# University of Alberta 2006 ACM ICPC World Finals Code Archive

Contents

```
double PI = 2*acos(0.0);
double EPS = 1E-8;
```

```
/* Complex Arithmetic and FFT -------------------------------------------
-*/
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; }
// N is a power of 2, x contains N elements
// Can generate W and rev globally if doing many transforms of same size
// Put a -1 into inv to get the inverse transform
vector<point> FFT(const vector<point> &y, int N, int inv = 1) {
  vector<point> W(N), x(N); vector<int> rev(N); double f = inv*2*PI;
  for (int i = 0; i < N; ++i) {
    W[i] = pol(1, i*f/N); rev[i] = rev[i>>1]>>1;
    if (i & 1) rev[i] += N >> 1;
    x[i] = y[rev[i]];
    if (inv == -1) x[i] /= N;
    }
  for (int i = 1; i < N; i <<= 1)
    for (int j = 0; j < N; j += i+i)
      for (int k = 0; k < i; ++k) {
        point T = x[j+k], B = W[k*(N/(i+i))]*x[j+k+i];
        x[j+k] = T+B; x[j+k+i] = T-B;
        }
  return x;
  }
```

```
/* 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;
  }
```

```
/* 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;
  }
```

```
/* 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;
  }
```

```
/* 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));
  }
```

```
/* 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;
  }
```

```
/* Area of intersection of two general polygons (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;
  }
// -1: p left of a->b, +1: p right of a->b, 0: p on a->b
int leftRight(const point &a, const point &b, const point &p) {
  double d = cross(a, b, a, p);
  if (d > EPS) return -1;
  if (d < -EPS) return 1;
  return 0;
  }
// tests if b in a->b->c is concave/flat
bool isConcave(point &a, point &b, point &c) {
  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);
  }
// Accepts a vector of n ordered vertices, returns triangulation.  No tri-
angles
// if n < 3.
vector<triangle> triangulate(vector<point> &orig) {
  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;
  }
// Finds intersection of segments a->b and c->d (returns 0 if none or infi-
nite)
bool isectLineSegs(point &a, point &b, point &c, point &d, point &p) {
  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;
  }
```

```
/* 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
    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;
  }
```

```
/* 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;
  }
```

```
/* 3D Geometry 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) {}
  // Direction unit vector a -> b
  point3 dir() { return (b - a).norm(); }
  };

// Closest point on an infinite line u to a given point p
point3 cpoint_iline(line u, point3 p) {
  point3 ud = u.dir();
  return u.a - ud*dot(u.a - p, ud);
  }
// Shortest distance between two infinite lines u and v
double dist_ilines(line u, line v) {
  return dot(v.a - u.a, cross(u.dir(), v.dir()).norm());
  }
// Finds the closest point on infinite line u to infinite line v.
// Note: if (uv*uv - uu*vv) is zero then the lines are parallel and such a
// single closest point does not exist.  Check for this if needed.
point3 cpoint_ilines(line u, line v) {
  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;
  }
// Closest point on a line segment u to a given point p
point3 cpoint_lineseg(line u, point3 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;
  }
// Planes
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) {}
  //Value of d for the equation ax + by + cz + d = 0
  double d() { return -dot(n, p); }
  };
//Closest point on a plane u to a given point p
point3 cpoint_plane(plane u, point3 p) {
  return p - u.n*(dot(u.n, p) + u.d());
  }
//Point of intersection between an infinite line v and a plane u.
//Note: if dot(u.n, vd) == 0 then the line and plane do not intersect at a
//single point. Check for this case if it is needed.
point3 iline_isect_plane(plane u, line v) {
  point3 vd = v.dir();
  return v.a - vd*((dot(u.n, v.a) + u.d())/dot(u.n, vd));
  }
//Infinite line of intersection between two planes u and v.
//Note: if dot(v.n, uvu) == 0 then the two planes do not intersect at a line.
//Check for this case if it is needed.
line isect_planes(plane u, plane v) {
  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);
  }
```

```
/* 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));
  }
```

```
/* Circle described by three points */
int circle(point p1, point p2, point p3, point &center, double &r) {
  double a,b,c,d,e,f,g;
  a = p2.x - p1.x;  b = p2.y - p1.y;
  c = p3.x - p1.x;  d = p3.y - p1.y;
  e = a*(p1.x + p2.x) + b*(p1.y + p2.y);
  f = c*(p1.x + p3.x) + d*(p1.y + p3.y);
  g = 2.0*(a*(p3.y - p2.y) - b*(p3.x - p2.x));
  if (fabs(g) < EPS) return 0;
  center.x = (d*e - b*f) / g; center.y = (a*f - c*e) / g;
  r = sqrt(SQR(p1.x-center.x)+SQR(p1.y-center.y));
  return 1;
  }
```

```
/* Arithmetic: Discrete Logarithm solver
   Description: Given prime P, B, and N, finds the smallest
                exponent L such that B^L == N (mod P) O(sqrt(P)) */

map<UI,UI> M;

UL times (UL a, UL b, UL m){
  return (ULL) a * b % m; }

UL power(UL val, UL power, UL m){
  UL res = 1, p;

  for(p = power; p; p=p>>1){
    if(p & 1) res = times(res, val, m);
    val = times(val, val, m);
  }
  return res;
}

int discrete_log(UI p, UI b, UI n){
  UL i, j, jump;

  M.clear();
  jump = (int)sqrt(p);
  for (i = 0; i < jump && i < p-1; i++){
    M[power(b,i,p)] = i+1;
  }
  for (i = 0; i < p-1; i+= jump){
    if (j = M[times(n,power(b,p-1-i,p),p)]) {
      j--;
      return (i+j)%(p-1);
    }
  }
  return -1;
}
```

```
/* Arithmetic: Fast Exponentition */

LL fast_exp(int b, int n){
  LL res = 1, x = b, p;
  for(p = n; p; p >>= 1, x *= x)
    if(p & 1) res *= x;
  return res;
}
```

```
/* Arithmatic: Simpson's Rule for Numerical Intergration */

double Simpson(double a, double b, int k, double (*f)(double)){
  double dx, x, t; int i;

  dx = (b-a)/(2.0*k);
  t = 0;
  for( i=0; i<k; i++ ) {
    t += (i==0 ? 1.0 : 2.0) * (*f)(a+2.0*i*dx);
    t += 4.0 * (*f)(a+(2.0*i+1.0)*dx);
  }
  t += (*f)(b);
  return t * (b-a)/6.0/k;
}
```

```
/* Arithmetic: Cubic equation solver */

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;
}
```

```
/* Miscellaneous: Coupons Problem
   Description: Coupons are given away in boxes of cereal.  There are
                'm' different kinds of coupons (with equiprobable
                distribution).  How many boxes of cereal would you
                have to buy, on average, to collect them all?
*/

double ncoupons(int m) {
  double num = 0.0; int i;
  for (i = 1; i <= m; i++) num += m/(double) i;
  return num;
}

/* A related problem:  If you buy 'n' boxes of cereal, what is the
   probability you get at least one of each of the 'm' coupons?  */

double nways[100][100];

void make_coupon_table() {
  double fact = 1.0;
  int i, j;

  for (i = 1; i < 100; i++) {
    nways[i][1] = 1.0;
    for (j = 2; j < i; j++)
      nways[i][j] = j*(nways[i-1][j] + nways[i-1][j-1]);
    nways[i][i] = fact *= i;
  }
}

double query_table(int m, int n) {
  if (n < m) return 0.0;
  if (m == 0) return 1.0;
  if (n >= 100 || m >= 100) exit(1);
  return nways[n][m]/pow(m,n);
}
```

```
/* Arithmetic: Binomial coefficient */

long double bin[MAXN+1][MAXN+1];
void getBinCoeff(){
  int i, k;
  for(k = 0; k <= MAXN; k++){
    bin[k][0] = bin[k][k] = 1;
    for(i = 1; i < k; i++)
      bin[k][i] = bin[k-1][i-1]+bin[k-1][i];
  }
}
```

```
/* Combinatorics: Digit Occurence count
   Description: Given a digit and a number N, return the number of
                times the digit occurs from 1..N. */

#include <stdio.h>
#include <string.h>
#include <math.h>

long long digit_count(int digit, int max){
  long long res = 0;
  char buff[15];
  int i, count;

  if(max <= 0) return 0;

  /* Number of times "digit" occurs in the one's place */
  res += max/10 + ((max % 10) >= digit ? 1 : 0);

  /* Since we start from 1, if digit = 0, remove 1 since "0"
     doesn't count */
  if(digit == 0) res--;

  /* Get the number of occurences in max/10-1, and multiply this by
     10 since we can choose 10 possible last digits [0-9] */
  res += digit_count(digit, max/10 - 1) * 10;

  /* The number of occurences in max/10 is equal to (1+max%10) * the
     number of times "digit" occurs in max/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: Digits in N!
   Description: 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! */
}
```

```
/* Combinatorics: Josephus Ring Survivor
   Description: Suppose that there are n people in a ring, [0..n-1].
                Count around the ring, starting from 0, and
               dismissing every m-th person.
*/

int survive[MAXN];
void josephus(int n, int m){
  int i;
  survive[1] = 0;
  for(i = 2; i <= n; i++)
    survive[i] = (survive[i-1]+(m%i))%i;
}
```

```
/* Combinatorics - Permutation index on distinct characters
   Description: Given a string formed of distinct characters,
                returns the index of the permutation from 0..N!-1.
   This does not work when characters can be the same for example: "aaba" */

int permdex (char *s){
  int i, j, size = strlen(s);
  int index = 0;

  for (i = 1; i < size; i++){
    for (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));
  /* assert(m && seq); */

  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 {
      /* assert(A[m[1]] <= A[c] && A[c] < 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;
      }
      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;
}

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));
  /* assert(m && seq); */

  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) {
        /* assert(A[m[h]] <= A[c] && A[c] < A[m[j]]); */
        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;
}
```

```
/* Dynamic Programming: Integer Parititoning
   Description: Template for calculating the number of ways of
                partitioning the integer N into M parts.

   Notes:       A partition of a number N is a representation of
                N as the sum of positive integers
               e.g. 5 = 1+1+1+1+1

               The number of ways of partitioning an integer N
               into M parts is equal to the number of ways of
               partitioning the number N with the largest element
               being of size M.  This is best seen with a Ferres-
               Young diagram:
               Suppose N = 8, M = 3:

               4 = * * * *
                3 = * * *
                1 = *
                   3 2 2 1
               By transposition from rows to columns, this equality
               can be seen.

               P(N, M) = P(N-1, M-1) + P(N-M, M)
               P(0, M) = P(N, 0) = 0
               P(N, 1) = 1
*/

#include <stdio.h>
#include <string.h>
#define MAXN 300
#define ULL unsigned long long

ULL A[MAXN+1][MAXN+1];

void Build(){
  int i, j;

  memset(A, 0, sizeof(A));
  A[0][0] = 1;
  for(i = 1; i <= MAXN; i++){
    A[i][1] = 1;
    for(j = 2; j <= i; j++)
      A[i][j] = A[i-1][j-1] + A[i-j][j];
  }
}
```

```
/* Generator: Catalan Numbers */

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

```
/* Generators: Binary Strings generator - (ordered by cardinality) */

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){
  int i;
  for(i = 0; i <= n; i++){
    printf(“Cardinality %d:\n”, i);
    recurse(n, 0, i);
  }
}
```

```
/* Graph Theory: Maximum Bipartite Matching
                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.
                                  ******      *****

   Notes:        - 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].
                - This code is roughly 2 times slower than the old
                  code since it doesn’t try multiple BFS paths at
                  once, however, it’s about 4 times shorter... */

#define 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){
  int i;
  memset(match, -1, sizeof(match));
  for(i = 0; i < n+m; i++) if(find(i,n,m)) i = 0;
}

int main(){
  int n, m, esize, x, y;
  int i, count;

  /* Read size of set U into n, size of set V into m */
  while(scanf(“%d %d”, &n, &m) == 2){

    memset(e, 0, sizeof(e));           /* Clear edges */
    scanf(“%d”, &esize);               /* get # of edges */
    while(esize--){
      scanf(“%d %d”, &x, &y);          /* add edges */
      addEdge(x,y,n);                  /* Edges [0,n-1]->[0,m-1] */
    }

    bipmatch(n, m);                    /* Perform matching */

    for(count = i = 0; i < n; i++){    /* Print results */
      if(match[i] != -1){
        printf(“%d->%d\n”, i, match[i]-n);
        count++;
      }
    }
    printf(“Matching size: %d\n”, count);
  }
  return 0;
}
```

```
/* Graph Theory: Eulerian Graphs
                Before adding edges, call Init() to initialize all
                necessary data structures.
                Use the provided function addEdge(x,y,c) which
                adds c number of 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 */

#define MAXN 105     /* Number of nodes */
#define 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];
int g[MAXN][MAXN];
int seq[MAXM], seqsize;
```

```
/* Uncomment if you need copy of graph
   int g2[MAXN][MAXN], deg2[MAXN];
*/

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;

  /* Check if graph is connected.  If all vertices
     are guaranteed to be used then use this:

     if(sets[getRoot(0)] != -n) return 0;

     Otherwise, count only vertices used like this: */
  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]%2){
      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;
  /* Uncomment if you need copy of graph
     memcpy(g, g2, sizeof(g));
     memcpy(deg, deg2, sizeof(deg));
  */
  getPath(n, start, end);
}

int main(){
  int i, x,y,start,end, n, m;

  while(scanf("%d %d", &n, &m) == 2){
    Init();
    for(i = 0; i < m; i++){
      scanf("%d %d", &x, &y);
      addEdge(x,y,1);
    }
    /* Uncomment if you need copy of graph
       memcpy(g2, g, sizeof(g2));
       memcpy(deg2, deg, sizeof(deg2));
    */
    switch(isEulerian(n, &start, &end)){
                        ...
  }
  return 0;
}
```

```
/* 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).
*/

#include <cstdio>
#include <vector>
#include <list>
```

```
using namespace std;

#define MAXN 200

//Edge u->v with capacity c
struct Edge {
  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 multiedge)*/
int deg[MAXN];          /* Degree count */
int A[MAXN+1];          /* Used by perfect matching generator */
int sum;                /* Sum of costs */
int odd;
int best;

void floyd(int n){
  int i, j, k;

  for(k = 0; k < n; k++) for(i = 0; i < n; i++) for(j = 0; j < n; j++)
    if(g[i][k] != -1 && g[k][j] != -1){
      int temp = g[i][k] + g[k][j];
      if(g[i][j] == -1 || g[i][j] > temp)
        g[i][j] = temp;
    }
  for(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));
  }
  return 0;
}
```

```
/* Graph Theory: Strongly Connected Components */

#define VI vector<int>
#define MAXN 1000
VI g[MAXN], curr;
vector< VI > 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]){
    VI 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);
}
```

```
/* Number Theory: Converting between bases (arbitrary precision)
   Description: Given a starting base b1, and a target 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 base1, int base2,
                                     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(“0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstu-
vwxyz”);

    // 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());
}}}

/* Java Template: BigInteger Reference
   =================================================================
   Description: This document is a reference for the use of the
                BigInteger class in Java.  It contains sample code
                that computes GCDs of pairs of integers.

   Constants:
   ----------
   [1.2]  BigInteger.ONE  - The BigInteger constant one.
   [1.2]  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);

   ----------------------------------------------------------------*/
```

```
/* Reads in lines of input until EOF is encountered.  For each line
   of input it will extract two integers and then print out their
   GCD. */

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());
        }
    }
}
```

```
/* Java Template: IO Reference
   ================================================================
   Description: This document is a reference for the use of the
                java for regular IO purposes.  It covers stdin and
                stdout as well as file IO.  It also shows how to use
                StringTokenizer for parsing.

   ----------------------------------------------------------------
   Author:      Patrick Earl
   Date:        Nov 14, 2002
   References:  Java API Documentation
   ----------------------------------------------------------------
   Reliability: 0
*/

import java.util.*;
import java.io.*;

class IO {
    public static void main(String[] args) {
        try {
            /*
            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++;

                /* Print out line number. */
                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());
        }
    }
}
```

```
/* Miscellaneous: Bit Count */

int bitcount(int a){
  int c = 0;

  while(a){
    c++; a &= a-1;
  } return c;
}
```

```
/* Number Theory: Euler Phi function */

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;
}
```

```
/* Number Theory: Primality Testing */

int isPrime(int x){
  int i;
  if( x == 1 ) return ONEPRIME;
  if( x == 2 ) return 1;
  if( x % 2 == 0) return 0;

  for(i = 3; i*i <= x; i+=2)
    if( x % i == 0) return 0;
  return 1;
}
```

```
/* Number Theory: Number of Divisors; O(sqrt(N)) */

#include <stdio.h>

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;
  }
}

typedef struct{
  int size;
  int f[32];
} Factors;

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 */

#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;
}

/* Returns whether or not a given POSITIVE number if prime. */
int isPrime(UL x){
  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)) */
```

```
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;
}
```

```
/* Chinese Remainder Theorem (cra.c)
 *
 * Author: Howard Cheng
 * Reference:
 *   Geddes, K.O., Czapor, S.R., and Labahn, G.  Algorithms for Computer
 *   Algebra, Kluwer Academic Publishers, 1992, p. 180
 *
 * 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].
 *
 * Added: 5 January 2000
 * Confirmed: Matthew McNaughton (mcnaught@cs.ualberta.ca)
 */

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

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) {
    /* assert(a1*A + a2*B == a); */
    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;
  /* assert(a >= 0); */
  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));
  /*  assert(gamma && v); */

  /* compute inverses */
  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];
    }
  }

  /* compute coefficients */
  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];
    }
  }

  /* convert from mixed-radix representation */
  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));
    assert(m && a);
    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);

    for (i = 0; i < n; i++) {
      assert((x-a[i]) % m[i] == 0);
    }
    free(m);
    free(a);
  }
  return 0;
}
```

```
/* Extended Euclidean Algorithm */

int gcd(int a, int b, int *s, int *t)
{
  int r, r1, r2, a1, a2, b1, b2, q;
  int A = a;
  int B = b;

  /* unnecessary if a, b >= 0 */
  if (a < 0) {
    r = gcd(-a, b, s, t);
    *s *= -1;
    return r;
  }
  if (b < 0) {
    r = gcd(a, -b, s, t);
    *t *= -1;
    return r;
  }

  a1 = b2 = 1;
  a2 = b1 = 0;

  while (b) {
    assert(a1*A + a2*B == a);
    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;
  assert(a >= 0);
  return a;
}
```

```
/* Fast Exponentiation mod m */

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;
}
```

```
//  Simplex Method for Linear Programming
//
//  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];
}

void print(int m, int n, char *msg) {  // not used -- debug only
   int i,j;
   printf(“%s\n”,msg);
   for(i=0;i<=m;i++) {
      for (j=0;j<=m;j++) printf(“ %10d”,i==j);
      for (j=0;j<=n;j++) printf(“ %10g”,A[i][j]);
      printf(“\n”);
   }
   for (i=0;i<=m;i++) printf(“ %10d”,basis[i]);
   for (j=0;j<n;j++) printf(“ %10d”,out[j]);
   printf(“\n”);
}
```

```
/* Gray code. Generates a b-bit gray code starting from 0. */
/* the i’th gray code is i^(i>>1). Magic. */

char *
pbits(char *s, int n, int b) {
  unsigned int i; char *t;
  t = s;
  for( i = 1 << (b-1); i != 0; i >>= 1 ) {
    *s++ = n&i ? ‘1’ : ‘0’;
  }
  *s++ = ‘\0’;
  return t;
}
```

```
/* Search: Golden section Search
   Description: 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 value of x
                The function is written globally as f(x)
   Notes:        - watch out for -0.000 */

#include <stdio.h>

#define GOLD 0.381966
#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;
}

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];
}
```

```
/* Searching: Suffix array
   =================================================================
   Description: Builds a suffix array of a string of N characters

   Complexity:  O(N log N)
   -----------------------------------------------------------------
   Author:      Howard Cheng
   Date:        Oct 30, 2003
   References:  Manber, U. and Myers, G.  "Suffix Arrays: a New
                Method for On-line String Searches."
                SIAM Journal on Computing.  22(5) p. 935-948, 1993.

                T. Kasai, G. Lee, H. Arimura, S. Arikawa, and
                K. Park.  "Linear-time Longest-common-prefix
                Computation in Suffix Arrays and Its Applications."
                Proc. 12th Annual Conference on Combinatorial
                Pattern Matching, LNCS 2089, p. 181-192, 2001

   -----------------------------------------------------------------
   Reliability: 1 (Spain 719 - Glass Beads)
   Notes:      The build_sarray routine takes in a string S of n
               characters (null-terminated), and constructs two
               arrays sarray and lcp.  The properties are:

             - If p = sarray[i], then the suffix of str starting at
               p (i.e. S[p..n-1] is the i-th suffix when all the
               suffixes are sorted in lexicographical order

             - NOTE: the empty suffix is not included in this list,
                     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].  lcp[0] is defined to be 0.

             - To see whether a pattern P occurs in str, you can
               look for it as the prefix of a suffix.  This can be
               done with a binary search in O(|P| log n) time.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <assert.h>

#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, a, c, d, e, f, h, i, j, x;

  n = strlen(str);

  /* sort the suffixes by first character */
  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;

/* inductive sort */
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];
  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];
        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;
}

int main(){
  char S[MAXN]; int sarray[MAXN], lcp[MAXN], i;
  char T[MAXN];
  int n, j;
  while (scanf(“%s”, S) == 1) {
    n = strlen(S);
    for(i = 0; i < n; i++) S[n+i] = S[i];
    S[n+n] = 0;
    build_sarray(S, sarray, lcp);
    for (i = 0; S[i]; i++)
      if(sarray[i] < n){
        printf(“%3d: %2d [%d]\n”, i, lcp[i], n);
        for(j = 0; j < n; j++){
          printf(“%c”, S[sarray[i]+j]);
        }
        printf(“\n”);
      }
  }
  return 0;
}
```

```
/* Graph Theory: Maximum Weighted Bipartite Matching
   Combinatorics: Assignment Problem
   ==================================================================
   Description: Given N workers and N jobs to complete, where each worker
has a
                certain compatibility (weight) to each job, find an assign-
ment
                (perfect matching) of workers to jobs which maximizes the
                compatibility (weight).

   Complexity:  O(n^3), where n is the number of workers or jobs.
   ------------------------------------------------------------------
   Author:      Jason Klaus
   Date:        February 18, 2004
   References:  www.cs.umd.edu/class/fall2003/cmsc651/lec07.ps
   ------------------------------------------------------------------
   Reliability: 3
   Notes:       - 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 could be used on non-negative floating point
                  weights as well.
*/

#include <stdio.h>

/* Maximum number of workers/jobs */
#define MAX_N 100

int W[MAX_N][MAX_N], U[MAX_N], V[MAX_N], Y[MAX_N]; /* <-- weight vari-
ables */
int M[MAX_N], N[MAX_N], P[MAX_N], Q[MAX_N], R[MAX_N], S[MAX_N],
T[MAX_N];

/* Returns the maximum weight, with the perfect matching stored in M. */
int Assign(int n)
{
  int w, y; /* <-- weight variables */
  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;
              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;
              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 variables */
  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]);
    }
  }

  return 0;
}
```

```
/* Number Theory: Generalized Chinese Remaindering
   =================================================================
   Description: 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)) )
   -----------------------------------------------------------------
   Author:      Zac Friggstad
   Date:        Nov 20, 2005
   References:  Elementary Number Theory, Kenneth H. Rosen
   -----------------------------------------------------------------
   Reliability: 0
   Notes:       lcm(m_0, ..., m_(n-1)) should be only
                single precision to avoid overflow
*/

#include <stdio.h>

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, m_tmp;

    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);
    }

    return 0;
}
```

```
/* KMP
   =================================
   Description: Given strings T and P, computes
   the indices of T where P occurs as a substring
   and stores in 'shifts'.  The return integer is the
   number of indices stored in 'shifts'

   Complexity: Linear in the sizes of T & P
   ----------------------------------------
   Author:      Zac Friggstad
   Date:        Jan 27, 2006
   References:  Introduction to Algorithms, CLRS
   ----------------------------------------
   Reliablity: 0
*/

#include <stdio.h>
#include <string.h>

#define MAX_LEN 1000

int pi[MAX_LEN];

void compute_prefix(char *P, int m, int *pi) {
    int k = -1;
    pi[0] = -1;

    for (int q = 1; q < m; ++q) {
        while (k >= 0 && P[k + 1] != P[q]) k = pi[k];
        if (P[k + 1] == P[q]) ++k;
        pi[q] = k;
    }
}

int kmp_match(char *T, char *P, int *shift) {
    int n, m, q = -1, shifts = 0;

    n = strlen(T);
    m = strlen(P);

    compute_prefix(P, m, pi);

    for (int i = 0; i < n; ++i) {
        while (q > -1 && P[q + 1] != T[i]) q = pi[q];

        if (P[q + 1] == T[i]) ++q;
        if (q == m - 1) {
            shift[shifts++] = i - m + 1;
            q = pi[q];
        }
    }
    return shifts;
}

int main() {
    char T[MAX_LEN + 1], P[MAX_LEN + 1];
    int shift[MAX_LEN], shifts;

    while (scanf("%s %s", T, P) != -1) {
        shifts = kmp_match(T, P, shift);
        if (shifts) {
            printf("Pattern occurs with shifts:");
            for (int i = 0; i < shifts; ++i) printf(" %d", shift[i]);
            printf("\n\n");
        }
        else printf("No matches.\n\n");
    }

    return 0;
}
```

```
/* Number Theory: Rational Reconstruction
   =============================================
   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.

   Complexity: O(log m)
   ------------------------------------------
   Author:      Zac Friggstad
   Date:        Nov 30, 2005
   References:  Modern Computer Algebra
                von zur Gathen & Gerhard
   ------------------------------------------
   Reliability: 10109 (Spain)
   Notes: Single precision numbers only.
          2*k*k < m must hold.
*/


#include <stdio.h>
#include <assert.h>

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;
  }
}

int main()
{
  LLI m, g, k, r, t;
  char c;

  scanf("%lld %lld %lld", &m, &g, &k);

  c = (rat_recon(m, g, k, &r, &t) ? 'y' : 'n');

  printf("%c %lld / %lld\n", c, r, t);
  return 0;
}
```

```
/* Min cost max flow * (Edmonds-Karp relabelling + 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.cpp.
 *
 * 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) */

#include <string.h>
#include <limits.h>
using namespace std;

// the maximum number of vertices + 1
#define NN 1024

// adjacency matrix (fill this up)
int cap[NN][NN];

// cost per unit of flow matrix (fill this up)
int cost[NN][NN];

// flow network and adjacency list
int fnet[NN][NN], adj[NN][NN], deg[NN];

// Dijkstra's successor and depth
int par[NN], d[NN];        // par[source] = source;

// Labelling function
int pi[NN];

#define CLR(a, x) memset( a, x, sizeof( a ) )
#define Inf (INT_MAX/2)

// Dijkstra's using non-negative edge weights (cost + potential)
#define Pot(u,v) (d[u] + pi[u] - pi[v])
bool dijkstra( int n, int s, int t )
{
    for( int i = 0; i < n; i++ ) d[i] = Inf, par[i] = -1;
    d[s] = 0;
    par[s] = -n - 1;

    while( 1 )
    {
        // find u with smallest d[u]
        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;

        // relax edge (u,i) or (i,u) for all i;
        par[u] = -par[u] - 1;
        for( int i = 0; i < deg[u]; i++ )
        {
            // try undoing edge v->u
            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;

            // try edge u->v
            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 )
{
    // build the adjacency list
    CLR( deg, 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;

    CLR( fnet, 0 );
    CLR( pi, 0 );
    int flow = fcost = 0;

    // repeatedly, find a cheapest path from s to t
    while( dijkstra( n, s, t ) )
    {
        // get the bottleneck capacity
        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] );

        // update the flow network
        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;
}

//----------------- EXAMPLE USAGE -----------------
#include <iostream>
#include <stdio.h>
```

```
using namespace std;

int main()
{
  int numV;
  //  while ( cin >> numV && numV ) {
  cin >> numV;
    memset( cap, 0, sizeof( cap ) );

    int m, a, b, c, cp;
    int s, t;
    cin >> m;
    cin >> 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
    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;
    int flow = mcmf3( numV, s, t, fcost );
    cout << “flow: “ << flow << endl;
    cout << “cost: “ << fcost << endl;

    return 0;
}
```

---

```
/* Min cost max flow * (Edmonds-Karp relabelling + fast heap 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 mcmf3.cpp FOR DENSE GRAPHS! */

#include <iostream>
using namespace std;

// the maximum number of vertices + 1
#define NN 1024

// adjacency matrix (fill this up)
int cap[NN][NN];

// cost per unit of flow matrix (fill this up)
int cost[NN][NN];

// flow network and adjacency list
int fnet[NN][NN], adj[NN][NN], deg[NN];

// Dijkstra’s predecessor, depth and priority queue
int par[NN], d[NN], q[NN], inq[NN], qs;

// Labelling function
int pi[NN];

#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; }

// Dijkstra’s using non-negative edge weights (cost + potential)
#define Pot(u,v) (d[u] + pi[u] - pi[v])
bool dijkstra( int n, int s, int t )
{
    CLR( d, 0x3F );
    CLR( par, -1 );
    CLR( inq, -1 );
    //for( int i = 0; i < n; i++ ) d[i] = Inf, par[i] = -1;
    d[s] = qs = 0;
    inq[q[qs++] = s] = 0;
    par[s] = n;

    while( qs )
    {
        // get the minimum from q and bubble down
        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[j] ) j++;
            if( d[q[j]] >= d[q[i]] ) break;
            BUBL;
        }

        // relax edge (u,i) or (i,u) for all i;
        for( int k = 0, v = adj[u][k]; k < deg[u]; v = adj[u][++k] )
        {
            // try undoing edge v->u
            if( fnet[v][u] && d[v] > Pot(u,v) - cost[v][u] )
                d[v] = Pot(u,v) - cost[v][par[v] = u];

            // try using edge u->v
            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 )
            {
                // bubble up or decrease key
                if( inq[v] < 0 ) { inq[q[qs] = v] = qs; qs++; }
                for( int i = inq[v], j = i/2, t;
                     d[q[i]] < d[q[j]]; i = j, j = i/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 )
{
    // build the adjacency list
    CLR( deg, 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;

    CLR( fnet, 0 );
    CLR( pi, 0 );
    int flow = fcost = 0;

    // repeatedly, find a cheapest path from s to t
    while( dijkstra( n, s, t ) )
    {
        // get the bottleneck capacity
        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] );

        // update the flow network
        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)
                -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.  Note
                 that if bridge (u,v) is represented in the array,
                 then (v,u) is not.
                -variable bridges is the number of bridges in the graph
   Complexity O(V + E)
   Notes: -index vertices from 0 to n-1
*/

#include <stdio.h>
#include <string.h>

#define MAX_N 200
#define min(a,b) (((a)<(b))?(a):(b))

typedef struct {
  int deg;
  int adj[MAX_N];
} Node;

Node alist[MAX_N];
bool art[MAX_N];
int df_num[MAX_N], low[MAX_N], father[MAX_N], count;
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;

  low[v] = df_num[v] = count++;
  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] = -1;
    father[i] = -1;
  }
  count = bridges = 0;
  search(0, 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”);
  }

  return 0;
}
```