```
/* Geometry: Complex Arithmetic ------------------------------------------*
// These two values are used in most of the geometry algorithms.
double PI = 2*acos(0.0);
double EPS = 1E-8;
struct pol { double r, t;
pol(double R = 0, double T = 0) : r(R), t(T) {} };
struct point { double x, y;
point(double X = 0, double Y = 0) : x(X), y(Y) {}
point(const pol &P) : x(P.r*cos(P.t)), y(P.r*sin(P.t)) {}
point conj() const { return point(x, -y); }
double mag2() const { return x*x + y*y; }
double mag() const { return sqrt(mag2()); }
double arg() const { return atan2(y, x); }
point operator-() const { return point(-x, -y); }
point& operator+=(const point &a) { x += a.x; y += a.y; return *this; }
point& operator-=(const point &s) { x -= s.x; y -= s.y; return *this; }
point& operator*=(const point &m) {
double tx = x*m.x - y*m.y, ty = x*m.y + y*m.x;
x = tx; y = ty; return *this; }
point& operator/=(const point &d) {
double tx = y*d.y + x*d.x, ty = y*d.x - x*d.y, t = d.mag2();
x = tx/t; y = ty/t; return *this; }
bool operator<(const point &q) const {
if (fabs(y-q.y) < EPS) return x < q.x;
return y < q.y; }
bool operator==(const point &q) const {
return (fabs(x-q.x) < EPS) && (fabs(y-q.y) < EPS); }
bool operator!=(const point &q) const { return !operator==(q); } };
point operator+(point a, const point &b) { return a += b; }
point operator-(point a, const point &b) { return a -= b; }
point operator*(point a, const point &b) { return a *= b; }
point operator/(point a, const point &b) { return a /= b; }

/* Geometry: Area of a polygon (positive <-> CCW orientation) ----------*/
double areaPoly(vector<point> &p) {
double sum = 0; int n = p.size();
for (int i = n-1, j = 0; j < n; i = j++) sum += (p[i].conj()*p[j]).y;
return sum/2; }

/* Geometry: Heron's formula for triangle area ----------------------------*/
// Given side lengths a, b, c, returns area or -1 if triangle is impossible
double area_heron(double a, double b, double c) {
if (a < b) swap(a, b); if (a < c) swap(a, c); if (b < c) swap(b, c);
if ((c-(a-b)) < 0) return -1;
return sqrt((a+(b+c))*(c-(a-b))*(c+(a-b))*(a+(b-c)))/4.0; }

/* Geometry: Closest point on line segment a-b to point c ----------------*/
point closest_pt_lineseg(point a, point b, point c) {
b -= a; c -= a; if (b == 0) return a;
double d = (c/b).x;
if (d < 0) d = 0; if (d > 1) d = 1;
return a + d*b; }

/* Geometry: Rectangle in rectangle test ----------------------------------*/
// Checks if rectangle of sides x,y fits inside one of sides X,Y
// Code as written rejects rectangles that just touch.
bool rect_in_rect(double X, double Y, double x, double y) {
if (Y > X) swap(Y, X); if (y > x) swap(y, x);
double diagonal = sqrt(X*X + Y*Y);
if (x < X && y < Y) return true;
else if (y >= Y || x >= diagonal) return false;
else { double w, theta, tMin = PI/4, tMax = PI/2;
while (tMax - tMin > EPS) { theta = (tMax + tMin)/2.0;
w = (Y-x*cos(theta))/sin(theta);
if (w < 0 || x * sin(theta) + w * cos(theta) < X) tMin = theta;
else tMax = theta; }
return (w > y); } }

/* Geometry: Centroid of a simple polygon [O(N)] ---------------------------*/
// Points must be oriented (either CW or CCW), and non-convex is OK
point centroid(point p[], int n) {
double sum = 0; point c;
for(int i = n-1, j = 0; j < n; i = j++) { double area = (p[i].conj()*p[j]).y;
sum += area; c += (p[i]+p[j])*area; } sum *= 3.0; c /= sum; return c; }

/* Geometry: Convex Hull ------------------------------------------------*/
struct polar_cmp {
point P0;
```

```
polar_cmp(point p = 0) : P0(p) {}
double turn(const point &p1, const point &p2) const {
return ((p2-P0)*(p1-P0).conj()).y; }
bool operator()(const point &p1, const point &p2) const {
double d = turn(p1, p2);
if (fabs(d) < EPS) return (p1-P0).mag2() < (p2-P0).mag2();
else return d > 0; } };
vector<point> convex_hull(vector<point> p) {
sort(p.begin(), p.end());
int n = unique(p.begin(), p.end()) - p.begin();
sort(p.begin()+1, p.begin()+n, polar_cmp(p[0]));
if (n <= 2) return vector<point>(p.begin(), p.begin()+n);
vector<point> hull(p.begin(), p.begin()+2); int h = 2;
for (int i = 2; i < n; ++i) {
while ((h > 1) && (polar_cmp(hull[h-2]).turn(hull[h-1], p[i]) < EPS)) {
hull.pop_back(); --h; }
hull.push_back(p[i]); ++h; }
return hull; }

/* Geometry: Area of intersection of two circles ----------------------------*/
struct circle { point c; double r; };
double CIArea(circle &a, circle &b) {
double d = (b.c-a.c).mag();
if (d <= (b.r - a.r)) return a.r*a.r*PI;
if (d <= (a.r - b.r)) return b.r*b.r*PI;
if (d >= a.r + b.r) return 0;
double alpha = acos((a.r*a.r+d*d-b.r*b.r)/(2*a.r*d));
double beta = acos((b.r*b.r+d*d-a.r*a.r)/(2*b.r*d));
return a.r*a.r*(alpha-0.5*sin(2*alpha))+b.r*b.r*(beta-0.5*sin(2*beta)); }

/* Geometry: Points of intersection of two circles -------------------------*/
// For identical circles, returns true with "indefinite" coordinates in p,q
// p, q will compare equal if there is only one intersection point
bool circintersect(circle &a, circle &b, point &p, point &q) {
double d2 = (b.c-a.c).mag2(), rS = a.r+b.r, rD = a.r-b.r;
if (d2 > rS*rS) return false;
if (d2 < rD*rD) return false;
double ca = 0.5*(1 + rS*rD/d2);
point z = point(ca, sqrt((a.r*a.r/d2)-ca*ca));
p = a.c + (b.c-a.c)*z; q = a.c + (b.c-a.c)*z.conj();
return true; }

/* Geometry: Line-circle intersection points --------------------------------*/
// Intersects (infinite) line through a,b with circle c, returns pts. p, q
// If a and b are the same, returns true with "indefinite" coordinates in p,q
// p, q will compare equal if there is only one intersection point
bool lineCircintersect(point a, point b, circle c, point &p, point &q) {
c.c -= a; b -= a; point m = b*(c.c/b).x;
double d2 = (m-c.c).mag2();
if (d2 > c.r*c.r) return false;
double L = sqrt((c.r*c.r-d2)/b.mag2());
p = a + m + L*b; q = a + m - L*b; return true; }

/* Geometry: Area of union of rectangles [O(N^2)] --------------------------*/
// Rectangle sides are parallel to the x & y axes
// May be desirable to add a constructor to 'rect' to ensure that the
// coordinates are properly sorted
struct rect { double minx, miny, maxx, maxy; };
struct edge { double x, miny, maxy;
char m;
bool operator<(const edge &e) const { return x < e.x; } };
double area_unionrect(vector<rect> R){ int n = R.size();
vector<double> ys(2*n);
vector<edge> e(2*n);
for (int i = 0; i < n; ++i) {
e[2*i].miny = e[2*i+1].miny = ys[2*i] = r[i].miny;
e[2*i].maxy = e[2*i+1].maxy = ys[2*i+1] = r[i].maxy;
e[2*i].x = r[i].minx; e[2*i].m = 1;
e[2*i+1].x = r[i].maxx; e[2*i+1].m = -1; }
sort(ys.begin(), ys.end());
sort(e.begin(), e.end());
double sum = 0, cur = 0;
for (int i = 0; i < 2*n; ++i) { if (i) sum += (ys[i]-ys[i-1])*cur;
int flag = 0; double sx = cur = 0;
for (int j = 0; j < 2*n; ++j) {
if (e[j].miny <= ys[i] && ys[i] < e[j].maxy) { if (!flag) sx = e[j].x;
flag += e[j].m;
if (!flag) curr += e[j].x-sx; } } } return sum; }
```

```
/* Geometry: Line segment a-b vs. c-d intersection (IP returned in p) -------*/
// returns 1 if intersect, 0 if not, -1 if coincident
int intersect_line(point a, point b, point c, point d, point &p) {
double num1 = ((a-c)*(d-c).conj()).y, num2 = ((a-c)*(b-a).conj()).y;
double denom = ((d-c)*(b-a).conj()).y;
if (fabs(denom) > EPS) {
double r = num1/denom, s = num2/denom;
if ((0 <= r) && (r <= 1) && (0 <= s) && (s <= 1)) {
p = a+r*(b-a); return 1; } return 0; }
if (fabs(num1) > EPS) return 0;
if (b < a) swap(a, b); if (d < c) swap(c, d);
if (a.y == b.y) { if (b.x == c.x) { p = b; return 1; }
else if (a.x == d.x) { p = a; return 1; }
else if ((b.x < c.x) || (d.x < a.x)) return 0; }
else { if (b.y == c.y) { p = b; return 1; }
else if (a.y == d.y) { p = a; return 1; }
else if ((b.y < c.y) || (d.y < a.y)) return 0; }
return -1; }

/* Geometry: Area of intersection of two general polygons [O(N^2)] ----------*/
int ORDER = -1; // CCW ordering, 1 for CW
struct triangle { point p[3]; };
double cross(point a, point b, point c, point d) {
d -= c; b -= a; return (d*b.conj()).y; }
int leftRight(const point &a, const point &b, const point &p) {
// -1: p left of a->b, +1: p right of a->b, 0: p on a->b
double d = cross(a, b, a, p);
if (d > EPS) return -1; if (d < -EPS) return 1; return 0; }
bool isConcave(point &a, point &b, point &c) {
// tests if b in a->b->c is concave/flat
return ORDER*leftRight(a, b, c) <= 0; }
bool isInsideTriangle(point &a, point &b, point &c, point &p) {
int r1 = leftRight(a,b,p), r2 = leftRight(b,c,p), r3 = leftRight(c,a,p);
return (ORDER*r1 >= 0) && (ORDER*r2 >= 0) && (ORDER*r3 >= 0); }
vector<triangle> triangulate(vector<point> &orig) {
// Accepts a vector of n ordered vertices, returns triangulation.
// No triangles if n < 3.
vector<triangle> T;
if (orig.size() < 3) return T;
list<point> P(orig.begin(), orig.end());
list<point>::iterator a, b, c, q;
for (a = b = P.begin(), c = ++b, ++c; c != P.end(); a = b, c = ++b, ++c)
if (!isConcave(*a, *b, *c)) {
q = P.begin(); if (q == a) { ++q; ++q; ++q; }
while ((q != P.end()) && !isInsideTriangle(*a, *b, *c, *q)) {
++q; if (q == a) { ++q; ++q; ++q; } }
if (q == P.end()) {
triangle t; t.p[0] = *a; t.p[1] = *b; t.p[2] = *c; T.push_back(t);
P.erase(b); b = a; if (b != P.begin()) --b; } } return T; }
bool isectLineSegs(point &a, point &b, point &c, point &d, point &p) {
// Finds intersection p of segments a-b and c-d (returns 0 if none/inf)
double n1 = cross(c, d, c, a), n2 = -cross(a, b, a, c);
double dn = cross(a, b, c, d);
if (fabs(dn) > EPS) { double r = n1/dn, s = n2/dn;
if ((0 <= r) && (r <= 1) && (0 <= s) && (s <= 1)) {
p = a+r*(b-a); return true; } } return false; }
struct radialLessThan { point P0;
radialLessThan(point p = 0) : P0(p) {}
bool operator()(const point &a, const point &b) const {
return (ORDER == leftRight(P0, a, b)); } };
double isectAreaTriangles(triangle &a, triangle &b) {
vector<point> P;
point p; triangle T[2] = {a, b};
for (int r = 1, t = 0; t < 2; r = t++)
for (int i = 2, j = 0; j < 3; i = j++) {
if (isInsideTriangle(T[r].p[0],T[r].p[1],T[r].p[2],T[t].p[i]))
P.push_back(T[t].p[i]);
for (int u = 2, v = 0; v < 3; u = v++)
if (isectLineSegs(T[t].p[i],T[t].p[j],T[r].p[u],T[r].p[v],p)) P.push_back(p); }
if (P.empty()) return 0;
sort(P.begin(), P.end());
vector<point> U; unique_copy(P.begin(), P.end(), back_inserter(U));
if (U.size() >= 3) { sort(++U.begin(), U.end(), radialLessThan(U[0]));
return areaPoly(U); } return 0; }
double isectAreaGpoly(vector<point> &P, vector<point> &Q) {
vector<triangle> S = triangulate(P), T = triangulate(Q); double area = 0;
for (vector<triangle>::iterator s = S.begin(); s != S.end(); ++s)
```

```
for (vector<triangle>::iterator t = T.begin(); t != T.end(); ++t)
area += isectAreaTriangles(*s, *t); return -ORDER*area; }

/* Geometry: Point in polygon ----------------------------------------------*/
bool pt_in_poly(vector<point> &p, const point &a) {
int n = p.size(); bool inside = false;
for (int i = 0, j = n-1; i < n; j = i++) {
if ((a-p[i]).mag()+(a-p[j]).mag()-(p[i]-p[j]).mag() < EPS)
return true; // Boundary case (pt on edge), you may want false here
if (((p[i].y<=a.y) && (a.y<p[j].y)) || ((p[j].y<=a.y) && (a.y<p[i].y)))
if (a.x-p[i].x < (p[j].x-p[i].x)*(a.y-p[i].y) / (p[j].y-p[i].y))
inside = !inside; } return inside; }

/* Geometry: Polygon midpoints -> vertices (n odd) -------------------------*/
vector<point> midpts2vert(vector<point> &midpts) {
int n = midpts.size(); vector<point> poly(n);
poly[0] = midpts[0];
for (int i = 1; i < n-1; i += 2) {
poly[0].x += midpts[i+1].x - midpts[i].x;
poly[0].y += midpts[i+1].y - midpts[i].y; }
for (int i = 1; i < n; i++) {
poly[i].x = 2.0*midpts[i-1].x - poly[i-1].x;
poly[i].y = 2.0*midpts[i-1].y - poly[i-1].y; } return poly; }

/* Geometry: 3D Primitives -------------------------------------------------*/
struct point3 { double x, y, z;
point3(double X=0, double Y=0, double Z=0) : x(X), y(Y), z(Z) {}
point3 operator+(point3 p) { return point3(x + p.x, y + p.y, z + p.z); }
point3 operator*(double k) { return point3(k*x, k*y, k*z); }
point3 operator-(point3 p) { return *this + (p*-1.0); }
point3 operator/(double k) { return *this*(1.0/k); }
double mag2() { return x*x + y*y + z*z; }
double mag() { return sqrt(mag2()); }
point3 norm() { return *this/this->mag(); } };
double dot(point3 a, point3 b) {
return a.x*b.x + a.y*b.y + a.z*b.z; }
point3 cross(point3 a, point3 b) {
return point3(a.y*b.z - b.y*a.z, b.x*a.z - a.x*b.z, a.x*b.y - b.x*a.y); }
struct line { point3 a, b;
line(point3 A=point3(), point3 B=point3()) : a(A), b(B) {}
point3 dir() { return (b - a).norm(); } };
point3 cpoint_iline(line u, point3 p) {
// Closest point on an infinite line u to a given point p
point3 ud = u.dir();
return u.a - ud*dot(u.a - p, ud); }
double dist_ilines(line u, line v) {
// Shortest distance between two infinite lines u and v
return dot(v.a - u.a, cross(u.dir(), v.dir()).norm()); }
point3 cpoint_ilines(line u, line v) {
// Finds the closest point on infinite line u to infinite line v.
// Assumes non-parallel lines
point3 ud = u.dir(); point3 vd = v.dir();
double uu = dot(ud, ud), vv = dot(vd, vd), uv = dot(ud, vd);
double t = dot(u.a, ud) - dot(v.a, ud); t *= vv;
t -= uv*(dot(u.a, vd) - dot(v.a, vd));
t /= (uv*uv - uu*vv); return u.a + ud*t; }
point3 cpoint_lineseg(line u, point3 p) {
// Closest point on a line segment u to a given point p
point3 ud = u.b - u.a; double s = dot(u.a - p, ud)/ud.mag2();
if (s < -1.0) return u.b; if (s > 0.0) return u.a; return u.a - ud*s; }
struct plane { point3 n, p;
plane(point3 ni = point3(), point3 pi = point3()) : n(ni), p(pi) {}
plane(point3 a, point3 b, point3 c) : n(cross(b-a, c-a).norm()), p(a) {}
double d() { return -dot(n, p); } };
point3 cpoint_plane(plane u, point3 p) {
// Closest point on a plane u to a given point p
return p - u.n*(dot(u.n, p) + u.d()); }
point3 iline_isect_plane(plane u, line v) {
// Point of intersection between an infinite line v and a plane u.
// Assumes line not parallel to plane.
point3 vd = v.dir();
return v.a - vd*((dot(u.n, v.a) + u.d())/dot(u.n, vd)); }
line isect_planes(plane u, plane v) {
// Infinite line of intersection between two planes u and v.
// Assumes planes not parallel.
point3 o = u.n*-u.d(), uv = cross(u.n, v.n);
point3 uvu = cross(uv, u.n);
```

```
point3 a = o - uvu*((dot(v.n, o) + v.d())/(dot(v.n, uvu)*uvu.mag2()));
return line(a, a + uv); }

/* Geometry: Great Circle distance (lat[-90,90], long[-180,180]) ----------*/
double greatcircle(double lt1, double lo1, double lt2, double lo2, double r) {
double a = PI*(lt1/180.0), b = PI*(lt2/180.0);
double c = PI*((lo2-lo1)/180.0);
return r*acos(sin(a)*sin(b) + cos(a)*cos(b)*cos(c)); }

/* Geometry: Circle described by three points ------------------------------*/
bool circle(point p1, point p2, point p3, point &center, double &r) {
double G = 2*((p2-p1).conj()*(p3-p2)).y;
if (fabs(G) < EPS) return false;
center = p1*(p3.mag2()-p2.mag2());
center += p2*(p1.mag2()-p3.mag2());
center += p3*(p2.mag2()-p1.mag2());
center /= point(0, G); r = (p1-center).mag();
return true; }

/* Arithmetic: Discrete Logarithm solver [O(sqrt(P))] ------------------------*/
// Given prime P, B, and N, finds least L such that B^L == N (mod P)
typedef unsigned int UI;
typedef unsigned long long ULL;
map<UI,UI> M;
UI times(UI a, UI b, UI m) {
return (ULL) a * b % m; }
UI power(UI val, UI power, UI m) { UI res = 1;
for (UI p = power; p; p >>= 1) {
if (p & 1) res = times(res, val, m);
val = times(val, val, m); }
return res; }
UI discrete_log(UI p, UI b, UI n) {
UI jump = sqrt(double(p)); M.clear();
for (UI i = 0; i < jump && i < p-1; ++i)
M[power(b,i,p)] = i+1;
for (UI i = 0, j; i < p-1; i += jump)
if (j = M[times(n,power(b,p-1-i,p),p)])
return (i+j-1)%(p-1);
return -1; }

/* Arithmetic: Cubic equation solver ----------------------------------------*/
struct Result { int n; // Number of solutions
double x[3]; // Solutions };
Result solve_cubic(double a, double b, double c, double d) {
long double a1 = b/a, a2 = c/a, a3 = d/a;
long double q = (a1*a1 - 3*a2)/9.0, sq = -2*sqrt(q);
long double r = (2*a1*a1*a1 - 9*a1*a2 + 27*a3)/54.0;
double z = r*r-q*q*q, theta;
Result s; if(z <= 0) {
s.n = 3; theta = acos(r/sqrt(q*q*q));
s.x[0] = sq*cos(theta/3.0) - a1/3.0;
s.x[1] = sq*cos((theta+2.0*PI)/3.0) - a1/3.0;
s.x[2] = sq*cos((theta+4.0*PI)/3.0) - a1/3.0; }
else { s.n = 1; s.x[0] = pow(sqrt(z)+fabs(r),1/3.0);
s.x[0] += q/s.x[0]; s.x[0] *= (r < 0) ? 1 : -1;
s.x[0] -= a1/3.0; } return s; }

/* Combinatorics: Digit Occurrence count ----------------------------------*/
// Given digit d and value N, returns # of times d occurs from 1..N
long long digit_count(int digit, int max) {
long long res = 0; char buff[15]; int i, count;
if(max <= 0) return 0;
res += max/10 + ((max % 10) >= digit ? 1 : 0);
if(digit == 0) res--;
res += digit_count(digit, max/10 - 1) * 10;
sprintf(buff, "%d", max/10);
for(i = 0, count = 0; i < strlen(buff); i++)
if(buff[i] == digit+'0') count++;
res += (1 + max%10) * count; return res; }

/* Combinatorics: Permutation index on distinct characters ----------------*/
// Returns perm. index of a string according to lex. ordering.
// Warning: does not work with repeated chars.
int permdex (char *s) { int size = strlen(s), index = 0;
for (int i = 1; i < size; ++i) { for (int j = i; j < size; ++j)
if (s[i-1] > s[j]) ++index; index *= size - i; } return index; }
```

```
/* Dynamic Programming: Longest Ascending Subsequence ----------*/
int asc_seq(int *A, int n, int *S) {
int *m, *seq, i, k, low, up, mid, start;
m = malloc((n+1) * sizeof(int));
seq = malloc(n * sizeof(int));
for (i = 0; i < n; i++) seq[i] = -1;
m[1] = start = 0;
for (k = i = 1; i < n; i++) { if (A[i] >= A[m[k]]) {
seq[i] = m[k++]; start = m[k] = i; }
else if (A[i] < A[m[1]]) m[1] = i;
else { low = 1; up = k;
while (low != up-1) {
mid = (low+up)/2;
if (A[m[mid]] <= A[i]) low = mid;
else up = mid; }
seq[i] = m[low]; m[up] = i; } }
for (i = k-1; i >= 0; i--) {
S[i] = A[start]; start = seq[start]; }
free(m); free(seq); return k; }

/* Dynamic Programming: Longest Strictly Ascending Subsequence ----------*/
int sasc_seq(int *A, int n, int *S) {
int *m, *seq, i, k, low, up, mid, start;
m = malloc((n+1) * sizeof(int));
seq = malloc(n * sizeof(int));
for (i = 0; i < n; i++) seq[i] = -1;
m[1] = start = 0;
for (k = i = 1; i < n; i++) {
if (A[i] > A[m[k]]) { seq[i] = m[k++]; start = m[k] = i; }
else if (A[i] < A[m[1]]) m[1] = i;
else if (A[i] < A[m[k]]) {
low = 1; up = k;
while (low != up-1) { mid = (low+up)/2;
if(A[m[mid]] <= A[i]) low = mid;
else up = mid; }
if (A[i] > A[m[low]]) { seq[i] = m[low]; m[up] = i; } } }
for (i = k-1; i >= 0; i--) { S[i] = A[start]; start = seq[start]; }
free(m); free(seq); return k; }

/* Generators: Catalan Numbers ----------*/
long long int cat[33];
void getcat() { cat[0] = cat[1] = 1;
for (int i = 2; i < 33; ++i) cat[i] = cat[i-1]*(4*i-6)/i; }

/* Generators: Binary Strings generator (cardinal order) ----------*/
char bit[MAXN];
void recurse(int n, int curr, int left) {
if(curr == n) Process(n);
else { if(curr+left < n) {
bit[curr] = 0; recurse(n, curr+1, left); }
if(left) { bit[curr] = 1; recurse(n, curr+1, left-1); } } }
void gen_bin_card(int n) {
for(int i = 0; i <= n; i++) {
printf("Cardinality %d:\n", i); recurse(n, 0, i); } }

/* Graph Theory: Maximum Bipartite Matching ----------*/
// How to use (sample at bottom):
// For vertex i of set U:
// match[i] = -1 means i is not matched
// match[i] = x means the edge i->(x-|U|) is selected
// For simplicity, use addEdge(i,j,n) to add edges, where
// 0 <= i < |U| and 0 <= j < |V| and |U| = n.
// If there is an edge from vertex i of U to vertex
// j of V then: e[i][j+|U|] = e[j+|U|][i] = 1.
// - If |U| = n and |V| = m, then vertices are assumed
// to be from [0,n-1] in set U and [0,m-1] in set V.
// - Remember that match[i]-n gives the edge from i, not just match[i].
const int MAXN 300 // How many vertices in U+V (in total)
char e[MAXN][MAXN]; // MODIFIED Adj. matrix (see note)
int match[MAXN], back[MAXN], q[MAXN], tail;
void addEdge(int x, int y, int n) {
e[x][y+n] = e[y+n][x] = 1; }
int find(int x, int n, int m) { int i, j, r;
if(match[x] != -1) return 0;
memset(back, -1, sizeof(back));
for(q[i=0]=x, tail = 1; i < tail; i++)
for(j = 0; j < n+m; j++) {
if(!e[q[i]][j]) continue;
```

```
if(match[j] != -1) { if(back[j] == -1) { back[j] = q[i];
back[q[tail++] = match[j]] = j; } }
else { match[match[q[i]] = j] = q[i];
for(r = back[q[i]]; r != -1; r = back[back[r]])
match[match[r] = back[r]] = r;
return 1; } } return 0; }
void bipmatch(int n, int m) {
memset(match, -1, sizeof(match));
for(int i = 0; i < n+m; i++) if(find(i,n,m)) i = 0; }

/* Graph Theory: Eulerian Graphs ----------------------------------------*/
// Before adding edges, call Init() to initialize all data structures.
// Use the provided addEdge(x,y,c) which adds c edges between x and y.
// isEulerian(int n, int *start, int *end) returns:
// 0 if the graph is not Eulerian
// 1 if the graph has a Euler cycle
// 2 if the graph a path, from start to end
// with n being the number of nodes in the graph
const int MAXN 105 // Number of nodes
const int MAXM 505 // Maximum number of edges
#define min(a,b) (((a)<(b))?(a):(b))
#define max(a,b) (((a)>(b))?(a):(b))
#define DEC(a,b) g[a][b]--;g[b][a]--;deg[a]--;deg[b]--
int sets[MAXN], deg[MAXN], g[MAXN][MAXN];
int seq[MAXM], seqsize;
int getRoot(int x) { if (sets[x] < 0) return x;
return sets[x] = getRoot(sets[x]); }
void Union(int a, int b) { int ra = getRoot(a), rb = getRoot(b);
if (ra != rb) { sets[ra] += sets[rb];
sets[rb] = ra; } }
void Init() { memset(sets, -1, sizeof(sets));
memset(g, 0, sizeof(g));
memset(deg, 0, sizeof(deg)); }
void addEdge(int x, int y, int count) {
g[x][y] += count; deg[x] += count;
g[y][x] += count; deg[y] += count;
Union(x,y); }
int isEulerian(int n, int *start, int *end) {
int odd = 0, i, count = 0, x;
for (i = 0; i < n; i++) if (deg[i]) { x = i; count++; }
if (sets[getRoot(x)] != -count) return 0;
for (i = 0; i < n; i++) { if (deg[i] & 1) { odd++;
if(odd == 1) *start = i; else if(odd == 2) *end = i;
else return 0; } } return odd ? 2 : 1; }
void getPath(int n, int start, int end) {
int temp[MAXM], tsize = 1, i, j;
temp[0] = start;
while(1) { j = temp[tsize-1];
for (i = 0; i < n; i++) { if (i == end) continue;
if (g[i][j]) { temp[tsize++] = i;
DEC(i,j); break; } }
if (i == n) { if (g[end][j]) {
temp[tsize++] = end;
DEC(j,end); }
break; } }
for (i = 0; i < tsize; i++) if (!deg[temp[i]])
seq[seqsize++] = temp[i];
else getPath(n, temp[i], temp[i]); }
void buildPath(int n, int start, int end) {
seqsize = 0; getPath(n, start, end); }

/* Graph Theory: Maximum Flow in a directed graph ---------------------------*/
// Multiple edges from u to v may be added. They are converted into a
// single edge with a capacity equal to their sum
// - Vertices are assumed to be numbered from 0..n-1
// - The graph is supplied as the number of nodes (n), the zero-based
// indexes of the source (s) and the sink (t), and a vector of edges u->v
// with capacity c (M).
const int MAXN 200
struct Edge { //Edge u->v with capacity c
int u, v, c; };
int F[MAXN][MAXN]; //Flow of the graph
int maxFlow(int n, int s, int t, vector<Edge> &M) {
int u, v, c, oh, min, df, flow, H[n], E[n], T[n], C[n][n];
vector<Edge>::iterator m;
list<int> N; list<int>::iterator cur;
vector<int> R[n]; vector<int>::iterator r;
for (u = 0; u < n; u++) { E[u] = H[u] = T[u] = 0;
```

```
R[u].clear();
for (v = 0; v < n; v++) C[u][v] = F[u][v] = 0; }
for (m = M.begin(); m != M.end(); m++) {
u = m->u; v = m->v; c = m->c;
if (c && !C[u][v] && !C[v][u]) { R[u].push_back(v);
R[v].push_back(u); }
C[u][v] += c; } H[s] = n;
for (r = R[s].begin(); r != R[s].end(); r++) { v = *r;
F[s][v] = C[s][v]; F[v][s] = -C[s][v];
E[v] = C[s][v]; E[s] -= C[s][v]; }
N.clear();
for (u = 0; u < n; u++) if ((u != s) && (u != t))  N.push_back(u);
for (cur = N.begin(); cur != N.end(); cur++) {
u = *cur; oh = H[u];
while (E[u] > 0) if (T[u] >= (int)R[u].size()) { min = 10000000;
for (r = R[u].begin(); r != R[u].end(); r++) { v = *r;
if ((C[u][v] - F[u][v] > 0) && (H[v] < min)) min = H[v]; }
H[u] = 1 + min; T[u] = 0; }
else { v = R[u][T[u]];
if ((C[u][v] - F[u][v] > 0) && (H[u] == H[v]+1)) { df = C[u][v] - F[u][v];
if (df > E[u]) df = E[u];
F[u][v] += df; F[v][u] = -F[u][v];
E[u] -= df; E[v] += df; }
else T[u]++; }
if (H[u] > oh) N.splice(N.begin(), N, cur); }
flow = 0;
for (r = R[s].begin(); r != R[s].end(); r++)
flow += F[s][*r]; return flow; }

/* Graph Theory: Chinese Postman Problem ------------------------------------*/
// The maximum # of vertices solvable is roughly 20
#define MAXN 20
#define DISCONNECT -1
int g[MAXN][MAXN]; // Adj matrix (keep lowest cost if multiedge)
int deg[MAXN]; // Degree count
int A[MAXN+1]; // Used by perfect matching generator
int sum; // Sum of costs
int odd, best;
void floyd(int n) {
for(int k = 0; k < n; k++)
for(int i = 0; i < n; i++) if (g[i][k] != -1)
for(int j = 0; j < n; j++) if (g[k][j] != -1)
if ((g[i][j] == -1) || (g[i][j] > g[i][k]+g[k][j])) g[i][j] = g[i][k]+g[k][j];
for(int i = 0; i < n; i++)
g[i][i] = 0; }
void checkSum() {
int i, temp;
for(i = temp = 0; i < odd/2; i++)
temp += g[A[2*i]][A[2*i+1]];
if(best == -1 || best > temp) best = temp; }
void perfmatch(int x) { int i, t;
if(x == 2) checkSum();
else { perfmatch(x-2);
for(i = x-3; i >= 0; i--) { t = A[i]; A[i] = A[x-2];
A[x-2] = t; perfmatch(x-2); } t = A[x-2];
for(i = x-2; i >= 1; i--) A[i] = A[i-1]; A[0] = t; } }
int postman(int n) { int i; floyd(n);
for(odd = i = 0; i < n; i++) if(deg[i]%2) A[odd++] = i;
if(!odd) return sum; best = -1;
perfmatch(odd);
return sum+best; }
int main() { int i, u, v, c, n, m;
while(scanf("%d %d", &n, &m) == 2){
// Clear graph and degree count
memset(g, -1, sizeof(g));
memset(deg, 0, sizeof(deg));
for(sum = i = 0; i < m; i++) {
scanf("%d %d %d", &u, &v, &c);
u--; v--; deg[u]++; deg[v]++;
if(g[u][v] == -1 || g[u][v] > c) g[u][v] = c;
if(g[v][u] == -1 || g[v][u] > c) g[v][u] = c;
sum += c; } printf("Best cost: %d\n", postman(n)); } }

/* Graph Theory: Strongly Connected Components ------------------------------*/
vector<int> g[MAXN], curr;
vector< vector<int> > scc;
int dfsnum[MAXN], low[MAXN], id; char done[MAXN];
void visit(int x) {
```

```
curr.push_back(x); dfsnum[x] = low[x] = id++;
for(size_t i = 0; i < g[x].size(); i++)
if(dfsnum[g[x][i]] == -1){
visit(g[x][i]); low[x] <?= low[g[x][i]]; }
else if(!done[g[x][i]])
low[x] <?= dfsnum[g[x][i]];
if(low[x] == dfsnum[x]) {
Vl c; int y;
do { done[y = curr[curr.size()-1]] = 1;
c.push_back(y); curr.pop_back(); } while(y != x);
scc.push_back(c); } }
void strong_conn(int n) {
memset(dfsnum, -1, n*sizeof(int)); memset(done, 0, sizeof(done));
scc.clear(); curr.clear();
for(int i = id = 0; i < n; i++) if(dfsnum[i] == -1) visit(i); }

/* Graph Theory: Min Cost Max Flow (Edmonds-Karp & Dijkstra) ----------*/
// Takes a directed graph where each edge has a capacity ('cap') and a
// cost per unit of flow ('cost') and returns a maximum flow network
// of minimal cost ('fcost') from s to t. USE THIS CODE FOR (MODERATELY)
// DENSE GRAPHS; FOR VERY SPARSE GRAPHS, USE mcmf4 (next)
// PARAMETERS:
// - cap (global): adjacency matrix where cap[u][v] is the capacity
// of the edge u->v. cap[u][v] is 0 for non-existent edges.
// - cost (global): a matrix where cost[u][v] is the cost per unit
// of flow along the edge u->v. If cap[u][v] == 0, cost[u][v] is
// ignored. ALL COSTS MUST BE NON-NEGATIVE!
// - n: the number of vertices ([0, n-1] are considered as vertices).
// - s: source vertex.
// - t: sink.
// RETURNS:
// - the flow
// - the total cost through 'fcost'
// - fnet contains the flow network. Careful: both fnet[u][v] and
// fnet[v][u] could be positive. Take the difference.
// COMPLEXITY:
// - Worst case: O(n^2*flow <? n^3*fcost)
// Watch for commas when typing this in!
#define NN 1024 // the maximum number of vertices + 1
int cap[NN][NN]; // adjacency matrix (fill this up)
int cost[NN][NN]; // cost per unit of flow matrix (fill this up)
int fnet[NN][NN], adj[NN][NN], deg[NN]; // flow network and adjacency list
int par[NN], d[NN]; // par[source] = source;
int pi[NN]; // Labelling function
#define CLR(a, x) memset(a, x, sizeof(a))
#define Inf (INT_MAX/2)
#define Pot(u,v) (d[u] + pi[u] - pi[v])
bool dijkstra(int n, int s, int t) {
// Dijkstra's using non-negative edge weights (cost + potential)
for (int i = 0; i < n; i++) d[i] = Inf, par[i] = -1;
d[s] = 0; par[s] = -n - 1;
while (1) { int u = -1, bestD = Inf;
for (int i = 0; i < n; i++) if (par[i] < 0 && d[i] < bestD)
bestD = d[u = i];
if(bestD == Inf) break;
par[u] = -par[u] - 1;
for (int i = 0; i < deg[u]; i++) { int v = adj[u][i];
if (par[v] >= 0) continue;
if (fnet[v][u] && d[v] > Pot(u,v) - cost[v][u])
d[v] = Pot(u,v) - cost[v][u], par[v] = -u-1;
if (fnet[u][v] < cap[u][v] && d[v] > Pot(u,v) + cost[u][v])
d[v] = Pot(u,v) + cost[u][v], par[v] = -u - 1; } }
for (int i = 0; i < n; i++) if (pi[i] < Inf) pi[i] += d[i];
return par[t] >= 0; }
#undef Pot
int mcmf3(int n, int s, int t, int &fcost) {
CLR(deg, 0); CLR(fnet, 0); CLR(pi, 0);
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
if (cap[i][j] || cap[j][i]) adj[i][deg[i]++] = j;
int flow = fcost = 0;
while (dijkstra(n, s, t)) { int bot = INT_MAX;
for (int v = t, u = par[v]; v != s; u = par[v = u])
bot <?= fnet[v][u] ? fnet[v][u] : (cap[u][v] - fnet[u][v]);
for (int v = t, u = par[v]; v != s; u = par[v = u])
if (fnet[v][u]) { fnet[v][u] -= bot; fcost -= bot * cost[v][u]; }
else { fnet[u][v] += bot; fcost += bot * cost[u][v]; }
flow += bot; } return flow; }
```

```
int main() { int numV; cin >> numV;
memset(cap, 0, sizeof(cap));
int m, a, b, c, cp, s, t; cin >> m >> s >> t;
// fill up cap with existing capacities.
// if the edge u->v has capacity 6, set cap[u][v] = 6.
// for each cap[u][v] > 0, set cost[u][v] to the
// cost per unit of flow along the edge i->v
// Uncomment the commented statements if caps/costs are bidirectional
for (int i=0; i<m; i++) { cin >> a >> b >> cp >> c;
cost[a][b] = c; // cost[b][a] = c;
cap[a][b] = cp; // cap[b][a] = cp; }
int fcost, flow = mcmf3(numV, s, t, fcost);
cout << "flow: " << flow << endl;
cout << "cost: " << fcost << endl; }

/* Graph Theory: Min Cost Max Flow (Edmonds-Karp & fast heap Dijkstra) ---*/
// Same as above, but better for sparse graphs
#define NN 1024 // the maximum number of vertices + 1
int cap[NN][NN]; // adjacency matrix (fill this up)
int cost[NN][NN]; // cost per unit of flow matrix (fill this up)
int fnet[NN][NN], adj[NN][NN], deg[NN]; // flow network and adjacency list
int par[NN], d[NN], q[NN], inq[NN], qs; // Dijkstra's variables
int pi[NN]; // Labelling function
#define CLR(a, x) memset(a, x, sizeof(a))
#define Inf (INT_MAX/2)
#define BUBL { \
t = q[i]; q[i] = q[j]; q[j] = t; \
t = inq[q[i]]; inq[q[i]] = inq[q[j]]; inq[q[j]] = t; }
#define Pot(u,v) (d[u] + pi[u] - pi[v])
bool dijkstra(int n, int s, int t) {
// Dijkstra's using non-negative edge weights (cost + potential)
CLR(d, 0x3F); CLR(par, -1); CLR(inq, -1);
d[s] = qs = 0; inq[q[qs++] = s] = 0;
par[s] = n;
while (qs) { int u = q[0]; inq[u] = -1;
q[0] = q[--qs];
if (qs) inq[q[0]] = 0;
for (int i = 0, j = 2*i + 1, t; j < qs; i = j, j = 2*i + 1) {
if (j + 1 < qs && d[q[j + 1]] < d[q[j]]) j++;
if (d[q[j]] >= d[q[i]]) break; BUBL; }
for (int k = 0, v = adj[u][k]; k < deg[u]; v = adj[u][++k]) {
if (fnet[v][u] && d[v] > Pot(u,v) - cost[v][u])
d[v] = Pot(u,v) - cost[v][par[v] = u];
if (fnet[u][v] < cap[u][v] && d[v] > Pot(u,v) + cost[u][v])
d[v] = Pot(u,v) + cost[par[v] = u][v];
if (par[v] == u) { if (inq[v] < 0) { inq[q[qs] = v] = qs; qs++; }
for (int i=inq[v], j=(i-1)/2, t; d[q[i]]<d[q[j]]; i=j, j=(i-1)/2)
BUBL; } } }
for (int i = 0; i < n; i++) if (pi[i] < Inf) pi[i] += d[i];
return par[t] >= 0; }
#undef Pot
int mcmf4(int n, int s, int t, int &fcost) {
CLR(deg, 0); CLR(fnet, 0); CLR(pi, 0);
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
if (cap[i][j] || cap[j][i]) adj[i][deg[i]++] = j;
int flow = fcost = 0;
while (dijkstra(n,s,t)) {
int bot = INT_MAX;
for (int v = t, u = par[v]; v != s; u = par[v = u])
bot <?= fnet[v][u] ? fnet[v][u] : (cap[u][v] - fnet[u][v]);
for (int v = t, u = par[v]; v != s; u = par[v = u])
if (fnet[v][u]) { fnet[v][u] -= bot; fcost -= bot * cost[v][u]; }
else { fnet[u][v] += bot; fcost += bot * cost[u][v]; }
flow += bot; } return flow; }

/* Graph Theory: Articulation Points & Bridges (adj list) [O(V+E)] ----------*/
// array entry art[v] is true iff vertex v is an articulation point
// - array entries bridge[i][0] and bridge[i][1] are the endpoints of a bridge
// in the graph. If bridge (u,v) is represented in the array, (v,u) is not.
// - 'bridges' is the number of bridges in the graph
// - index vertices from 0 to n-1
#define MAX_N 200
#define min(a,b) (((a)<(b))?(a):(b))
struct Node { int deg;
int adj[MAX_N]; }; Node alist[MAX_N];
bool art[MAX_N], seen[MAX_N];
int df_num[MAX_N], low[MAX_N], father[MAX_N], cnt;
```

```
int bridge[MAX_N*MAX_N][2], bridges;
void add_edge(int v1, int v2) {
alist[v1].adj[alist[v1].deg++] = v2;
alist[v2].adj[alist[v2].deg++] = v1; }
void add_bridge(int v1, int v2) {
bridge[bridges][0] = v1; bridge[bridges][1] = v2; ++bridges; }
void clear() { for (int i = 0; i < MAX_N; ++i)
alist[i].deg = 0; }
void search(int v, bool root) {
int w, child = 0; seen[v] = true;
low[v] = df_num[v] = cnt++;
for (int i = 0; i < alist[v].deg; ++i) {
w = alist[v].adj[i];
if (df_num[w] == -1) { father[w] = v; ++child;
search(w, false);
if (low[w] > df_num[v]) add_bridge(v, w);
if (low[w] >= df_num[v] && !root) art[v] = true;
low[v] = min(low[v], low[w]); }
else if (w != father[v]) {
low[v] = min(low[v], df_num[w]); } }
if (root && child > 1) art[v] = true; }
void articulate(int n) { int child = 0;
for (int i = 0; i < n; ++i) { art[i] = false;
df_num[i] = father[i] = -1; }
cnt = bridges = 0;
memset(seen, false, sizeof(seen));
for (int i = 0; i < n; ++i) if (!seen[i])
search(i, true); }
int main() { int n, m, v1, v2, c = 0;
while (true) { scanf("%d %d", &n, &m);
if (!n && !m) break; clear();
for (int i = 0; i < m; ++i) { scanf("%d %d", &v1, &v2);
add_edge(v1 - 1, v2 - 1); } articulate(n);
printf("Articulation Points:");
for (int i = 0; i < n; ++i) if (art[i]) printf(" %d", i + 1); printf("\n");
printf("Bridges:");
for (int i = 0; i < bridges; ++i)
printf(" (%d,%d)", bridge[i][0] + 1, bridge[i][1] + 1); printf("\n\n"); } }
```

```
/* Graph Theory: Maximum Weighted Bipartite Matching [O(n^3)] ----------*/
// Given N workers and N jobs to complete, where each worker has a certain
// compatibility (weight) to each job, find an assignment (perfect matching)
// of workers to jobs which maximizes the compatibility (weight).
// - W is a 2 dimensional array where W[i][j] is the weight of worker i
// doing job j. Weights must be non-negative. If there is no weight
// assigned to a particular worker and job pair, set it to zero. If there
// is a different number of workers than jobs, create dummy workers or jobs
// accordingly with zero weight edges.
// - M is a 1 dimensional array populated by the algorithm where M[i] is the
// index of the job matched to worker i.
// - This algorithm can be used with non-negative floating point weights.
#define MAX_N 100 // Max number of workers/jobs
int W[MAX_N][MAX_N], U[MAX_N], V[MAX_N], Y[MAX_N]; // weight vars
int M[MAX_N], N[MAX_N], P[MAX_N], Q[MAX_N], R[MAX_N], S[MAX_N], T[MAX_N];
int Assign(int n) {
// Returns max weight, corresponding matching inside global M
int w, y; // weight vars
int i, j, m, p, q, s, t, v;
for (i = 0; i < n; i++) { M[i] = N[i] = -1; U[i] = V[i] = 0;
for (j = 0; j < n; j++) if (W[i][j] > U[i]) U[i] = W[i][j]; }
for (m = 0; m < n; m++) { for (p = i = 0; i < n; i++) { T[i] = 0; Y[i] = -1;
if (M[i] == -1) { S[i] = 1; P[p++] = i; }
else S[i] = 0; }
while (1) { for (q = s = 0; s < p; s++) { i = P[s];
for (j = 0; j < n; j++)
if (!T[j]) { y = U[i] + V[j] - W[i][j];
if (y == 0) { R[j] = i; if (N[j] == -1)
goto end_phase; // I hate goto's!
T[j] = 1; Q[q++] = j; }
else if ((Y[j] == -1) || (y < Y[j])) {
Y[j] = y; R[j] = i; } } }
if (q == 0) { y = -1;
for (j = 0; j < n; j++)
if (!T[j] && ((y == -1) || (Y[j] < y))) y = Y[j];
for (j = 0; j < n; j++) { if (T[j]) V[j] += y; if (S[j])
U[j] -= y; }
for (j = 0; j < n; j++) if (!T[j]) { Y[j] -= y;
if (Y[j] == 0) {
```

```
if (N[j] == -1)
goto end_phase; // again!
T[j] = 1; Q[q++] = j; } } }
for (p = t = 0; t < q; t++) {
i = N[Q[t]]; S[i] = 1; P[p++] = i; } }
end_phase:
i = R[j]; v = M[i];
M[i] = j; N[j] = i;
while (v != -1) { j = v; i = R[j];
v = M[i]; M[i] = j; N[j] = i; } }
for (i = w = 0; i < n; i++) w += W[i][M[i]]; return w; }
int main() { int w; // weight var
int n, i, j;
while ((scanf("%d", &n) == 1) && (n != 0)) {
for (i = 0; i < n; i++)
for (j = 0; j < n; j++) scanf("%d", &W[i][j]);
w = Assign(n);
printf("Optimum weight: %d\n", w);
printf("Matchings:\n");
for (i = 0; i < n; i++)
printf("%d matched to %d\n", i, M[i]); } }
```

/* Graph Theory: Minimum weight Steiner tree [O(|V|*3^|S|+|V|^3)] -------*/

```
// Given a weighted undirected graph G = (V, E) and a subset S of V,
// finds a minimum weight tree T whose vertices are a superset of S.
// NP-hard -- this is a pseudo-polynomial algorithm.
// Minimum stc[(1<<s)-1][v] (0 <= v < n) is weight of min. Steiner tree
// Minimum stc[i][v] (0 <= v < n) is weight of min. Steiner tree for
// the i'th subset of Steiner vertices
// S is the list of Steiner vertices, s = |S|
// d is the adjacency matrix (use infinities, not -1), and n = |V|
const int N = 32; const int K = 8;
int d[N][N], n, S[K], s, stc[1<<K][N];
void steiner() {
for (int k = 0; k < n; ++k)
for (int i = 0; i < n; ++i)
for (int j = 0; j < n; ++j) d[i][j] <?= d[i][k] + d[k][j];
for(int i = 1; i < (1<<s); ++i) { if (!(i&(i-1))) { int u;
for (int j = i, k = 0; j; u = S[k++], j >>= 1);
for (int v = 0; v < n; ++v)
stc[i][v] = d[v][u]; }
else for (int v = 0; v < n; ++v) {
stc[i][v] = 0xffffff;
for (int j = 1; j < i; ++j)
if ((j|i) == i) { int x1 = j, x2 = i&(~j);
for (int w = 0; w < n; ++w)
stc[i][v] <?= d[v][w] + stc[x1][w] + stc[x2][w]; } } } }
```

/* Linear Programming: Simplex Method ---------------------------------------*/

```
// m - number of (less than) inequalities
// n - number of variables
// C - (m+1) by (n+1) array of coefficients:
// row 0 - objective function coefficients
// row 1:m - less-than inequalities
// column 0:n-1 - inequality coefficients
// column n - inequality constants (0 for objective function)
// X[n] - result variables
// return value - maximum value of objective function
// (-inf for infeasible, inf for unbounded)
#define MAXM 400 // leave one extra
#define MAXN 400 // leave one extra
#define EPS 1e-9
#define INF 1.0/0.0
double A[MAXM][MAXN];
int basis[MAXM], out[MAXN];
void pivot(int m, int n, int a, int b) { int i,j;
for (i=0;i<=m;i++) if (i!=a)
for (j=0;j<=n;j++) if (j!=b)
A[i][j] -= A[a][j] * A[i][b] / A[a][b];
for (j=0;j<=n;j++) if (j!=b) A[a][j] /= A[a][b];
for (i=0;i<=m;i++) if (i!=a) A[i][b] = -A[i][b]/A[a][b];
A[a][b] = 1/A[a][b];
i = basis[a]; basis[a] = out[b]; out[b] = i; }
double simplex(int m, int n, double C[][MAXN], double X[]) {
int i,j,ii,jj; // i,ii are row indexes; j,jj are column indexes
for (i=1;i<=m;i++) for (j=0;j<=n;j++) A[i][j] = C[i][j];
for (j=0;j<=n;j++) A[0][j] = -C[0][j];
for (i=0;i<=m;i++) basis[i] = -i;
```

```
for (j=0;j<=n;j++) out[j] = j;
for(;;) { for (i=ii=1;i<=m;i++)
if (A[i][n]<A[ii][n] || (A[i][n]==A[ii][n] && basis[i]<basis[ii]))
ii=i;
if (A[ii][n] >= -EPS) break;
for (j=jj=0;j<n;j++)
if (A[ii][j]<A[ii][jj]-EPS || (A[ii][j]<A[ii][jj]-EPS && out[i]<out[j]))
jj=j;
if (A[ii][jj] >= -EPS) return -INF;
pivot(m,n,ii,jj); }
for(;;) { for (j=jj=0;j<n;j++)
if (A[0][j]<A[0][jj] || (A[0][j]==A[0][jj] && out[j]<out[jj]))
jj=j;
if (A[0][jj] > -EPS) break;
for (i=1,ii=0;i<=m;i++)
if ((A[i][jj]>EPS) && (!ii || (A[i][n]/A[i][jj] < A[ii][n]/A[ii][jj]-EPS) ||
((A[i][n]/A[i][jj] < A[ii][n]/A[ii][jj]+EPS) &&
(basis[i] < basis[ii])))
ii=i;
if (A[ii][jj] <= EPS) return INF;
pivot(m,n,ii,jj); }
for (j=0;j<n;j++) X[j] = 0;
for (i=1;i<=m;i++) if (basis[i] >= 0)
X[basis[i]] = A[i][n]; return A[0][n]; }
```

```
/* Java Template: IO Reference ----------------------------------------------*/
// Description: This document is a reference for the use of java for regular
// IO purposes. It covers stdin and stdout as well as file IO.
// It also shows how to use StringTokenizer for parsing.
import java.util.*;
import java.io.*;
class IO {
public static void main(String[] args) {
try {
// For file IO, use:
// BufferedReader in=new BufferedReader(new FileReader("prob1.dat"));
// PrintWriter out=new PrintWriter(
// new BufferedWriter(new FileWriter("prob1.out")));
// For stdin/stdout IO, use:
PrintStream out = System.out;
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
String line;
int num=0;
StringTokenizer st;
while(true) {
// Newlines are removed by readLine()
line = in.readLine();
if(line == null) break;
num++;
out.println("Line #" + num);
// Split on whitespace
st = new StringTokenizer(line);
while(st.hasMoreTokens()) {
out.print("Token: ");
out.println(st.nextToken()); }
// To split on something else, use:
// st = new StringTokenizer(line, delim);
// Or use this to change in the middle of parsing:
// line = st.nextToken(delim); }
// You must flush for files!
out.flush(); }
catch (Exception e) {
System.err.println(e.toString()); } } }
```

```
/* Java Template: BigInteger Reference ----------------------------------------*/
// Description: This document is a reference for the use of the BigInteger
// class in Java. It contains code to compute GCDs of integers.
// Constants:
// ----------
// BigInteger.ONE - The BigInteger constant one.
// BigInteger.ZERO - The BigInteger constant zero.
// Creating BigIntegers
// -------------
// 1. From Strings
// a) BigInteger(String val);
// b) BigInteger(String val, int radix);
// 2. From byte arrays
// a) BigInteger(byte[] val);
```

```
// b) BigInteger(int signum, byte[] magnitude)
// 3. From a long integer
// a) static BigInteger BigInteger.valueOf(long val)
// Math operations:
// ----------------
// A + B = C --> C = A.add(B);
// A - B = C --> C = A.subtract(B);
// A * B = C --> C = A.multiply(B);
// A / B = C --> C = A.divide(B);
// A % B = C --> C = A.remainder(B);
// A % B = C where C > 0 --> C = A.mod(B);
// A / B = Q & A % B = R --> C = A.divideAndRemainder(B);
// (Q = C[0], R = C[1])
// A ^ b = C --> C = A.pow(B);
// abs(A) = C --> C = A.abs();
// -(A) = C --> C = A.negate();
//
// gcd(A,B) = C --> C = A.gcd(B);
// (A ^ B) % M --> C = A.modPow(B,M);
// C = inverse of A mod M --> C = A.modInverse(M);
// max(A,B) = C --> C = A.max(B);
// min(A,B) = C --> C = A.min(B);
// Bit Operations
// ------------------
// ~A = C (NOT) --> C = A.not();
// A & B = C (AND) --> C = A.and(B);
// A | B = C (OR) --> C = A.or(B);
// A ^ B = C (XOR) --> C = A.xor(B);
// A & ~B = C (ANDNOT) --> C = A.andNot(B);
// A << n = C (LSHIFT) --> C = A.shiftLeft(n);
// A >> n = C (RSHIFT) --> C = A.shiftRight(n);
// Clear n'th bit of A --> C = A.clearBit(n);
// Set n'th bit of A --> C = A.setBit(n);
// Flip n'th bit of A --> C = A.flipBit(n);
// Test n'th bit of A --> C = A.testBit(n);
//
// Bitcount of A = n --> n = A.bitCount();
// Bitlength of A = n --> n = A.bitLength();
// Lowest set bit of A --> n = A.getLowestSetBit();
// Comparison Operations
// ---------------------
// A < B --> A.compareTo(B) == -1;
// A == B --> A.compareTo(B) == 0
// or A.equals(B);
// A > B --> A.compareTo(B) == 1;
// A < 0 --> A.signum() == -1;
// A == 0 --> A.signum() == 0;
// A > 0 --> A.signum() == 1;
// Conversion:
// -----------
// double --> A.doubleValue();
// float --> A.floatValue();
// int --> A.intValue();
// long --> A.longValue();
// byte[] --> A.toByteArray();
// String --> A.toString();
// String (base b) --> A.toString(b);
import java.math.*;
import java.io.*;
import java.util.*;
class BigIntegers {
public static void main(String[] args) {
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
String line;
StringTokenizer st;
BigInteger a;
BigInteger b;
try {
while(true) {
line = in.readLine();
if(line == null) break;
st = new StringTokenizer(line);
a = new BigInteger(st.nextToken());
b = new BigInteger(st.nextToken());
System.out.println( a.gcd(b) ); } }
catch (Exception e) {
System.err.println(e.toString()); } } }
```

```
/* Number Theory: Converting between bases (Java, arb. precision) ----------*/
// Converts from base b1 to base b2
import java.math.*;
import java.io.*;
import java.util.*;
class base_convert {
// invalid is the string that is returned if the N is not valid
static String invalid = new String("Number is not valid");
private static String convert_base(int b1, int b2, String n, String key) {
int i, x;
String n2 = "", n3 = "";
BigInteger a = BigInteger.ZERO,
b1 = BigInteger.valueOf(base1),
b2 = BigInteger.valueOf(base2);
for (i = 0; i < n.length(); i++) {
a = a.multiply(b1);
x = key.indexOf(n.charAt(i));
if (x == -1 || x >= base1) return invalid;
a = a.add(BigInteger.valueOf(x)); }
while (a.signum() == 1) {
BigInteger r[] = a.divideAndRemainder(b2);
n2 += key.charAt(r[1].intValue());
a = r[0]; }
for (i = n2.length()-1; i >= 0; i--) n3 += n2.charAt(i);
if (n3.length() == 0) n3 += '0';
return n3; }
public static void main(String[] args) {
try {
String line, n;
int tnum, base1, base2;
StringTokenizer st;
// key is the base system that you may change as needed
String key = new
String("0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz");
// Standard IO
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
PrintStream out = System.out;
// File IO
// BufferedReader in = new BufferedReader(new FileReader("prob1.dat"));
// PrintWriter out = new BufferedWriter(new FileWriter("prob1.out"));
line = in.readLine(); // Get number of test cases
st = new StringTokenizer(line);
tnum = Integer.parseInt(st.nextToken());
for (int t = 0; t < tnum; t++) {
line = in.readLine();
st = new StringTokenizer(line);
base1 = Integer.parseInt(st.nextToken());
base2 = Integer.parseInt(st.nextToken());
n = st.nextToken();
String result = convert_base(base1, base2, n, key2);
out.println(result); } }
catch (Exception e) {
System.err.println(e.toString()); } } }
```

```
/* Number Theory: Mayor exponente de un primo que divide a n! ----------*/
int pow_div_fact(int n, int p) { int sd = 0;
for (int t = n; t > 0; t /= p) sd += t % p;
return (n-sd)/(p-1); }
```

```
/* Number Theory: Potencia modular ----------*/
// Returns (b^n)%m
// using <assert.h>
int fast_exp(int b, int n, int m) { int res = 1; int x = b;
while (n > 0) { if (n & 0x01) { n--; res = (res * x) % m; }
else { n >>= 1; x = (x * x) % m; } } return res; }
```

```
/* Number Theory: Primality Testing ----------*/
bool isPrime(int x) {
if(x == 1) return ONEPRIME;
if(x == 2) return true;
if(!(x & 1)) return false;
for(int i = 3; i*i <= x; i += 2) // watch for overflow
if (!(x % i)) return false; return true; }
```

```
/* Number Theory: Number of Divisors [O(sqrt(N))] ----------*/
int num_divisors(int n) {
int i, count, res = 1;
```

```
for(i = 2; i*i <= n; i++) { count = 0;
while(!(n%i)) { n /= i; count++; }
if(count) res *= (count+1); }
if (n > 1) res *= 2; return res; }

/* Number Theory: Prime Factorization ----------------------------------------*/
int primes[MAXP]; int psize;
void getPrimes() {
int i, j, isprime; psize = 0; primes[psize++] = 2;
for (i = 3; i <= MAXN; i+= 2) {
for (isprime = j = 1; j < psize; j++) { if (i % primes[j] == 0) { isprime = 0; break; }
if (1.0*primes[j]*primes[j] > i) break; }
if(isprime) primes[psize++] = i; } }
struct Factors { int size; int f[32]; };
Factors getPFactor(int n) { Factors x; int i; x.size = 0;
for (i = 0; i < psize; i++) { while (n % primes[i] == 0) {
x.f[x.size++] = primes[i]; n /= primes[i]; }
if(1.0*primes[i]*primes[i] > n) break; }
if(n > 1) x.f[x.size++] = n; return x; }

/* Number Theory: Primality testing with a sieve ----------------------------*/
// Consider using typedefs and functions instead of defines...
#define TEST(f,x) (*(f+(x)/16)&(1<<(((x)%16L)/2)))
#define SET(f,x) *(f+(x)/16)|=1<<(((x)%16L)/2)
#define ONEPRIME 0 // whether or not 1 is considered to be prime
#define UL unsigned long
#define UC unsigned char
UC *primes = NULL;
UL getPrimes(UL maxn) { UL x, y, psize=1;
primes = calloc(((maxn)>>4)+1L, sizeof(UC));
for (x = 3; x*x <= maxn; x+=2)
if (!TEST(primes, x)) for (y = x*x; y <= maxn; y += x<<1) SET (primes, y);
// Comment out if you don't need # of primes <= maxn
for(x = 3; x <= maxn; x+=2)
if(!TEST(primes, x)) psize++;
return psize; }
int isPrime(UL x) { // Returns whether or not a given POSITIVE number is prime
if(x == 1) return ONEPRIME;
if(x == 2) return 1;
if(x % 2 == 0) return 0; return (!TEST(primes, x)); }

/* Number Theory: Sum of divisors [O(sqrt(N))] ------------------------------*/
typedef long long int LL;
LL sum_divisors(LL n) { int i, count; LL res = 1;
for (i = 2; i*i <= n; i++) { count = 0;
while (n % i == 0) { n /= i; count++; }
if (count) res *= (pow(i, count+1)-1)/(i-1); }
if(n > 1) res *= (pow(n, 2)-1)/(n-1); return res; }

/* Number Theory: Chinese Remainder Theorem ---------------------------------*/
// Given n relatively prime modular in m[0], ..., m[n-1], and right-hand
// sides a[0], ..., a[n-1], the routine solves for the unique solution
// in the range 0 <= x < m[0]*m[1]*...*m[n-1] such that x = a[i] mod m[i]
// for all 0 <= i < n. The algorithm used is Garner's algorithm, which
// is not the same as the one usually used in number theory textbooks.
// It is assumed that m[i] are positive and pairwise relatively prime.
// a[i] can be any integer.
// If the system of equations is
// x = a[0] mod m[0]
// x = a[1] mod m[1]
// then a[i] should be reduced mod m[i] first.
// Also, if 0 <= a[i] < m[i] for all i, then the answer will fall
// in the range 0 <= x < m[0]*m[1]*...*m[n-1].
int gcd(int a, int b, int *s, int *t) {
int r, r1, r2, a1, a2, b1, b2, q;
a1 = b2 = 1; a2 = b1 = 0;
while (b) { q = a / b; r = a % b; r1 = a1 - q*b1;
r2 = a2 - q*b2; a = b; a1 = b1; a2 = b2;
b = r; b1 = r1; b2 = r2; }
*s = a1; *t = a2; return a; }
int cra(int n, int *m, int *a) {
int x, i, k, prod, temp;
int *gamma, *v;
gamma = malloc(n*sizeof(int));
v = malloc(n*sizeof(int));
for (k = 1; k < n; k++) { prod = m[0] % m[k];
for (i = 1; i < k; i++) { prod = (prod * m[i]) % m[k]; }
```

```
gcd(prod, m[k], gamma+k, &temp);
gamma[k] %= m[k];
if (gamma[k] < 0) gamma[k] += m[k]; } v[0] = a[0];
for (k = 1; k < n; k++) { temp = v[k-1];
for (i = k-2; i >= 0; i--) { temp = (temp * m[i] + v[i]) % m[k];
if (temp < 0) temp += m[k]; }
v[k] = ((a[k] - temp) * gamma[k]) % m[k];
if (v[k] < 0) v[k] += m[k]; }
x = v[n-1]; for (k = n-2; k >= 0; k--)
x = x * m[k] + v[k]; free(gamma); free(v); return x; }
int main(void) { int n, *m, *a, i, x;
while (scanf("%d", &n) == 1 && n > 0) {
m = malloc(n*sizeof(int)); a = malloc(n*sizeof(int));
printf("Enter moduli:\n");
for (i = 0; i < n; i++) scanf("%d", m+i);
printf("Enter right-hand side:\n");
for (i = 0; i < n; i++) scanf("%d", a+i);
x = cra(n, m, a); printf("x = %d\n", x); free(m); free(a); } }
```

```
/* Number Theory: Extended Euclidean Algorithm ------------------------------*/
// Assumes non-negative input. Returns d s.t. d = a*x + b*y
// x,y passed in by reference, #include <algorithm> for swap function
int gcd(int a, int b, int &x, int &y) {
x = 1; y = 0; int nx = 0, ny = 1;
while (b) { int q = a/b;
x -= q*nx; swap(x, nx); y -= q*ny; swap(y, ny); a -= q*b; swap(a, b); }
return a; }
```

```
/* Number Theory: Generalized Chinese Remaindering -------------------------*/
// Given [a_0, ..., a_(n-1)] and [m_0, ..., m_(n-1)]
// Computes 0 <= x < lcm(m_0, ..., m_(n-1)) such that
// x == a_0 mod m_0, ..., x == a_(n-1) mod m_(n-1), if
// such an x exists.
// True is returned iff such an x exists. If x does not exist then the value
// at the address of x will not be affected.
// Complexity: O(n log(MAX(m_0, ..., m_(n-1)) )
typedef long long int LLI;
LLI safe_mod(LLI a, LLI m) {
if (a < 0) return (a + m + m * (-a/m)) % m;
else return a % m; }
LLI abs(LLI a) { return a < 0 ? -a : a; }
LLI gcdex(LLI a, LLI b, LLI *ss, LLI *tt) {
LLI q, r[150], s[150], t[150];
int num = 2;
r[0] = a; r[1] = b; s[0] = t[1] = 1; s[1] = t[0] = 0;
while (r[num - 1]) { q = r[num - 2] / r[num - 1];
r[num] = r[num - 2] % r[num - 1];
s[num] = s[num - 2] - q * s[num - 1];
t[num] = t[num - 2] - q * t[num - 1]; ++num; }
*ss = s[num - 2]; *tt = t[num - 2];
return r[num - 2]; }
bool gen_chrem(LLI *a, LLI *m, int n, LLI *x) {
LLI g, s, t, a_tmp = safe_mod(a[0], m[0]), m_tmp = m[0];
for (int i = 1; i < n; ++i) {
g = gcdex(m_tmp, m[i], &s, &t);
if (abs(a_tmp - a[i]) % g) return false;
a_tmp = safe_mod(a_tmp + (a[i] - a_tmp) / g * s * m_tmp, m_tmp/g*m[i]);
m_tmp = m[i]; }
x = a_tmp; return true; }
int main() { int n; LLI a[20], m[20], x;
while (true) { scanf("%lld", &n);
if (!n) break;
for (int i = 0; i < n; ++i) scanf("%lld %lld", &a[i], &m[i]);
if (!gen_chrem(a, m, n, &x)) printf("No solution.\n\n");
else printf("X = %lld\n\n", x); } }
```

```
/* Number Theory: Rational Reconstruction [O(log m)] ------------------------*/
// Description: Given integers m, g and k, computes integers 'num' and 'den'
// (if they exist) such that num == g*den mod m where |num| < k and
// 0 < den < g/k. True is returned iff den is invertible mod m. This algorithm
// is useful if computations on rational numbers is to be used when the input
// and output numbers have small numerators and denominators but intermediate
// results can have very large numerators and denominators. To use in this
// fashion, reduce the input rationals modulo some number m (probably a prime),
// perform the operations modulo m and then use rational reconstruction to
// recover the results. m and k must be selected such that |num|, den < k
// and 2*k*k < m for all input and output rational numbers.
typedef long long int LLI;
```

```
int gcd_table(LLI a, LLI b, LLI *r, LLI *q, LLI *s, LLI *t) {
int n = 2; assert(0 <= a && 0 < b);
r[0] = a; r[1] = b; s[0] = t[1] = 1; s[1] = t[0] = 0;
while (r[n - 1]) { r[n] = r[n - 2] % r[n - 1];
q[n - 1] = r[n - 2] / r[n - 1];
s[n] = s[n - 2] - s[n - 1] * q[n - 1];
t[n] = t[n - 2] - t[n - 1] * q[n - 1]; ++n; }
return n; }
LLI gcd(LLI a, LLI b) { if (a < 0) return gcd(-a, b);
if (b < 0) return gcd(a, -b);
if (!b) return a;
return gcd(b, a % b); }
bool rat_recon(LLI m, LLI g, LLI k, LLI *num, LLI *den) {
int n, j;
LLI r[200], q[200], s[200], t[200], quo, tj, rj;
assert(0 <= g && g < m && 1 <= k && k <= m);
n = gcd_table(m, g, r, q, s, t);
q[0] = q[n - 1] = 0;
for (j = 0; j < n && r[j] >= k; ++j);
if (t[j] > 0) { *num = r[j]; *den = t[j]; }
else { *num = -r[j]; *den = -t[j];  }
if (gcd(r[j], t[j]) == 1) return true;
else { quo = (j == n - 1 ? 0 : (k - r[j-1]) / r[j] + 1);
rj = r[j - 1] - quo*r[j];
tj = t[j - 1] - quo*t[j];
if (gcd(rj, tj) != 1 || (tj > 0 ? tj : -tj) * k > m)
return false;
if (tj > 0) { *num = rj; *den = tj; }
else { *num = -rj; *den = -tj; } return true; } }
```

/* Search: Golden section search ----------*/

```
// Given an function f(x) with a single local minimum, a lower and upper
// bound on x, and a tolerance for convergence, this function finds the
// minimizing value of x. f(x) should evaluate globally.
#define GOLD 0.381966 // 1/phi^2 = 1/(phi+1) = (phi-1)^2
#define move(a,b,c) x[a]=x[b];x[b]=x[c];fx[a]=fx[b];fx[b]=fx[c]
double f(double x) { return x*x; } // Just an example
double golden(double xlow, double xhigh, double tol) {
double x[4], fx[4], L;
int iter = 0, left = 0, mini, i;
fx[0] = f(x[0]=xlow); fx[3] = f(x[3]=xhigh);
while (1) { L = x[3]-x[0];
if (!iter || left) { x[1] = x[0]+GOLD*L; fx[1] = f(x[1]); }
if (!iter || !left) { x[2] = x[3]-GOLD*L; fx[2] = f(x[2]); }
for (mini = 0, i = 1; i < 4; i++) if (fx[i] < fx[mini]) mini = i;
if (L < tol) break;
if (mini < 2) { left = 1; move(3,2,1); }
else { left = 0; move(0,1,2); }
iter++; } return x[mini]; }
```

/* Search: Suffix array [O(N log N)] ----------*/

```
// Notes: The build_sarray routine takes in a string S of n characters
// (null-terminated), and constructs two arrays 'sarray' and 'lcp'.
// - If p = sarray[i], then the suffix of str starting at p (i.e. S[p..n-1])
// is the i-th suffix (lexographically ordered)
// - NOTE: the empty suffix is not considered, so sarray[0] != n.
// - lcp[i] contains the length of the longest common prefix of the suffixes
// pointed to by sarray[i-1] and sarray[i] (but lcp[0] = 0).
// - To find a pattern P in str, you can look for it as the prefix of a
// suffix. This takes O(|P| log n) time with a binary search.
// You probably need to #include <climits> here.
#define MAXN 100000
int bucket[CHAR_MAX-CHAR_MIN+1];
int prm[MAXN], count[MAXN];
char bh[MAXN+1];
void build_sarray(char *str, int* sarray, int *lcp) {
int n = strlen(str), a, c, d, e, f, h, i, j, x;
memset(bucket, -1, sizeof(bucket));
for (i = 0; i < n; i++) { j = str[i] - CHAR_MIN;
prm[i] = bucket[j]; bucket[j] = i; }
for (a = c = 0; a <= CHAR_MAX - CHAR_MIN; a++)
for (i = bucket[a]; i != -1; i = j) {
j = prm[i]; prm[i] = c; bh[c++] = (i == bucket[a]); }
bh[n] = 1;
for (i = 0; i < n; i++) sarray[prm[i]] = i;
x = 0;
for (h = 1; h < n; h *= 2) { for (i = 0; i < n; i++) { if (bh[i] & 1) {
x = i; count[x] = 0; } prm[sarray[i]] = x; }
```

```
d = n - h; e = prm[d]; prm[d] = e + count[e]++;
bh[prm[d]] |= 2; i = 0;
while (i < n) { for (j = i; (j == i || !(bh[j] & 1)) && j < n; j++) {
d = sarray[j] - h;
if (d >= 0) { e = prm[d]; prm[d] = e + count[e]++; bh[prm[d]] |= 2; } }
for (j = i; (j == i || !(bh[j] & 1)) && j < n; j++) {
d = sarray[j] - h;
if (d >= 0 && (bh[prm[d]] & 2)) {
for (e = prm[d]+1; bh[e] == 2; e++);
for (f = prm[d]+1; f < e; f++) bh[f] &= 1; } } i = j; }
for (i = 0; i < n; i++) {
sarray[prm[i]] = i;
if (bh[i] == 2) bh[i] = 3; } } h = 0;
for (i = 0; i < n; i++) { e = prm[i]; if (e > 0) {
j = sarray[e-1];
while (str[i+h] == str[j+h]) h++; lcp[e] = h;
if (h > 0) h--; } } lcp[0] = 0; }
```

/* Misceláneas: Convertir de un sistema numérico a otro ----------*/

```
int sini, send, i, j, l, b, dec1, r; char n[100], sol[100];
void decimal() {
for (i = l; i >= 0; i--) { n[i] -= 48;
if (n[i] >= 10) n[i] -= 7;
dec1 += n[i] * b; b *= sini; } }
void sistem() {
while (dec1 != 0) { r = dec1 % send; dec1 /= send;
if (r >= 10) r += 7;
r += 48; sol[i] = r; i++; } }
int main() { scanf ("%s %d %d", &n, &sini, &send);
l = strlen(n) - 1; b = 1; decimal(); i = 0;
sistem();
for (j = i - 1; j >= 0; j--) printf ("%c", sol[j]);
return 0; }
```

/* Misceláneas: Combinar las letras de una palabra ----------*/

```
int len, cnt; string a, b;
void Comb(int pos, int k){
for (int i = k; i < a.size(); i++){ b[pos] = a[i];
if (pos < len - 1) Comb(pos + 1, i + 1);
else printf("%s\n", b.c_str()), cnt++; } }
int main(){ cin >> a;
for (len = 1; len <= a.size(); len++){ b = "";
for (int j = 0; j < len; j++) b += ' ';
Comb(0, 0); } printf("%d\n", cnt); return 0; }
```

/* Misceláneas: Disjoin Set ----------*/

```
int N, M, Q, Set[maxg], Rank[maxg];
void join_set ( int nodo, int newn ) {
if ( Rank[nodo] > Rank[newn] ) { Rank[nodo] += Rank[newn];
Set[newn] = nodo; }
else { Rank[newn] += Rank[nodo];
Set[nodo] = newn; } }
int Find_Set ( int nodo ) {
if ( nodo != Set[nodo] ) Find_Set ( Set[nodo] );
return Set[nodo]; }
int main () { scanf ("%d%d%d\n", &N, &M, &Q);
for ( int i = 1; i <= N; i ++ ) Set[i] = i, Rank[i] = 1;
int a, b, setnodo, setnewn;
for ( int i = 1; i <= M; i ++ ) { scanf ("%d%d\n", &a, &b);
setnodo = Find_Set ( a ); setnewn = Find_Set ( b );
join_set (setnodo, setnewn); }
for ( int i = 1; i <= Q; i ++ ) { scanf ("%d%d\n", &a, &b);
setnodo = Find_Set ( a ); setnewn = Find_Set ( b );
if ( setnodo == setnewn ) printf ("Pertenecen al mismo Grupo\n");
else printf ("No pertenecen al mismo Grupo\n"); } return 0; }
```

/* Misceláneas: Bitwise operations ----------*/

```
__builtin_clz(unsigned int x) // Retorna 32 menos la cantidad de dígitos en binario.
__builtin_ctz(unsigned int x) // Retorna la cantidad de zeros a la derecha.
__builtin_popcount(unsigned int x) // Retorna la cantidad de unos.
__builtin_parity(unsigned int x) // Retorna la cantidad de unos modulo 2.
```

/* Misceláneas: Find whether a 2d matrix is subset of another 2d matrix --- */

```
const int N = 100; const int M = N; int n, m;
string haystack[N], needle[M];
int A[N][N]; // filled by successive calls to match
int p[N]; // pattern to search for in columns of A
```

```
struct Node { Node *a[2]; // alphabet is binary
Node *suff; // pointer to node whose prefix = longest proper suffix of this node
int flag;
Node() { a[0] = a[1] = 0; suff = 0; flag = -1; } };
void insert(Node *x, string s)
{ static int id = 0; static int p_size = 0;
for(int i = 0; i < s.size(); i++) { char c = s[i];
if(x->a[c - '0'] == 0) x->a[c - '0'] = new Node;
x = x->a[c - '0']; }
if(x->flag == -1) x->flag = id++;
// update pattern
p[p_size++] = x->flag; }
Node *longest_suffix(Node *x, int c) {
while(x->a[c] == 0) x = x->suff; return x->a[c]; }
Node *mk_automaton(void) { Node *trie = new Node;
for(int i = 0; i < m; i++) insert(trie, needle[i]);
queue<Node*> q;
// level 1
for(int i = 0; i < 2; i++) {
if(trie->a[i]) { trie->a[i]->suff = trie;
q.push(trie->a[i]); }
else trie->a[i] = trie; }
// level > 1
while(q.empty() == false) { Node *x = q.front(); q.pop();
for(int i = 0; i < 2; i++) {
if(x->a[i] == 0) continue;
x->a[i]->suff = longest_suffix(x->suff, i);
q.push(x->a[i]); } }
return trie; }
// search for patterns in haystack[j]
void match(Node *x, int j) {
for(int i = 0; i < n; i++) {
x = longest_suffix(x, haystack[j][i] - '0');
if(x->flag != -1) { A[j][i-m+1] = x->flag; } } }
int match2d(Node *x) { int matches = 0;
static int z[M+N]; static int z_str[M+N+1];
// init
memset(A, -1, sizeof(A));
// fill the A matrix
for(int i = 0; i < n; i++) match(x, i);
// build string for z algorithm
z_str[n+m] = -2; // acts like `\0` for strings
for(int i = 0; i < m; i++) z_str[i] = p[i];
for(int i = 0; i < n; i++) { /* search for pattern in column i */
for(int j = 0; j < n; j++) z_str[j + m] = A[j][i];
// run z algorithm
int l, r; l = r = 0; z[0] = n + m;
for(int j = 1; j < n + m; j++) {
if(j > r) { l = r = j;
while(z_str[r] == z_str[r - l]) r++;
z[j] = r - l; r--; }
else { if(z[j - l] < r - j + 1) z[j] = z[j - l];
else { l = j;
while(z_str[r] == z_str[r - l]) r++;
z[j] = r - l; r--; } } }
// locate matches
for(int j = m; j < n + m; j++) { if(z[j] >= m) {
printf("match at (%d,%d)\n", j - m, i); matches++; } } }
return matches; }

int main() { cin >> n >> m;
for(int i = 0; i < n; i++) cin >> haystack[i];
for(int i = 0; i < m; i++) cin >> needle[i];
Node *trie = mk_automaton(); match2d(trie);
return 0; }

/* Misceláneas: Roman Numerals ----------------------------------------------*/
map<string, int, less <string> > dict;
char nums[5000][20];
void gen_roman() {
char *roman[13] = {"M","CM","D","CD","C","XC","L","XL","X","IX","V","IV","I"};
int i, j, n, arab[13] = {1000,900,500,400,100,90,50,40,10,9,5,4,1};
string key;
for (i = 0; i < 5000; i++) { nums[i][0] = 0;
for (n = i, j = 0; n; j++) for (; n >= arab[j]; n -= arab[j]) strcat(nums[i],roman[j]);
key = nums[i]; dict[key] = i; } }
char *to_roman(int n) {
if (n < 1 || n >= 5000) return 0; return nums[n]; }
```

```
int to_arabic(char *in) { string key = in;
if (!dict.count(key)) return -1; return dict[key]; }

int main() { int i; gen_roman();
for (i = 1; i < 5000; i++)
printf("%d = %s\n",to_arabic(to_roman(i)),to_roman(i)); return 0; }

/* Misceláneas: Euler Phi function ----------------------------------------------*/
// returns the number of positive integers less than N that are relatively prime to N
int phi(int n){ int i, count, res = 1;
for(i = 2; i*i <= n; i++){ count = 0;
while(n % i == 0){ n /= i; count++; }
if(count > 0) res *= (pow(i, count)-pow(i, count-1)); }
if(n > 1) res *= (n-1); return res; }

/* Misceláneas: Farey Sequence Generator ----------------------------------------*/
// The Farey Sequence of order n is the list of all reduced fractions between 0 and 1
// (inclusive) in sorted order.
// e.g. order 6:
// 0/1, 1/6, 1/5, 1/4, 2/5, 1/3, 1/2, 2/3, 3/5, 3/4, 4/5, 5/6, 1/1
// Given any positive integer n, this algorithm will generate the Farey sequence in order
// with one term being generated per loop iteration.
void farey(int n) { int h = 0, k = 1, x = 1, y = 0;
do { cout << h << '/' << k << endl; int r = (n-y)/k;
y += r*k; x += r*h; swap(x,h); swap(y,k); x = -x; y = -y; } while (k > 1);
cout << "1/1" << endl; }

/* Misceláneas: Cubic equation solver -------------------------------------------*/
// Finds solutions to the cubic equation: ax^3+bx^2+cx+d = 0
typedef struct { int n; // Number of solutions
double x[3]; // Solutions
} Result;
double PI;
Result solve_cubic(double a, double b, double c, double d){ Result s;
long double a1 = b/a, a2 = c/a, a3 = d/a;
long double q = (a1*a1 - 3*a2)/9.0, sq = -2*sqrt(q);
long double r = (2*a1*a1*a1 - 9*a1*a2 + 27*a3)/54.0;
double z = r*r-q*q*q; double theta;
if(z <= 0){ s.n = 3;
theta = acos(r/sqrt(q*q*q));
s.x[0] = sq*cos(theta/3.0) - a1/3.0;
s.x[1] = sq*cos((theta+2.0*PI)/3.0) - a1/3.0;
s.x[2] = sq*cos((theta+4.0*PI)/3.0) - a1/3.0; }
else { s.n = 1;
s.x[0] = pow(sqrt(z)+fabs(r),1/3.0);
s.x[0] += q/s.x[0];
s.x[0] *= (r < 0) ? 1 : -1;
s.x[0] -= a1/3.0; } return s; }

int main(){ double a,b,c,d; Result r; int i;
PI = acos(-1);
while(scanf("%lf %lf %lf %lf", &a, &b, &c, &d) == 4){
r = solve_cubic(a,b,c,d); printf("%d solution(s)\n", r.n);
for(i = 0; i < r.n; i++){ printf("x = %f\n", r.x[i]); } } return 0; }

/* Misceláneas: Digits in N! ----------------------------------------------------*/
// Given N, computes the number of digits that N! will occupy in base B.
long long fac_digit(int n, int b) { double sum = 0; int i;
for (i = 2; i <= n; i++) sum += log(i);
return (long long) floor(1+sum/log(b)); } // don't use ceil!
```