAXT];
int Time;
void Clear(int node){
   Next[node].clear();
   End[node].clear();
   edgeLink[node].clear();
```

```
   perNodeText[node].clear();
}
void init(){
   Time = 0;
   Root = Nnode = 0;
   Clear(Root);
}
void insertword(string p,int ind){
   int len = p.size();
   int now = Root;
   for(int i=0; i<len; i++){
       if(!Next[now][p[i]]){
           Next[now][p[i]] = ++Nnode;
           Clear(Nnode);
       }
       now = Next[now][p[i]];
   }
   End[now].push_back(ind);
}
void push_links(){
   queue<int> q;
   Link[0] = -1;
   q.push(0);
   while(!q.empty()){
       int u = q.front();
       q.pop();
       for(auto edge : Next[u]){
           char ch = edge.first;
           int v = edge.second;
           int j = Link[u];
           while(j != -1 && !Next[j][ch]) j =
Link[j];
           if(j != -1) Link[v] = Next[j][ch];
           else Link[v] = 0;
           q.push(v);
           edgeLink[Link[v]].push_back(v);
           // for(int x : End[Link[v]])
End[v].push_back(x);
       }
   }
}
void traverse(string s){
   int len = s.size();
   int now = Root;
   for(int i = 0; i < len; i++)     {
       while(now != -1 && !Next[now][s[i]]) now =
Link[now];
       if(now!=-1) now = Next[now][s[i]];
       else now = 0;
       perNodeText[now].push_back(i+1);  // using
1 based indexing for text indices
       // for(int x=0;x<End[now].size();x++)
Occ[End[now][x]].push_back(i);
   }
}

// After dfs, the occurence of ith query string
will be the count of
// all the occurrence of the subtree under the
endNode of ith string
void dfs(int pos){
   for(int q : End[pos]) in[q] = Time + 1;
   for(int val : perNodeText[pos]) euler[++Time] =
val;
   for(int to : edgeLink[pos]) dfs(to);
   for(int q : End[pos]) out[q] = Time;
}
int main(){
   // init();
   // insert(keys[i], i); for inserting the ith
keyword
   // push_links();
   // traverse(s);
   // dfs(Root);
}
```

**KMP:**

```
const ll MAX_N = 1e5+10;
char s[MAX_N], pat[MAX_N];  // 1-indexed
ll lps[MAX_N];     // lps[i] = longest proper
prefix-suffix in i length's prefix
void gen_lps(ll plen){
   ll now;
   lps[0] = lps[1] = now = 0;
   for(ll i = 2; i <= plen; i++) {
       while(now != 0 && pat[now+1] != pat[i])
           now = lps[now];
       if(pat[now+1] == pat[i]) lps[i] = ++now;
       else lps[i] = now = 0;
   }
}
ll KMP(ll slen, ll plen){
   ll now = 0;
   for(ll i = 1; i <= slen; i++) {
       while(now != 0 && pat[now+1] != s[i])
           now = lps[now];
       if(pat[now+1] == s[i]) ++now;
       else now = 0;
      // now is the length of the longest prefix
of pat, which
      // ends as a substring of s in index i.
       if(now == plen) return 1;
   }
   return 0;
}

// slen = length of s, plen = length of pat
// call gen_lps(plen); to generate LPS (failure)
array
// call KMP(slen, plen) to find pat in s
```

**Lagrange Interpolation:**

```
ll bigMod(ll n, ll r) {
   if (r==0)   return 1LL;
   ll ret = bigMod (n, r/2);
   ret = (ret * ret)% MOD;
   if (r %2 ) ret = (ret * n) % MOD;
   return ret;
}
ll Point[MAXN];
ll Fact[MAXN];
// Calculate first k + 1 points (0 to k) on the
polynomial
// where k = degree of the polynomial
// Then find f(x) for any x using interpolation in
O(n log(MOD))
ll interpolate(int n,ll x) {
   if(x <= n) return Point[x];
   ll num = 1;
   for(int i=0; i<=n; i++) num=(num*(x-i)) % MOD;
   ll ret = 0;
   for(int i=0; i<=n; i++) {
       ll nn = (num * bigMod(x-i, MOD-2)) % MOD;
       ll dd = (Fact[n-i] * Fact[i]) % MOD;
       if((n-i) & 1) dd = MOD -dd;
       nn = (Point[i] * nn) % MOD;
       ret = (ret + nn * bigMod(dd, MOD-2))%MOD;
   }
   return ret;
}
```

**NTT:**

```
/**Iterative Implementation of Number Theoretic
Transform
Complexity: O(N log N)
Slower than regular fft
Possible Optimizations:
1. Remove trailing zeroes
2. Keep the mod const
Suggested mods (mod, root, inv, pw) :
7340033, 5, 4404020, 1<<20
13631489, 11799463,6244495, 1<<20
23068673, 177147,17187657, 1<<21
463470593, 428228038, 182429, 1<<21
415236097, 73362476, 247718523, 1<<22
918552577, 86995699, 324602258, 1<<22
998244353, 15311432, 469870224, 1<<23
167772161, 243, 114609789, 1<<25
469762049, 2187, 410692747, 1<<26
If required mod is not above, use nttdata function
OFFLINE.
If pw=1<<k, a polynomial can have at most (1<<k)
degree.**/
namespace ntt {
   int N;
   vector<int> perm;
   vector<int>wp[2][30];
   const int mod = 998244353, root = 15311432, inv
= 469870224, pw = 1<<23;
   int power(int a, int p) {
       if (p==0)   return 1;
       int ans = power(a, p/2);
       ans = (ans * 1LL * ans)%mod;
       if (p%2)    ans = (ans * 1LL * a)%mod;
       return ans;
   }
   void precalculate() {
       perm.resize(N);
       perm[0] = 0;
       for (int k=1; k<N; k<<=1) {
           for (int i=0; i<k; i++) {
               perm[i] <<= 1;
               perm[i+k] = 1 + perm[i];
           }
       }
       for (int b=0; b<2; b++) {
           for (int idx = 0, len = 2; len <= N;
idx++, len <<= 1) {
               int factor = b ? inv : root;
               for (int i = len; i < pw; i <<= 1)
                   factor =
(factor*1LL*factor)%mod;
               wp[b][idx].resize(N);
               wp[b][idx][0] = 1;
               for (int i = 1; i < len; i++)
                   wp[b][idx][i] =
(wp[b][idx][i-1]*1LL*factor)%mod;
           }
       }
   }
   void fft(vector<int> &v, bool invert = false) {
       if (v.size() != perm.size()) {
           N = v.size();
           assert(N && (N&(N-1)) == 0);
           precalculate();
       }
       for (int i=0; i<N; i++)
           if (i < perm[i])
               swap(v[i], v[perm[i]]);
       for (int idx = 0, len = 2; len <= N; idx++,
len <<= 1) {
           for (int i=0; i<N; i+=len) {
               for (int j=0; j<len/2; j++) {
                   int x = v[i+j];
                   int y =
(wp[invert][idx][j]*1LL*v[i+j+len/2])%mod;
                   v[i+j] = (x+y>=mod ? x+y-mod :
x+y);
                   v[i+j+len/2] = (x-y>=0 ? x-y :
x-y+mod);
               }
           }
       }
       if (invert) {
           int n1 = power(N, mod-2);
           for (int &x : v) x = (x*1LL*n1)%mod;
       }
   }
   vector<int> multiply(vector<int> a,vector<int> b){
       int n = 1;
       while(a.back()==0&&!a.empty()) a.pop_back();
       while(b.back()==0&&!b.empty()) b.pop_back();
       while (n < a.size()+ b.size())  n<<=1;
       a.resize(n), b.resize(n);
       fft(a), fft(b);
       for(int i=0; i<n; i++) a[i] = (a[i]*1LL*
b[i])%mod;
       fft(a, true);
       return a;
   }
};
const int M = 998244353, N = 2e6;
int main() {
   std::ios_base::sync_with_stdio(false);
   cin.tie(NULL); cout.tie(NULL);
   vector<int> a(N), b(N);
   long long asum = 0, bsum = 0, csum = 0;
   for (int i=0; i<N; i++) asum += (a[i] =
rand()%M);
   for (int i=0; i<N; i++) bsum += (b[i] =
rand()%M);
   vector<int> c = NTT::multiply(a, b);
   for (int x: c)   csum += x;
   cout<<csum<<endl;
}
int power(int a, int p, int mod) {
   if (p==0)   return 1;
   int ans = power(a, p/2, mod);
   ans = (ans * 1LL * ans)%mod;
   if (p%2)    ans = (ans * 1LL * a)%mod;
   return ans;
}
/** Find primitive root of p assuming p is prime.
if not, we must add calculation of phi(p).
Complexity : O(Ans * log (phi(n)) * log n +
sqrt(p)) (if exists)
             O(p * log (phi(n)) * log n + sqrt(p))
(if does not exist)
Returns -1 if not found.*/
int primitive_root(int p) {
   if (p == 2) return 1;
   vector<int> factor;
   int phi = p-1,  n = phi;
   for (int i=2; i*i<=n; ++i)
       if (n%i == 0) {
           factor.push_back (i);
           while (n%i==0)  n/=i;
       }
   if (n>1)  factor.push_back(n);
   for (int res=2; res<=p; ++res) {
       bool ok = true;
       for (int i=0; i<factor.size() && ok; ++i)
           ok &= power(res, phi/factor[i], p)!=1;
       if (ok)  return res;
   }
   return -1;
}
/**
```

```
 Generates necessary info for NTT (for offline
usage :3).
 Returns maximum k such that 2^k % mod = 1,
 NTT can only be applied for arrays not larger
than this size.
 mod MUST BE PRIME!!!!!
 We use the fact that if primes have the form
p=c*2^k+1,
 there always exists the 2^k-th root of unity.
 It can be shown that g^c is such a 2^k-th root
 of unity, where g is a primitive root of p.
*/
int nttdata(int mod,int &root, int &inv, int &pw){
   int c = 0, n = mod-1;
   while (n%2 == 0) c++, n/=2;
   pw = (mod-1)/n;
   int g = primitive_root(mod);
   if(g == -1) return -1; // No primitive root
exists
   root = power(g, n, mod);
   inv = power(root, mod-2, mod);
   return c;
}
```

**Extra Notes**

1. if n=$a^p.b^q.c^r$, S.O.D = $\frac{a^{p+1}-1}{a-1} * \frac{b^{q+1}-1}{b-1} * \frac{c^{r+1}-1}{c-1}$
2. সমান্তর ধারা: nতম পদ=a+(n-1)d, sum=$\frac{n\{2a+(n-1)d\}}{2}$
3. গুণোত্তর ধারা: nতম পদ=$ar^{n-1}$, sum=$\frac{a(r^n-1)}{r-1}$
4. $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$
5. $\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$
6. Catalan Numbers: 1, 1, 2, 5, 14, 42, 132…..
   $C_n = \frac{(2n)!}{(n+1)!n!}$ ; n >= 0
7. $(a+b)^p = \sum_{k=0}^{p} (p\ k) \times a^p \times b^{p-k}$
8. Suppose, there are n unlabelled objects to be placed into k bins, ways= $(n-1\ k-1)$
9. Statement of 5no. and empty bins are valid, ways= $(n+k-1\ k-1)$
10. Sine Rule of a Triangle: $\frac{a}{sinA} = \frac{b}{sinB} = \frac{c}{sinC}$
11. Cosine Rule of a Triangle: $cosA = \frac{b^2+c^2-a^2}{2bc}$
12. Surface Area & Volumes:
    Sphere: SA = $4\pi r^2$, V= $\frac{4}{3}\pi r^3$
    Cone: SA= $\pi r^2 + \pi rs$, V= $\frac{1}{3}\pi r^2 h$, [$side,\ s = \sqrt{h^2+r^2}$]
    Cylinder: SA= $2\pi r^2 + 2\pi rh$, V= $\pi r^2 h$
    Cuboid: SA= $2(wh + lw + lh)$, V= $lwh$
    Trapezoid: Area= $\frac{1}{2}(b1 + b2)h$
13. Area of a Circle Sector= $\frac{\theta}{360}\pi r^2\ (in\ degree)$
14. Number of permutations of n elements with k disjoint cycles
= Str1(n,k) = (n-1) * Str1(n-1,k) + Str1(n-1,k-1)
15. n! = Sum(Str1(n,k)) (for all 0 <= k <= n).
16. Ways to partition n labelled objects into k unlabeled subsets = Str2(n,k) = k * Str2(n-1,k) + Str2(n-1,k-1)
17. Parity of Str2(n,k) : ((n-k) & Floor((k-1)/2))==0)
18. Ways to partition n labelled objects into k unlabelled subsets, with each subset containing at least r elements : SR(n,k) = k * SR(n-1,k) + C(n-1,r-1) * SR(n-r,k-1)
19. Number of ways to partition n labelled objects 1,2,3, ... n into k non-empty subsets so that for any integers i and j in a given subset |i-j| >= d : Str2(n-d+1, k-d+1), n >= k >= d
20. Total number of paths from point P(x1, y1) to point Q(x2, y2) where x2 >= x1 and y2 >= y1:
Let x = x2 - x1 and y = y2 - y1. Then ans = C(x+y, x).
21. Total number of paths from point P(x1, y1) to point Q(x2, y2), where x2 >= x1 and y2 >= y1 without crossing the line X = Y + c:
Let x = x2 - x1 and y = y2 - y1. Then ans = C(x+y, x) - C(x+y, x+c-1).
Special Case : x = n, y = n, c = 0, then ans = C(2n, n) - C(2n, n-1) [Catalan Number]
22. Catalan triangle : Total number of permutation having n X and k Y so that Count(X)-Count(Y)>=0 in any prefix (Non-negative Partial Sum):
ans = C(n+k,k) - C(n+k, k-1)
23. Catalan trapezoid: Total number of permutation having n X and k Y so that Count(Y) - Count(X) < m in any prefix, then :
when 0 <= k < m, ans = C(n+k,k)
when m <= k <= n+m-1, ans = C(n+k,k) - C(n+k,k-m)
when k > n+m-1, ans = 0
24. Eulerian number of the first kind :
A1(n,k) is the number of permutations of 1 to n in which exactly k elements are greater than their previous element.
Then : A1(n,k) = (n-k) * A1(n-1,k-1) + (k+1) * A1(n-1,k).
25. Eulerian number of the second kind :
Number of permutations of the multiset {1,1,2,2,..,n,n} such that for each k, all the numbers appearing between the two occurrences of k are greater than k = (2n - 1)!
A2(n,m) is the number of such permutations with m ascents.
Then : A2(n,m) = (2n-m-1) * A2(n-1, m-1) + (m+1) * A2(n-1,m)
[ex: 332211: 0 ascent, 233211: 1 ascent, 112233: 2 ascents]
26. In 2-SAT: $A,\ \overline{A}$ mustn't be in the same SCC.
$(B) = TRUE\ is\ eqv\ to\ (\overline{A} \Rightarrow B)\&(\overline{B} \Rightarrow A)$.
27. GCC Optimization:

```
#pragma GCC optimize("Ofast,unroll-loops")
#pragma GCC target("avx,avx2,fma")
```

28. checker.sh: run "bash checker.sh"

```
for((i = 1; ; ++i)); do
    echo $i
    ./gen $i > int
    diff -w <(./a < int) <(./brute < int) || break
done
```

29. Pick's Theorem: A = I + (B/2) -1
A = Area of Polygon, B = Number of integral points on edges of polygon, I = Number of integral points strictly inside the polygon.
30. Python Fast I/O:

```
import io, os, sys
input=io.BytesIO(os.read(0,os.fstat(0).st_size)).readline
a,b = map(int, input().decode())
sys.stdout.write(str(n)+"\n")
sys.stdout.write(" ".join(map(str, li))+"\n")
```