ACM ICPC World Finals 2020 Code Booklet
University of Lethbridge

1 Setup

```
#!/bin/bash
clear

echo running... $1
g++ $1 -g -Og -std=c++17 -Wall -Wextra -Wconversion -Wfatal-errors -fsanitize=address,
    undefined -o sol || exit

for i in *.in; do
  # run sample in
  echo test $i
  ./sol < $i

done

# for d in {A..K}; do mkdir $d && cp ~/template.cc $d/d.cc; done
```

```
#include <bits/stdc++.h>
using namespace std;

#define debug(a) cerr << #a << " = " << (a) << endl;
#define fst first
#define snd second
#define sz(x) (int)(x).size()
#define all(X) begin(X), end(X)

template<typename T, typename U> ostream& operator<<(ostream& o, const pair<T, U>& x)
     {
  o << "(" << x.fst << ", " << x.snd << ")"; return o;
}

template<typename T> ostream& operator<<(ostream& o, const vector<T>& x) {
  o << "["; int b = 0; for (auto& a : x) o << (b++ ? ", " : "") << a; o << "]"; return
       o;
}

template<typename T> ostream& operator<<(ostream& o, const set<T>& x) {
  o << "{"; int b = 0; for (auto& a : x) o << (b++ ? ", " : "") << a; o << "}"; return
       o;
}

template<typename T, typename U> ostream& operator<<(ostream& o, const map<T, U>& x) {
  o << "{"; int b = 0; for (auto& a : x) o << (b++ ? ", " : "") << a; o << "}"; return
       o;
}

int main() {
  ios::sync_with_stdio(0); cin.tie(0);

}
```

```
" setxkbmap -option caps:escape "
set nowrap
set nobackup
set nowritebackup
set smarttab
set expandtab
set tabstop=2
set softtabstop=0
set shiftwidth=0
set number relativenumber
set ai
set si

" fix shift O lag on some terminals "
set timeout timeoutlen=5000 ttimeoutlen=100

" tabs "
map <C-t> :tabnew<Space>
map <C-n> :tabn<CR>

" testing "
map <F10> :! ~/run %<CR>
map <leader>i :tabnew test.in<CR>
```

2 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

typedef long long ll;
bool dEqual(ll x, ll y) { return x == y; }  // replaces dEqual from double code
const ll EPS = 0;                           // replaces EPS from double code
struct Point {
  ll x, y;
  // safe ranges for 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;
}
```

3 Math

```
/* Polynomial algebra and modular arithmetic
 */
namespace algebra {

  #define double long double
  typedef complex<ftype> point;
  typedef double ftype;
  typedef long long ll;
  const double pi = acos(-1);
  const int maxn = 1 << 18;
  const int inf = 1 << 30;

  point w[maxn];
  bool initiated = 0;
  void init() {
    if (!initiated) {
      for(int i = 1; i < maxn; i *= 2)
        for(int j = 0; j < i; j++)
          w[i + j] = polar(ftype(1), pi * j / i);
      initiated = 1;
    }
  }

  template<typename T>
  void fft(T *in, point *out, int n, int k = 1) {
    if (n == 1) {
      *out = *in;
    } else {
      n /= 2;
      fft(in, out, n, 2 * k);
      fft(in + k, out + n, n, 2 * k);
      for (int i = 0; i < n; i++) {
        auto t = out[i + n] * w[i + n];
        out[i + n] = out[i] - t;
        out[i] += t;
```

```cpp
      }
    }
  }

  template<typename T>
  void slow(vector<T> &a, const vector<T> &b) {
    vector<T> res(max(sz(a) + sz(b) - 1, 0));
    for (int i = 0; i < sz(a); i++) {
      for (int j = 0; j < sz(b); j++) {
        res[i + j] += a[i] * b[j];
      }
    }
    a = res;
  }


  template<typename T>
  void mult(vector<T> &a, const vector<T> &b) {
    if (min(sz(a), sz(b)) < 200) { slow(a, b); return; }
    init();
    static const int shift = 15, mask = (1 << shift) - 1;
    int n = sz(a) + sz(b) - 1;
    while (__builtin_popcount(n) != 1) n++;
    a.resize(n);
    static point A[maxn], B[maxn], C[maxn], D[maxn];
    for (int i = 0; i < n; i++) {
      A[i] = point(a[i] & mask, a[i] >> shift);
      if (i < sz(b)) B[i] = point(b[i] & mask, b[i] >> shift);
      else B[i] = 0;
    }
    fft(A, C, n); fft(B, D, n);
    for (int i = 0; i < n; i++) {
      point c0 = C[i] + conj(C[(n - i) % n]);
      point c1 = C[i] - conj(C[(n - i) % n]);
      point d0 = D[i] + conj(D[(n - i) % n]);
      point d1 = D[i] - conj(D[(n - i) % n]);
      A[i] = c0 * d0 - point(0, 1) * c1 * d1;
      B[i] = c0 * d1 + d0 * c1;
    }
    fft(A, C, n); fft(B, D, n);
    reverse(C + 1, C + n); reverse(D + 1, D + n);
    int t = 4 * n;
    for (int i = 0; i < n; i++) {
      ll A0 = llround(real(C[i]) / t);
      T A1 = llround(imag(D[i]) / t);
      T A2 = llround(imag(C[i]) / t);
      a[i] = A0 + (A1 << shift) + (A2 << 2 * shift);
    }
  }

  template<typename T>
        T bpow(T x, ll n) { return n ? n % 2 ? x * bpow(x, n - 1) : bpow(x * x, n / 2)
             : T(1); }
        template<typename T>
        T bpow(T x, ll n, T m) { return n ? n % 2 ? x * bpow(x, n - 1, m) % m : bpow(x
             * x % m, n / 2, m) : T(1); }
        template<typename T>
        T gcd(const T &a, const T &b) { return b == T(0) ? a : gcd(b, a % b); }
        template<typename T>
        T nCr(T n, int r) { T res(1); for (int i = 0; i < r; i++) { res *= (n - T(i));
             res /= (i + 1); } return res; }

  template<int m>
        struct modular {
                ll r;
                modular() : r(0) {}
                modular(ll r) : r(r) { if (abs(r) >= m) r %= m; if (r < 0) r += m; }
                modular inv() const { return bpow(*this, m - 2); }
                modular operator * (const modular &t) const { return (r * t.r) % m; }
                modular operator / (const modular &t) const { return *this * t.inv();
                     }
                modular operator += (const modular &t) { r += t.r; if (r >= m) r -= m;
                      return *this; }
                modular operator -= (const modular &t) { r -= t.r; if (r < 0) r += m;
                     return *this; }
                modular operator + (const modular &t) const { return modular(*this) +=
                      t; }
                modular operator - (const modular &t) const { return modular(*this) -=
                      t; }
                modular operator *= (const modular &t) { return *this = *this * t; }
                modular operator /= (const modular &t) { return *this = *this / t; }
                bool operator == (const modular &t) const { return r == t.r; }
                bool operator != (const modular &t) const { return r != t.r; }
                operator ll() const { return r; }
        };

        template<int T>
        istream& operator << (istream &out, modular<T> &x) {
                return out << x.r;
        }

        template<int T>
        istream& operator >> (istream &in, modular<T> &x) {
                return in >> x.r;
        }

  template<typename T>
  struct poly {
    vector<T> a;
    poly() {}
    poly(T a0) : a{a0} { normalize(); }
    poly(vector<T> t) : a(t) { normalize(); }
    void normalize() { while (!a.empty() && a.back() == T(0)) a.pop_back(); }

    poly operator += (const poly &t) {
      a.resize(max(sz(a), sz(t.a)));
      for (int i = 0; i < sz(t.a); i++) a[i] += t.a[i];
      normalize();return *this;
    }
    poly operator -= (const poly &t) {
      a.resize(max(sz(a), sz(t.a)));
      for (int i = 0; i < sz(a); i++) a[i] -= t.a[i];
      normalize(); return *this;
    }
    poly operator + (const poly &t) const { return poly(*this) += t; }
    poly operator - (const poly &t) const { return poly(*this) -= t; }
    poly operator *= (const poly &t) { mult(a, t.a); normalize(); return *this; }
    poly operator * (const poly &t) const { return poly(*this) *= t; }

    // for division and remainder
    poly mod_xk(int k) const { k = min(k, sz(a)); return vector<T>(begin(a), begin(a)
         + k); }
    poly mul_xk(int k) const { poly res(*this); res.a.insert(begin(res.a), k, 0);
      return res; }
    poly div_xk(int k) const { k = min(k, sz(a)); return vector<T>(begin(a) + k, end(a
         )); }
    poly substr(int l, int r) const { l = min(l, sz(a)); r = min(r, sz(a)); return
         vector<T>(begin(a) + l, begin(a) + r); }
    poly inv(int n) const { // get inverse series mod x^n
      assert(!is_zero()); poly ans = a[0].inv(); int a = 1;
      while (a < n) { poly C = (ans * mod_xk(2 * a)).substr(a, 2 * a); ans -= (ans * C
          ).mod_xk(a).mul_xk(a); a *= 2; }
```

```
      return ans.mod_xk(n);
  }
  poly reverse(int n, bool rev = 0) const {
    poly res(*this);
    if (rev) res.a.resize(max(n, sz(res.a)));
    std::reverse(all(res.a)); return res.mod_xk(n);
  }
  pair<poly, poly> divmod(const poly &b) const {
    if (deg() < b.deg()) return {poly{0}, *this};
    int d = deg() - b.deg();
    poly D = (reverse(d + 1) * b.reverse(d + 1).inv(d + 1)).mod_xk(d + 1).reverse(d
         + 1, 1);
    return {D, *this - D * b};
  }
  poly operator /= (const poly &t) { return *this = divmod(t).first; }
  poly operator %= (const poly &t) { return *this = divmod(t).second; }
  poly operator / (const poly &t) const { return divmod(t).first; }
  poly operator % (const poly &t) const { return divmod(t).second; }
  poly operator *= (const T &x) { for (auto &it: a) it *= x; normalize(); return *
      this; }
  poly operator /= (const T &x) { for (auto &it: a) it /= x; normalize(); return *
      this; }
  poly operator * (const T &x) const { return poly(*this) *= x; }
  poly operator / (const T &x) const { return poly(*this) /= x; }

  T eval(T x) const { T res(0); for (int i = sz(a) - 1; i >= 0; i--) res *= x, res
      += a[i]; return res; }
  T& lead() { return a.back(); }
  int deg() const { return a.empty() ? -inf : sz(a) - 1; }
  bool is_zero() const { return a.empty(); }
  T operator [](int idx) const { return idx >= sz(a) || idx < 0 ? T(0) : a[idx]; }
  T& coef(int idx) { return a[idx]; }
  bool operator == (const poly &t) const { return a == t.a; }
  bool operator != (const poly &t) const { return a != t.a; }
  poly deriv() { vector<T> res; for (int i = 1; i <= deg(); i++) res.push_back(T(i)
      * a[i]); return res; }
  poly integr() { vector<T> res = {0}; for (int i = 0; i <= deg(); i++) res.
      push_back(a[i] / T(i + 1)); return res; }
  int leading_xk() const { if (is_zero()) return inf; int res = 0; while (a[res] ==
      T(0)) res++; return res; }

  template<typename iter>
  vector<T> eval(vector<poly> &tree, int v, iter l, iter r) {
    if (r - l == 1) {
      return {eval(*l)};
    } else {
      auto m = l + (r - l) / 2;
      auto A = (*this % tree[2 * v]).eval(tree, 2 * v, l, m);
      auto B = (*this % tree[2 * v + 1]).eval(tree, 2 * v + 1, m, r);
      A.insert(end(A), begin(B), end(B));
      return A;
    }
  }

  // evaluate polynomial in (x1, ..., xn)
  vector<T> eval(vector<T> x) {
    int n = sz(x);
    if (is_zero()) return vector<T>(n, T(0));
    vector<poly> tree(4 * n);
    build(tree, 1, all(x));
    return eval(tree, 1, all(x));
  }

  template<typename iter>
  poly inter(vector<poly> &tree, int v, iter l, iter r, iter ly, iter ry) {
    if (r - l == 1) {
      return {*ly / a[0]};
    } else {
      auto m = l + (r - l) / 2;
      auto my = ly + (ry - ly) / 2;
      auto A = (*this % tree[2 * v]).inter(tree, 2 * v, l, m, ly, my);
      auto B = (*this % tree[2 * v + 1]).inter(tree, 2 * v + 1, m, r, my, ry);
      return A * tree[2 * v + 1] + B * tree[2 * v];
    }
  }
};

template<typename T, typename iter>
poly<T> build(vector<poly<T>> &res, int v, iter L, iter R) {
  if (R - L == 1) {
    return res[v] = vector<T>{-*L, 1};
  } else {
    iter M = L + (R - L) / 2;
    return res[v] = build(res, 2 * v, L, M) * build(res, 2 * v + 1, M, R);
  }
}

// interpolates minimum polynomial from (xi, yi) pairs
template<typename T>
poly<T> inter(vector<T> x, vector<T> y) {
  int n = sz(x); vector<poly<T>> tree(4 * n);
  return build(tree, 1, all(x)).deriv().inter(tree, 1, all(x), all(y));
}

};

using namespace algebra;
const ll p = 1e9+7;
typedef modular<p> b;

struct piecewise {
  vector<int> r;
  vector<poly<b>> f;
  piecewise() {}
  piecewise(int c) : r(1, {1 << 30}), f(1, {c}) {}
};

piecewise integrate(piecewise& p, int bound) {
  auto& r = p.r; auto& f = p.f; poly<b> c(0); piecewise ans;
  ans.f.push_back({0}); ans.r.push_back(0);
  for (int i = 1; i < sz(f); i++) {
    if (r[i] <= bound) {
      f[i] = f[i].integr();
      ans.f.push_back(poly<b>(f[i].eval(min(r[i], bound))) - f[i] + c);
      ans.r.push_back(min(r[i], bound));
      c += poly<b>(f[i].eval(min(r[i], bound))) - f[i].eval(r[i-1]);
    }
  }
  return ans;
}

piecewise mult(piecewise& a, piecewise& b) {
  auto& r = a.r; auto& f = a.f;
  auto& s = b.r; auto& g = b.f;
  piecewise ans; int i = 0, j = 0;
  while (i < sz(f) and j < sz(g)) {
    ans.f.push_back(f[i]*g[j]);
    ans.r.push_back(min(r[i], s[j]));
    if (s[i] == r[j]) i++, j++;
    else if (s[i] < r[j]) i++;
    else if (s[i] > r[j]) j++;
  }
```

```
  return ans;
}
```

```
typedef long long ll;
const ll mod = 1e9+9;
// square matrix struct with fast mod exp
struct mat {
  int n; vector<vector<ll>> A;
  mat(int n, ll v) : n(n), A(n, vector<ll>(n, v)) {}
  mat(int n) : n(n), A(n, vector<ll>(n, 0)) { for (int i = 0; i < n; i++) A[i][i] = 1;
      }
  vector<ll>& operator[](int i) { return A[i]; }
  mat operator*(mat& left) {
    auto& a = *this;
    auto& b = left;
    mat r(n, 0);
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
          r[i][j] += (a[i][k] * b[k][j]) % mod,
          r[i][j] %= mod;
    return r;
  }
  mat operator^(ll e) {
    auto b = *this;
    mat r(n);
    while (e > 0) {
      if (e & 1) r = r * b, e--;
      else b = b * b, e /= 2;
    }
    return r;
  }
};
```

3.1 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, vector<int>& m, vector<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;
}
```

```
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; }
}
```

```
// 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;
}
```

```
/*  number theoretic transform.
 *  pick prime p such that
 *  p = c * 2^k + 1, then
 *  ord = 2^k, and
 *  r = g^c, where g is a primitive root
 *  common values: p, r, ord.
 *  7340033, 5, 1 << 20
 *  469762049, 13, 1 << 25
 *  998244353, 31, 1 << 23
 *  1107296257, 8, 1 << 24
 *  ... need __int128 for these
 *  10000093151233, 366508, 1 << 26
 *  1000000523862017, 2127080, 1 << 26
 *  To find solution mod arbitrary modulus, use CRT
 *  ! watch for 64 bit int overflow
 */

struct ntconv {
  ll p, r, rinv, ord;
  ntconv(ll p, ll r, ll ord) : p(p), r(r), rinv(modinv(r, p)), ord(ord) {}
  void ntt(vector<ll>& A, bool inv) {
    ll n = sz(A);
    for (ll i = 1, j = 0; i < n; i++) {
      ll b = n >> 1;
      for (; j & b; b >>= 1) j ^= b;
      j ^= b; if (i < j) swap(A[i], A[j]);
    }
    for (ll l = 2; l <= n; l <<= 1) {
      ll wl = inv ? rinv : r;
      for (ll i = l; i < ord; i <<= 1) wl = wl * wl % p;
      for (ll i = 0; i < n; i += l) {
        ll w = 1;
        for (ll j = 0; j < l/2; j++) {
          ll u = A[i+j], v = A[i+j+l/2] * w % p;
          A[i+j] = u + v < p ? u + v : u + v - p;
          A[i+j+l/2] = u - v >= 0 ? u - v : u - v + p;
          w = w * wl % p;
        }
      }
    }
    if (inv) {
      ll ninv = modinv(n, p);
      for (auto& a : A) a = a * ninv % p;
    }
  }
  vector<ll> mult(vector<ll> A, vector<ll> B) {
    int n = sz(A), m = sz(B), N = 1;
    while (N < n + m) N <<= 1;
    A.resize(N); B.resize(N);
    ntt(A, 0); ntt(B, 0); vector<ll> ans(N);
    for (int i = 0; i < N; i++) ans[i] = A[i] * B[i] % p;
    ntt(ans, 1);
    return ans;
  }
};
```

3.2 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 MAXN 50
#define MAXM 50
int triangulate(int A[MAXN+1][MAXM+MAXN+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) {
            for (int j = ci; j < n; j++) A[i][j] -= d.quot*A[pi][j];
            k++;
          }
        }
      }
      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[MAXM][MAXN], int b[MAXM], int m, int n, int xp[MAXN],
     int hom_basis[MAXN][MAXN]) {
  int mat[MAXN+1][MAXM+MAXN+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];
  }
}
```

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++) {
    // use lower_bound for strict increasing subsequence
    int j = upper_bound(last.begin()+1, last.begin()+len+1, A[i]) - last.begin();
    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;
}
```

```
// max sum is in [start,end]
int vecsum(int v[], int n, int &start, int &end)
{
  int maxval = 0, max_end = 0, max_end_start, max_end_end;
  start = max_end_start = 0;       end = max_end_end = -1;
  for (int i = 0; i < n; i++) {
    if (v[i] + max_end >= 0) {  max_end = v[i] + max_end;    max_end_end = i;
    } else { max_end_start = i+1;     max_end_end = -1;    max_end = 0; }


    if (maxval < max_end) {
      start = max_end_start;    end = max_end_end;    maxval = max_end;
    } else if (maxval == max_end) {    }  /* tie-breaking here */
  }
  return maxval;
}
```

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
      }
}
```

```
// 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);
}
```

```
// 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 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;
}
```

```
// 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;
}
```

```
/*  Minimum weight perfect matching in O(n^2 m)
 *  where n = #people, m = #tasks and n <= m.
 *  A[i][j] = cost to assign person i task j.
 *  returns the min weight and a vector containing the optimal assignment
 */
template<typename T>
pair<T, vector<int>> hungarian(const vector<vector<T>>& A) {
  int n = sz(A), m = sz(A[0]); T inf = numeric_limits<T>::max() / 2;
  vector<int> way(m + 1), p(m + 1), used(m + 1), ans(n); vector<T> u(n + 1), v(m + 1),
        minv(m + 1);
  for (int i = 1; i <= n; i++) {
    int j0 = 0, j1 = 0; p[0] = i; minv.assign(m + 1, inf), used.assign(m + 1, 0);
    do {
      int i0 = p[j0]; j1 = 0; T delta = inf; used[j0] = true;
      for (int j = 1; j <= m; j++) if (!used[j]) {
        T cur = A[i0 - 1][j - 1] - u[i0] - v[j];
        if (cur < minv[j]) minv[j] = cur, way[j] = j0;
        if (minv[j] < delta) delta = minv[j], j1 = j;
      }
      for (int j = 0; j <= m; j++) {
        if (used[j]) u[p[j]] += delta, v[j] -= delta;
        else minv[j] -= delta;
      }
    } while (j0 =  j1, p[j0]);
    do { int j1 = way[j0]; p[j0] = p[j1]; j0 = j1; } while (j0);
  }
  for (int i = 1; i <= m; i++) if (p[i] > 0) ans[p[i] - 1] = i - 1;
  return {-v[0], ans};
}
```

```
/*  Maximum unweighted bipartite matching in O(n sqrt(n))
 *  returns the size of matching and vector containing an optimal match
 *  match().snd[i] = -1 if i'th node (on the left) has no match
 *                   j if i'th node matched with j'th node(on the right)
 *
 *  NOTE: matching on bipartite graph can be used to solve:
 *        min vertex cover, min edge cover and max independant set
 */
struct matching {
  int l, r, p; vector<int> M, U, D; vector<vector<int>> A; queue<int> Q;
  matching (int l, int r) : l(l), r(r), D(r+1), A(r) {}
  void add_edge(int u, int v) { A[v].push_back(u); }
  bool bfs() {
    for (int v = 0; v < r; v++) if (!U[v]) D[v] = p, Q.push(v);
    while (!Q.empty()) {
      int v = Q.front(); Q.pop();
      if (D[v] != D[r]) for (int u : A[v]) if (D[M[u]] < p)
        D[M[u]] = D[v] + 1, Q.push(M[u]);
    }
    return D[r] >= p;
  }
  int dfs(int v) {
    if (v == r) return 1;
    for (int u : A[v]) if (D[M[u]] == D[v] + 1 and dfs(M[u]))
      return M[u] = v, 1;
    D[v] = D[r]; return 0;
  }
  pair<int, vector<int>> match() {
    int res = 0; M.assign(l, r), U.assign(r+1, 0);
    for (p = 0; bfs(); p = D[r] + 1) for (int v = 0; v < r; v++)
      if (!U[v] and dfs(v)) U[v] = 1, res++;
    replace(all(M), r, -1); return {res, M};
  }
};
```

```
/*  O(n) find strongly connected components in a digraph
 *  comp[i] = component containing vertex i.
 *  dag[i] = adjacency list of the i'th strongly connected component
 */
struct SCC {
```

```
  int n, c;
  vector<vector<int>> G, H;
  vector<int> ord, comp;
  vector<bool> V;
  SCC(int n) : n(n), G(n), H(n) { };
  void add_edge(int u, int v) {
    G[u].push_back(v);
    H[v].push_back(u);
  }
  void dfs1(int v) {
    V[v] = true;
    for (auto& u : G[v])
      if (!V[u]) dfs1(u);
    ord.push_back(v);
  }
  void dfs2(int v) {
    comp[v] = c;
    for (auto& u : H[v])
      if (comp[u] == -1) dfs2(u);
  }
  vector<int> scc() {
    V.assign(n, 0);
    for (int i = 0; i < n; i++)
      if (!V[i]) dfs1(i);
    comp.assign(n, -1); c = 0;
    for (int i = 0; i < n; i++) {
      int v = ord[n-1-i];
      if (comp[v] == -1) dfs2(v), c++;
    }
    return comp;
  }
  vector<vector<int>> dag() {
    set<pair<int, int>> S;
    vector<vector<int>> dag(c);
    for (int a = 0; a < n; a++) {
      for (auto& b : G[a]) {
        if (comp[a] == comp[b]) continue;
        if (!S.count({comp[a], comp[b]})) {
          dag[comp[a]].push_back(comp[b]);
          S.insert({comp[a], comp[b]});
        }
      }
    }
    return dag;
  }
};
```

```
/*  Include SCC code
 *  O(n) solve 2SAT problem:
 *  solve().fst = T/F if there is a valid assignment
 *  solve().snd = vector<bool> containing the valid assignments.
 */
int VAR(int i) { return 2*i; }
int NOT(int i) { return i^1; }
struct SAT {
  int n; SCC scc;
  SAT(int n) : n(n), scc(2*n) {}
  void add_or(int a, int b) {
    if (a == NOT(b)) return;
    scc.add_edge(NOT(a), b);
    scc.add_edge(NOT(b), a);
  }
  void add_true(int a) { add_or(a, a); }
  void add_false(int a) { add_or(NOT(a), NOT(a)); }
  void add_xor(int a, int b) { add_or(a, b); add_or(NOT(a), NOT(b)); }
  pair<bool, vector<bool>> solve() {
    auto comp = scc.scc(); vector<bool> ans(n);
    for (int i = 0; i < 2*n; i += 2) {
      if (comp[i] == comp[i+1]) return {false, {}};
      ans[i/2] = (comp[i] > comp[i+1]);
    }
    return {true, ans};
  }
};
```

6 Data Structures

```
// dynamic prefix sums O(log n) queries and updates
// add(i, v) = add v to A[i] | i in [1, n]
// query(i) = range sum [1, i]
struct fenwick {
  int n; vector<int> A;
  fenwick(int n) : n(n+1), A(n+1) { }
  void add(int i, int v) { while (i < n) A[i] += v, i += i & -i; }
  int query(int i) { int s = 0; while (i > 0) s += A[i], i -= i & -i; return s; }
};
```

```
// found on codeforces blog
// short non-recursive implementation.
template<typename T>
struct segment {
  int n; T id; function<T(T, T)> op;
  vector<T> S;
  segment(int n, T id, function<T(T, T)> op, const vector<T>& A = {})
    : n(n), id(id), op(op), S(2*n, id) {
    for (int i = 0; i < sz(A); i++) S[n+i] = A[i];
    for (int i = n-1; i > 0; i--) S[i] = op(S[2*i], S[2*i+1]);
  }
  // add v to A[x] (can change to = for setting)
  void update(int x, T v) {
    for (S[x += n] += v; x > 1; x /= 2)
      S[x/2] = op(S[x], S[x^1]);
  }
  // query range A[l], ... , A[r-1].
  T query(int l, int r) {
    int ans = id;
    for (l += n, r += n; l < r; l /= 2, r /= 2) {
      if (l & 1) ans = op(ans, S[l++]);
      if (r & 1) ans = op(ans, S[--r]);
    }
    return ans;
  }
};

// examples
int n = 7;
vector<int> A(n, 1);
segment<int> stadd(n, 0, [] (int a, int b) { return a + b; });
segment<int> stmin(n, 1<<30, [] (int a, int b) { return min(a, b); }, A);
segment<int> stmax(n, -(1<<30), [] (int a, int b) { return max(a, b); }, A);
```

```
// segment tree with lazy prop, log(n) range query and range update.
// st.update(l, r, v) -> apply(i, v) where i ranges in [l, r]
// st.query(l, r) -> compute op of range [l, r]
// think about non-commutative ops!!
```

```
template<typename T>
struct segment {

  // these will work for min/max query and range add.
  // most other ops will require modification here.
  void apply(int i, int v) {
    S[i] += v;
    D[i] += v;
  }

  void prop(int i) {
    if (depth(i) != d and D[i]) {
      apply(2*i+1, D[i]);
      apply(2*i, D[i]);
      D[i] = 0;
    }
  }

  // initialize tree with size n, op: (T, T) -> (T), identity value and optional
       initial data.
  int n, d; T id; function<T(T, T)>op;
  vector<int> L, R, D; vector<T> S;
  int depth(int i) { return 31 - __builtin_clz(i); }
  segment(int n, T id, function<T(T, T)> op, const vector<T>& A = {}) : n(n), d(depth(
       n) + (n != 1 << depth(n))),
  id(id), op(op), L(1 << (d+1), 0), R(1 << (d+1), 0), D(1 << (d+1), 0), S(1 << (d+1),
       id) {
    for (int i = 0; i <= d; i++)
      for (int j = (1 << i); j < (1 << (i+1)); j++)
        L[j] = (j % (1 << i)) * (1 << (d - i)),
        R[j] = L[j] + (1 << (d - i)) - 1;
    for (int i = 0; i < sz(A); i++) S[(1<<d)+i] = A[i];
    for (int i = (1 << d) - 1; i > 0; i--) S[i] = op(S[2*i], S[2*i+1]);
  }
  // update range [l, r]
  void update(int l, int r, int v, int i = 1) {
    if (r < l) return;
    if (L[i] == l and R[i] == r) apply(i, v);
    else {
      prop(i);
      update(l, min(r, R[2*i]), v, 2*i);
      update(max(l, L[2*i+1]), r, v, 2*i+1);
      S[i] = op(S[2*i], S[2*i+1]);
    }
  }
  // query op in range [l, r]
  T query(int l, int r, int i = 1) {
    if (r < l) return id;
    if (L[i] == l and R[i] == r) return S[i];
    else {
      prop(i);
      return op(query(l, min(r, R[2*i]), 2*i), query(max(l, L[2*i+1]), r, 2*i+1));
    }
  }
};

// example
int n = 1 << 20;
vector<int> A(n, 0);
segment<int> stmin(n, 1 << 30, [] (int a, int b) { return min(a, b); }, A);
segment<int> stmax(n, -(1 << 30), [] (int a, int b) { return max(a, b); }, A);
```

```
struct UF {
  int n; vector<int> A;
  UF (int n) : n(n), A(n) { iota(begin(A), end(A), 0); }
  int find (int a) { return a == A[a] ? a : A[a] = find(A[a]); }
  bool connected (int a, int b) { return find(a) == find(b); }
  void merge (int a, int b) { A[find(b)] = find(a); }
};
```

```
/* add lines of the form y = ax + b
 * query maximum value at point x
 * both add and query run in O(log n)
 */
template<typename T> struct DynamicHull {
  struct Line {
    typedef typename multiset<Line>::iterator It;
    T a, b; mutable It me, endit, none;
    Line(T a, T b, It endit) : a(a), b(b), endit(endit) {}
    bool operator<(const Line& rhs) const {
      if (rhs.endit != none) return a < rhs.a;
      if (next(me) == endit) return 0;
      return (b - next(me)->b) < (next(me)->a - a) * rhs.a;
    }
  };
  multiset<Line> lines;
  void add(T a, T b) {
    auto bad = [&](auto y) {
      auto z = next(y);
      if (y == lines.begin()) {
        if (z == lines.end()) return false;
        return y->a == z->a and z->b >= y->b;
      }
      auto x = prev(y);
      if (z == lines.end()) return y->a == x->a and x->b >= y->b;
      return (x->b-y->b) * (z->a-y->a) >= (y->b-z->b) * (y->a-x->a);
    };
    auto it = lines.emplace(a, b, lines.end()); it->me = it;
    if (bad(it)) { lines.erase(it); return; }
    while (next(it) != lines.end() and bad(next(it))) lines.erase(next(it));
    while (it != lines.begin() and bad(prev(it))) lines.erase(prev(it));
  }
  T query(T x) {
    auto it = lines.lower_bound(Line{x, 0, {}});
    return it->a * x + it->b;
  }
};
```

```
// croot = root of centroid tree
// par[v] = parent of v in centroid tree
// cadj[v] = decendants of v in centroid tree
struct Centroid {
  int n, cnt = 0, croot; vector<vector<int>> adj, cadj; vector<int> par, mark, size;
  Centroid(int n) : n(n), adj(n), cadj(n), par(n, -1), mark(n), size(n) {}
  void add_edge(int u, int v) { adj[u].push_back(v), adj[v].push_back(u); }
  int dfs(int u, int p) {
    size[u] = 1;
    for (int v : adj[u]) if (v != p and !mark[v]) dfs(v, u), size[u] += size[v];
    return size[u];
  }
  int find_centroid(int u, int p, int s) {
    for (int v : adj[u]) if (v != p and !mark[v])
      if (size[v] * 2 > s) return find_centroid(v, u, s);
    return u;
  }
  int find_centroid(int src) { return find_centroid(src, -1, dfs(src, -1)); }
  int decompose(int src = 0) {
    int c = find_centroid(src); mark[c] = 1;
    for (int u : adj[c]) if (!mark[u]) {
```

```
      int v = decompose(u);
      cadj[c].push_back(v), par[v] = c;
    }
    return croot = c;
  }
};
```

```
// MinQueue: maintain a standard queue while being able to query the min element
//  Constant time (amortized) per push/pop operation can be changed to maintain
//  max (or both min/max). No checks for empty queues anywhere!

class MinQueue {
private:
  stack<pair<int,int> > s1;   stack<int> s2;   int m1, m2;

  void move() {
    if (!s1.empty()) return;
    while (!s2.empty()) {
      s1.push(make_pair(s2.top(), m1));
      m1 = ::min(s2.top(), m1);
      s2.pop();
    }
    m2 = INT_MAX;          // min of empty queue
  }

public:
  // whatever the min of an empty queue should be
  MinQueue() : m1(INT_MAX), m2(INT_MAX) { }

  int min()     const { return ::min(m1, m2); }
  bool empty() const { return s1.empty() && s2.empty();  }
  void push(int x)    { m2 = ::min(m2, x); s2.push(x); }
  int front()         { move(); return s1.top().first; }
  void pop()          { move(); m1 = s1.top().second; s1.pop(); }
};
```

7 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
}
```

```
const string alphabet = "abcdefghijklmnopqrstuvwxyz";
const int s = 26;
const int logn = 20;
const int MAXNODES = 202020;
int index(char c) { return (int) alphabet.find(c); }

struct trie {
  int n; vector<vector<int>> A;
  trie() : n(1), A(MAXNODES, vector<int>(s+1, 0)) { }

  // returns vertex of last char in w
  int add(string w, int v) {
    int i = 0, j = 0;
    int l = sz(w);
    while (j < l) {
      int& k = A[i][index(w[j])];
      if (k != 0) i = k, j++;
      else i = k = n++, j++;
    }
    A[i][s] = v;
    return i;
  }

  // returns value of w if it exists
  int find(string w) {
    int i = 0;
    for (auto& l : w){
      int c = index(l);
      i = A[i][c];
      if (!i) return -1;
    }
    return A[i][s];
  }

  // D[i] = depth of node i
  vector<int> D;
  void dfs(int v, int d) {
    D[v] = d;
    for (int i = 0; i < s; i++) {
      if (A[v][i]) dfs(A[v][i], d+1);
    }
  }

  // P[i][j] = the node that is 2^j levels above i
  vector<vector<int>> P;
  void initlca() {
    D.assign(n, -1); dfs(0, 0);
    P.assign(n, vector<int>(logn, 0));
    for (int i = 0; i < n; i++)
      for (int j = 0; j < s; j++)
        if (A[i][j]) P[A[i][j]][0] = i;
    for (int i = 0; i < n; i++)
      for (int j = 1; j < logn; j++)
        P[i][j] = P[P[i][j-1]][j-1];
  }

  int lca(int a, int b) {
    if (D[b] > D[a]) swap(a, b);
    for (int j = logn-1; D[a] > D[b]; j--)
      while (D[P[a][j]] >= D[b]) a = P[a][j];
    assert(D[a] == D[b]);
    if (a == b) return a;
    for (int j = logn-1; j >= 0; j--)
```

```
    if (P[a][j] != P[b][j])
      a = P[a][j], b = P[b][j];
  // lca doesnt exist ?
  assert(P[a][0] == P[b][0]);
  return P[a][0];
 }
};
```

```
struct SA {
  int n; string str; vector<int> sarray, lcp;
  suffix_array(string s) : n(s.size()), str(move(s)) { build_sarray(); build_lcp(); }
  void bucket(vector<int>& a, vector<int>& b, vector<int>& r, int n, int K, int off=0)
      {
    vector<int> 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];
  }
  void build_sarray() {
    sarray.assign(n, 0); vector<int> r(2*n, 0), sa(2*n), tmp(2*n); if (n <= 1) return;
    for (int i = 0; i < n; i++) r[i] = (int) str[i] - CHAR_MIN + 1, sa[i] = i;
    for (int k = 1; k < n; k *= 2) {
      bucket(sa, tmp, r, n, max(n, 256), k), bucket(tmp, sa, r, n, max(n, 256), 0);
      tmp[sa[0]] = 1;
      for (int i = 1; i < n; i++) {
        tmp[sa[i]] = tmp[sa[i-1]];
        if ((r[sa[i]] != r[sa[i-1]]) || (r[sa[i]+k] != r[sa[i-1]+k])) tmp[sa[i]]++;
      }
      copy(tmp.begin(), tmp.begin()+n, r.begin());
    }
    copy(sa.begin(), sa.begin()+n, sarray.begin());
  }
  void build_lcp() {
    int h = 0; vector<int> rank(n); lcp.assign(n, 0);
    for (int i = 0; i < n; i++) rank[sarray[i]] = i;
    for (int i = 0; i < n; i++) {
      if (rank[i] > 0) {
        int j = sarray[rank[i] - 1];
        while (i + h < n and j + h < n and str[i+h] == str[j+h]) h++;
        lcp[rank[i]] = h;
      }
      if (h > 0) h--;
    }
  }
};
```

```
// Find lex least rotation of a string, and smallest period of a string: O(n)
// pos = start of lex least rotation, period = the period
void compute(string s, int &pos, int &period) {
  s += s;
  int len = s.length(), i = 0, j = 1;
  for (int k = 0; i+k < len && j+k < len; k++) {
    if (s[i+k] > s[j+k]) {
      i = max(i+k+1, j+1);      k = -1;
    } else if (s[i+k] < s[j+k]) {
      j = max(j+k+1, i+1);      k = -1;
    }
  }
  pos = min(i, j);
  period = (i > j) ? i - j : j - i;
}
```

8 Algorithms and Misc

```
// 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;
}
```

```
// -- 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);
}
```

```
// 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];
}
```

```
// 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;
}
```

```
primes for hashing:
1e9+7, 1e9+9,  1e9+21, 1e9+33, 1e3+9, 1e3+13, 1e3+19, 1e3+21
999999733, 999999491, 999999193, 999996901, 999996227
```

9 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.

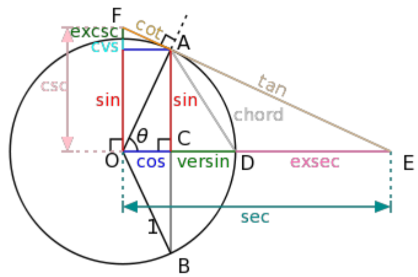
Cosine law: $c^2 = a^2 + b^2 - 2ab\cos(\gamma)$

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

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



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.

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

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

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 $b + 2c + 3d + 4e + ... = n$ and the number of nonnegative solutions to $2c + 3d + 4e + ... \leq n$)

	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

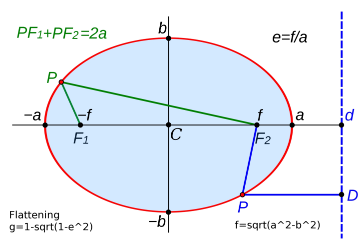
Lagrange Interpolation: Given $(x_0, y_0), \cdots, (x_n, y_n)$, the polynomial is:

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

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

9.0.1 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 $q|a_n$.

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

9.1 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

10 Tips

If You Are Stuck, Read These!

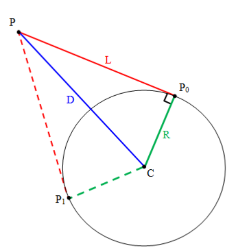
- Can you write the question as a whole bunch of inequalities? (Simplex?)
- Can you hash to reduce time? (Normally cuts a factor of N)
- Can you only have one “item” on a location at a time? Can only one “item” move through a hallway at one time?
- Can you break the problem into two disjoint sets? (Even/Odd, 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 $f(t) = v \cdot t +$ 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.

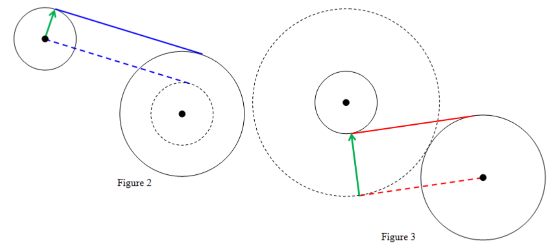
General Things

- **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

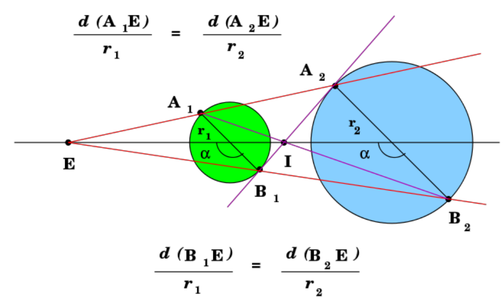


Now Intersect two circles: (C,R) and $(P, L = \sqrt{D^2 - R^2})$.



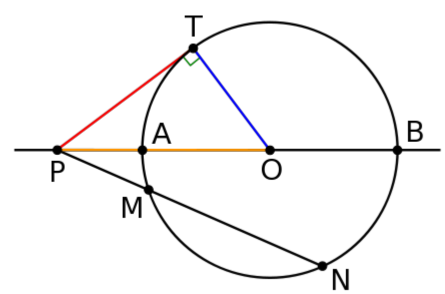
Two circles of radii $r_1 \le r_2$. For **outer tangent** (Left Picture), make a circle of radius $r_2 - r_1$ around C_2 (dashed circle) and find tangent lines from C_1 (dashed blue line), then translate it r_1 units (solid blue line). For **inner tangent** (Right Picture), make a circle of radius $r_1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



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

- Type this command: `source .bashrc`