ACM ICPC World Finals 2014 Code Booklet
University of Lethbridge

1 Geometry

```
const double EPS = 1e-8;
bool dEqual(double x,double y) { return fabs(x-y) < EPS; }

struct Point {
  double x, y;
  bool operator==(const Point &p) const { return dEqual(x, p.x) && dEqual(y, p.y); }
  bool operator<(const Point &p) const { return y < p.y || (y == p.y && x < p.x); }
};

Point operator-(Point p,Point q){ p.x -= q.x; p.y -= q.y; return p; }
Point operator+(Point p,Point q){ p.x += q.x; p.y += q.y; return p; }
Point operator*(double r,Point p){ p.x *= r; p.y *= r; return p; }
double operator*(Point p,Point q){ return p.x*q.x + p.y*q.y; }
double len(Point p){ return sqrt(p*p); }
double cross(Point p,Point q){ return p.x*q.y - q.x*p.y; }
Point inv(Point p){ Point q = {-p.y,p.x}; return q; }

enum Orientation {CCW, CW, CNEITHER};

//----------------------------------------------------------------------------
// Colinearity test
bool colinear(Point a, Point b, Point c) { return dEqual(cross(b-a,c-b),0); }

//----------------------------------------------------------------------------
// Orientation test  (When pts are colinear: ccw: a-b-c  cw: c-a-b  neither: a-c-b)
Orientation ccw(Point a, Point b, Point c) { //
  Point d1 = b - a, d2 = c - b;
  if (dEqual(cross(d1,d2),0))
    if (d1.x * d2.x < 0 || d1.y * d2.y < 0)
      return (d1 * d1 >= d2*d2 - EPS) ? CNEITHER : CW;
    else return CCW;
  else return (cross(d1,d2) > 0) ? CCW : CW;
}

//----------------------------------------------------------------------------
// Signed Area of Polygon
double area_polygon(Point p[], int n) {
  double sum = 0.0;
  for (int i = 0; i < n; i++)  sum += cross(p[i],p[(i+1)%n]);
  return sum/2.0;
}

//----------------------------------------------------------------------------
// Convex hull: Contains co-linear points. To remove colinear points:
//   Change ("< -EPS" and "> EPS") to ("< EPS" and "> -EPS")
int convex_hull(Point P[], int n, Point hull[]){
  sort(P,P+n); n = unique(P,P+n) - P;  vector<Point> L,U;
  if(n <= 2) { copy(P,P+n,hull); return n; }
  for(int i=0;i<n;i++){
    while(L.size()>1 && cross(P[i]-L.back(),L[L.size()-2]-P[i]) < -EPS) L.pop_back();
    while(U.size()>1 && cross(P[i]-U.back(),U[U.size()-2]-P[i]) >  EPS) U.pop_back();
    L.push_back(P[i]); U.push_back(P[i]);
  }
  copy(L.begin(),L.end(),hull); copy(U.rbegin()+1,U.rend()-1,hull+L.size());
  return L.size()+U.size()-2;
}

//----------------------------------------------------------------------------
// Point in Polygon Test
const bool BOUNDARY = true;  // is boundary in polygon?
bool point_in_poly(Point poly[], int n, Point p) {
  int i, j, c = 0;
  for (i = 0; i < n; i++)
    if (poly[i] == p || ccw(poly[i], poly[(i+1)%n], p) == CNEITHER) return BOUNDARY;

  for (i = 0, j = n-1; i < n; j = i++)
    if (((poly[i].y <= p.y && p.y < poly[j].y) ||
         (poly[j].y <= p.y && p.y < poly[i].y)) &&
        (p.x < (poly[j].x - poly[i].x) * (p.y - poly[i].y) /
         (poly[j].y - poly[i].y) + poly[i].x))
      c = !c;
  return c;
}

//----------------------------------------------------------------------------
// Computes the distance from "c" to the infinite line defined by "a" and "b"
double dist_line(Point a, Point b, Point c) { return fabs(cross(b-a,a-c)/len(b-a)); }

//----------------------------------------------------------------------------
// Intersection of lines (line segment or infinite line)
//      (1 == 1 intersection pt, 0 == no intersection pts, -1 == infinitely many
int intersect_line(Point a, Point b, Point c, Point d, Point &p,bool segment) {
  double num1 = cross(d-c,a-c), num2 = cross(b-a,a-c),denom = cross(b-a,d-c);
  if (!dEqual(denom, 0)) {
    double r = num1 / denom, s = num2 / denom;
    if (!segment || (0-EPS <= r && r <= 1+EPS && 0-EPS <= s && s <= 1+EPS)) {
      p = a + r*(b-a); return 1;
    } else return 0;
  }
  if(!segment) return dEqual(num1,0) ? -1 : 0; // For infinite lines, this is the end
  if (!dEqual(num1, 0)) return 0;
  if(b < a) swap(a,b); if(d < c) swap(c,d);
  if (a.x == b.x) {
    if (b.y == c.y) { p = b; return 1; }
    if (a.y == d.y) { p = a; return 1; }
    return (b.y < c.y || d.y < a.y) ? 0 : -1;
  } else 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;
  return -1;
}

//----------------------------------------------------------------------------
// Intersect 2 circles: 3 -> infinity, or 0-2 intersection points
// Does not deal with radius of 0 (AKA points)
#define SQR(X) ((X) * (X))
struct Circle{ Point c; double r; };
int intersect_circle_circle(Circle c1,Circle c2,Point& ans1,Point& ans2) {
  if(c1.c == c2.c && dEqual(c1.r,c2.r)) return 3;
  double d = len(c1.c-c2.c);
  if(d > c1.r + c2.r + EPS || d < fabs(c1.r-c2.r) - EPS) return 0;
  double a = (SQR(c1.r) - SQR(c2.r) + SQR(d)) / (2*d);
  double h = sqrt(abs(SQR(c1.r) - SQR(a)));
  Point P = c1.c + a/d*(c2.c-c1.c);
  ans1 = P + h/d*inv(c2.c-c1.c); ans2 = P - h/d*inv(c2.c-c1.c);
  return dEqual(h,0) ? 1 : 2;
}

//----------------------------------------------------------------------------
// Intersect circle and line
// -> # of intersection points, in ans1 (and ans2)
struct Line{  Point a,b;  }; // distinct  points
int intersect_iline_circle(Line l,Circle c, Point& ans1, Point& ans2) {
  Point a = l.a - c.c, b = l.b - c.c; Point d = b - a;
  double dr = d*d, D = cross(a,b); double desc = SQR(c.r)*dr - SQR(D);
  if(dEqual(desc,0)){ ans1 = c.c-D/dr*inv(d); return 1; }
  if(desc < 0) return 0; double sgn = (d.y < -EPS ? -1 : 1);
  Point f = (sgn*sqrt(desc)/dr)*d; d = c.c-D/dr*inv(d);
  ans1 = d + f; ans2 = d - f; return 2;
}

//------------------------------------------------------------------------------
// Circle From Points
bool circle3pt(Point a, Point b, Point c, Point &center, double &r) {
  double g = 2*cross((b-a),(c-b)); if (dEqual(g, 0)) return false; // colinear points
  double e = (b-a)*(b+a)/g, f = (c-a)*(c+a)/g;
  center = inv(f*(b-a) - e*(c-a));
  r = len(a-center);
  return true;
}

//------------------------------------------------------------------------------
// Closest Pair of Points
Point M;
bool left_half(Point p){ return p.x<M.x || (p.x==M.x && p.y>M.y); }
double cp(Point P[],int n,vector<Point>& X,int l,int h){
  if(h - l == 2) return len(P[l]-P[l+1]);
  if(h - l == 3) return min(len(P[l]-P[l+1]),
                            min(len(P[l]-P[l+2]),len(P[l+1]-P[l+2])));
  M = X[(h+l)/2]; int m = stable_partition(P+l,P+h,left_half)-P;
  double d = min(cp(P,n,X,l,m),cp(P,n,X,m,h));
  M.x += d, M.y = LARGE_NUM; int t=stable_partition(P+m,P+h,left_half)-P;
  for(int i=l,j=m;i<m && j<t;i++){ if(P[m].x - P[i].x >= d) continue;
    while(j < t && P[i].y - P[j].y >= d) j++;
    for(int k=j;k<t && P[k].y-P[i].y < d;k++)
      if(len(P[k]-P[i]) < d) d=len(P[k]-P[i]);
  }
  inplace_merge(P+m,P+t,P+h); inplace_merge(P+l,P+m,P+h);
  return d;
}
double closest_pair(Point P[],int n){ // Call this from your program
  sort(P,P+n); if(n == 1) return -1; // Undefined
  Point* u = adjacent_find(P,P+n); if(u != P+n) return 0;
  vector<Point> X(n);      for(int i=0;i<n;i++) X[i]=inv(P[i]);
  sort(X.begin(),X.end()); for(int i=0;i<n;i++) X[i]=-1*inv(X[i]);
  return cp(P,n,X,0,n);
}

//------------------------------------------------------------------------------
// Minimum Enclosing Circle [Expected O(n) if you use the random_shuffle]
// inf needs to be bigger than the largest distance between points
Point tmp_c,pL,pR,mid; double tmp_r,inf=1e12;
bool all_of(Point* first,Point* last,bool (*f)(Point p)){
  for(;first != last;++first) if(!f(*first)) return false;
  return true;
}
bool in_circle(Point p){ return len(p-tmp_c) <= tmp_r + EPS; }
void circle2pt(Point a,Point b,Point& c,double& r){ c=0.5*(a+b); r=len(c-a); }
void minimum_enclosing_circle(Point P[],int N,Point& c,double& r){
  if(N <= 1) { c = P[0]; r = 0; return; } random_shuffle(P,P+N);
  circle2pt(P[0],P[1],c,r);

  for(int i=2;i<N;i++){
    if(len(c-P[i]) <= r + EPS) continue;
    circle2pt(P[0],P[i],c,r);
    for(int j=1;j<i;j++){
      if(len(c-P[j]) <= r + EPS) continue;
      circle2pt(P[i],P[j],mid,r); pL = pR = mid;

      double distL = -inf, distR = -inf;
      for(int k=0;k<j;k++)
        if(circle3pt(P[i],P[j],P[k],c,r)){
          double dist = (ccw(P[i],mid,P[k]) == ccw(P[i],mid,c) ? 1 : -1)*len(mid-c);
          if(ccw(P[i],mid,P[k]) == CCW && dist > distL) { pL = c; distL = dist; }
          if(ccw(P[i],mid,P[k]) ==  CW && dist > distR) { pR = c; distR = dist; }
        }
      if(len(P[i]-pL) > len(P[i]-pR)) swap(pL,pR);
      c=tmp_c=mid; r=tmp_r=len(c-P[i]); if(all_of(P,P+j,in_circle)) continue;
      c=tmp_c=pL;  r=tmp_r=len(c-P[i]); if(all_of(P,P+j,in_circle)) continue;
      c=pR;        r=len(c-P[i]);
    }
  }
}
```

```
const double PI = acos(-1.0), EPS = 1e-8;

struct Vector {
  double x, y, z;

  Vector(double xx = 0, double yy = 0, double zz = 0) : x(xx), y(yy), z(zz) { }
  Vector(const Vector &p1, const Vector &p2)
    : x(p2.x - p1.x), y(p2.y - p1.y), z(p2.z - p1.z) { }
  Vector(const Vector &p1, const Vector &p2, double t)
    : x(p1.x + t*p2.x), y(p1.y + t*p2.y), z(p1.z + t*p2.z) { }
  double norm() const { return sqrt(x*x + y*y + z*z); }
  bool operator==(const Vector&p) const{
    return abs(x - p.x) < EPS && abs(y - p.y) < EPS && abs(z - p.z) < EPS;
  }
};

double dot(Vector p1, Vector p2) { return p1.x * p2.x + p1.y * p2.y + p1.z * p2.z; }
double angle(Vector p1,Vector p2) {  return acos(dot(p1, p2)/p1.norm()/p2.norm()); }
Vector cross(Vector p1, Vector p2) {
  return Vector(p1.y*p2.z-p2.y*p1.z, p2.x*p1.z-p1.x*p2.z, p1.x*p2.y-p2.x*p1.y); }
Vector operator+(Vector p1,Vector p2){ return Vector(p1.x+p2.x,p1.y+p2.y,p1.z+p2.z); }
Vector operator-(Vector p1,Vector p2){ return Vector(p1.x-p2.x,p1.y-p2.y,p1.z-p2.z); }
Vector operator*(double c,Vector v){ return Vector(c*v.x, c*v.y, c*v.z); }

double dist_pt_to_pt(Vector p1,Vector p2) { return Vector(p1, p2).norm(); }

// distance from p to the line segment defined by a and b
double dist_pt_to_segment(Vector p,Vector a,Vector b) {
  Vector u(a, p), v(a, b); double s = dot(u,v) / dot(v,v);
  if (s < 0 || s > 1) return min(dist_pt_to_pt(p, a), dist_pt_to_pt(p, b));
  return dist_pt_to_pt(Vector(a, v, s), p);
}

// distance from p to the infinite line defined by a and b
double dist_pt_to_line(Vector p, Vector a,Vector b) {
  Vector u(a, p), v(a, b); double s = dot(u,v) / dot(v,v);
  return dist_pt_to_pt(Vector(a, v, s), p);
}

// distance from p to the triangle defined by a, b, c
double dist_pt_to_triangle(Vector p, Vector a, Vector b, Vector c) {
  Vector u(a, p), v1(a, b), v2(a, c); Vector normal = cross(v1, v2);
  double s = dot(u, normal) / (normal.norm() * normal.norm());
  Vector proj(p, normal, -s);
  Vector wa(proj, a), wb(proj, b), wc(proj, c);
  double a1 = angle(wa, wb), a2 = angle(wa, wc), a3 = angle(wb, wc);
  if (fabs(a1 + a2 + a3 - 2*PI) < EPS) return dist_pt_to_pt(proj, p);
  return min(dist_pt_to_segment(p, a, b), min(dist_pt_to_segment(p, a, c),
                                               dist_pt_to_segment(p, b, c)));
}

// distance from p to the infinite plane defined by a, b, c
double dist_pt_to_plane(Vector p, Vector a, Vector b, Vector c) {
  Vector u(a, p), v1(a, b), v2(a, c); Vector normal = cross(v1, v2);
  double s = dot(u, normal) / (normal.norm() * normal.norm());
  return dist_pt_to_pt(Vector(p, normal, -s), p);
}

// distance from segment p1->q1 to p2->q2
double dist_segment_to_segment(Vector p1, Vector q1, Vector p2, Vector q2) {
  Vector v1(p1, q1), v2(p2, q2);
  Vector rhs(dot(v1, p2) - dot(v1, p1), dot(v2, p1) - dot(v2, p2));
  double det = v1.norm()*v1.norm()*v2.norm()*v2.norm() - dot(v1, v2)*dot(v1, v2);
  if (det > EPS){
    double t = (rhs.x*v2.norm()*v2.norm() + rhs.y * dot(v1, v2)) / det;
    double s = (v1.norm()*v1.norm()*rhs.y + dot(v1, v2) * rhs.x) / det;
    if (0 <= s && s <= 1 && 0 <= t && t <= 1)
      return dist_pt_to_pt(Vector(p1, v1, t), Vector(p2, v2, s));
  }
  return min(min(dist_pt_to_segment(p1, p2, q2), dist_pt_to_segment(q1, p2, q2)),
             min(dist_pt_to_segment(p2, p1, q1), dist_pt_to_segment(q2, p1, q1)));
}

// distance from infinite lines defined by p1->q1 and p2->q2
double dist_line_to_line(Vector p1, Vector q1, Vector p2, Vector q2) {
  Vector v1(p1, q1), v2(p2, q2);
  Vector rhs(dot(v1, p2) - dot(v1, p1), dot(v2, p1) - dot(v2, p2));
  double det = v1.norm()*v1.norm()*v2.norm()*v2.norm() - dot(v1, v2)*dot(v1, v2);
  if (det < EPS) return dist_pt_to_line(p1, p2, q2);
  double t = (rhs.x*v2.norm()*v2.norm() + rhs.y * dot(v1, v2)) / det;
  double s = (v1.norm()*v1.norm()*rhs.y + dot(v1, v2) * rhs.x) / det;
  return dist_pt_to_pt(Vector(p1, v1, t), Vector(p2, v2, s));
}

// Rotate a point (P) around a line (defined by two points L1 and L2) by theta
//   Note: Rotation is counterclockwise when looking through L2 to L1.
Point rotate(Point P,Point L1,Point L2,double theta){
  double a=L1.x,b=L1.y,c=L1.z, u=(L2-L1).x,v=(L2-L1).y,w=(L2-L1).z;
  double x=P.x,y=P.y,z=P.z,L = sqrt(u*u+v*v+w*w); u /= L, v /= L, w /= L;
  double C=cos(theta),S=sin(theta),D=1-cos(theta),E=u*x+v*y+w*z;

  Point ans;
  ans.x = D*(a*(v*v+w*w) - u*(b*v+c*w-E)) + x*C + S*(b*w-c*v-w*y+v*z);
  ans.y = D*(b*(u*u+w*w) - v*(a*u+c*w-E)) + y*C + S*(c*u-a*w+w*x-u*z);
  ans.z = D*(c*(u*u+v*v) - w*(a*u+b*v-E)) + z*C + S*(a*v-b*u-v*x+u*y);

  return ans;
}

// 3D Convex Hull -- O(n^2)
//  -- To use:
//   vector<Vector> pts;
//   vector<hullFinder::hullFace> hull = hullFinder(pts).findHull();
//  -- Each entry in hull will represent indices of a triangle on the hull (u,v,w)
//  -- Some points may be coplanar
Vector tNorm(Vector a,Vector b,Vector c){ return cross(a,b)+cross(b,c)+cross(c,a); }
const Vector Zero;

class hullFinder {
  const vector<Vector> &pts;
public:
  hullFinder(const vector<Vector> &PTS) : pts(PTS), halfE(pts.size(),-1) {}
  struct hullFace {
    int u, v, w; Vector n;
    hullFace(int U, int V, int W, const Vector &N) : u(U), v(V), w(W), n(N) {}
  };
  vector<hullFinder::hullFace> findHull() {
    vector<hullFace> hull; int n = pts.size(), p3, p4; Vector t; edges.clear();
    if (n < 4) return hull;   // Not enough points  (hull is empty)
    for(p3 = 2  ; (p3 < n) && (t=tNorm(pts[0], pts[1], pts[p3])) == Zero ; p3++) {}
    for(p4=p3+1 ; (p4 < n) && (abs(dot(t, pts[p4] - pts[0])) < EPS)       ; p4++) {}
    if (p4 >= n) return hull; // All points coplanar (hull is empty)
    edges.push_front(hullEdge(0, 1)),setF1(edges.front(),p3),setF2(edges.front(),p3);
    edges.push_front(hullEdge(1,p3)),setF1(edges.front(), 0),setF2(edges.front(), 0);
    edges.push_front(hullEdge(p3,0)),setF1(edges.front(), 1),setF2(edges.front(), 1);
    addPt(p4); for (int i = 2; i < n; ++i) if ((i != p3) && (i != p4)) addPt(i);
    for (list<hullEdge>::iterator e = edges.begin(); e != edges.end(); ++e){
      if((e->u < e->v) && (e->u < e->f1))
        hull.push_back(hullFace(e->u, e->v, e->f1, e->n1));
      else if ((e->v < e->u) && (e->v < e->f2))
        hull.push_back(hullFace(e->v, e->u, e->f2, e->n2));
    }
    return hull; // Good hull
  }
private:
  struct hullEdge {
    int u, v, f1, f2; Vector n1, n2;
    hullEdge(int U, int V) : u(U), v(V), f1(-1), f2(-1) {}
  };
  list<hullEdge> edges; vector<int> halfE;
  void setF1(hullEdge &e,int f1) { e.f1=f1, e.n1=tNorm(pts[e.u],pts[e.v],pts[e.f1]); }
  void setF2(hullEdge &e,int f2) { e.f2=f2, e.n2=tNorm(pts[e.v],pts[e.u],pts[e.f2]); }
  void addPt(int i) {
    for (list<hullEdge>::iterator e = edges.begin(); e != edges.end(); ++e) {
      bool v1 = dot(pts[i] - pts[e->u], e->n1) > EPS;
      bool v2 = dot(pts[i] - pts[e->u], e->n2) > EPS;
      if(v1 && v2) e = --edges.erase(e);
      else if(v1) setF1(*e, i), addCone(e->u, e->v, i);
      else if(v2) setF2(*e, i), addCone(e->v, e->u, i);
    }
  }
  void addCone(int u, int v, int apex) {
    if (halfE[v] != -1){
      edges.push_front(hullEdge(v, apex));
      setF1(edges.front(), u), setF2(edges.front(), halfE[v]);
      halfE[v] = -1;
    } else halfE[v] = u;
    if (halfE[u] != -1){
      edges.push_front(hullEdge(apex, u));
      setF1(edges.front(), v); setF2(edges.front(), halfE[u]);
      halfE[u] = -1;
    } else halfE[u] = v;
  }
};

// Compute the volume of a convex polyhedron (input is an array of triangular faces)
typedef tuple<Vector,Vector,Vector> tvvv;
double volume_polyhedron(vector<tvvv>& p){
  Vector c,p0,p1,p2; double v, volume = 0;
  for(int i=0;i<p.size();i++)
    c = c + get<0>(p[i]) + get<1>(p[i]) + get<2>(p[i]);
  c = 1/(3.0*p.size())*c;
  for(int i=0;i<p.size();i++){
    tie(p0,p1,p2) = p[i], v = dot(p0,cross(p1,p2)) / 6;
    if(dot(cross(p2-p1,p0-p1),c-p0) > 0) volume -= v;
    else volume += v;
  }
  return volume;
}
```

```
// Delauney Triangulation -- O(n^2)
//  -- Triangulation of a set of points so that no point P is inside the circumcircle
//     of any triangle.
//  -- Maximizes the minimum angle of all angles of the triangles in the triangulation
//  -- 'triangles' is a vector of the indices of the vertices of triangles in the
      triangulation
```

```
// Include 3D convex hull code.
typedef tiii tuple<int,int,int>;
void delauney_triangulation(vector<Vector>& pts,vector<tiii>& triangles){
  triangles.clear();
  for(int i=0;i<pts.size();i++) pts[i].z = pts[i].x*pts[i].x + pts[i].y*pts[i].y;
  vector<hullFinder::hullFace> hull = hullFinder(pts).findHull();
  for(int i=0;i<hull.size();i++)
    if(hull[i].n.z < -EPS)
      triangles.push_back(make_tuple(hull[i].u,hull[i].v,hull[i].w));
}
```

```
// Great Circle computations ////////////////////////////////////////////
// lat [-90,90], long [-180,180]
double greatcircle(double lat1, double long1, double lat2, double long2,
                   double radius) {
  lat1 *= PI/180.0; lat2 *= PI/180.0; long1 *= PI/180.0; long2 *= PI/180.0;
  double dlong = long2 - long1, dlat = lat2 - lat1;
  double a = sin(dlat/2)*sin(dlat/2) + cos(lat1)*cos(lat2)*sin(dlong/2)*sin(dlong/2);
  return radius * 2 * atan2(sqrt(a), sqrt(1-a));
}

void longlat2cart(double lat, double lon, double radius,
                  double &x, double &y, double &z) {
  lat *= PI/180.0; lon *= PI/180.0;  x = radius * cos(lat) * cos(lon);
  y = radius * cos(lat) * sin(lon);  z = radius * sin(lat);
}

void cart2longlat(double x, double y, double z,
                  double &lat, double &lon, double &radius) {
  radius = sqrt(x*x + y*y + z*z);
  lat = (PI/2 - acos(z / radius)) * 180.0 / PI;  lon = atan2(y, x) * 180.0 / PI;
}
```

```
double area_heron(double a, double b, double c) { // assumes triangle valid
  return sqrt((a+b+c)*(c-a+b)*(c+a-b)*(a+b-c))/4.0;
}
```

```
typedef tuple<double,int,int> seg;

// (x1,y1) , (x2,y2) are corners of axis-aligned rectangles
struct rectangle{ double x1,y1,x2,y2; };

struct segment_tree{
  int n; const vector<double>& v;  vector<int> pop;  vector<double> len;
  segment_tree(const vector<double>& y) : n(y.size()),v(y),pop(2*n-3),len(2*n-3) {}

  double add(pair<double,double> s,int a){ return add(s,a,0,n-2); }
  double add(const pair<double,double>& s, int a, int lo, int hi){
    int m = (lo+hi)/2 + (lo == hi ? n-2 : 0);
    if(a && (v[lo] < s.second) && (s.first < v[hi+1])){
      if((s.first <= v[lo]) && (v[hi+1] <= s.second)){
        pop[m] += a;
        len[m] = (lo == hi ? 0 : add(s,0,lo,m) + add(s,0,m+1,hi));
      } else len[m] = add(s,a,lo,m) + add(s,a,m+1,hi);
      if(pop[m] > 0) len[m] = v[hi+1] - v[lo];
    }
    return len[m];
  }
};

double area_union_rectangles(vector<rectangle>& R){
  vector<double> y; vector<seg> v;
  for(int i=0;i<R.size();i++){
    if(R[i].x1 == R[i].x2 || R[i].y1 == R[i].y2) continue;
    y.push_back(R[i].y1), y.push_back(R[i].y2);
    if(R[i].y1 > R[i].y2) swap(R[i].y1,R[i].y2);
    v.push_back(seg(min(R[i].x1,R[i].x2),i, 1));
    v.push_back(seg(max(R[i].x1,R[i].x2),i,-1));
  }
  sort(v.begin(),v.end());  sort(y.begin(),y.end());
  y.resize(unique(y.begin(),y.end()) - y.begin());
  segment_tree s(y); double area = 0, amt = 0, last = 0;
  for(int i=0;i<v.size();i++){
    area += amt * (get<0>(v[i]) - last);
    last = get<0>(v[i]); int t = get<1>(v[i]);
    amt = s.add(make_pair(R[t].y1,R[t].y2),get<2>(v[i]));
  }
  return area;
}
```

```
//--------------------------------------------------------------------------
// 2D Integer geometry starts here

// change dEqual, make EPS = 0
struct Point {
  ll x, y;
  // safe ranges fro x and y:
  // SR1 : -10^18<=x,y<=10^18,   SR2 : -10^9<=x,y<=10^9
  // SR3 : -10^6<=x,y<=10^6,     SR4 : -3*10^4<=x,y<=3*10^4

  // operator== and operator<: use double geometry code
};

// +, -, inv: SR1
//  *, cross: SR2
ll len2(const Point &p){ return p*p; } // len2=len*len // SR2

//--------------------------------------------------------------------------
// Colinearity test // SR2
// Orientation test // SR2
// Signed Area of Polygon (*2) // SR2 divided by n, don't divide by 2
//--------------------------------------------------------------------------
// Convex hull:
//   To remove colinear pts: Change ("<0" and ">0") to ("<=0" and ">=0") // SR2
//--------------------------------------------------------------------------
// Point in Polygon Test // SR2

//--------------------------------------------------------------------------
// Squared distance from "c" to the infinite line defined by "a" and "b"
frac dist_line2(Point a, Point b, Point c) // SR4
{ ll cr=cross(b-a,a-c);return make_frac(cr*cr,len2(b-a)); }

//--------------------------------------------------------------------------
// Intersection of lines (line segment or infinite line) // SR3
//   (1 == 1 intersection pt, 0 == no intersection pts, -1 == infinitely many
int intersect_line(Point a, Point b, Point c, Point d,
                   frac &px, frac &py,bool segment) {
  ll num1 = cross(d-c,a-c), num2 = cross(b-a,a-c),denom = cross(b-a,d-c);
  if (denom!=0) {
    if(!segment || (denom<0 && num1<=0 && num1>=denom && num2<=0 && num2>=denom) ||
       (denom>0 && num1>=0 && num1<=denom && num2>=0 && num2<=denom)) {
      px=make_frac(a.x,1)+make_frac(num1,denom)*make_frac((b-a).x,1);
      py=make_frac(a.y,1)+make_frac(num1,denom)*make_frac((b-a).y,1); return 1;
    } else return 0;
  }
  if(!segment) return (num1==0) ? -1 : 0; // For infinite lines, this is the end
  if (num1!=0) return 0;
  if(b < a) swap(a,b); if(d < c) swap(c,d);
  if (a.x == b.x) {
    if (b.y == c.y) { px=make_frac(b.x,1); py=make_frac(b.y,1); return 1; }
    if (a.y == d.y) { px=make_frac(a.x,1); py=make_frac(a.y,1); return 1; }
    return (b.y < c.y || d.y < a.y) ? 0 : -1;
  } else if (b.x == c.x) { px=make_frac(b.x,1); py=make_frac(b.y,1); return 1; }
  else if (a.x == d.x) { px=make_frac(a.x,1); py=make_frac(a.y,1); return 1; }
  else if (b.x < c.x || d.x < a.x) return 0;
  return -1;
}

//--------------------------------------------------------------------------------
// Circle From 3 Points // SR3
bool circle3pt(Point a, Point b, Point c, // r2= r*r to avoid irrational numbers
               frac &centerx, frac & centery, frac &r2) {
  ll g = 2*cross((b-a),(c-b)); if (g==0) return false; // colinear points
  frac e= make_frac((b-a)*(b+a),g), f=make_frac((c-a)*(c+a),g);

  centerx= (f*make_frac((b-a).y,1) - e*make_frac((c-a).y,1)) * make_frac(-1 ,1);
  centery= f*make_frac((b-a).x,1) - e*make_frac((c-a).x,1);

  frac tx=make_frac(a.x,1)-centerx,  ty=make_frac(a.y,1)-centery;
  r2=tx*tx+ty*ty;
  return true;
}
```

2 Number Theory

```
// solve x = a[i] mod m[i] where gcd(m[i],m[j]) | a[i]-a[j]
// x0 in [0, lcm(m's)], x = x0 + t*lcm(m's) for all t.
int cra(int n, int m[], int a[]) {
  int u = a[0], v = m[0], p, q, r, t;
  for (int i = 1; i < n; i++) {
    r = gcd(v, m[i], p, q);  t = v;
    if ((a[i] - u) % r != 0) { }   // no solution!
    v = v/r * m[i];           u = ((a[i]-u)/r * p * t + u) % v;
  }
  if (u < 0) u += v;
  return u;
}
```

```
// Discrete Log Solver -- O(sqrt(p))

ll discrete_log(ll p,ll b,ll n){
  map<ll,ll> M; ll jump = ceil(sqrt(p));
  for(int i=0;i<jump && i<p;i++) M[fast_exp_mod(b,i,p)] = i+1;
  for(int i=0;i<p-1;i+=jump){
    ll x = (n*fast_exp_mod(b,p-i-1,p)) % p;
    if(M.find(x) != M.end()) return (i+M[x]-1) % (p-1);
  }
  return -1;
}
```

```
// Euler Phi
int exp(int b, int n) {
  return (n == 0) ? 1 : b * exp(b, n-1);
}

int phi(int n) {
  int k, res = 1;
  for (k = 0; n % 2 == 0; k++)       n /= 2;
  if (k) res *= exp(2, k-1);
  for (long long p = 3; p*p <= n; p += 2) {
    for (k = 0; n % p == 0; k++)     n /= p;
    if (k) res *= exp(p, k-1) * (p-1);
  }
  if (n > 1) res *= n-1;
  return res;
}
```

```
long long fast_exp_mod(long long b, long long n, long long m) {
  if (n == 0) return 1 % m;
  if (n % 2 == 0) return fast_exp_mod((b*b)%m, n/2, m);
  return (fast_exp_mod(b, n-1, m) * b) % m;
}
```

```
int gcd(int a, int b, int &s, int &t) { // a*s+b*t = g
  if (b==0) { t = 0; s = (a < 0) ? -1 : 1; return (a < 0) ? -a : a;
  } else { int g = gcd(b, a%b, t, s);  t -= a/b*s;  return g; }
}
```

```
// To factor large numbers (x >= 2^40):
//  - Check all primes up to CUBE_ROOT(x) via trial division
//     -- At this point, x has AT MOST 2 unknown prime divisors
//  - Check if remaining value is perfect square (ll(sqrt(x))*ll(sqrt(x)) == x)
//  - Check if remaining value is prime (is_probable_prime(x,20))
//  - Find a prime divisor (using q=pollardRho(x))
//     -- q and x/q are the factors

typedef long long int ll;

// Helper functions...
ll q_mod(ll x,ll m){ return (x >= m) ? x-m : x; }
ll mult_mod(ll x,ll y,ll m){ // Use int128
  ll r = 0;
  while(y){
    if(y % 2) r = q_mod(r+x,m);
    y >>= 1; x = q_mod(x << 1,m);
  } return r;
}

ll F(ll x,ll n,ll c){ x=mult_mod(x,x,n)-c; return (x < 0 ? x + n : x); }

// Returns one (not necessarily prime) factor of n.
//  Works best on semi-primes (where n = p*q for distinct primes).
// Does not work well on perfect powers -- check for those separately.
ll pollardRho(ll n){
  ll i,c,b,x,y,z,g;
  for(g=0,c=3; g % n == 0 ;c++)
    for(g=b=x=y=z=1 ; g==1 ; b *= 2,g = gcd(z,n),z = 1, y = x)
      for(i=0;i<b;i++){ x = F(x,n,c); z = mult_mod(z,abs(x-y),n); }
  return g;
}

// Works for any long long. Do some trial division. Pick an appropriate val:
const val[] = {2,7,61};                               // n <= 2^32
const val[] = {2,13,23,1662803};                      // n <= 10^12
const val[] = {2,3,5,7,11,13,17,19,23,29,31,37}; // n <= 2^64
bool is_prime(ll n){
  if(n < 2) return false;
  for(int i=0;i<NUM_SMALL_PRIMES;i++) if(n % pr[i] == 0) return n == pr[i];

  ll s = __builtin_ctzll(n-1), d = (n-1) >> s;
  for(int i=0;i<NUM_ENTRIES_IN_VAL;i++){
    if(val[i] >= n) break;
    ll x = fast_exp_mod(val[i],d,n); // Use int128 in here
    if(x == 1 || x == n-1) continue;
    for(ll r=1;r<s;r++) if((x = mult_mod(x,x,n)) == n-1) goto nextPr;
    return false;
    nextPr:;
  }
  return true;
}
```

3 Big Integer

```
// Big integer implementation
using namespace std::rel_ops;

typedef long long Digit;
#define BASE 1000000000
#define LOG_BASE 9

#define pbb pair<BigInteger,BigInteger>
#define VEC(v,i) ((0 <= i && i < v.mag.size()) ? v.mag[i] : 0)

bool isZ(Digit x){ return x; }
struct BigInteger {
  BigInteger(Digit n = 0);
  BigInteger(string s);    // no error checking
  int sign;            // +1 = positive, 0 = zero, -1 = negative
  vector<Digit> mag;   // magnitude
  void clear() { sign = 0;   mag.clear(); }
  void normalize(){
    mag.resize(mag.rend()-find_if(mag.rbegin(),mag.rend(),isZ));
    if(mag.empty()) clear(); }

  string toString() const;        // convert to string
  long long toLongLong() const { return strtoll(toString().c_str(),NULL,10); }

  bool isZero() const { return sign == 0; }
  bool operator<(const BigInteger &a) const;
  bool operator==(const BigInteger &a) const { return sign==a.sign && mag==a.mag; }

  BigInteger operator-() const { BigInteger t(*this); t.sign *= -1; return t; }
  friend BigInteger add_sub(BigInteger a,const BigInteger& b,int m);
  friend pbb divide(const BigInteger &a,const BigInteger &b);
  BigInteger &operator+=(const BigInteger &a){ return *this = add_sub(*this,a,1); }
  BigInteger &operator-=(const BigInteger &a){ return *this = add_sub(*this,a,-1); }
  BigInteger &operator*=(const BigInteger &a);
  BigInteger &operator/=(const BigInteger &a){ return *this = divide(*this,a).first; }
  BigInteger &operator%=(const BigInteger &a){ return *this = divide(*this,a).second;}
  // This is (*= BASE^a), not (*= 2^a)
  BigInteger &operator<<=(Digit a){ if(sign)mag.insert(mag.begin(),a,0);return *this;}
  bool sqrt(BigInteger &root) const;
};

BigInteger operator+(BigInteger a, const BigInteger &b) { return a += b; }
BigInteger operator-(BigInteger a, const BigInteger &b) { return a -= b; }
BigInteger operator*(BigInteger a, const BigInteger &b) { return a *= b; }
BigInteger operator/(BigInteger a, const BigInteger &b) { return a /= b; }
BigInteger operator%(BigInteger a, const BigInteger &b) { return a %= b; }
ostream &operator<<(ostream &os, const BigInteger &a){ return (os << a.toString()); }

BigInteger::BigInteger(Digit n){
  if (n == 0){ clear(); return; } sign = (n<0 ? -1 : 1); n *= sign;
  while (n > 0) { mag.push_back(n % BASE); n /= BASE; }
}

BigInteger::BigInteger(string s){
  clear(); sign = 1; if(s[0] == '-'){ sign = -1; s[0] = '0'; }
  int n = s.length(); mag.resize((n+LOG_BASE-1)/LOG_BASE,0); vector<Digit>::
      reverse_iterator p = mag.rbegin();
  for(int i=0;i<n;i++){
    if(i && ((n-i) % 9 == 0)) p++;
    (*p) = (*p)*10 + (s[i] - '0');
  } normalize();
}

string BigInteger::toString() const {
  if(sign == 0) return "0";
  stringstream ss; if(sign == -1) ss << "-"; ss << *mag.rbegin();
  for(int i=mag.size()-2;i>=0;i--) ss << setw(LOG_BASE) << setfill('0') << mag[i];
  return ss.str();
}

bool BigInteger::operator<(const BigInteger &a) const {
  if (sign != a.sign) return sign < a.sign;
  if (sign < 0) return -a < -(*this);
  if (mag.size() != a.mag.size()) return mag.size() < a.mag.size();
  return lexicographical_compare(mag.rbegin(),mag.rend(),a.mag.rbegin(),a.mag.rend());
}

BigInteger add_sub(BigInteger a,const BigInteger& b,int m){
  if(b.sign == 0) return a; if(a.sign == 0) return (m == 1 ? b : -b);
  if(a.sign != b.sign) return add_sub(a,-b,-m);
  if(a.sign == -1) return -add_sub(-a,-b,m);
  if(a < b){ BigInteger x = add_sub(b,a,m); return (m == 1 ? x : -x); }
  Digit bc = 0,lim = a.mag.size();
  for(Digit i=0;i<lim;i++){
    Digit ds = VEC(a,i) + m*VEC(b,i) + m*bc;
    if((m>0 && ds>=BASE) || (m<0 && ds<0)){ a.mag[i] = ds - m*BASE; bc = 1; }
    else{ a.mag[i] = ds; bc = 0; }
  }
  if(bc) a.mag.push_back(1); a.normalize(); return a;
}

BigInteger int_mult(BigInteger b,Digit a){
  if (b.sign == 0 || a == 0){ b.clear(); return b; }
  if (a < 0){ b.sign *= -1; a = -a; }
  Digit carry = 0, n = b.mag.size();
  for (int i = 0; i < n; i++) {
    Digit x = a * b.mag[i] + carry; b.mag[i] = x % BASE; carry = x / BASE; }
  if (carry) b.mag.push_back(carry); return b;
}

BigInteger &BigInteger::operator*=(const BigInteger &a){
  BigInteger t(*this),c;
  if (this == &a) c = a; const BigInteger &b = (this == &a) ? c : a; clear();
  for (int i=0;i<b.mag.size();i++){ *this += int_mult(t,b.mag[i]); t <<= 1; }
  sign *= b.sign; return (*this);
}

pbb divide(const BigInteger &a,const BigInteger &b){
  if(a.sign*b.sign == 0) return make_pair(0,0); // WARNING: x/0 == 0, x%0 == 0
  if(b.sign == -1){ pbb t=divide( a,-b); t.first=-t.first; return t; }
  if(a.sign == -1){ pbb t=divide(-a, b); BigInteger q(t.first),r(t.second);
    q=-q; r=-r; if(r < 0){ r+=b; q-=1; } return make_pair(q,r); }
  if(a < b) return make_pair(0,a);
  BigInteger q,r; q.sign = 1;
  if (b.mag.size() == 1){
    Digit R = 0;
    for (int i=a.mag.size()-1;i>=0;i--){
      Digit t = R * BASE + a.mag[i];
      q.mag.insert(q.mag.begin(), t / b.mag[0]);
      R = t - q.mag[0] * b.mag[0];
    }
    q.normalize(); r = R;
    return make_pair(q,r);
  }

  Digit t,q2,r2,n,m,d;
  r = a;
  n = b.mag.size(), m = a.mag.size() - n, d = BASE / (*b.mag.rbegin() + 1);
  q.mag.resize(m+1); q.sign = 1; r.mag.resize(m+n+1,0); BigInteger v(b);
  r *= d;    v *= d;

  for (int j = m; j >= 0; j--) {
    t=r.mag[j+n]*BASE+r.mag[j+n-1]; q2=t/v.mag[n-1]; r2=t-q2*v.mag[n-1];

#define XXX (q2 == BASE || q2 * v.mag[n-2] > BASE * r2 + r.mag[j+n-2])
    if (XXX){ q2--; r2 += v.mag[n-1];
      if (r2 < BASE && XXX){ q2--; r2 += v.mag[n-1]; } }

    Digit carry = 0, borrow = 0;
    for (int i = 0; i <= n; i++) {
      t = q2 * VEC(v,i) + carry; carry = t / BASE; t %= BASE;
      Digit diff = r.mag[j+i] - t - borrow;
      r.mag[j+i] = diff + BASE*(diff < 0 && i < n); borrow = (diff < 0);
    }

    q.mag[j] = q2;
    if (r.mag[n+j] < 0) {
      q.mag[j]--;
      carry = 0;
      for (int i = 0; i < n; i++) {
        t = r.mag[j+i] + v.mag[i] + carry;
        r.mag[j+i] = t % BASE; carry = t / BASE;
      }
      r.mag[j+n] += carry;
    }
  }
  q.normalize(); r.normalize(); return make_pair(q,r/d);
}

bool BigInteger::sqrt(BigInteger &root) const {
  root.clear(); if (sign == 0) return true;
  BigInteger x, r; int d = mag.size(), root_d = (d+1)/2;
  r.sign = 1, root.sign = 1;

  if(d % 2 == 0) r.mag.push_back(mag[--d]);
  r <<= 1; r.mag[0] = mag[--d];

  for (int k = root_d - 1; k >= 0; k--) {
    x = root * 2;   x <<= 1;
    Digit lo = 0, hi = BASE;
    while (hi - lo > 1) {
      Digit mid = x.mag[0] = (lo + hi) / 2;
      (x*mid <= r ? lo : hi) = mid;
    }
    root <<= 1; root.mag[0] = x.mag[0] = lo; r -= x * lo;
    r <<= 1;  r += (d > 0) ? mag[--d] : 0;
    r <<= 1;  r += (d > 0) ? mag[--d] : 0;
  }
  return r.isZero();
}
```

4 Dynamic Programming

```
int asc_seq(int A[], int n, int S[]) {
  vector<int> last(n+1), pos(n+1), pred(n);

  if (n == 0) return 0;
  int len = 1;                    last[1] = A[pos[1] = 0];
  for (int i = 1; i < n; i++) {
    int j = upper_bound(last.begin()+1, last.begin()+len+1, A[i]) -
      last.begin();      // use lower_bound for strict increasing subsequence
    pred[i] = (j-1 > 0) ? pos[j-1] : -1;
    last[j] = A[pos[j] = i];    len = max(len, j);
  }
  int start = pos[len];
  for (int i = len-1; i >= 0; i--) {  S[i] = A[start];  start = pred[start];  }
  return len;
}
```

```
// Find the longest palindromic substrings (or all)
// Returns the starting index and the length of the palindrome
pair<int,int> longest_palindrome(vector<int> input){
  int a1=-1,a2=-2,a3=-3; // Three DIFFERENT numbers that do NOT appear in your input
  int C,R,n = 2*input.size()+3;   vector<int> v(n,a1), P(n,0);
  v[0] = a2, v[n-1] = a3;
  for(int i=0;i<input.size();i++) v[2*i+2] = input[i];
  for(int i=1;i<n-1;i++){
    for(P[i]=(R>i ? min(R-i,P[2*C-i]) : 0) ; v[i+1+P[i]] == v[i-1-P[i]] ; P[i]++) {}
    if(P[i]+i > R) C = i, R = P[i]+i;
  }
  int loc = max_element(v.begin(),v.end()) - v.begin(); // All ties here are also
  return make_pair((loc-1-v[loc])/2,v[loc]);             // longest palindromes
}
```

5 Graph Theory

```
// Graph layout
//  -- Each problem has its own Edge structure.
// If you see "typedef int Edge;" at the top of an algorithm, change
//    vector<vector<Edge> > nbr; --->  vector<vector<int> > nbr;

struct Graph {
  vector<vector<Edge> > nbr;
  int num_nodes;
  Graph(int n) : nbr(n), num_nodes(n) { }

  // No check for duplicate edges!
  // Add (or remove) any parameters that matter for your problem
  void add_edge_directed(int u, int v, int weight, double cost, ...) {
    Edge e = {v,weight,cost, ...};    nbr[u].push_back(e);
  }
  void add_edge_undirected(int u, int v, int weight, double cost, ...) {
    Edge e1 = {v,weight,cost, ...};   nbr[u].push_back(e1);
    Edge e2 = {u,weight,cost, ...};   nbr[v].push_back(e2);
  }
  // Does not allow for duplicate edges between u and v.
  //     (Note that if "typedef int Edge;", do not write the ".to")
  void add_edge_directed_no_dup(int u, int v, int weight, double cost, ...) {
    for(int i=0;i<nbr[u].size();i++)
      if(nbr[u][i].to == v) {
        // An edge between u and v is already here.
        // Add tie breaking here if necessary (for example, keep the smallest cost).
        nbr[u][i].cost = min(nbr[u][i].cost,cost);
        return;
      }
    Edge e = {v,weight,cost, ...};    nbr[u].push_back(e);
  }
  void add_edge_undirected_no_dup(int u, int v, int weight, double cost, ...) {
    add_edge_directed_no_dup(u,v,weight,cost, ...);
    add_edge_directed_no_dup(v,u,weight,cost, ...);
  }
};

// Get path from (src) to (v). Stored in path[0], .. ,path[k-1]
int get_path(int v, int P[], int path[]) {
  int k = 0;
  path[k++] = v;
  while (P[v] != -1) path[k++] = v = P[v];
  reverse(path,path+k);
  return k;
}
```

```
// Bellman-Ford (Directed and Undirected) -- O(nm)
//   -- May use get_path to obtain the path.

struct Edge{ int to,weight; }; // weight may be any data-type

void bellmanford(const Graph& G, int src, int D[], int P[]){
  int n = G.num_nodes;
  fill_n(D,n,INT_MAX); fill_n(P,n,-1);
  D[src] = 0;
  for (int k = 0; k < n-1; k++)
    for (int v = 0; v < n; v++)
      for (int w = 0; D[v] != INT_MAX && w < G.nbr[v].size(); w++) {
        Edge p = G.nbr[v][w];
        if (D[p.to] == INT_MAX || D[p.to] > D[v] + p.weight) {
          D[p.to] = D[v] + p.weight; P[p.to] = v;
        } else if (D[p.to] == D[v] + p.weight) { } // tie-breaking
      }

  for (int v = 0; v < n; v++) // negative cycle detection
    for (int w = 0; w < G.nbr[v].size(); w++)
      if (D[v] != INT_MAX) {
        Edge p = G.nbr[v][w];
        if (D[p.to] == INT_MAX || D[p.to] > D[v] + p.weight)
          { } // Found a negative cycle
      }
}
```

```
// Biconnected Components (Undirected Only) -- O(n+m)
//   -- Some articulation points may be processed multiple times.

typedef int Edge;

int dfn, dfs[MAX_N], back[MAX_N];
bool root_art(const Graph& G,int v,int k,int child){
  if(child > 1) return true;
  for(int i=k+1;i<G.nbr[v].size();i++)
    if(!dfs[G.nbr[v][i]]) return true;
  return false;
}

void do_dfs(const Graph& G, int v, int pred, stack<pair<int,int> > &e_stack){
  int child = 0;
  dfs[v] = back[v] = ++dfn;
  for (int i = 0; i < G.nbr[v].size(); i++) {
    int w = G.nbr[v][i];
    if (dfs[w] < dfs[v] && w != pred) e_stack.push(make_pair(v,w));
    if (!dfs[w]) {
      do_dfs(G, w, v, e_stack);      child++;

      if (back[w] >= dfs[v]) {                        // new biconnected component
        pair<int,int> e,E = make_pair(v,w);
        do{
          e = e_stack.top(); e_stack.pop();           // e belongs to this component
        } while(e != E);

        if(pred != -1 || root_art(G,v,i,child)){ }   // v is articulation point
      } else back[v] = min(back[v],back[w]);
    } else back[v] = min(back[v],dfs[w]);
  }
}

void bicomp(const Graph& G) {
  stack<pair<int,int> > e_stack;
  dfn = 0; fill_n(dfs,G.num_nodes,0);
  for (int i=0;i<G.num_nodes;i++) // get rid of loop to process only one component
    if (dfs[i] == 0) do_dfs(G, i, -1, e_stack);
}
```

```
// Dijkstra's Algorithm [Sparse Graphs] (Directed and Undirected) -- O((n+m)*log(n+m))
//   -- Edge weight >= 0.  May use get_path to obtain the path.

struct Edge{ int to,weight; }; // weight can be double or other numeric type
typedef vector<Edge>::const_iterator EdgeIter;

void dijkstra(const Graph &G, int src, vector<int> &D, vector<int> &P) {
  typedef pair<int,int> pii;
  int n = G.num_nodes;
  vector<bool> used(n, false);
  priority_queue<pii, vector<pii>,  greater<pii> > fringe;

  D.resize(n);  fill(D.begin(), D.end(), -1);
  P.resize(n);  fill(P.begin(), P.end(), -1);

  D[src] = 0;
  fringe.push(make_pair(0, src));

  while (!fringe.empty()) {
    pii next = fringe.top();    fringe.pop();
    int u = next.second;
    if (used[u]) continue;
    used[u] = true;

    for (EdgeIter it = G.nbr[u].begin(); it != G.nbr[u].end(); ++it) {
      int v = it->to, weight = it->weight + next.first;
      if (used[v]) continue;
      if (D[v] == -1 || weight < D[v]) {
        D[v] = weight;    P[v] = u;    fringe.push(make_pair(D[v], v));
      }
    }
  }
}
```

```
// Eulerian Tour (Undirected or Directed) -- O(mn) [Change to adj list --> O(m+n)]
//  -- Returns one arbitrary Eulerian tour: destroys original graph!
// To run: tour.clear(), then call find_tour on any vertex with a non-zero degree
//
// If there are self loops, make sure graph[u][u] is incremented twice.
//
// FACTS:
// 1. Undirected G has CLOSED Eulerian <--> (G connected) && (every vertex has
//    even degree)
// 2. Directed G has CLOSED Eulerian <--> (G strongly connected) &&
//    (in-degree==out-degree)
// 3. G has an OPEN Eulerian <--> All but two vertices satisfy the right
//    condition above, and adding an edge between them satisfies both conditions.

int graph[MAX_N][MAX_N];

vector<int> tour;
void find_tour(int u,int n){ // n is the number of vertices
  for(int v=0;v<n;v++)
    while(graph[u][v]){
      graph[u][v]--;
      graph[v][u]--;       // this line is only for undirected graphs!!!
      find_tour(v,n);
    }
  tour.push_back(u);
}
```

```
// Floyd's Algorithm with path information (Undirected and Directed) -- O(n^3)
//  -- Length = -1 if no path exists
const int DISCONNECT = -1;

int graph[MAX_N][MAX_N], dist[MAX_N][MAX_N], succ[MAX_N][MAX_N];

void floyd(int n){
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++) {
      dist[i][j] = graph[i][j];
      succ[i][j] = (i == j || graph[i][j] == DISCONNECT) ? -1 : j;
    }

  for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        if (i != k && dist[i][k] != DISCONNECT && dist[k][j] != DISCONNECT) {
          int temp = dist[i][k] + dist[k][j];
          if (dist[i][j] == DISCONNECT || dist[i][j] > temp) {
            dist[i][j] = temp;     succ[i][j] = succ[i][k];
          } else if (dist[i][j] == temp && succ[i][k] < succ[i][j]) {
            // put tie-breaking on paths here: the one above kind of
            // chooses lex smallest path (but ignores the number of
            // vertices in the path!)
            succ[i][j] = succ[i][k];
          }
        }

  for (int i = 0; i < n; i++) dist[i][i] = 0;
}

int extract_path(int u, int v, int path[]) {
  int len = 0;
  if (dist[u][v] == DISCONNECT) return -1;

  path[len++] = u;
  while (u != v) {
    u = succ[u][v];    path[len++] = u;
  }
  return len;
}
```

```
// Hungarian Algorithm (Undirected Only) -- O(n^3)
// Each half of the graph has exactly N vertices (0 to N-1).
// G[i][j] = weight (left_i, right_j), matching[i] = right vertex matched to left_i
//  -- If the set of U and V are different sizes, pad the smaller one with enough
//     nodes to make them equal sizes. Connect these nodes that you have added with
//     EVERY node in the larger one with a weight of 0.
//   Absent edge: DISCONNECT
const int DISCONNECT = INT_MIN;

// Global Variables (Used internally):
int lx[MAX_N],ly[MAX_N],slack[MAX_N],slack_x[MAX_N],pre[MAX_N],revmatch[MAX_N];
bool S[MAX_N], T[MAX_N];

bool locate_path(int& x,int& y,int G[MAX_N][MAX_N],int N,queue<int>& q,bool phase1){
  if(phase1){ x = q.front(); q.pop(); }
  for (y = 0; y < N; y++){
    if(!phase1) x = slack_x[y];
    if (T[y]) continue;
    if((phase1 && G[x][y] == lx[x] + ly[y]) || (!phase1 && slack[y] == 0)){
      if (revmatch[y] == -1) return true;
      T[y] = true;
      if(phase1 || !S[revmatch[y]]){
        q.push(revmatch[y]);
        int tmp;   S[revmatch[y]] = true;  pre[revmatch[y]] = x;
        x = revmatch[y];
        for(int i=0;i<N;i++)
          if(G[x][i] != DISCONNECT && (tmp = lx[x] + ly[i] - G[x][i]) < slack[i])
            slack[i] = tmp , slack_x[i] = x;
      }
    }
  }
  return false;
}

int max_weight_matching(int G[MAX_N][MAX_N], int N, int matching[MAX_N]) {
  fill_n(matching, N, -1);  fill_n(revmatch, N, -1); fill_n(ly, N, 0);
  for(int i=0;i<N;i++) lx[i] = *max_element(G[i], G[i]+N);

  for(int max_match=0;max_match < N;max_match++) {
    queue<int> q;
    fill_n(S, N, false);  fill_n(T, N, false);  fill_n(pre, N, -1);

    int root = find(matching, matching+N, -1) - matching;
    q.push(root);       pre[root] = -2;      S[root] = true;

    fill_n(slack_x, N, root);
    for (int y = 0; y < N; y++)
      slack[y] = (G[root][y]==DISCONNECT) ? INT_MAX : lx[root]+ly[y]-G[root][y];

    int x, y;
    while (true) {
      while(!q.empty()) if(locate_path(x,y,G,N,q,true)) goto path_found;
      int delta = INT_MAX;
      for (y = 0; y < N; y++) if (!T[y]) delta = min(delta,slack[y]);
      for (x = 0; x < N; x++) if (S[x]) lx[x] -= delta;
      for (y = 0; y < N; y++) if (T[y]) ly[y] += delta; else slack[y] -= delta;
      if(locate_path(x,y,G,N,q,false)) goto path_found;
    }
    path_found:  // <-- That is a colon, not a semi-colon
    for (int cx = x, cy = y, ty; cx != -2; cx = pre[cx], cy = ty) {
      ty = matching[cx];   revmatch[cy] = cx;   matching[cx] = cy;
    }
  }

  // return the final answer
  int weight = 0;
  for (int x = 0; x < N; x++) weight += G[x][matching[x]];
  return weight;
}

int min_weight_matching(int G[MAX_N][MAX_N], int N, int matching[MAX_N]) {
  int M = INT_MIN;
  for (int i = 0; i < N; i++)
    M = max(M, *max_element(G[i],G[i]+N));

  int newG[MAX_N][MAX_N];
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      newG[i][j] = (G[i][j] == DISCONNECT) ? DISCONNECT : M - G[i][j];

  return N*M - max_weight_matching(newG, N, matching);
}
```

```
// include RMQ code (Minimum) -- MAX_N must be 2*MAX_NODES
//   call constructLCA once before starting      O(n)
//   call LCA to find lca of vertex u and v      O(log n)
typedef pair<int,int> Type;             // These two lines replace the
const Type oo = make_pair(INT_MIN,-1);  // corresponding  lines in rmq.cc
typedef int Edge;

void preLCA(const Graph& G,int r,int p,Type A[MAX_N],int loc[MAX_N],int d,int& idx){
  for(int i=0;i<G.nbr[r].size();i++){ int w = G.nbr[r][i];
    if(w != p) { A[idx++] = make_pair(d,r); preLCA(G,w,r,A,loc,d+1,idx); } }
  loc[r] = idx; A[idx++] = make_pair(d,r);
}
void constructLCA(const Graph& G,int root,pti M[4*MAX_N],int loc[MAX_N]){
  Type A[MAX_N]; int idx=0; preLCA(G,root,-1,A,loc,0,idx);
  constructRMQ(A,M,idx);
}
int LCA(pti M[4*MAX_N],int loc[MAX_N],int u,int v){
  return getmin(M,min(loc[u],loc[v]),max(loc[u],loc[v])).first.second;
}
```

```
// Unweighted Bipartite Matching (Undirected Only) -- O(m*sqrt(n))
//   -- Your match is stored in "mate". (mate == -1 if there is no match)
//   -- adj is an adjacency list that indexes the other set
//         Ex: U[0].adj[0] == x means there is an edge from U[0] to V[x]

struct Node{ vector<int> adj; int mate,pred; }; // Ignore "pred" -- For internal use.

void add_edge(Node U[],int u_node,Node V[],int v_node){
  U[u_node].adj.push_back(v_node);
  V[v_node].adj.push_back(u_node);
}

// u is the number of elements in U
// v is the number of elements in V
int match(Node U[],int u,Node V[],int v){
  for(int i=0;i<u;i++) U[i].mate = -1;
  for(int i=0;i<v;i++) V[i].mate = -1;

  int numMatches = 0;
  while(true){
    queue<int> q1,q2;
    for(int i=0;i<u;i++) if(U[i].mate == -1) q1.push(i);
    for(int i=0;i<u;i++) U[i].pred = -1;
    for(int i=0;i<v;i++) V[i].pred = -1;

    while(!q1.empty()){
      int x = q1.front(); q1.pop();
      for(int i=0;i<U[x].adj.size();i++){ int w = U[x].adj[i];
        if(V[w].pred != -1) continue;
        if(V[w].mate == -1) V[w].pred = x , q2.push(w);
        else if(V[w].mate != x && U[V[w].mate].pred == -1)
          V[w].pred = x , U[V[w].mate].pred = w , q1.push(V[w].mate);
      }
    }

    if(q2.empty()) break;
    while(!q2.empty()){
      Node* W = V; int i,x = q2.front(); q2.pop();
      for(i = x;i >= 0;W=(W == U ? V : U)) i = W[i].pred;
      if(i == -2) continue; numMatches++;
      for(i = x;i >= 0;){
        int p = V[i].pred;  V[i].pred = -2;  V[i].mate = p;
        U[p].mate = i;      i = U[p].pred;   U[p].pred = -2;
      }
    }
  }
  return numMatches;
}

// Other interesting things: (Don't forget about vertices of degree 0)
//  - Minimum Vertex Cover (size == maximum matching cardinality -- Konig's Thm)
//  - Maximum Independent Set (Complement of minimum vertex cover -- see code)
//  - Minimum Edge Cover (size == max indep. set): Take all edges in the matching +
//     for every node (in U and V) that does not have a mate, include ANY adjacent edge

int vertex_cover(Node U[],int u,Node V[],int v,
                 vector<int>& coverU,vector<int>& coverV){
  coverU.clear(); coverV.clear(); match(U,u,V,v);
  // If you want max independent set, put a ! around both if-statements
  for(int i=0;i<u;i++) if(U[i].pred == -1 && U[i].mate != -1) coverU.push_back(i);
  for(int i=0;i<v;i++) if(V[i].pred != -1) coverV.push_back(i);
  return coverU.size() + coverV.size();
}
```

```
// General Graph Matching
// match[i] = j and match[j] = i if i <-> j is matched.  -1 means no match
// returns size of maximum matching O(|V|^3)
const int MAX_N = 100;

int lca(int match[], int base[], int p[], int a, int b)
{
  bool used[MAX_N] = {false};
  while (true) {
    a = base[a];  used[a] = true;  if (match[a] == -1) break; a = p[match[a]]; }
  while (true) { b = base[b];  if (used[b]) return b;  b = p[match[b]]; }
}

void mark_path(int match[], int base[], bool blossom[], int p[], int v, int b, int c)
{
  for (; base[v] != b; v = p[match[v]]) {
    blossom[base[v]] = blossom[base[match[v]]] = true;   p[v] = c;   c = match[v];  }
}

int find_path(const Graph &G, int match[], int p[], int root)
{
  int n = G.num_nodes;   bool used[MAX_N] = {false};   int base[MAX_N];
  fill(p, p + n, -1);    for (int i = 0; i < n; i++) base[i] = i;

  used[root] = true;     queue<int> q;   q.push(root);
  while (!q.empty()) {
    int v = q.front();   q.pop();
    for (auto to : G.nbr[v]) {
      if (base[v] == base[to] || match[v] == to) continue;
      if (to == root || (match[to] != -1 && p[match[to]] != -1)) {
        int cb = lca(match, base, p, v, to);
        bool blossom[MAX_N] = {false};
        mark_path(match, base, blossom, p, v, cb, to);
        mark_path(match, base, blossom, p, to, cb, v);
        for (int i = 0; i < n; i++)
          if (blossom[base[i]]) {
            base[i] = cb;
            if (!used[i]) { used[i] = true;  q.push(i); } }
      } else if (p[to] == -1) {
        p[to] = v;   if (match[to] == -1) return to;
        to = match[to];   used[to] = true;  q.push(to); } } }
  return -1;
}

int max_matching(const Graph &G, int match[])
{
  int p[MAX_N], n = G.num_nodes;
  fill(match, match + n, -1);
  for (int i = 0; i < n; i++) {
    if (match[i] != -1) continue;
    int v = find_path(G, match, p, i);
    while (v != -1) {
      int pv = p[v];   int ppv = match[pv];
      match[v] = pv;   match[pv] = v;   v = ppv; } }
  return (n - count(match, match + n, -1)) / 2;
}
```

```
// Min Cost Max Flow for Dense graphs
// cap[i][j] is the capacity, cost[i][j] >= 0 is the cost/unit (**directed!**)
// returns maximum flow, fcost = min cost for max flow, fnet contains flow network.
// O(min(n^2 * flow, n^3*fcost)), cap[i][j] = 0 if edge is not there
const int NN = 1024; // the maximum number of vertices + 1
int cap[NN][NN], cost[NN][NN], fnet[NN][NN], adj[NN][NN], deg[NN];
int par[NN], d[NN], pi[NN];
const int Inf = INT_MAX/2;

#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) {
    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 mcmf( int n, int s, int t, int &fcost ){
  fill(deg, deg+NN, 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;

  for (int i = 0; i < NN; i++)  fill(fnet[i], fnet[i]+NN, 0);
  fill(pi, pi+NN, 0);
  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 = min(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;
  }

  for (int u = 0; u < NN; u++)
    for (int v = u; v < NN; v++) {
      int diff = fnet[v][u] - fnet[u][v];
      if (diff > 0) {         fnet[v][u] = diff;   fnet[u][v] = 0; }
      else          {         fnet[u][v] = -diff;  fnet[v][u] = 0; }
    }

  return flow;
}
```

```
// Min Cost Max Flow for Sparse Graph
// O(min((n+m)*log(n+m)*flow, n*(n+m)*log(n+m)*fcost))

struct Edge;
typedef vector<Edge>::iterator EdgeIter;
typedef pair<int,int> pii;
const int oo = INT_MAX / 2;

struct Edge {
  int to, cap, flow, cost;
  bool is_real;
  pair<int,int> part;
  EdgeIter partner;

  int residual() const { return cap - flow; }
};

// Use this instead of G.add_edge_directed in your actual program
void add_edge_with_capacity_directed(Graph& G,int u,int v,int cap,int cost){
  int U = G.nbr[u].size(), V = G.nbr[v].size();
  G.add_edge_directed(u,v,cap,0, cost,true ,make_pair(v,V));
  G.add_edge_directed(v,u,0  ,0,-cost,false,make_pair(u,U));
}

void push_path(Graph& G, int s, int t, const vector<EdgeIter>& path, int flow, int&
     fcost) {
  for (int i = 0; s != t; s = path[i++]->to){
    fcost += flow*path[i]->cost;
    if (path[i]->is_real) {
      path[i]->flow += flow; path[i]->partner->cap += flow;
    } else {
      path[i]->cap -= flow; path[i]->partner->flow -= flow;
    }
  }
}

int augmenting_path(Graph& G, int s, int t, vector<EdgeIter>& path, vector<int>& pi) {
  vector<int> d(G.num_nodes,oo); vector<EdgeIter> pred(G.num_nodes);
  priority_queue<pii,vector<pii>,greater<pii> > pq;
  d[s] = 0; pq.push(make_pair(d[s],s));

  while(!pq.empty()){
    int u = pq.top().second, ud = pq.top().first; pq.pop();
    if(u == t) break; if(d[u] < ud) continue;
    for (EdgeIter it = G.nbr[u].begin(); it != G.nbr[u].end(); ++it) {
      int v = it->to;
      if (it->residual() > 0 && d[v] > d[u] + pi[u] - pi[v] + it->cost) {
        pred[v] = it->partner;  d[v] = d[u] + pi[u] - pi[v] + it->cost;
        pq.push(make_pair(d[v],v));
      }
    }
  }
  if(d[t] == oo) return 0;

  int len = 0 , flow = pred[t]->partner->residual();
  for(int v=t;v!=s;v=pred[v]->to){ path[len++] = pred[v]->partner;
    flow = min(flow,pred[v]->partner->residual());
  }
  reverse(path.begin(),path.begin()+len);
  for(int i=0;i<G.num_nodes;i++) if(pi[i] < oo) pi[i] += d[i];
  return flow;
}

int mcmf(Graph& G, int s, int t, int& fcost) { // note that the graph is modified
  for(int i=0;i<G.num_nodes;i++)
    for(EdgeIter it=G.nbr[i].begin(); it != G.nbr[i].end(); ++it)
      G.nbr[it->part.first][it->part.second].partner = it;

  vector<int> pi(G.num_nodes, 0); vector<EdgeIter> path(G.num_nodes);
  int flow = 0, f; fcost = 0;
  while((f = augmenting_path(G, s, t, path, pi)) > 0){
    push_path(G, s, t, path, f, fcost);    flow += f;
  }
  return flow;
}
```

```
// Minimum Cut (Undirected Only) -- O(n^3)
int min_cut(int G[MAX_N][MAX_N],int n){ // DISCONNECT == 0
  int w[MAX_N],p,j,J,best = -1,A[MAX_N];

  for(n++ ; n-- ; ){
    fill(A,A+n,true), A[p = 0] = false, copy(G[0],G[0]+n,w);
    for(int i=1;i<n;i++){
      for(j=1,J=0;j<n;j++) if(A[j] && (!J || w[j] > w[J])) J = j;
      A[J] = false;
      if(i == n-1){
        if(best < 0 || best > w[J]) best = w[J];
        for(int i=0;i<n;i++) G[i][p] = G[p][i] += G[i][J];
        for(int i=0;i<n-1;i++) G[i][J] = G[J][i]  = G[i][n-1];
        G[J][J] = 0;
      }
      for(p=J,j=1;j<n;j++) if(A[j]) w[j] += G[J][j];
    }
  }
  return best;
}
```

```
// Minimum Spanning Tree (Undirected Only) -- O(m*log(m))
//  -- Do NOT type the Graph structure (not needed)
//  -- Include unionfind code

typedef double Weight;     // can be int instead

struct Edge {
  int v1, v2;          // two endpoints of edge
  Weight w;
  Edge(int i=-1, int j=-1, Weight weight=0)  :  v1(i), v2(j), w(weight) { }
  bool operator<(const Edge& e) const { return w < e.w; }
};

Weight mst(int n, int m, Edge elist[], int index[], int& size) {
  UnionFind uf(n);
  sort(elist, elist+m);

  Weight w = 0;    size = 0;
  for (int i = 0; i < m && size < n-1; i++) {
    if (uf.merge(elist[i].v1, elist[i].v2)) {
      index[size++] = i;   w += elist[i].w;
    }
  }
  return w;
}
```

```
// Network Flow (Directed and Undirected) -- O(fm) where f = max flow
// To recover flow on an edge, it's in the flow field provided is_real == true.
// Note: if you have an undirected network. simply call add_edge twice
// with an edge in both directions (same capacity).  Note that 4 edges
// will be added (2 real edges and 2 residual edges).  To discover the
// actual flow between two vertices u and v, add up the flow of all
// real edges from u to v and subtract all the flow of real edges from
// v to u.

struct Edge;
typedef vector<Edge>::iterator EdgeIter;

struct Edge {
  int to, cap, flow;
  bool is_real;
  pair<int,int> part;
  EdgeIter partner;

  int residual() const { return cap - flow; }
};

// Use this instead of G.add_edge_directed in your actual program
void add_edge_with_capacity_directed(Graph& G,int u,int v,int cap){
  int U = G.nbr[u].size(), V = G.nbr[v].size();
  G.add_edge_directed(u,v,cap,0,true ,make_pair(v,V));
  G.add_edge_directed(v,u,0  ,0,false,make_pair(u,U));
}

void push_path(Graph& G, int s, int t, const vector<EdgeIter>& path, int flow) {
  for (int i = 0; s != t; s = path[i++]->to)
    if (path[i]->is_real) {
      path[i]->flow += flow;    path[i]->partner->cap += flow;
    } else {
      path[i]->cap -= flow;     path[i]->partner->flow -= flow;
    }
}

int augmenting_path(Graph& G, int s, int t, vector<EdgeIter>& path,
                    vector<bool>& visited, int step = 0) {
  if (s == t) return -1;  visited[s] = true;
  for (EdgeIter it = G.nbr[s].begin(); it != G.nbr[s].end(); ++it) {
    int v = it->to;
    if (it->residual() > 0 && !visited[v]) {
      path[step] = it;
      int flow = augmenting_path(G, v, t, path, visited, step+1);
      if (flow == -1)    return it->residual();
      else if (flow > 0) return min(flow, it->residual());
    }
  }
  return 0;
}

int network_flow(Graph& G, int s, int t) { // note that the graph is modified
  for(int i=0;i<G.num_nodes;i++)
    for(EdgeIter it=G.nbr[i].begin(); it != G.nbr[i].end(); ++it)
      G.nbr[it->part.first][it->part.second].partner = it;

  vector<EdgeIter> path(G.num_nodes);
  int flow = 0, f;
  do {
    vector<bool> visited(G.num_nodes, false);
    if ((f = augmenting_path(G, s, t, path, visited)) > 0) {
      push_path(G, s, t, path, f);    flow += f;
    }
  } while (f > 0);
  return flow;
}
```

```
// Network flow (Directed and Undirected) -- O(n^3)
// returns max flow.  Look for positive entries in flow array for the flow.

void push(int graph[MAX_N][MAX_N], int flow[MAX_N][MAX_N],
          int e[], int u, int v) {
  int cf = graph[u][v] - flow[u][v],   d = (e[u] < cf) ? e[u] : cf;
  flow[u][v] += d;      flow[v][u] = -flow[u][v];
  e[u] -= d;            e[v] += d;
}

void relabel(int graph[MAX_N][MAX_N], int flow[MAX_N][MAX_N],
             int n, int h[], int u) {
  h[u] = -1;
  for (int v = 0; v < n; v++)
    if (graph[u][v] - flow[u][v] > 0 && (h[u] == -1 || 1 + h[v] < h[u]))
      h[u] = 1 + h[v];
}

void discharge(int graph[MAX_N][MAX_N], int flow[MAX_N][MAX_N],
               int n, int e[], int h[], list<int>& NU,
               list<int>::iterator &current, int u) {
  while (e[u] > 0)
    if (current == NU.end()) {
      relabel(graph, flow, n, h, u);
      current = NU.begin();
    } else {
      int v = *current;
      if (graph[u][v] - flow[u][v] > 0 && h[u] == h[v] + 1)
        push(graph, flow, e, u, v);
      else ++current;
    }
}

int network_flow(int graph[MAX_N][MAX_N], int flow[MAX_N][MAX_N],
                 int n, int s, int t) {
  int e[MAX_N], h[MAX_N], u, v, oh;
  list<int> N[MAX_N], L;
  list<int>::iterator current[MAX_N], p;

  for (u = 0; u < n; u++) h[u] = e[u] = 0;
  for (u = 0; u < n; u++)
    for (v = 0; v < n; v++) {
      flow[u][v] = 0;
      if (graph[u][v] > 0 || graph[v][u] > 0) N[u].push_front(v);
    }

  h[s] = n;
  for (u = 0; u < n; u++) {
    if (graph[s][u] > 0) {
      e[u] = flow[s][u] = graph[s][u];
      e[s] += flow[u][s] = -graph[s][u];
    }
    if (u != s && u != t) L.push_front(u);
    current[u] = N[u].begin();
  }

  for (p = L.begin(); p != L.end(); ++p) {
    u = *p;         oh = h[u];
    discharge(graph, flow, n, e, h, N[u], current[u], u);
    if (h[u] > oh) {
      L.erase(p);    L.push_front(u);    p = L.begin();
    }
  }

  int maxflow = 0;
  for (u = 0; u < n; u++)
    if (flow[s][u] > 0) maxflow += flow[s][u];
  return maxflow;
}
```

```
// Strongly Connected Components (Directed Only) -- O(n+m)
//  -- Each vertex's component number is stored in comp[].
//  -- The components are in REVERSE topological order
//  -- Also can construct a DAG of connected components

typedef int Edge;

int po[MAX_N],comp[MAX_N],num_scc,dfn;

void DFS(const Graph& G,int v,stack<int>& P,stack<int>& S){
  po[v] = dfn++;
  S.push(v);  P.push(v);
  for(int i=0;i<G.nbr[v].size();i++){
    int w = G.nbr[v][i];
    if(po[w] == -1) DFS(G,w,P,S);
    else if(comp[w] == -1)
      while(!P.empty() && (po[P.top()] > po[w]))
        P.pop();
  }
  if(!P.empty() && P.top() == v){
    while(!S.empty()){
      int t = S.top();       S.pop();
      comp[t] = num_scc;
      if(t == v) break;
    }
    P.pop();    num_scc++;
  }
}

int SCC(const Graph& G){
  num_scc = dfn = 0;
  stack<int> P,S;
  fill(po,po+G.num_nodes,-1);
  fill(comp,comp+G.num_nodes,-1);
  for(int i=0;i<G.num_nodes;i++)
    if(po[i] == -1) DFS(G,i,P,S);
  return num_scc;
}

// Make sure you call SCC first
Graph get_DAG(const Graph& G){
  Graph G_scc(num_scc);
  for(int i=0;i<G.num_nodes;i++){
    for(int j=0;j<G.nbr[i].size();j++){
      int w = G.nbr[i][j];
      if(comp[i] != comp[w])
        G_scc.add_edge_directed_no_dup(comp[i],comp[w]);
    }
  }
  return G_scc;
}
```

```
// 2SAT solver: returns T/F whether it is satisfiable -- O(n+m)
//   - allocate 2*n nodes in graph where n is the number of variables
//   - use NOT() to negate a variable (works on negated ones too!)
//   - ALWAYS use VAR() to talk about the non-negated version of the var i
//   - use add_clause to add a clause
//   - one possible satisfying assignment is returned in val[], if
//     it exists
//   - To FORCE i to be true:  add_clause(G,VAR(i),VAR(i));
//   - To implement XOR -- say (i XOR j) :
//       add_clause(G,VAR(i),VAR(j)); add_clause(G,NOT(VAR(i)),NOT(VAR(j)));
//     NOTE: val[] is indexed by i for var i, not by VAR(i)!!!

// copy SCC code except get_DAG
int VAR(int i) { return 2*i; }
int NOT(int i) { return i ^ 1; }

void add_clause(Graph &G, int v, int w) { // adds (v || w)
  if (v == NOT(w)) return;
  G.add_edge_directed(NOT(v), w);
  G.add_edge_directed(NOT(w), v);
}

bool twoSAT(const Graph &G, bool val[]) {   // assumes graph is built
  SCC(G);
  for (int i = 0; i < G.num_nodes; i += 2) {
    if (comp[i] == comp[i+1]) return false;
    val[i/2] = (comp[i] < comp[i+1]);
  }
  return true;
}
```

```
// Topological sort (Directed Only) -- O(n+m)
bool topological_sort(const Graph &G, vector<int> &order) {
  vector<int> indeg(G.num_nodes, 0);
  for (int i = 0; i < G.num_nodes; i++)
    for (int j = 0; j < G.nbr[i].size(); j++)
      indeg[G.nbr[i][j]]++;

  queue<int> q;  // use priority queue if you want tie-breaking by lex ordering
  for (int i = 0; i < G.num_nodes; i++)
    if (indeg[i] == 0) q.push(i);

  order.clear();
  while (!q.empty()) {
    int v = q.front();    q.pop();
    order.push_back(v);
    for (int i = 0; i < G.nbr[v].size(); i++)
      if (--indeg[G.nbr[v][i]] == 0) q.push(G.nbr[v][i]);
  }
  return order.size() == G.num_nodes;
}
```

```
// NOTE: For trees only!

// Returns a node that is the furthest from u -- O(n)
int furthest(const Graph& G,int u,int& max_depth,int par[],int p=-1,int d=0){
  if(d == 0 || d > max_depth) max_depth = d;
  int D,v,ans = u;  par[u] = p;
  for(int i=0;i<G.nbr.size();i++){
    if(p == G.nbr[i]) continue;
    D = max_depth; v = furthest(G,G.nbr[i],max_depth,u,d+1);
    if(max_depth > D) ans = v;
  }
  return ans;
}

// The eccentricity of u is the distance to the furthest away node from u -- O(n)
int eccentricity(const Graph& G,int u){
  int max_d,p[MAX_N]; furthest(G,u,max_d,p); return max_d;
}

// The diameter of G is the maximum distance between two nodes -- O(n)
int diam(const Graph& G){
  int max_d,p[MAX_N]; furthest(G,furthest(G,0,max_d,p),max_d,p); return max_d;
}

// The center of G is/are the node(s) with minimum eccentricity -- O(n)
//    (.second == -1 if there is only one center)
pii center(const Graph& G){
  int max_d,v,p[MAX_N]; v = furthest(G,furthest(G,0,max_d,p),max_d,p);
  for(int i=0;i<max_d/2;i++,v = p[v]);
  return make_pair(v,(max_d % 2 ? p[v] : -1));
}
```

6 Linear Algebra

```
// System of linear diophantine equations   A*x = b
// Returns dim(null space), or -1 if there is no solution.
// xp: a particular solution
// hom_basis: an n x n matrix whose first dim columns form a basis of the nullspace.
// All solutions are obtained by adding integer multiples the basis elements to xp.
```

```
#define MAX_N 50
#define MAX_M 50
int triangulate(int A[MAX_N+1][MAX_M+MAX_N+1], int m, int n, int cols) {
  div_t d;
  int ri = 0, ci = 0;
  while (ri < m && ci < cols) {
    int pi = -1;
    for (int i = ri; i < m; i++)
      if (A[i][ci] && (pi == -1 || abs(A[i][ci]) < abs(A[pi][ci]))) pi = i;
    if (pi == -1) ci++;
    else {
      int k = 0;
      for (int i = ri; i < m; i++)
        if (i != pi) {
          d = div(A[i][ci], A[pi][ci]);
          if (d.quot) {
            k++;
            for (int j = ci; j < n; j++) A[i][j] -= d.quot*A[pi][j]; } }
      if (!k) {
        for (int i = ci; i < n && ri != pi; i++)  swap(A[ri][i], A[pi][i]);
        ri++;   ci++; } } }
  return ri;
}

int diophantine_linsolve(int A[MAX_M][MAX_N], int b[MAX_M], int m, int n,
                         int xp[MAX_N], int hom_basis[MAX_N][MAX_N]) {
  int mat[MAX_N+1][MAX_M+MAX_N+1], i, j, rank, d;

  for (i = 0; i < m; i++) mat[0][i] = -b[i];
  for (i = 0; i < m; i++) for (j = 0; j < n; j++) mat[j+1][i] = A[i][j];

  for (i = 0; i < n+1; i++) for (j = 0; j < n+1; j++) mat[i][j+m] = (i == j);

  rank = triangulate(mat, n+1, m+n+1, m+1);
  d = mat[rank-1][m];
  if (d != 1 && d != -1) return -1;    // no integer solutions

  for (i = 0; i < m; i++)
    if (mat[rank-1][i]) return -1;     // inconsistent system

  for (i = 0; i < n; i++) {
    xp[i] = d*mat[rank-1][m+1+i];
    for (j = 0; j < n+1-rank; j++) hom_basis[i][j] = mat[rank+j][m+1+i];
  }
  return n+1-rank;
}
```

```
// solves Ax = b.  Returns det...solution is x_star[i]/det
// A and b may be modified!
int fflinsolve(int A[MAX_N][MAX_N], int b[], int x_star[], int n) {
  int k_c, k_r, pivot, sign = 1, d = 1;
  for (k_c = k_r = 0; k_c < n; k_c++) {
    for (pivot = k_r; pivot < n && !A[pivot][k_r]; pivot++) ;
    if (pivot < n) {
      if (pivot != k_r) {
        for (int j = k_c; j < n; j++) swap(A[pivot][j], A[k_r][j]);
        swap(b[pivot], b[k_r]);     sign *= -1;
      }

      for (int i = k_r+1; i < n; i++) {
        for (int j = k_c+1; j < n; j++)
          A[i][j] = (A[k_r][k_c]*A[i][j]-A[i][k_c]*A[k_r][j])/d;
        b[i] = (A[k_r][k_c]*b[i]-A[i][k_c]*b[k_r])/d;
        A[i][k_c] = 0;
      }
      if (d) d = A[k_r][k_c];
      k_r++;
    } else d = 0;
  }
  if (!d) {
    for (int k = k_r; k < n; k++) if (b[k]) return 0;   // inconsistent system
    return 0;                                           // multiple solutions
  }
  for (int k = n-1; k >= 0; k--) {
    x_star[k] = sign*d*b[k];
    for (int j = k+1; j < n; j++) x_star[k] -= A[k][j]*x_star[j];
    x_star[k] /= A[k][k];
  }
  return sign*d;
}
```

```
// Solves Ax = b in floating-point
// - first call LU_decomp on A (returns determinant)
// - then use LU_solve on A, pivot, b to find solution.

double LU_decomp(double A[MAX_N][MAX_N], int n, int pivot[MAX_N]) {
  double s[MAX_N], c, t, det = 1.0;

  for (int i = 0; i < n; i++) {
    s[i] = 0.0;
    for (int j = 0; j < n; j++) s[i] = max(s[i], fabs(A[i][j]));
    if (s[i] < EPS) return 0; // Singular
  }

  for (int k = 0; k < n; k++){
    c = fabs(A[k][k]/s[k]), pivot[k] = k;
    for (int i = k+1; i < n; i++)
      if ((t = fabs(A[i][k]/s[i])) > c) { c = t; pivot[k] = i; }
    if (c < EPS) return 0; // Singular

    if (k != pivot[k]) {
      det *= -1.0;
      swap_ranges(A[k]+k,A[k]+n,A[pivot[k]]+k);
      swap(s[k],s[pivot[k]]);
    }

    for (int i = k+1; i < n; i++) {
      A[i][k] /= A[k][k];
      for (int j = k+1; j < n; j++) A[i][j] -= A[i][k] * A[k][j];
    }
    det *= A[k][k];
  }
  return det;
}

void LU_solve(double A[MAX_N][MAX_N], int n, int pivot[], double b[], double x[]) {
  copy(b, b+n, x);
  for (int k = 0; k < n-1; k++) {
    if (k != pivot[k]) swap(x[k], x[pivot[k]]);
    for (int i = k+1; i < n; i++) x[i] -= A[i][k] * x[k];
  }

  for (int i = n-1; i >= 0; i--) {
    for (int j = i+1; j< n; j++) x[i] -= A[i][j] * x[j];
    x[i] /= A[i][i];
  }
}
```

7 Data Structures

```
class FenwickTree{  // All entries must be >= 0 even after decrement
public:             // Every function is O(log n)
  FenwickTree(int n) : N(n), iBM(1), tree(n,0) {
    while (iBM < N) iBM *= 2;
  }

  // inc/dec the entry at position idx by val
  void incEntry(int idx, int val) {
    do tree[idx] += val; while(idx && (idx += (idx & (-idx))) < N);
  }

  // return the cumulative sum val[0] + val[1] + ... + val[idx]
  int cumulativeSum(int idx) const {
    int sum = tree[0];
    for( ; idx > 0 ; idx &= idx-1) sum += tree[idx];
    return sum;
  }

  // return the entry indexed by idx
  int getEntry(int idx) const {
    int val = tree[idx], par = idx & (idx-1);
    if (idx--) for( ; par != idx ; idx &= idx-1) val -= tree[idx];
    return val;
  }

  // return the largest index such that the cumulative frequency is
  // what is given, or -1 if it is not found
  int getIndex(int sum) const {
    if ((sum -= tree[0]) < 0) return -1;
    int idx = 0;
    for(int bM = iBM ; bM != 0 && idx < N-1 ; bM >>= 1)
      if (sum >= tree[idx+bM]) sum -= tree[idx += bM];
    return (sum != 0) ? -1 : min(N-1,idx);
  }

private:
  int N, iBM; vector<int> tree;
};

// You can extend this to n-D if you want
class FenwickTree2D{  // All entries must be >= 0 even after decrement
public:               // Every function is O(log^2(n))
  FenwickTree2D(int m,int n) : M(m),N(n),tree(m,vector<int>(n,0)) {} // Array is m x n

  // inc/dec the entry at (i,j) by val
  void incEntry(int i, int j, int val) {
    do{
      int idx = j;
      do tree[i][idx] += val; while(idx && (idx += (idx & (-idx))) < M);
    } while(x && (x += (x & (-x))) < N);
  }

  // return the sum tree[0][0] + ... + tree[i][j]
  int cumSum(int i,int j) const {
    int sum = tree[0][0];
    for( ; i > 0 ; i &= i-1 )
      for(int idx=j ; idx > 0 ; idx &= idx-1) sum += tree[i][idx];
    return sum;
  }

private:
  int M,N; vector<vector<int> > tree;
};
```

```
// Note: Applying operations to reduced fractions should yield a reduced answer
//       Make sure you reduce the fraction when you store it into the structure.
//  EXCEPT: 0 may be 0/x until reduce is called (then changed to 0/1)
typedef long long ll;
struct frac{  ll num,den; };

frac make_frac(ll n,ll d){ frac f; f.num = n,f.den = d; return f; }

frac reduce(frac a){
  if(a.num == 0) return make_frac(0,1); if(a.den < 0) { a.num *= -1; a.den *= -1; }
  ll g = gcd(a.num,a.den); return make_frac(a.num/g,a.den/g);
}
frac recip(frac a){ return make_frac(a.den,a.num); }

frac operator+(frac a,frac b){
  ll g = gcd(a.den,b.den);
  return reduce(make_frac(a.num*(b.den/g) + b.num*(a.den/g), (a.den/g)*b.den));
}
frac operator-(frac a,frac b){ return a + make_frac(-b.num,b.den); }
frac operator*(frac a,frac b){
  ll g1 = gcd(a.num,b.den), g2 = gcd(a.den,b.num);
  return make_frac((a.num / g1) * (b.num / g2),(a.den / g2) * (b.den / g1));
}
frac operator/(frac a,frac b){ return a * recip(b); } // Watch division by 0

bool operator==(frac a,frac b){
  a=reduce(a); b=reduce(b);
  return a.num==b.num && a.den==b.den;
}

// Choose one. First one may overflow. Second one has rounding errors.
bool operator<(frac a,frac b){ return (a.num*b.den) < (b.num*a.den); }
bool operator<(frac a,frac b){ return !(a==b) && a.num/1.0/a.den < b.num/1.0/b.den);}
```

```
struct UnionFind
{
  vector<int> uf;
  UnionFind(int n) : uf(n) {
    for (int i = 0; i < n; i++) uf[i] = i;
  }

  int find(int x) {
    return (uf[x] == x) ? x : (uf[x] = find(uf[x]));
  }

  bool merge(int x, int y) {
    int res1 = find(x), res2 = find(y);
    if (res1 == res2) return false;
    uf[res2] = res1;
    return true;
  }
};
```

```
// Date:    dates from Jan 1, 1753 to after
using namespace std;
using namespace std::rel_ops;

struct Date {
  int yyyy, mm, dd;
  static int const BASE_YEAR = 1753;
  enum dayName {SUN,MON,TUE,WED,THU,FRI,SAT};

  static bool validDate(int yr, int mon, int day) {
    return yr >= BASE_YEAR && mon >= 1 && mon <= 12 && day > 0 && day <= daysIn(mon,
         yr);
  }
  bool isValid() const { return validDate(yyyy, mm, dd); }

  Date(int yr = 1970, int mon = 1, int day = 1)   // assume valid date
    : yyyy(yr), mm(mon), dd(day) { }

  dayName dayOfWeek() const {
    int a = (14 - mm) / 12, y = yyyy - a, m = mm + 12 * a - 2;
    return (dayName)((dd + y + y/4 - y/100 + y/400 + 31 * m / 12) % 7);
  }

  bool operator==(const Date &d) const {
    return dd == d.dd && mm == d.mm && yyyy == d.yyyy;
  }

  bool operator<(const Date &d) const {
    return yyyy < d.yyyy || (yyyy == d.yyyy && mm < d.mm) ||
      (yyyy == d.yyyy && mm == d.mm && dd < d.dd);
  }

  static bool leapYear(int y) {
    return (y % 400 ==0 || (y % 4 == 0 && y % 100 != 0));
  }

  static int daysIn(int m, int y) {
    switch (m) {
    case 4:  case 6:  case 9: case 11: return 30;
    case 2:  return leapYear(y) ? 29 : 28;
    default: return 31;
    }
  }

  // increment by day, month, or year, can use negative
  // if result is invalid date, the result is rounded down to the last valid date
  void addDay(int n = 1) {    // about n/30 iterations
    dd += n;
    while (dd > daysIn(mm,yyyy)) {
      dd -= daysIn(mm,yyyy);
      if (++mm > 12) { mm = 1;   yyyy++;    }
    }

    while (dd < 1) {
      if (--mm < 1) {  mm = 12;  yyyy--;    }
      dd += daysIn(mm,yyyy);
    }
  }

  void addMonth(int n = 1) {   // about n/12 iterations
    mm += n;
    while (mm > 12) {  mm -= 12;   yyyy++;  }
    while (mm < 1)  {  mm += 12;   yyyy--;  }
    if (dd > daysIn(mm,yyyy)) dd = daysIn(mm,yyyy);
  }

  void addYear(int n = 1) {
    yyyy += n;
    if (!leapYear(yyyy) && mm == 2 && dd == 29) dd = 28;
  }

  int daysFromStart() const {  // number of days since 1753/01/01, incl. current date
    int c = 0;
    Date d(BASE_YEAR, 1, 1);
    Date d2(d);

    d2.addYear(1);
    while (d2 < *this) {
      c += leapYear(d.yyyy) ? 366 : 365;
      d = d2;    d2.addYear(1);
    }

    d2 = d;      d2.addMonth(1);
    while (d2 < *this) {
      c += daysIn(d.mm, d.yyyy);
      d = d2;    d2.addMonth(1);
    }
    while (d <= *this) { d.addDay();  c++; }
    return c;
  }
};
```

```
// RMQ:
//   call constructRMQ to get data structure M                   O(n)
//   call getmax to get the maximum from [a..b] inclusive        O(log n)
//     returns a pair of the form (maximum value,index of maximum value)
//   call update to change a value in the array                  O(log n)
typedef int Type;            // Change this as needed
const Type oo = INT_MAX;     // Change to INT_MIN for min.

typedef pair<Type,int> pti;
const pti p_oo = make_pair(oo,-1);
int size;

void constructRMQ(Type A[MAX_N], pti M[4*MAX_N], int n) {
  size = 1; while(size < n) size <<= 1;
  for (int i=0; i < size; i++) M[size-1+i] = (i < n ? make_pair(A[i],i) : p_oo);
  for (int i=size-2; i>=0; i--) M[i] = max(M[2*i+1],M[2*i+2]);
}

pti getmax(pti M[4*MAX_N], int a, int b, int st=0,int en=size,int ind=0) {
  if (a > b || a >= en || b < st)         return p_oo;
  if ((a <= st && en <= b) || st+1 >= en) return M[ind];
  int mid = (st+en)/2;
  return max(getmax(M,a,b,st,mid,2*ind+1), getmax(M,a,b,mid,en,2*ind+2));
}

void update(Type A[MAX_N], pti M[4*MAX_N], int i, Type val){
  A[i] = val; M[i += size-1] = make_pair(val,i);
  while((i = (i-1)/2)) M[i] = max(M[2*i+1],M[2*i+2]);
}
```

```
// Given a set of linear lines (non-vertical of the form y=mx+b):
//   gives you the minimum value at any given x-value.
// Look at lines with comments like this //// for max_value

typedef long long ll;
typedef long double ld;

const ll oo = LONG_LONG_MAX; // Infinity
bool XXX = true;
struct Line{
  ld m,b; bool d;       mutable ld xL,xR;
  Line(ld x_)             : m(0) , b(0) , d(false), xL(x_), xR(x_) { }
  Line(ld m_,ld b_)       : m(m_), b(b_), d(false), xL(0) , xR(0)  { }
  Line(ld x_,ld m_,ld b_) : m(m_), b(b_), d(true) , xL(x_), xR(x_) { }
  bool operator<(const Line& L) const {
    if(XXX) return (xR != L.xR ? xR > L.xR : xL > L.xL);
    return (m != L.m ? m < L.m : b < L.b);
  }
};

bool cw(const Line& L1,const Line& L2,const Line& L3){
  complex<ld> a(L1.m,L1.b), b(L2.m,L2.b), c(L3.m,L3.b);
  return (L1.d || L2.d || L3.d || imag(conj(b-a)*(c-b)) < 0);
}

ld intersect(const Line& L1,const Line& L2){ return (L1.b-L2.b) / (L1.m-L2.m); }

// Completely general (no assumptions)
struct Hull{ // add_line = O(log n)   min_value = O(log n)
  set<Line> H; set<Line>::iterator it,itL,itR;
  Hull() { H.insert(Line(oo,-oo,-oo)); H.insert(Line(-oo,oo,-oo)); }

  void add_line(ld m,ld b){
    Line L(m,-b);    // For max_value: Line L(-m,b);
    if(H.size() == 2){ L.xL = oo, L.xR = -oo;  H.insert(L); return; }
    XXX = 0; itL = itR = H.upper_bound(L); --itL;

    if((L.m == itL->m && L.b <= itL->b) || (L.m == itR->m && L.b <= itR->b)) return;
    if(!cw(*itL,L,*itR)) return;

    if(!itL->d) for(it=itL--;(it->m==L.m || !cw(*itL,*it,L));it=itL--) H.erase(it);
    if(!itR->d) for(it=itR++;(it->m==L.m || !cw(L,*it,*itR));it=itR++) H.erase(it);
    it = itL = itR = H.insert(itR,L);  --itL; ++itR;

    it->xL = itL->xR = (itL->d ?  oo : intersect(*it,*itL));
    it->xR = itR->xL = (itR->d ? -oo : intersect(*it,*itR));
  }

  ld min_value(ld x){
    XXX = 1;  it = H.lower_bound(x);
    return (it->m)*x - (it->b);     ////   return -(it->m)*x + (it->b);
  }

  ld min_value_inc_x(ld x){ // If you know every x is larger (or the same)
    itR = H.end(); advance(itR,-2);
    for(it=itR-- ; x > it->xL ; it=itR-- ) H.erase(it);
    it->xR = -oo;
    return (it->m)*x - (it->b);     ////   return -(it->m)*x + (it->b);
  }

  ld min_value_dec_x(ld x){ // If you know every x is smaller (or the same)
    itL = H.begin(); ++itL;
    for(it=itL++ ; x < it->xR ; it=itL++ ) H.erase(it);
    it->xL = oo;
    return (it->m)*x - (it->b);     ////   return -(it->m)*x + (it->b);
  }
};

// Assumes that slopes are non-increasing (or non-decreasing)
struct Hull2{ // add_line = O(1)    min_value = O(log n)
  deque<Line> H;

  void add_line_slope_inc(ld m,ld b){
    Line L(m,-b);                    //// Line L(-m,b);
    if(H.empty()) { L.xL = oo, L.xR = -oo; H.push_back(L); return; }
    if(H.back().m == L.m && H.back().b >= L.b) return;
    while(H.size() > 1 && !cw(H[H.size()-2],H.back(),L)) H.pop_back();
    L.xR = -oo; L.xL = H.back().xR = intersect(H.back(),L);
    H.push_back(L);
  }

  void add_line_slope_dec(ld m,ld b){
    Line L(m,-b);                  //// Line L(-m,b);
    if(H.empty()) { L.xL = oo, L.xR = -oo; H.push_front(L); return; }
    if(H[0].m == L.m && H[0].b >= L.b) return;
    while(H.size() > 1 && !cw(L,H[0],H[1])) H.pop_front();
    L.xR = -oo; L.xL = H[0].xR = intersect(H[0],L);
    H.push_front(L);
  }

  ld min_value(ld x){
    XXX = 1; int i = lower_bound(H.begin(),H.end(),x) - H.begin();
    return H[i].m*x - H[i].b;          ////   return -H[i].m*x + H[i].b;
  }

  ld min_value_inc_x(ld x){ // If you know every x is larger (or the same)
    while(H.back().xL < x) { H.pop_back(); } H.back().xR = -oo;
    return (H.back().m)*x - (H.back().b); //// return -(H.back().m)*x+(H.back().b);
  }

  ld min_value_dec_x(ld x){ // If you know every x is smaller (or the same)
    while(H[0].xR > x) { H.pop_front(); }  H[0].xL = oo;
    return (H[0].m)*x - (H[0].b); //// return -(H[0].m)*x + (H[0].b);
  }
};
```

8 String Processing

```
// KMP
void prepare_pattern(const string &pat, vector<int> &T) {
  int n = pat.length();
  T.resize(n+1);
  fill(T.begin(), T.end(), -1);
  for (int i = 1; i <= n; i++) {
    int pos = T[i-1];
    while (pos != -1 && pat[pos] != pat[i-1])
      pos = T[pos];
    T[i] = pos + 1;
  }
}

int find_pattern(const string &s, const string &pat, const vector<int> &T) {
  int sp = 0, kp = 0;
  int slen = s.length(), plen = pat.length();
  while (sp < slen) {
    while (kp != -1 && (kp == plen || pat[kp] != s[sp]))  kp = T[kp];
    kp++;   sp++;
    if (kp == plen)
      return sp - plen;   // continue with kp = T[kp] for more
  }
  return -1;  // not found
}
```

```
// Suffix array: O(n)  and  O(n log n)
// 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.
typedef pair<int,int> pii;
typedef tuple<int,int,int> tiii;
typedef vector<int> vi;

void radixPass(vi &a, vi &b, vi &r, int n, int K, int off=0) {
  vi c(K+1, 0);
  for (int i = 0;  i < n;  i++) c[r[a[i]+off]]++;
  for (int i = 0, sum = 0;  i <= K;  i++) {
     int t = c[i];  c[i] = sum;  sum += t;
  }
  for (int i = 0;  i < n;  i++) b[c[r[a[i]+off]]++] = a[i];
}

#define GetI() (SA12[t] < n0 ? SA12[t] * 3 + 1 : (SA12[t] - n0) * 3 + 2)
void sarray_int(vi &s, vi &SA, int n, int K) {
  int n0=(n+2)/3, n1=(n+1)/3, n2=n/3, n02=n0+n2;
  vi s12(n02 + 3, 0), SA12(n02 + 3, 0), s0(n0), SA0(n0);

  for (int i=0, j=0;  i < n+(n0-n1);  i++) if (i%3 != 0) s12[j++] = i;

  radixPass(s12 , SA12, s, n02, K, 2);
  radixPass(SA12, s12 , s, n02, K, 1);
  radixPass(s12 , SA12, s, n02, K, 0);

  int name = 0, c0 = -1, c1 = -1, c2 = -1;
  for (int i = 0;  i < n02;  i++) {
    if (s[SA12[i]] != c0 || s[SA12[i]+1] != c1 || s[SA12[i]+2] != c2) {
      name++;  c0 = s[SA12[i]];  c1 = s[SA12[i]+1];  c2 = s[SA12[i]+2];
    }
    if (SA12[i] % 3 == 1) { s12[SA12[i]/3]      = name; }
    else                  { s12[SA12[i]/3 + n0] = name; }
  }

  if (name < n02) {
    sarray_int(s12, SA12, n02, name);
    for (int i = 0;  i < n02;  i++) s12[SA12[i]] = i + 1;
  } else
    for (int i = 0;  i < n02;  i++) SA12[s12[i] - 1] = i;

  for (int i=0, j=0;  i < n02;  i++) if (SA12[i] < n0) s0[j++] = 3*SA12[i];
  radixPass(s0, SA0, s, n0, K);

  for (int p=0,  t=n0-n1,  k=0;  k < n;  k++) {
    int i = GetI(), j = SA0[p];
    if (SA12[t] < n0 ?
        (pii(s[i], s12[SA12[t] + n0]) < pii(s[j], s12[j/3])) :
        (tiii(s[i],s[i+1],s12[SA12[t]-n0+1]) < tiii(s[j],s[j+1],s12[j/3+n0]))) {
      SA[k] = i;  t++;
      if (t == n02)
        for (k++;  p < n0;  p++, k++) SA[k] = SA0[p];
    } else {
      SA[k] = j;  p++;
      if (p == n0)
        for (k++;  t < n02;  t++, k++) SA[k] = GetI();
    }
  }
}

// O(n)
void build_sarray(string str, int sarray[]) {
  int n = str.length();
  if (n <= 1) { for (int i = 0; i < n; i++)  sarray[i] = i; return; }

  vi s(n+3, 0), SA(n+3);
  for (int i = 0; i < n; i++) s[i] = (int)str[i] - CHAR_MIN + 1;
  sarray_int(s, SA, n, 256);
  copy(SA.begin(), SA.begin()+n, sarray);
}

// O(n log n) -- Will work for n <= 10^6
void build_sarray(string str, int sarray[]) {
  int n = str.length();
  if (n <= 1) { for (int i = 0; i < n; i++)  sarray[i] = i; return; }

  vi RA(2*n, 0), SA(2*n), tmp(2*n);
  for (int i = 0; i < n; i++) RA[i] = (int)str[i] - CHAR_MIN + 1;
  for (int i = 0; i < n; i++) SA[i] = i;
  for (int k = 1; k < n; k <<= 1){
    radixPass(SA,tmp,RA,n,max(n,256),k);
    radixPass(tmp,SA,RA,n,max(n,256),0);
    tmp[SA[0]] = 1;
    for(int i=1;i<n;i++){
      tmp[SA[i]] = tmp[SA[i-1]];
      if((RA[SA[i]] != RA[SA[i-1]]) || (RA[SA[i]+k] != RA[SA[i-1]+k]))
        tmp[SA[i]]++;
    }
    copy(tmp.begin(),tmp.begin()+n,RA.begin());
  }
  copy(SA.begin(), SA.begin()+n, sarray);
}

// O(n)
void compute_lcp(string str, int sarray[], int lcp[]) {
  int n = str.length(), h = 0;  vi rank(n);
  for (int i = 0; i < n; i++)   rank[sarray[i]] = i;

  for (int i = 0; i < n; i++) {
    int k = rank[i];
    if (k > 0) {
      int j = sarray[k-1];
      while (i + h < n && j + h < n && str[i+h] == str[j+h]) h++;
      lcp[k] = h;
    }
    if (h > 0) h--;
  }
  lcp[0] = 0;
}

struct Comp{
  const string &s; int m;
  Comp(const string &str,int M) : s(str),m(M) { }
  bool operator()(int i, const string& p) const { return s.compare(i,m,p,0,m) < 0; }
  bool operator()(const string& p, int i) const { return s.compare(i,m,p,0,m) > 0; }
};

pii find(const string &str, const int sarray[], const string &pattern) {
  pair<const int *, const int *> p =
    equal_range(sarray, sarray+str.size(), pattern, Comp(str,pattern.size()));
  return pii(p.first - sarray, p.second - sarray);
}
```

9 Game Theory

```
// alpha-beta pruning: Exponential time, but a good heuristic
// -- Use for mini-max searches (Player 1 is maximizing, Player -1 is minimizing).
// -- Call from main with f(start,-inf,inf,1);

int f(state S,int alpha,int beta,int p){
  if(s.is_done()) return p*s.value();

  for_all_states_from(s,p){     // We want "next" to run through all possible
    state next = child_of(S,p); // moves that player p can take from state s.
    alpha = max(alpha,-f(next,-beta,-alpha,-p));
    if(beta <= alpha) return alpha;
  }
  return alpha;
}
```

10 Algorithms and Misc

```
// Gives the solutions a*x^3 + b*x^2 + c*x + d = 0.
//  Note: Does NOT work well when a is NEAR 0.

typedef long double ld;
int cubic(ld a,ld b,ld c,ld d,ld s[3]){
  b /= a, c /= a, d /= a; // Ensure that a is not zero before hand!
  ld q = (b*b - 3*c)/9, r = (2*pow(b,3) - 9*b*c + 27*d) / 54;
  ld z = pow(r,2) - pow(q,3);
  if(z <= EPS){
    ld theta = acos(r/pow(q,1.5));
    for(int i=0;i<3;i++) s[i] = -2*sqrt(q)*cos((theta+i*2*PI)/3) - b/3;
    return 3;
  }
  s[0] = pow(sqrt(z)+fabs(r),1.0/3);
  s[0] = (s[0] + q/s[0]) * (r < 0 ? 1 : -1) - b/3;
  return 1;
}
```

```
// Use built-in hashing for numbers, strings, pointers, vector<bool>,
// bitset. Others require custom hashing.

struct XYZ {
  int a; string b;
  bool operator==(const XYZ& r) const { // unordered_map only needs equality
    return a == r.a && b == r.b; }
};

namespace std {
  template<> struct hash<XYZ> {
    size_t operator()(const XYZ& xyz) const {
      return hash<int>()(xyz.a) ^ hash<string>()(xyz.b); }
  };
}

unordered_map<XYZ,string> um; // Elements are NOT sorted. Access is O(1).
```

```
// Infix expressions evaluation
struct Token {  // modify as needed
  enum Type {NUMBER, PLUS, MINUS, TIMES, DIVIDE, LEFT_PAREN, RIGHT_PAREN};
  int priority[7]; // priority of the operators: bigger number means higher priority
  bool left_assoc[7];  // is operator left assoc
  Type type;
  long val;

  Token() {
    priority[1] = priority[2] = 1;
    priority[3] = priority[4] = 2;
    priority[5] = 0;
    left_assoc[1] = left_assoc[2] = left_assoc[3] = left_assoc[4] = true;
  }

  int get_priority() const { return priority[type]; };
  bool is_left_assoc() const { return left_assoc[type]; }

  bool next_token(string &expr, int &start, bool &error) {
    int len = expr.length();
    error = false;
    while (start < len && isspace(expr[start])) start++;
    if (start >= len) return false;

    switch (expr[start]) {
    case '(': type = LEFT_PAREN;   break;
    case ')': type = RIGHT_PAREN;  break;
    case '*': type = TIMES;        break;
    case '/': type = DIVIDE;       break;
    case '+': type = PLUS;         break;
    case '-': type = MINUS;        break;
    default:
      const char *s = expr.c_str() + start; char *p;
      val = strtol(s, &p, 10);
      if (s == p) { error = true;  return false; }
      type = NUMBER;     start += (p - s);
    }
    if (type != NUMBER) start++;
    return true;
  }
};

#define F(T) case Token::T: \
  b = operands.top(); operands.pop(); a = operands.top(); operands.pop();

// returns true if operation is successful
bool apply_op(stack<long> &operands, Token token){  // modify for more tokens
  long a, b;
  if (operands.size() < 2) return false;
  switch(token.type){
    F(PLUS) operands.push(a+b); break;
    F(MINUS) operands.push(a-b); break;
    F(TIMES) operands.push(a*b); break;
    F(DIVIDE) if(b == 0) return false; operands.push(a/b); break;
    default: return false;
  }
  return true;
}

long evaluate(string expr, bool &error){
  stack<Token> s;
  stack<long> operands;
  int i;
  Token token;

  error = false;       i = 0;
  while (token.next_token(expr, i, error) && !error) {
    switch (token.type) {
    case Token::NUMBER:
      operands.push(token.val);      break;
    case Token::LEFT_PAREN:
      s.push(token);                 break;
    case Token::RIGHT_PAREN:
      while (!(error = s.empty()) && s.top().type != Token::LEFT_PAREN) {
        if ((error = !apply_op(operands, s.top()))) break;
        s.pop();
      }
      if (!error) s.pop();
      break;
    default:
      while (!error && !s.empty() &&
             (token.get_priority() < s.top().get_priority() ||
              (token.get_priority() == s.top().get_priority() &&
               token.is_left_assoc()))) {
        error = !apply_op(operands, s.top());           s.pop();
      }
      if (!error) s.push(token);
    }
    if (error) break;
  }
  while (!error && !s.empty()) {
    error = !apply_op(operands, s.top());               s.pop();
  }
  error |= (operands.size() != 1);
  return (error) ? 0 : operands.top();
}
```

```
// Josephus Problem (0-based) -- Kill the k'th person first
//   1. Determine the survivor            -- O(n)
//       -- Do not include inner for-loop (j)
//       -- A[i] is the survivor with i people, killing every k'th.
//   2. Determine the full killing order -- O(n^2)
//       -- Include inner for-loop (j)
//       -- A[i] is the i'th person who is killed (A[n] is survivor)
void josephus(int A[], int n,int k){
  A[1] = 0;
  for(int i=2;i<=n;i++){ A[i] = (A[i-1]+(k%i))%i;
    for(int j=1;j<i;j++) A[j] = (A[j]+(k%i))%i;
  }
}
```

```
// Multiplies two polynomials in O((n+m)*log(n+m))
//  There will be rounding errors. Check for them.

typedef vector<complex<double> > vcd;
vcd DFT(const vcd& a,double inv,int st=0,int step=1){
  int n = a.size()/step;
  if(n == 1) return vcd(1,a[st]);
  complex<double> w_n = polar(1.0,inv*2*PI/n), w = 1;
  vcd y_0 = DFT(a,inv,st,2*step), y_1 = DFT(a,inv,st+step,2*step), c(n);

  for(int k=0 ; k<n/2 ; k++,w *= w_n){
    c[k]     = y_0[k] + w*y_1[k];     c[k+n/2] = y_0[k] - w*y_1[k];
  }
  return c;
}

vcd poly_mult(vcd p,vcd q){
  int m = p.size()+q.size(),s=1;
  while(s < m) s *= 2;
  p.resize(s,0); q.resize(s,0);
  vcd P = DFT(p,1), Q = DFT(q,1), R = P;
  for(int i=0;i<R.size();i++) R[i] *= Q[i];
  vcd ans = DFT(R,-1);
  for(int i=0;i<ans.size();i++) ans[i] /= s;
  return ans;
}
```

```
// Be greedy from large items to small  -- Only works for (0 < x < 4000)
string Roman[13] = {"M" ,"CM","D","CD","C","XC","L","XL","X","IX","V","IV","I"};
int Arabic[13]   = {1000, 900,500, 400,100,  90, 50,  40, 10,   9,  5,   4,  1};
```

```
// -- n is the number of intervals -- IT MUST BE EVEN. O(n)
// -- If K is an upper bound on the 4th derivative of f for all x in [a,b],
//     then the maximum error is ( K*(b-a)^5 ) / ( 180*n^4 )
double integrate(double (*f)(double), double a, double b, int n){
  double ans = f(a) + f(b), h = (b-a)/n;
  for(int i=1;i<n;i++) ans += f(a+i*h) * (i%2 ? 4 : 2);
  return ans * h / 3;
}

// -- h is the step size. Error is O(h^4).
double differentiate(double (*f)(double), double x, double h){
  return (-f(x+2*h) + 8*(f(x+h) - f(x-h)) + f(x-2*h)) / (12*h);
}
```

```
// Rubik's Cube: This is how the cube labelled. Note that you fold the box away
// from you, so C is the closest face to you and E is the farthest:
//
//         +-------+                            +-------+
//         |       |                            | 1 2 3 |
//         |   A   |                            | 8 0 4 |
//         |       |                            | 7 6 5 |
// +-------+-------+-------+-------+    +-------+-------+-------+-------+
// |       |       |       |       |    | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 |
// |   B   |   C   |   D   |   E   |    | 8 0 4 | 8 0 4 | 8 0 4 | 8 0 4 |
// |       |       |       |       |    | 7 6 5 | 7 6 5 | 7 6 5 | 7 6 5 |
// +-------+-------+-------+-------+    +-------+-------+-------+-------+
//         |       |                            | 1 2 3 |
//         |   F   |                            | 8 0 4 |
//         |       |                            | 7 6 5 |
//         +-------+                            +-------+
//
//   cube[i][j] is the color of index j on face (i+'A')
// To rotate Face X 90 degrees clockwise, call rotateFace(X)
// To rotate Face X 90 degrees counterclockwise, call rotateFace(X) 3 times

string rot[6] = {"BEDC1111","ACFE7773","ADFB5713","AEFC3733","ABFD1753","BCDE5555"};
int cube[6][9],t[3]; // t is a tmp variable
int m9(int x){ return (x % 9 == 0 ? 1 : x % 9); }

void rotateFace(char F){
   int ind = F - 'A'; rotate(cube[ind]+1,cube[ind]+7,cube[ind]+9);

   string r = rot[ind];
   for(int i=0;i<3;i++) t[i] = cube[r[3]-'A'][m9(r[7]-'0'+i)];

   for(int i=7;i>4;i--) for(int j=0;j<3;j++)
         cube[r[i-4]-'A'][m9(r[i]-'0'+j)] = cube[r[i-5]-'A'][m9(r[i-1]-'0'+j)];
   for(int j=0;j<3;j++) cube[r[0]-'A'][m9(r[4]-'0'+j)] = t[j];
}

void printCube(){
   string o = "123804765";
   for(int F=0;F<6;F++){
      cout << "Face␣" << char(F+'A') << endl;
      for(int i=0;i<3;i++){
         for(int j=0;j<3;j++) cout << setw(3) << cube[F][o[i*3+j]-'0'];
         cout << endl;
      }
   }
}
```

```
// simplex: A is (m+1)x(n+1).
// First row obj. function (maximize), next m rows are <= constraints
const int MAX_M = 101, MAX_N = 101; // MAX_CONSTRAINTS+1 and MAX_VARS+1
const double EPS = 1e-9, INF = 1.0/0.0;

void pivot(double A[MAX_M][MAX_N],int m, int n, int a, int b,int basis[],int out[]){
  for (int i = 0; i <= m; i++)
    if (i != a)
      for (int j = 0; j <= n; j++)
        if (j != b) A[i][j] -= A[a][j] * A[i][b] / A[a][b];
  for (int j = 0; j <= n; j++) if (j != b) A[a][j] /=  A[a][b];
  for (int i = 0; i <= m; i++) if (i != a) A[i][b] /= -A[a][b];
  A[a][b] = 1 / A[a][b];
  swap(basis[a], out[b]);
}

bool pless(double a1,double a2,double b1,double b2){
  return (a1 < b1-EPS || (a1 < b1+EPS && a2 < b2));
}

// A is altered
double simplex(int m, int n, double A[MAX_M][MAX_N], double X[MAX_N]){
  int i, j, I, J, basis[MAX_M], out[MAX_N];
  for (i = 1; i <= m; i++) basis[i] = -i;
  for (j = 0; j <= n; j++) A[0][j] = -A[0][j], out[j] = j;
  A[0][n] = 0;
  while(true) {
    for (i = I = 1; i <= m; i++)
      if (make_pair(A[i][n],basis[i]) < make_pair(A[I][n],basis[I])) I = i;
    if (A[I][n] > -EPS) break;
    for (j = J = 0; j < n; j++)
      if (pless(A[I][j],out[J],A[I][J],out[j])) J = j;
    if (A[I][J] > -EPS) return -INF; // No solution
    pivot(A, m, n, I, J, basis, out);
  }
  while(true) {
    for (j = J = 0; j < n; j++)
      if (make_pair(A[0][j],out[j]) < make_pair(A[0][J],out[J])) J = j;
    if (A[0][J] > -EPS) break;
    for (i=1, I=0; i <= m; i++){
      if (A[i][J] < EPS) continue;
      if (!I || pless(A[i][n]/A[i][J],basis[i],A[I][n]/A[I][J],basis[I])) I = i;
    }
    if (A[I][J] < EPS) return INF; // Unbounded
    pivot(A, m, n, I, J, basis, out);
  }
  fill(X, X+n, 0);
  for (i = 1; i <= m; i++) if (basis[i] >= 0) X[basis[i]] = A[i][n];
  return A[0][n];
}
```

11 Formulas

Triangles

Sine law: $\frac{\sin(\alpha)}{a} = \frac{\sin(\beta)}{b} = \frac{\sin(\gamma)}{c}$, a, b, c = side lengths, α, β, γ = opposite angles.

Cosine law: $c^2 = a^2 + b^2 - 2ab\cos(\gamma)$

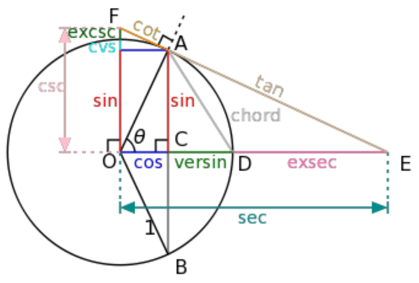
Circle inscribed in triangle: radius = $\sqrt{\frac{(s-a)(s-b)(s-c)}{s}}$, $s = \frac{a+b+c}{2}$.

Circumcircle: radius = $\frac{abc}{4A}$, A = area of triangle.

Trig Identities

$$\sin^2(u) = \tfrac{1}{2}(1-\cos(2u)) \qquad \cos^2(u) = \tfrac{1}{2}(1+\cos(2u))$$

$$\begin{aligned}
\sin(u)+\sin(v) &= 2\sin\left(\tfrac{u+v}{2}\right)\cos\left(\tfrac{u-v}{2}\right) & \sin(u)-\sin(v) &= 2\sin\left(\tfrac{u-v}{2}\right)\cos\left(\tfrac{u+v}{2}\right)\\
\cos(u)+\cos(v) &= 2\cos\left(\tfrac{u+v}{2}\right)\cos\left(\tfrac{u-v}{2}\right) & \cos(u)-\cos(v) &= -2\sin\left(\tfrac{u+v}{2}\right)\sin\left(\tfrac{u-v}{2}\right)\\
\sin(u)\sin(v) &= \tfrac{1}{2}(\cos(u-v)-\cos(u+v)) & \cos(u)\cos(v) &= \tfrac{1}{2}(\cos(u-v)+\cos(u+v))\\
\sin(u)\cos(v) &= \tfrac{1}{2}(\sin(u+v)+\cos(u-v)) & \cos(u)\sin(v) &= \tfrac{1}{2}(\sin(u+v)-\cos(u-v))
\end{aligned}$$



Length of a Chord: $2r\sin\theta$

Other Geometry

Rotation matrix: $\begin{bmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{bmatrix}$ (counter-clockwise by θ)

Dot product: $\vec{u}\cdot\vec{v} = \|\vec{u}\|\|\vec{v}\|\cos\theta$.

Sphere through 4 Points: Given (x_i, y_i, z_i), find (x, y, z) and r.

$$x = 0.5\cdot M_{12}/M_{11},\ y = -0.5\cdot M_{13}/M_{11},\ z = 0.5\cdot M_{14}/M_{11},\ r = d((x,y,z),(x_1,y_1,z_1))$$

where $\begin{vmatrix} x^2+y^2+z^2 & x & y & z & 1\\ x_1^2+y_1^2+z_1^2 & x_1 & y_1 & z_1 & 1\\ x_2^2+y_2^2+z_2^2 & x_2 & y_2 & z_2 & 1\\ x_3^2+y_3^2+z_3^2 & x_3 & y_3 & z_3 & 1\\ x_4^2+y_4^2+z_4^2 & x_4 & y_4 & z_4 & 1 \end{vmatrix} = 0$

Number Theory

Number and sum of divisors: multiplicative, $\tau(p^k) = k+1$, $\sigma(p^k) = \frac{p^{k+1}-1}{p-1}$.

Linear Diophantine equations: $a\cdot s + b\cdot t = c$ iff $\gcd(a,b)|c$.

Solutions are $(s_0, t_0) + k\cdot\left(\frac{b}{\gcd(a,b)}, -\frac{a}{\gcd(a,b)}\right)$.

Misc

Pick's Theorem: $A = i + \frac{b}{2} - 1$, A = area, i = interior lattice points, b = boundary lattice points.

Euler formula: $V - E + F - C = 1$, V = vertices, E = edges, F = faces, C = number of connected components. True for planar graphs and regular polyhedra (assume $C = 1$ in the latter).

Catalan numbers: $C_n = \frac{1}{n+1}\binom{2n}{n}$. Recurrence: $C_0 = 1$, and $C_{n+1} = \sum_{i=0} C_i C_{n-i}$.

Derangements: $!0 = 1$, $!1 = 0$, $!n = (n-1)(!(n-1) + !(n-2))$.

Burnside's Lemma: $|X/G| = \frac{1}{|G|}\sum_{g\in G}|X_g|$ (Points fixed by g) $[\frac{1}{24}(n^6 + 3n^4 + 12n^3 + 8n^2)]$

Number of solutions: $x_1 + \cdots + x_k = r$ with $x_i \geq 0$: $\binom{r+k-1}{r}$

Integer Partitions of n: (Also number of nonnegative solutions to $b+2c+3d+4e+\ldots = n$ and the number of nonnegative solutions to $2c+3d+4e+\ldots \leq n$)

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9
0x	1	1	2	3	5	7	11	15	22	30
1x	42	56	77	101	135	176	231	297	385	490
2x	627	792	1002	1255	1575	1958	2436	3010	3718	4565
3x	5604	6842	8349	10143	12310	14883	17977	21637	26015	31185
4x	37338	44583	53174	63261	75175	89134	105558	124754	147273	179525

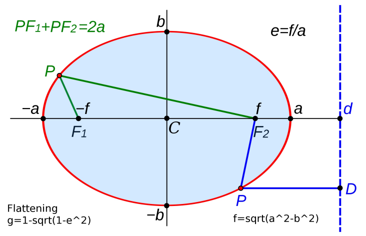
Lagrange Interpolation: Given $(x_0, y_0), \cdots, (x_n, y_n)$, the polynomial is:

$$P(x) = \sum_{j=1}^{n} P_j(x) \text{ where } P_j(x) = y_j \prod_{0 \leq k \leq n, k \neq j} \frac{x - x_k}{x_j - x_k}$$

Usable Chooses: $\binom{n}{k}$ is safe assuming 50,000,000 is not TLE: $\binom{28}{k}$ is okay for all $k \leq n$.

n	29	$30-31$	$32-33$	$34-38$	$39-45$	$46-59$	$60-92$	$93-187$	$188-670$
k	11	10	9	8	7	6	5	4	3

11.0.1 Physics



Circumference: $4a \int_0^{\pi/2} \sqrt{1 - \varepsilon^2 \sin^2(\theta)}\, d\theta$

Polar form relative to focus: $r(\theta) = \frac{a(1-\varepsilon)}{1-\varepsilon\cos(\theta-\phi)}$, where ϕ is the angle of rotation of ellipse.

Polar form relative to centre: $r(\theta) = \frac{ab}{\sqrt{(b\cos\theta)^2 + (a\sin\theta)^2}}$

Minimal Surface of Revolution (Rotating around x-axis): $y = a\cosh(\frac{x-b}{a})$
Do binary search on a using secant lines – (a,b) is the extrema

Rational Roots: $a_n x^n + \cdots + a_0 = 0$. If $\frac{p}{q}$ is a solution, where $(p,q) = 1$, then $p | a_0$ and $q | a_n$.

$$r^2 \frac{d\theta}{dt} = \frac{2\pi}{p} ab$$

11.1 Rotating Calipers

Computing distances: The diameter of a convex polygon, The width of a convex polygon, The maximum distance between 2 convex polygons, The minimum distance between 2 convex polygons.

Enclosing rectangles: The minimum area enclosing rectangle, The minimum perimeter enclosing rectangle

Triangulations: Onion triangulations, Spiral triangulations Quadrangulations

Properties of convex polygons: Merging convex hulls, Finding common tangents, Intersecting convex polygons, Critical support lines, Vector sums of convex polygons

Thinnest transversals: Thinnest-strip transversals

12 Tips

If You Are Stuck, Read These!

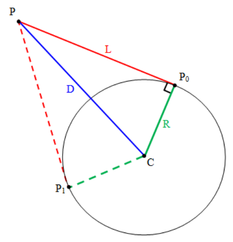
- Can you write the question as a whole bunch of inequalities? (Simplex?)
- Can you hash to reduce time? (Normally cuts a factor of N)
- Can you only have one "item" on a location at a time? Can only one "item" move through a hallway at one time?
- Can you break the problem into two disjoint sets? (Even/Odd, Black/White, 2-player games)
- Is $n \approx 40$? Consider $O(2^{n/2} \log(2^{n/2}))$.
- Would $\sqrt{N}$ blocks of size $\sqrt{N}$ help?
- Read the Table of Contents!
- Binary search and check (often greedy)
- Sweep line/circle (often with extra data structures)
- DP:
  - subsets (e.g. TSP type)
  - on trees: state = (root, extra info)
  - on DAG
  - incremental convex hull/envelope code
  - probability/expected value in a state transition graphs, deal with cycles through infinite series or linear equations.
- Represent moving objects as $f(t) = v \cdot t + \text{init. pos.}$ and use geometry.
- Coordinate compression
- Meet-in-the-middle
- Max flow of some kind, but need to formulate right graph
- Brute Force:
  - Are there very few different solutions?
  - Are there very few different (effective) inputs?
  - Pruning
- Math:
  - integration/area computation
  - physics: make sure you read all the rules

- Game Theory (2-player):
  - Can you duplicate your opponent's move?
  - Can formulate it so one person is maximizing something and one person minimizing?
  - Write a program to brute force small cases and look for a pattern.
- Try to looking at the problem in reverse?
- Cycle decomposition of permutation.

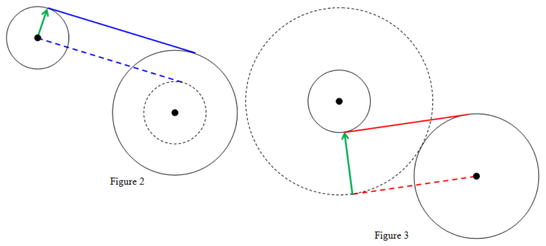
General Things

- **RTFQ**
- Step away from the computer. Go to the bathroom.
- Print after every submission, debug on paper.
- Did you remember to handle the empty cases (e.g. n = 0).
- Graphs: is it directed or undirected?
- Floating-point computation: be careful about -0.0
- atan2 can return -pi and +pi
- Watchout for stack overflow (DFS and large variables)

Point and Circle Tangent

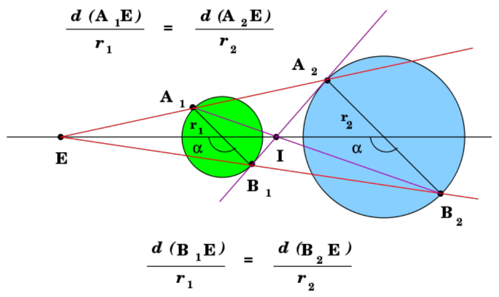


Now Intersect two circles: (C,R) and $(P, L = \sqrt{D^2 - R^2})$.



Two circles of radii $r_1 \le r_2$. For **outer tangent** (Left Picture), make a circle of radius $r_2 - r_1$ around C_2 (dashed circle) and find tangent lines from C_1 (dashed blue line), then translate it r_1 units (solid blue line). For **inner tangent** (Right Picture), make a circle of radius $r_1 + r_2$ around C_1 (dashed circle) and find tangent lines from C_2 (dashed red line), then translate it r_2 units (solid red line).

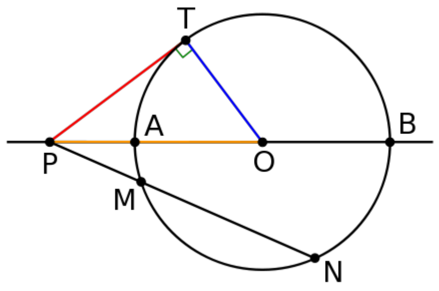
Holomorphic Centre


$\frac{d(A_1E)}{r_1} = \frac{d(A_2E)}{r_2}$
A1
A2
r1
r2
α
α
E
I
B1
B2
$\frac{d(B_1E)}{r_1} = \frac{d(B_2E)}{r_2}$

Inner tangent lines go through I:

$I = (x,y) = \frac{r_2}{r_1+r_2}(x_1,y_1) + \frac{r_1}{r_1+r_2}(x_2,y_2)$ $\quad$ $E = (x,y) = \frac{-r_2}{r_1-r_2}(x_1,y_1) + \frac{r_1}{r_1-r_2}(x_2,y_2)$

Power Points



$$\overline{PT}^2 = \overline{PM} \cdot \overline{PN} = \overline{PA} \cdot \overline{PB} = \overline{PO}^2 - \overline{TO}^2$$

Start of Contest

- Put this somewhere in the `.bashrc` file:

```
function amake(){
  g++ -g -std=g++0x -static -Wall ${1}.cc -o ${1}
}
ulimit -c unlimited
function e { emacs "$@" & }
```

- Type this command: `source .bashrc`