UNdead's notebook (2017)

Contents

1 Graphs

1.1 Articulation points

```
/*Tarjan Algorithm to find articulation points
 * single dfs O(|v| + |e|)
 * visited =[false]
 * disc = [0]
 * low = [0]
 * parent = [-1]
 * ap = [false] */

void articulation(vector<vector<int> > G, int u, bool visited[], int disc[], int low[], int parent
      [], bool ap[]) {

    static int time = 0;

    int children = 0;
    visited[u] = true;

    disc[u] = low[u] = ++time;

    for(int i = 0; i < G[u].size(); i++){
        int v = G[u][i];

        if(!visited[v]){
            children++;
            parent[v] = u;
            articulation(G, v, visited, disc, low, ap);

            low[u] = min(low[u], low[v]);

            if(parent[u] == -1 && children > 1) ap[u] = true;

            if(parent[u] != -1 && low[v] >= disc[u]) ap[u] = true;
        }
        else if(v != parent[u])
            low[u] = min(low[u], disc[v]);
    }
}
```

1.2 Biconected graph

```
/* Tarjan Algorithm to find Biconnected graph
 * single dfs O(|v| + |e|)
 * visited =[false]
 * disc = [0]
 * low = [0]
 * parent = [-1]  */

bool isBiconnected(vector<vector<int> > G, int u, bool visited[], int disc[], int low[], int parent
      []){
    static int time = 0;

    int children = 0;

    visited[u] = true;

    disc[u] = low[u] = ++time;

    for(int i = 0; i<G[u].size(); i++){
        int v = G[u][i];
```

```
        if(!visited[v]){
            children++;
            parent[v] = u;

            if (isBiconnected(G, v, visited, disc, low, parent)) return true;

            low[u] = min(low[u], low[v]);

            if(parent[u] == -1 && children > 1) return true;

            if(parent[u] != -1 && low[v] >= disc[u]) return true;
        }
        else if(v != parent[v]) low[u] = min(low[u], disc[v]);
    }
    return false;
}
```

1.3 Bridges

```
/* Tarjan Algorithm to find bridges
 * single dfs O(|v| + |e|)
 * visited =[false]
 * disc = [0]
 * low = [0]
 * parent = [-1]  */

 void bridges(vector<vector<int> > G, int u, bool visited[], int disc[], int low[], int parent[],
       priority_queue< pair<int, int> > *bridge) {

    static int time = 0;

    int children = 0;
    visited[u] = true;

    disc[u] = low[u] = ++time;

    for(int i = 0; i < G[u].size(); i++){
        int v = G[u][i];

        if(!visited[v]){
            children++;
            parent[v] = u;
            bridges(G, v, visited, disc, low, bridge);

            low[u] = min(low[u], low[v]);

            if(low[v] > disc[u])bridge->push({u,v});
        }
        else if(v != parent[u])
            low[u] = min(low[u], disc[v]);
    }
}
```

1.4 Tarjan SCC

```
/* Tarjan Algorithm to find connected components
 * single dfs O(|v| + |e|)
 * visited =[false]
 * disc = [0]
 * low = [0]
 * parent = [-1]  */

 void dfsSCC(vector<vector<int> > G, int u, int disc[], int low[], stack<int> *st, bool stackMember
       []){
     static int time = 0;

     disc[u] = low[u] = ++time;

     st->push(u);
     stackmember[u] = true;

     for(int i = 0; i < G[u].size(); i++){
         int v = G[u][i];

         if(disc[v] == -1){

             dfsSCC(G, v, disc, low, st, stackmember);

             low[u] = min(low[u], low[v]);
         }
         else if(stackmember[v] == true) low[u] = min(low[u], disc[v]);
     }

     int w = 0;

     if(low[u] == disc[u]){

         while(st->top() != u){
             w = st->top();
             cout<<w<<" ";
             stackmember[w] = false;
             st->pop();
         }

         w = st->top();
         cout<<w<<"\n";
         stackmember[w] = false;
         st->pop();
     }
 }

 void scc(G){
    int *disc = new int[V];
    int *low = new int[V];
    bool *stackMember = new bool[V];
    stack<int> *st = new stack<int>();

    memset(disc, -1, sizeof(disc));
    memset(low, 0, sizeof(low));
    memset(stackMember, false, sizeof(stackMember));

    for(int i = 0; i < G.size(); i++)
        if(disc[i] == -1) dfsScc(G, i, disc, low, st, stackMember);
 }
```

1.5 Fast Dijkstra's algorithm

```
// Implementation of Dijkstra's algorithm using adjacency lists
// and priority queue for efficiency.
//
// Running time: O(|E| log |V|)

#include <queue>
#include <cstdio>

using namespace std;
const int INF = 2000000000;
typedef pair<int, int> PII;

int main() {

        int N, s, t;
        scanf("%d%d%d", &N, &s, &t);
        vector<vector<PII> > edges(N);
        for (int i = 0; i < N; i++) {
                int M;
                scanf("%d", &M);
                for (int j = 0; j < M; j++) {
                        int vertex, dist;
                        scanf("%d%d", &vertex, &dist);
                        edges[i].push_back(make_pair(dist, vertex)); // note order of arguments here
                }
        }

        // use priority queue in which top element has the "smallest" priority
        priority_queue<PII, vector<PII>, greater<PII> > Q;
        vector<int> dist(N, INF), dad(N, -1);
        Q.push(make_pair(0, s));
        dist[s] = 0;
        while (!Q.empty()) {
                PII p = Q.top();
                Q.pop();
                int here = p.second;
                if (here == t) break;
                if (dist[here] != p.first) continue;

                for (vector<PII>::iterator it = edges[here].begin(); it != edges[here].end(); it++)
                      {
```

```
				if (dist[here] + it->first < dist[it->second]) {
					dist[it->second] = dist[here] + it->first;
					dad[it->second] = here;
					Q.push(make_pair(dist[it->second], it->second));
				}
			}
		}

	printf("%d\n", dist[t]);
	if (dist[t] < INF)
		for (int i = t; i != -1; i = dad[i])
			printf("%d%c", i, (i == s ? '\n' : ' '));
	return 0;
}

/*
Sample input:
5 0 4
2 1 2 3 1
2 2 4 4 5
3 1 4 3 3 4 1
2 0 1 2 3
2 1 5 2 1

Expected:
5
4 2 3 0
*/
```

1.6 Bellman Ford

```
// This function runs the Bellman-Ford algorithm for single source
// shortest paths with negative edge weights.  The function returns
// false if a negative weight cycle is detected.  Otherwise, the
// function returns true and dist[i] is the length of the shortest
// path from start to i.
//
// Running time: O(|V|^3)
//
//   INPUT:   start, w[i][j] = cost of edge from i to j
//   OUTPUT:  dist[i] = min weight path from start to i
//            prev[i] = previous node on the best path from the
//                      start node

#include <iostream>
#include <queue>
#include <cmath>
#include <vector>

using namespace std;

typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

typedef vector<int> VI;
typedef vector<VI> VVI;

bool BellmanFord (const VVT &w, VT &dist, VI &prev, int start){
  int n = w.size();
  prev = VI(n, -1);
  dist = VT(n, 1000000000);
  dist[start] = 0;

  for (int k = 0; k < n; k++){
    for (int i = 0; i < n; i++){
      for (int j = 0; j < n; j++){
        if (dist[j] > dist[i] + w[i][j]){
          if (k == n-1) return false;
          dist[j] = dist[i] + w[i][j];
          prev[j] = i;
        }
      }
    }
  }

  return true;
}
```

1.7 Eulerian Path

```
struct Edge;
typedef list<Edge>::iterator iter;

struct Edge
{
	int next_vertex;
	iter reverse_edge;

	Edge(int next_vertex)
		:next_vertex(next_vertex)
		{ }
};

const int max_vertices = ;
int num_vertices;
list<Edge> adj[max_vertices];		// adjacency list

vector<int> path;

void find_path(int v)
{
	while(adj[v].size() > 0)
	{
		int vn = adj[v].front().next_vertex;
		adj[vn].erase(adj[v].front().reverse_edge);
		adj[v].pop_front();
		find_path(vn);
	}
	path.push_back(v);
}

void add_edge(int a, int b)
{
	adj[a].push_front(Edge(b));
	iter ita = adj[a].begin();
	adj[b].push_front(Edge(a));
	iter itb = adj[b].begin();
	ita->reverse_edge = itb;
	itb->reverse_edge = ita;
}
```

1.8 Topological sort

```
char c[TAM];
int l[TAM];
int r[TAM];
int in[TAM];

//can be priority queue
queue<int> Q;

void reset(){
    memset(l, 0, sizeof l);
    memset(r, 0, sizeof r);
    memset(in, 0, sizeof in);
    memset(balls, 0, sizeof balls);
    c[0] = 'L';
}

void topo(vector<vector<int> > G, int u){
    while(!Q.empty()){
        u = Q.front();Q.pop();
        update(u);
        for(int i = 0; i <G[u].size(); i++){
            int v = G[u][i];
            in[v]--;
            if(in[v] == 0) Q.push(v);
        }
    }
}

int main(){
    ll n;
    int m;
    while(cin>>n>>m){
        reset();
        vector< vector<int> > G(m + 1);
```

```
        for(int i = 1; i <=m; i++){
            int u,v;
            cin>>c[i]>>u>>v;
            G[i].push_back(u);
            G[i].push_back(v);
            in[u] ++; in[v]++;
            l[i] = u;r[i] = v;
        }
        for(int i = 1; i <=m; i++){
            if(in[i] == 0)Q.push(i);
        }
        topo(G, 1);
    }
}
```

1.9 Kruskal's minimum spanning tree

```
/*
Uses Kruskal's Algorithm to calculate the weight of the minimum spanning
forest (union of minimum spanning trees of each connected component) of
a possibly disjoint graph, given in the form of a matrix of edge weights
(-1 if no edge exists). Returns the weight of the minimum spanning
forest (also calculates the actual edges - stored in T). Note: uses a
disjoint-set data structure with amortized (effectively) constant time per
union/find. Runs in O(E*log(E)) time.
*/

#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>

using namespace std;

typedef int T;

struct edge
{
  int u, v;
  T d;
};

struct edgeCmp
{
  int operator()(const edge& a, const edge& b) { return a.d > b.d; }
};

int find(vector <int>& C, int x) { return (C[x] == x) ? x : C[x] = find(C, C[x]); }

T Kruskal(vector <vector <T> >& w)
{
  int n = w.size();
  T weight = 0;

  vector <int> C(n), R(n);
  for(int i=0; i<n; i++) { C[i] = i; R[i] = 0; }

  vector <edge> T;
  priority_queue <edge, vector <edge>, edgeCmp> E;

  for(int i=0; i<n; i++)
    for(int j=i+1; j<n; j++)
      if(w[i][j] >= 0)
      {
        edge e;
        e.u = i; e.v = j; e.d = w[i][j];
        E.push(e);
      }

  while(T.size() < n-1 && !E.empty())
  {
    edge cur = E.top(); E.pop();

    int uc = find(C, cur.u), vc = find(C, cur.v);
    if(uc != vc)
    {
      T.push_back(cur); weight += cur.d;

      if(R[uc] > R[vc]) C[vc] = uc;
      else if(R[vc] > R[uc]) C[uc] = vc;
      else { C[vc] = uc; R[uc]++; }
    }
  }

  return weight;
}

int main()
{
  int wa[6][6] = {
    { 0, -1, 2, -1, 7, -1 },
    { -1, 0, -1, 2, -1, -1 },
    { 2, -1, 0, -1, 8, 6 },
    { -1, 2, -1, 0, -1, -1 },
    { 7, -1, 8, -1, 0, 4 },
    { -1, -1, 6, -1, 4, 0 } };

  vector <vector <int> > w(6, vector <int>(6));

  for(int i=0; i<6; i++)
    for(int j=0; j<6; j++)
      w[i][j] = wa[i][j];

  cout << Kruskal(w) << endl;
  cin >> wa[0][0];
}
```

1.10 Lowest Common ancestor

```
const int max_nodes, log_max_nodes;
int num_nodes, log_num_nodes, root;

vector<int> children[max_nodes];        // children[i] contains the children of node i
int A[max_nodes][log_max_nodes+1];      // A[i][j] is the 2^j-th ancestor of node i, or -1 if that
      ancestor does not exist
int L[max_nodes];                       // L[i] is the distance between node i and the root

// floor of the binary logarithm of n
int lb(unsigned int n)
{
    if(n==0)
        return -1;
    int p = 0;
    if (n >= 1<<16) { n >>= 16; p += 16; }
    if (n >= 1<< 8) { n >>=  8; p +=  8; }
    if (n >= 1<< 4) { n >>=  4; p +=  4; }
    if (n >= 1<< 2) { n >>=  2; p +=  2; }
    if (n >= 1<< 1) {           p +=  1; }
    return p;
}

void DFS(int i, int l)
{
    L[i] = l;
    for(int j = 0; j < children[i].size(); j++)
        DFS(children[i][j], l+1);
}

int LCA(int p, int q)
{
    // ensure node p is at least as deep as node q
    if(L[p] < L[q])
        swap(p, q);

    // "binary search" for the ancestor of node p situated on the same level as q
    for(int i = log_num_nodes; i >= 0; i--)
        if(L[p] - (1<<i) >= L[q])
            p = A[p][i];

    if(p == q)
        return p;

    // "binary search" for the LCA
    for(int i = log_num_nodes; i >= 0; i--)
        if(A[p][i] != -1 && A[p][i] != A[q][i])
        {
            p = A[p][i];
            q = A[q][i];
        }

    return A[p][0];
}

int main(int argc,char* argv[])
{
    // read num_nodes, the total number of nodes
    log_num_nodes=lb(num_nodes);
```

```
    for(int i = 0; i < num_nodes; i++)
    {
        int p;
        // read p, the parent of node i or -1 if node i is the root

        A[i][0] = p;
        if(p != -1)
            children[p].push_back(i);
        else
            root = i;
    }

    // precompute A using dynamic programming
    for(int j = 1; j <= log_num_nodes; j++)
        for(int i = 0; i < num_nodes; i++)
            if(A[i][j-1] != -1)
                A[i][j] = A[A[i][j-1]][j-1];
            else
                A[i][j] = -1;

    // precompute L
    DFS(root, 0);


    return 0;
}
```

2 Geometry

2.1 Geometry

```
// C++ routines for computational geometry.

#include <iostream>
#include <vector>
#include <cmath>
#include <cassert>

using namespace std;

double INF = 1e100;
double EPS = 1e-12;

struct PT {
  double x, y;
  PT() {}
  PT(double x, double y) : x(x), y(y) {}
  PT(const PT &p) : x(p.x), y(p.y)    {}
  PT operator + (const PT &p)  const { return PT(x+p.x, y+p.y); }
  PT operator - (const PT &p)  const { return PT(x-p.x, y-p.y); }
  PT operator * (double c)     const { return PT(x*c,   y*c  ); }
  PT operator / (double c)     const { return PT(x/c,   y/c  ); }
};

double dot(PT p, PT q)     { return p.x*q.x+p.y*q.y; }
double dist2(PT p, PT q)   { return dot(p-q,p-q); }
double cross(PT p, PT q)   { return p.x*q.y-p.y*q.x; }
ostream &operator<<(ostream &os, const PT &p) {
  os << "(" << p.x << "," << p.y << ")";
}

// rotate a point CCW or CW around the origin
PT RotateCCW90(PT p)   { return PT(-p.y,p.x); }
PT RotateCW90(PT p)    { return PT(p.y,-p.x); }
PT RotateCCW(PT p, double t) {
  return PT(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t));
}

// project point c onto line through a and b
// assuming a != b
PT ProjectPointLine(PT a, PT b, PT c) {
  return a + (b-a)*dot(c-a, b-a)/dot(b-a, b-a);
}

// project point c onto line segment through a and b
PT ProjectPointSegment(PT a, PT b, PT c) {
  double r = dot(b-a,b-a);
  if (fabs(r) < EPS) return a;
  r = dot(c-a, b-a)/r;
  if (r < 0) return a;
  if (r > 1) return b;
  return a + (b-a)*r;
}

// compute distance from c to segment between a and b
double DistancePointSegment(PT a, PT b, PT c) {
  return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}

// compute distance between point (x,y,z) and plane ax+by+cz=d
double DistancePointPlane(double x, double y, double z,
                          double a, double b, double c, double d)
{
  return fabs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);
}

// determine if lines from a to b and c to d are parallel or collinear
bool LinesParallel(PT a, PT b, PT c, PT d) {
  return fabs(cross(b-a, c-d)) < EPS;
}

bool LinesCollinear(PT a, PT b, PT c, PT d) {
  return LinesParallel(a, b, c, d)
      && fabs(cross(a-b, a-c)) < EPS
      && fabs(cross(c-d, c-a)) < EPS;
}

// determine if line segment from a to b intersects with
// line segment from c to d
bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
  if (LinesCollinear(a, b, c, d)) {
    if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
      dist2(b, c) < EPS || dist2(b, d) < EPS) return true;
    if (dot(c-a, c-b) > 0 && dot(d-a, d-b) > 0 && dot(c-b, d-b) > 0)
      return false;
    return true;
  }
  if (cross(d-a, b-a) * cross(c-a, b-a) > 0) return false;
  if (cross(a-c, d-c) * cross(b-c, d-c) > 0) return false;
  return true;
}

// compute intersection of line passing through a and b
// with line passing through c and d, assuming that unique
// intersection exists; for segment intersection, check if
// segments intersect first
PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {
  b=b-a; d=c-d; c=c-a;
  assert(dot(b, b) > EPS && dot(d, d) > EPS);
  return a + b*cross(c, d)/cross(b, d);
}

// compute center of circle given three points
PT ComputeCircleCenter(PT a, PT b, PT c) {
  b=(a+b)/2;
  c=(a+c)/2;
  return ComputeLineIntersection(b, b+RotateCW90(a-b), c, c+RotateCW90(a-c));
}

// determine if point is in a possibly non-convex polygon (by William
// Randolph Franklin); returns 1 for strictly interior points, 0 for
// strictly exterior points, and 0 or 1 for the remaining points.
// Note that it is possible to convert this into an *exact* test using
// integer arithmetic by taking care of the division appropriately
// (making sure to deal with signs properly) and then by writing exact
// tests for checking point on polygon boundary
bool PointInPolygon(const vector<PT> &p, PT q) {
  bool c = 0;
  for (int i = 0; i < p.size(); i++){
    int j = (i+1)%p.size();
    if ((p[i].y <= q.y && q.y < p[j].y ||
      p[j].y <= q.y && q.y < p[i].y) &&
      q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) / (p[j].y - p[i].y))
      c = !c;
  }
  return c;
}

// determine if point is on the boundary of a polygon
bool PointOnPolygon(const vector<PT> &p, PT q) {
  for (int i = 0; i < p.size(); i++)
    if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.size()], q), q) < EPS)
      return true;
    return false;
}

// compute intersection of line through points a and b with
// circle centered at c with radius r > 0
vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r) {
  vector<PT> ret;
```

```
  b = b-a;
  a = a-c;
  double A = dot(b, b);
  double B = dot(a, b);
  double C = dot(a, a) - r*r;
  double D = B*B - A*C;
  if (D < -EPS) return ret;
  ret.push_back(c+a+b*(-B+sqrt(D+EPS))/A);
  if (D > EPS)
    ret.push_back(c+a+b*(-B-sqrt(D))/A);
  return ret;
}

// compute intersection of circle centered at a with radius r
// with circle centered at b with radius R
vector<PT> CircleCircleIntersection(PT a, PT b, double r, double R) {
  vector<PT> ret;
  double d = sqrt(dist2(a, b));
  if (d > r+R || d+min(r, R) < max(r, R)) return ret;
  double x = (d*d-R*R+r*r)/(2*d);
  double y = sqrt(r*r-x*x);
  PT v = (b-a)/d;
  ret.push_back(a+v*x + RotateCCW90(v)*y);
  if (y > 0)
    ret.push_back(a+v*x - RotateCCW90(v)*y);
  return ret;
}

// This code computes the area or centroid of a (possibly nonconvex)
// polygon, assuming that the coordinates are listed in a clockwise or
// counterclockwise fashion.  Note that the centroid is often known as
// the "center of gravity" or "center of mass".
double ComputeSignedArea(const vector<PT> &p) {
  double area = 0;
  for(int i = 0; i < p.size(); i++) {
    int j = (i+1) % p.size();
    area += p[i].x*p[j].y - p[j].x*p[i].y;
  }
  return area / 2.0;
}

double ComputeArea(const vector<PT> &p) {
  return fabs(ComputeSignedArea(p));
}

PT ComputeCentroid(const vector<PT> &p) {
  PT c(0,0);
  double scale = 6.0 * ComputeSignedArea(p);
  for (int i = 0; i < p.size(); i++){
    int j = (i+1) % p.size();
    c = c + (p[i]+p[j])*(p[i].x*p[j].y - p[j].x*p[i].y);
  }
  return c / scale;
}

// tests whether or not a given polygon (in CW or CCW order) is simple
bool IsSimple(const vector<PT> &p) {
  for (int i = 0; i < p.size(); i++) {
    for (int k = i+1; k < p.size(); k++) {
      int j = (i+1) % p.size();
      int l = (k+1) % p.size();
      if (i == l || j == k) continue;
      if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
        return false;
    }
  }
  return true;
}

int main() {

  // expected: (-5,2)
  cerr << RotateCCW90(PT(2,5)) << endl;

  // expected: (5,-2)
  cerr << RotateCW90(PT(2,5)) << endl;

  // expected: (-5,2)
  cerr << RotateCCW(PT(2,5),M_PI/2) << endl;

  // expected: (5,2)
  cerr << ProjectPointLine(PT(-5,-2), PT(10,4), PT(3,7)) << endl;

  // expected: (5,2) (7.5,3) (2.5,1)
  cerr << ProjectPointSegment(PT(-5,-2), PT(10,4), PT(3,7)) << " "
       << ProjectPointSegment(PT(7.5,3), PT(10,4), PT(3,7)) << " "
       << ProjectPointSegment(PT(-5,-2), PT(2.5,1), PT(3,7)) << endl;

  // expected: 6.78903
  cerr << DistancePointPlane(4,-4,3,2,-2,5,-8) << endl;

  // expected: 1 0 1
  cerr << LinesParallel(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
       << LinesParallel(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
       << LinesParallel(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;

  // expected: 0 0 1
  cerr << LinesCollinear(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
       << LinesCollinear(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
       << LinesCollinear(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;

  // expected: 1 1 1 0
  cerr << SegmentsIntersect(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) << " "
       << SegmentsIntersect(PT(0,0), PT(2,4), PT(4,3), PT(0,5)) << " "
       << SegmentsIntersect(PT(0,0), PT(2,4), PT(2,-1), PT(-2,1)) << " "
       << SegmentsIntersect(PT(0,0), PT(2,4), PT(5,5), PT(1,7)) << endl;

  // expected: (1,2)
  cerr << ComputeLineIntersection(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) << endl;

  // expected: (1,1)
  cerr << ComputeCircleCenter(PT(-3,4), PT(6,1), PT(4,5)) << endl;

  vector<PT> v;
  v.push_back(PT(0,0));
  v.push_back(PT(5,0));
  v.push_back(PT(5,5));
  v.push_back(PT(0,5));

  // expected: 1 1 1 0 0
  cerr << PointInPolygon(v, PT(2,2)) << " "
       << PointInPolygon(v, PT(2,0)) << " "
       << PointInPolygon(v, PT(0,2)) << " "
       << PointInPolygon(v, PT(5,2)) << " "
       << PointInPolygon(v, PT(2,5)) << endl;

  // expected: 0 1 1 1 1
  cerr << PointOnPolygon(v, PT(2,2)) << " "
       << PointOnPolygon(v, PT(2,0)) << " "
       << PointOnPolygon(v, PT(0,2)) << " "
       << PointOnPolygon(v, PT(5,2)) << " "
       << PointOnPolygon(v, PT(2,5)) << endl;

  // expected: (1,6)
  //           (5,4) (4,5)
  //           blank line
  //           (4,5) (5,4)
  //           blank line
  //           (4,5) (5,4)
  vector<PT> u = CircleLineIntersection(PT(0,6), PT(2,6), PT(1,1), 5);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
  u = CircleLineIntersection(PT(0,9), PT(9,0), PT(1,1), 5);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
  u = CircleCircleIntersection(PT(1,1), PT(10,10), 5, 5);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
  u = CircleCircleIntersection(PT(1,1), PT(8,8), 5, 5);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
  u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 10, sqrt(2.0)/2.0);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
  u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 5, sqrt(2.0)/2.0);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;

  // area should be 5.0
  // centroid should be (1.1666666, 1.166666)
  PT pa[] = { PT(0,0), PT(5,0), PT(1,1), PT(0,5) };
  vector<PT> p(pa, pa+4);
  PT c = ComputeCentroid(p);
  cerr << "Area: " << ComputeArea(p) << endl;
  cerr << "Centroid: " << c << endl;

  return 0;
}
```

2.2 Geometry (Java)

```
// In this example, we read an input file containing three lines, each
// containing an even number of doubles, separated by commas.  The first two
// lines represent the coordinates of two polygons, given in counterclockwise
// (or clockwise) order, which we will call "A" and "B".  The last line
// contains a list of points, p[1], p[2], ...
//
// Our goal is to determine:
//   (1) whether B - A is a single closed shape (as opposed to multiple shapes)
//   (2) the area of B - A
//   (3) whether each p[i] is in the interior of B - A
```

```
//
// INPUT:
//   0 0 10 0 0 10
//   0 0 10 10 10 0
//   8 6
//   5 1
//
// OUTPUT:
//   The area is singular.
//   The area is 25.0
//   Point belongs to the area.
//   Point does not belong to the area.

import java.util.*;
import java.awt.geom.*;
import java.io.*;

public class JavaGeometry {

    // make an array of doubles from a string
    static double[] readPoints(String s) {
        String[] arr = s.trim().split("\\s++");
        double[] ret = new double[arr.length];
        for (int i = 0; i < arr.length; i++) ret[i] = Double.parseDouble(arr[i]);
        return ret;
    }

    // make an Area object from the coordinates of a polygon
    static Area makeArea(double[] pts) {
        Path2D.Double p = new Path2D.Double();
        p.moveTo(pts[0], pts[1]);
        for (int i = 2; i < pts.length; i += 2) p.lineTo(pts[i], pts[i+1]);
        p.closePath();
        return new Area(p);
    }

    // compute area of polygon
    static double computePolygonArea(ArrayList<Point2D.Double> points) {
        Point2D.Double[] pts = points.toArray(new Point2D.Double[points.size()]);
        double area = 0;
        for (int i = 0; i < pts.length; i++){
            int j = (i+1) % pts.length;
            area += pts[i].x * pts[j].y - pts[j].x * pts[i].y;
        }
        return Math.abs(area)/2;
    }

    // compute the area of an Area object containing several disjoint polygons
    static double computeArea(Area area) {
        double totArea = 0;
        PathIterator iter = area.getPathIterator(null);
        ArrayList<Point2D.Double> points = new ArrayList<Point2D.Double>();

        while (!iter.isDone()) {
            double[] buffer = new double[6];
            switch (iter.currentSegment(buffer)) {
            case PathIterator.SEG_MOVETO:
            case PathIterator.SEG_LINETO:
                points.add(new Point2D.Double(buffer[0], buffer[1]));
                break;
            case PathIterator.SEG_CLOSE:
                totArea += computePolygonArea(points);
                points.clear();
                break;
            }
            iter.next();
        }
        return totArea;
    }

    // notice that the main() throws an Exception -- necessary to
    // avoid wrapping the Scanner object for file reading in a
    // try { ... } catch block.
    public static void main(String args[]) throws Exception {

        Scanner scanner = new Scanner(new File("input.txt"));
        // also,
        //   Scanner scanner = new Scanner (System.in);

        double[] pointsA = readPoints(scanner.nextLine());
        double[] pointsB = readPoints(scanner.nextLine());
        Area areaA = makeArea(pointsA);
        Area areaB = makeArea(pointsB);
        areaB.subtract(areaA);
        // also,
        //   areaB.exclusiveOr (areaA);
        //   areaB.add (areaA);
        //   areaB.intersect (areaA);

        // (1) determine whether B - A is a single closed shape (as
        //     opposed to multiple shapes)
        boolean isSingle = areaB.isSingular();
        // also,
        //   areaB.isEmpty();

        if (isSingle)
            System.out.println("The area is singular.");
        else
            System.out.println("The area is not singular.");

        // (2) compute the area of B - A
        System.out.println("The area is " + computeArea(areaB) + ".");

        // (3) determine whether each p[i] is in the interior of B - A
        while (scanner.hasNextDouble()) {
            double x = scanner.nextDouble();
            assert(scanner.hasNextDouble());
            double y = scanner.nextDouble();

            if (areaB.contains(x,y)) {
                System.out.println ("Point belongs to the area.");
            } else {
                System.out.println ("Point does not belong to the area.");
            }
        }

        // Finally, some useful things we didn't use in this example:
        //
        //   Ellipse2D.Double ellipse = new Ellipse2D.Double (double x, double y,
        //                                                    double w, double h);
        //
        //     creates an ellipse inscribed in box with bottom-left corner (x,y)
        //     and upper-right corner (x+y,w+h)
        //
        //   Rectangle2D.Double rect = new Rectangle2D.Double (double x, double y,
        //                                                     double w, double h);
        //
        //     creates a box with bottom-left corner (x,y) and upper-right
        //     corner (x+y,w+h)
        //
        // Each of these can be embedded in an Area object (e.g., new Area (rect)).

    }
}
```

2.3 Convex Hull Monotone Chain

```
#include<iostream>
#include<algorithm>
#include<complex>
#include<cstdio>
#include<iomanip>
#include<vector>
#define x real()
#define y imag()
#define dot(A,B)                real(conj((A))*(B))
#define cross(A,B)              imag(conj((A))*(B))
#define PI 3.1415926
#define EPS 1e-9

using namespace std;
typedef double lf;
typedef complex<lf> pt;

istream& operator >> ( istream& in, pt& p ) {
  lf a,n; in >> a >> n;
  p = pt(a,n); return in;
}

bool cmp(pt &p, pt &q) {
    if(p.x != q.x) return p.x < q.x;
    return p.y < q.y;
}

bool is_zero( lf x ){
   return -EPS <= x && x <= EPS;
}

inline bool same ( lf a, lf b ) {
  return a+EPS > b && b+EPS > a;
}

int ccw(pt& p1, pt& p2, pt& p3) {
  lf ans = (cross(p1 - p3, p2 - p3));
  if(-EPS <= ans && ans <= EPS)
```

```
    return 0;
  else if(ans <= -EPS)
    return -1;
  else
    return 1;
}

lf dist ( pt A, pt B ) { return abs(A-B); }


vector<pt> convex_hull(vector<pt> P){
        int n = P.size(); int k = 0;
        vector<pt> H(2*n);

        sort(P.begin(), P.end(), cmp);

        for (int i = 0; i < n; i++) {
                while (k >= 2 && ccw(H[k-2], H[k-1], P[i]) == 1) k--;
                H[k++] = P[i];
        }


        for (int i = n-2, t = k+1; i >= 0; i--) {
                while (k >= t && ccw(H[k-2], H[k-1], P[i]) == 1) k--;
                H[k++] = P[i];
        }

        if(n>1)
    k--;
        H.resize(k);
        return H;
}


int main(){
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  #ifdef LOCAL
  //freopen("in.txt","r",stdin);
  //freopen("out.txt","w",stdout);
  #endif // LOCAL
  int n;
  cin>>n;
  vector<pt> p(n);
  for(int i = 0; i < n; i++)
    cin>>p[i];
  vector<pt> hull = convex_hull(p);
  for(auto &pt: hull)
    cout<<pt<<endl;
}
```

2.4 Delaunay triangulation

```
// Slow but simple Delaunay triangulation. Does not handle
// degenerate cases (from O'Rourke, Computational Geometry in C)
//
// Running time: O(n^4)
//
// INPUT:    x[] = x-coordinates
//           y[] = y-coordinates
//
// OUTPUT:   triples = a vector containing m triples of indices
//                     corresponding to triangle vertices

#include<vector>
using namespace std;

typedef double T;

struct triple {
    int i, j, k;
    triple() {}
    triple(int i, int j, int k) : i(i), j(j), k(k) {}
};

vector<triple> delaunayTriangulation(vector<T>& x, vector<T>& y) {
        int n = x.size();
        vector<T> z(n);
        vector<triple> ret;

        for (int i = 0; i < n; i++)
            z[i] = x[i] * x[i] + y[i] * y[i];

        for (int i = 0; i < n-2; i++) {
            for (int j = i+1; j < n; j++) {
                for (int k = i+1; k < n; k++) {
                    if (j == k) continue;
                    double xn = (y[j]-y[i])*(z[k]-z[i]) - (y[k]-y[i])*(z[j]-z[i]);
                    double yn = (x[k]-x[i])*(z[j]-z[i]) - (x[j]-x[i])*(z[k]-z[i]);
                    double zn = (x[j]-x[i])*(y[k]-y[i]) - (x[k]-x[i])*(y[j]-y[i]);
                    bool flag = zn < 0;
                    for (int m = 0; flag && m < n; m++)
                        flag = flag && ((x[m]-x[i])*xn +
                                        (y[m]-y[i])*yn +
                                        (z[m]-z[i])*zn <= 0);
                    if (flag) ret.push_back(triple(i, j, k));
                }
            }
        }
        return ret;
}

int main()
{
    T xs[]={0, 0, 1, 0.9};
    T ys[]={0, 1, 0, 0.9};
    vector<T> x(&xs[0], &xs[4]), y(&ys[0], &ys[4]);
    vector<triple> tri = delaunayTriangulation(x, y);

    //expected: 0 1 3
    //          0 3 2

    int i;
    for(i = 0; i < tri.size(); i++)
        printf("%d %d %d\n", tri[i].i, tri[i].j, tri[i].k);
    return 0;
}
```

2.5 Delaunay triangulation (java)

```
// Slow but simple Delaunay triangulation.  (from O'Rourke,
// Computational Geometry in C)
//
// Running time: O(n^4)
//
// INPUT:    x[] = x-coordinates
//           y[] = y-coordinates
//
// OUTPUT:   ret[][] = an mx3 matrix containing m triples of indices
//                     corresponding to triangle vertices

import java.util.*;

public class Delaunay {
    int[][] triangulate(double[] x, double[] y) {
        int n = x.length;
        double z[] = new double[n];
        ArrayList<int[]> ret = new ArrayList<int[]>();

        for (int i = 0; i < n; i++)
            z[i] = x[i] * x[i] + y[i] * y[i];

        for (int i = 0; i < n-2; i++) {
            for (int j = i+1; j < n; j++) {
                for (int k = i+1; k < n; k++) {
                    if (j == k) continue;
                    double xn = (y[j]-y[i])*(z[k]-z[i]) - (y[k]-y[i])*(z[j]-z[i]);
                    double yn = (x[k]-x[i])*(z[j]-z[i]) - (x[j]-x[i])*(z[k]-z[i]);
                    double zn = (x[j]-x[i])*(y[k]-y[i]) - (x[k]-x[i])*(y[j]-y[i]);
                    boolean flag = zn < 0;
                    for (int m = 0; flag && m < n; m++)
                        flag = flag && ((x[m]-x[i])*xn +
                                        (y[m]-y[i])*yn +
                                        (z[m]-z[i])*zn <= 0);
                    if (flag) ret.add(new int[]{i, j, k});
                }
            }
        }
        return ret.toArray(new int[0][0]);
    }
}
```

3 Data Structures

3.1 Mo's algorithm

```
#include<bits/stdc++.h>
#define TAM 30000 + 7
#define QTAM 200000 + 7
#define MTAM 1000000 + 7
#define whatis(x) cerr<<#x<<" is "<<x<<endl

using namespace std;

int a[TAM], r[QTAM], cnt[MTAM];
int ans, BLOCK, currL , currR ;

struct node{
    int L, R, idx;
}q[QTAM];

bool comp(node a, node b){
    if(a.L/BLOCK < b.L/BLOCK) return true;
    if(a.L/BLOCK > b.L/BLOCK) return false;
    return a.R < b.R;
}

void remove(int i){
    cnt[a[i]]--;
    if(cnt[a[i]] == 0)ans--;
}

void add(int i){
    cnt[a[i]]++;
    if(cnt[a[i]]==1)ans++;
}

int query(node i){
    while(currL< i.L){
        remove(currL);
        currL++;
    }
    while(currL > i.L){
        currL--;
        add(currL);
    }
    while(currR< i.R){
        currR++;
        add(currR);
    }
    while(currR > i.R){
        remove(currR);
        currR--;
    }
    return ans;
}

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

    #ifdef LOCAL
    freopen("in","r",stdin);
    #endif
    int n, que;
    cin>>n;
    BLOCK = sqrt(n);
    for(int i = 1; i <= n; i++)cin>>a[i];

    cin>>que;
    for(int i = 1; i <= que; i++){
        cin>>q[i].L>>q[i].R;
        q[i].idx = i;
    }
    sort(q +1, q + que+1, comp);

    for(int i = 1; i <=que; i++)
        r[q[i].idx] = query(q[i]);

    for(int i = 1; i <= que; i++)
        cout<<r[i]<<"\n";
}
```

3.2 Segment Trees with lazy propagation

```
//querys and build takes O(log n)
//example with segment sum
#include<bits/stdc++.h>

using namespace std;

long long *p;
//long long *lazy;

struct SegmentTree{
    SegmentTree *L, *R;
    long long sum = 0;
    long long lazy = 0;
    int l, r;

    long long query2(int a, int b){
        if(a == l && b == r) return sum;
        if(b <= L->r) return L->query(a,b);
        if(a >= R->l) return R->query(a,b);
        return (L->query2(a,L->r) + R->query2(R->l, b));
    }

    void update(int a, int val){
        if(l == r){
            sum += val;
            return;
        }
        int mid = (l + r)/2;
        if(l <= a && a<= mid)
            L->update(a, val);
        else
            R->update(a, val);
        sum = L->sum + R->sum;
    }

    void updateRange2(int a, int b, long long val){
        if(b < l or a > r)
            return;
        if(l == r){
            sum += val;
            return;
        }
        L->updateRange2(a, b, val);
        R->updateRange2(a,b,val);
        sum = L->sum + R->sum;
    }

    void updateRange(int a, int b, long long val){
        if(lazy != 0){
            sum += (r-l+1)*lazy;
            //sum += lazy;
            if(l != r){
                R->lazy = lazy + R->lazy;
                L->lazy = lazy + L->lazy;
            }
            lazy = 0;
        }
        if(b < l or a > r)
            return;
        if(l >= a && r <= b){
            sum += (r-l+1)*val;
            //sum += val;
            if(l != r){
                R->lazy = val + R->lazy;
                L->lazy = val + L->lazy;
            }
            return;
        }
        L->updateRange(a, b, val);
        R->updateRange(a,b,val);
        sum = L->sum + R->sum;
    }

    long long query(int a, int b){
        if(b < l or a > r)
            return 0;
        if(lazy != 0){
            sum += (r-l+1)*lazy;
            //sum += lazy;
            if(l != r){
                R->lazy = lazy + R->lazy;
                L->lazy = lazy + L->lazy;
            }
            lazy = 0;
```

```
        }
        if(a == l && b == r) return sum;
        if(b <= L->r) return L->query(a,b);
        if(a >= R->l) return R->query(a,b);
        return (L->query(a,L->r) + R->query(R->l, b));
    }

    SegmentTree(int a, int b): l(a), r(b){
        if(a == b){
            sum = p[a];
            L = R = nullptr;
        }
        else{
            L = new SegmentTree ( a, (a+b)/2 );
            R = new SegmentTree ( (a+b)/2 + 1, b );
            sum = L->sum + R->sum;
        }
    }
};


int main(){
    cin.tie(0);
    ios_base::sync_with_stdio(0);
    #ifdef LOCAL
        freopen("input.txt", "r", stdin);
    #endif // LOCAL
        long long T;
        cin >> T;
    while(T--){
        long long n, c;
        cin >> n >> c;
        long long l[n];
        memset(l,0,sizeof(l));
        p = l;
        SegmentTree *stree = new SegmentTree(0, n-1);
        while(c--){
            long long aux, p, q;
            cin >> aux >> p >> q;
            if(aux == 0){
                long long val;
                cin >> val;
                stree->updateRange(p-1, q-1, val);
            }
            else
                cout << stree->query(p-1, q-1) << endl;
        }
    }
}
```

3.3 Segment Trees with lazy propagation (Java)

```
public class SegmentTreeRangeUpdate {
        public long[] leaf;
        public long[] update;
        public int origSize;
        public SegmentTreeRangeUpdate(int[] list)       {
                origSize = list.length;
                leaf = new long[4*list.length];
                update = new long[4*list.length];
                build(1,0,list.length-1,list);
        }
        public void build(int curr, int begin, int end, int[] list)     {
                if(begin == end)
                        leaf[curr] = list[begin];
                else    {
                        int mid = (begin+end)/2;
                        build(2 * curr, begin, mid, list);
                        build(2 * curr + 1, mid+1, end, list);
                        leaf[curr] = leaf[2*curr] + leaf[2*curr+1];
                }
        }
        public void update(int begin, int end, int val) {
                update(1,0,origSize-1,begin,end,val);
        }
        public void update(int curr,  int tBegin, int tEnd, int begin, int end, int val)        {
                if(tBegin >= begin && tEnd <= end)
                        update[curr] += val;
                else    {
                        leaf[curr] += (Math.min(end,tEnd)-Math.max(begin,tBegin)+1) * val;
                        int mid = (tBegin+tEnd)/2;
                        if(mid >= begin && tBegin <= end)
                                update(2*curr, tBegin, mid, begin, end, val);
                        if(tEnd >= begin && mid+1 <= end)
                                update(2*curr+1, mid+1, tEnd, begin, end, val);
                }
        }
        public long query(int begin, int end)   {
                return query(1,0,origSize-1,begin,end);
        }
        public long query(int curr, int tBegin, int tEnd, int begin, int end)   {
                if(tBegin >= begin && tEnd <= end)      {
                        if(update[curr] != 0)   {
                                leaf[curr] += (tEnd-tBegin+1) * update[curr];
                                if(2*curr < update.length){
                                        update[2*curr] += update[curr];
                                        update[2*curr+1] += update[curr];
                                }
                                update[curr] = 0;
                        }
                        return leaf[curr];
                }
                else    {
                        leaf[curr] += (tEnd-tBegin+1) * update[curr];
                        if(2*curr < update.length){
                                update[2*curr] += update[curr];
                                update[2*curr+1] += update[curr];
                        }
                        update[curr] = 0;
                        int mid = (tBegin+tEnd)/2;
                        long ret = 0;
                        if(mid >= begin && tBegin <= end)
                                ret += query(2*curr, tBegin, mid, begin, end);
                        if(tEnd >= begin && mid+1 <= end)
                                ret += query(2*curr+1, mid+1, tEnd, begin, end);
                        return ret;
                }
        }
}
```

3.4 Fenwick Tree

```
#include <iostream>
using namespace std;

#define LOGSZ 17

int tree[(1<<LOGSZ)+1];
int N = (1<<LOGSZ);

// add v to value at x
void set(int x, int v) {
  while(x <= N) {
    tree[x] += v;
    x += (x & -x);
  }
}

// get cumulative sum up to and including x
int get(int x) {
  int res = 0;
  while(x) {
    res += tree[x];
    x -= (x & -x);
  }
  return res;
}

// get largest value with cumulative sum less than or equal to x;
// for smallest, pass x-1 and add 1 to result
int getind(int x) {
  int idx = 0, mask = N;
  while(mask && idx < N) {
    int t = idx + mask;
    if(x >= tree[t]) {
      idx = t;
      x -= tree[t];
    }
    mask >>= 1;
  }
  return idx;
}
```

3.5 Union Find (Short)

```
#include <iostream>
#include <vector>
using namespace std;
int find(vector<int> &C, int x) { return (C[x] == x) ? x : C[x] = find(C, C[x]); }
void merge(vector<int> &C, int x, int y) { C[find(C, x)] = find(C, y); }
int main()
{
	int n = 5;
	vector<int> C(n);
	for (int i = 0; i < n; i++) C[i] = i;
	merge(C, 0, 2);
	merge(C, 1, 0);
	merge(C, 3, 4);
	for (int i = 0; i < n; i++) cout << i << " " << find(C, i) << endl;
	return 0;
}
```

3.6 Union Find

```
/*---------------- disjoint sets-----------*/
#include<bits/stdc++.h>
#define TAM 10000

using namespace std;

class UnionFind{
    private:
        vector<int> p, rank, ssize;
        int numSets;
    public:
        UnionFind(int N){
            rank.assign(N, 0);
            ssize.assign(N, 1);
            numSets = N;
            p.assign(N,0);
            for(int i = 0; i < N; i++)
                p[i] = i;
        }
        int findSet(int i){
            return (p[i] == i)? i: (p[i] = findSet(p[i]));
        }
        bool isSameSet(int i, int j){
            return findSet(i) == findSet(j);
        }

        void unionSet(int i, int j){
            if(!isSameSet(i, j)){
                numSets--;
                int x = findset(i), y = findSet(j);
                if(rank[x] > rank[y]){
                    p[y] = x;
                    ssize[x] += ssize[y];
                }
                else{
                    p[x] = y;
                    ssize[y] += ssize[x];
                    if(rank[x] == rank[y])
                        rank[y]++;
                }

            }
        }

        int numDisjointSets(){
            return numSets;
        }

        int sizeOfSet(int i){
            return ssize[findSet(i)];
        }
};
```

4 Strings

4.1 KMP

```
/*
Finds all occurrences of the pattern string p within the
text string t. Running time is O(n + m), where n and m
are the lengths of p and t, respecitvely.
*/

#include <iostream>
#include <string>
#include <vector>

using namespace std;

typedef vector<int> VI;

void buildPi(string& p, VI& pi)
{
  pi = VI(p.length());
  int k = -2;
  for(int i = 0; i < p.length(); i++) {
    while(k >= -1 && p[k+1] != p[i])
      k = (k == -1) ? -2 : pi[k];
    pi[i] = ++k;
  }
}

int KMP(string& t, string& p)
{
  VI pi;
  buildPi(p, pi);
  int k = -1;
  for(int i = 0; i < t.length(); i++) {
    while(k >= -1 && p[k+1] != t[i])
      k = (k == -1) ? -2 : pi[k];
    k++;
    if(k == p.length() - 1) {
      // p matches t[i-m+1, ..., i]
      cout << "matched at index " << i-k << ": ";
      cout << t.substr(i-k, p.length()) << endl;
      k = (k == -1) ? -2 : pi[k];
    }
  }
  return 0;
}

int main()
{
  string a = "AABAACAADAABAABA", b = "AABA";
  KMP(a, b); // expected matches at: 0, 9, 12
  return 0;
}
```

4.2 Suffix Array

```
// Suffix array construction in O(L log^2 L) time.  Routine for
// computing the length of the longest common prefix of any two
// suffixes in O(log L) time.
//
// INPUT:   string s
//
// OUTPUT:  array suffix[] such that suffix[i] = index (from 0 to L-1)
//          of substring s[i...L-1] in the list of sorted suffixes.
//          That is, if we take the inverse of the permutation suffix[],
//          we get the actual suffix array.

#include <vector>
#include <iostream>
#include <string>

using namespace std;
```

```
struct SuffixArray {
  const int L;
  string s;
  vector<vector<int> > P;
  vector<pair<pair<int,int>,int> > M;

  SuffixArray(const string &s) : L(s.length()), s(s), P(1, vector<int>(L, 0)), M(L) {
    for (int i = 0; i < L; i++) P[0][i] = int(s[i]);
    for (int skip = 1, level = 1; skip < L; skip *= 2, level++) {
      P.push_back(vector<int>(L, 0));
      for (int i = 0; i < L; i++)
        M[i] = make_pair(make_pair(P[level-1][i], i + skip < L ? P[level-1][i + skip] : -1000), i);
      sort(M.begin(), M.end());
      for (int i = 0; i < L; i++)
        P[level][M[i].second] = (i > 0 && M[i].first == M[i-1].first) ? P[level][M[i-1].second] : i;
    }
  }

  vector<int> GetSuffixArray() { return P.back(); }

  // returns the length of the longest common prefix of s[i...L-1] and s[j...L-1]
  int LongestCommonPrefix(int i, int j) {
    int len = 0;
    if (i == j) return L - i;
    for (int k = P.size() - 1; k >= 0 && i < L && j < L; k--) {
      if (P[k][i] == P[k][j]) {
        i += 1 << k;
        j += 1 << k;
        len += 1 << k;
      }
    }
    return len;
  }
};

// BEGIN CUT
// The following code solves UVA problem 11512: GATTACA.
#define TESTING
#ifdef TESTING
int main() {
  int T;
  cin >> T;
  for (int caseno = 0; caseno < T; caseno++) {
    string s;
    cin >> s;
    SuffixArray array(s);
    vector<int> v = array.GetSuffixArray();
    int bestlen = -1, bestpos = -1, bestcount = 0;
    for (int i = 0; i < s.length(); i++) {
      int len = 0, count = 0;
      for (int j = i+1; j < s.length(); j++) {
        int l = array.LongestCommonPrefix(i, j);
        if (l >= len) {
          if (l > len) count = 2; else count++;
          len = l;
        }
      }
      if (len > bestlen || len == bestlen && s.substr(bestpos, bestlen) > s.substr(i, len)) {
        bestlen = len;
        bestcount = count;
        bestpos = i;
      }
    }
    if (bestlen == 0) {
      cout << "No repetitions found!" << endl;
    } else {
      cout << s.substr(bestpos, bestlen) << " " << bestcount << endl;
    }
  }
}

#else
// END CUT
int main() {

  // bobocel is the 0'th suffix
  //  obocel is the 5'th suffix
  //   bocel is the 1'st suffix
  //    ocel is the 6'th suffix
  //     cel is the 2'nd suffix
  //      el is the 3'rd suffix
  //       l is the 4'th suffix
  SuffixArray suffix("bobocel");
  vector<int> v = suffix.GetSuffixArray();

  // Expected output: 0 5 1 6 2 3 4
  //                  2
  for (int i = 0; i < v.size(); i++) cout << v[i] << " ";
  cout << endl;
  cout << suffix.LongestCommonPrefix(0, 2) << endl;
}
// BEGIN CUT
#endif
// END CUT
```

4.3 Fast Fourier Transform (convolution)

```
// Convolution using the fast Fourier transform (FFT).
//
// INPUT:
//     a[1...n]
//     b[1...m]
//
// OUTPUT:
//     c[1...n+m-1] such that c[k] = sum_{i=0}^k a[i] b[k-i]
//
// Alternatively, you can use the DFT() routine directly, which will
// zero-pad your input to the next largest power of 2 and compute the
// DFT or inverse DFT.

#include <iostream>
#include <vector>
#include <complex>

using namespace std;

typedef long double DOUBLE;
typedef complex<DOUBLE> COMPLEX;
typedef vector<DOUBLE> VD;
typedef vector<COMPLEX> VC;

struct FFT {
  VC A;
  int n, L;

  int ReverseBits(int k) {
    int ret = 0;
    for (int i = 0; i < L; i++) {
      ret = (ret << 1) | (k & 1);
      k >>= 1;
    }
    return ret;
  }

  void BitReverseCopy(VC a) {
    for (n = 1, L = 0; n < a.size(); n <<= 1, L++) ;
    A.resize(n);
    for (int k = 0; k < n; k++)
      A[ReverseBits(k)] = a[k];
  }

  VC DFT(VC a, bool inverse) {
    BitReverseCopy(a);
    for (int s = 1; s <= L; s++) {
      int m = 1 << s;
      COMPLEX wm = exp(COMPLEX(0, 2.0 * M_PI / m));
      if (inverse) wm = COMPLEX(1, 0) / wm;
      for (int k = 0; k < n; k += m) {
        COMPLEX w = 1;
        for (int j = 0; j < m/2; j++) {
          COMPLEX t = w * A[k + j + m/2];
          COMPLEX u = A[k + j];
          A[k + j] = u + t;
          A[k + j + m/2] = u - t;
          w = w * wm;
        }
      }
    }
    if (inverse) for (int i = 0; i < n; i++) A[i] /= n;
    return A;
  }

  // c[k] = sum_{i=0}^k a[i] b[k-i]
  VD Convolution(VD a, VD b) {
    int L = 1;
    while ((1 << L) < a.size()) L++;
    while ((1 << L) < b.size()) L++;
    int n = 1 << (L+1);

    VC aa, bb;
    for (size_t i = 0; i < n; i++) aa.push_back(i < a.size() ? COMPLEX(a[i], 0) : 0);
```

```
    for (size_t i = 0; i < n; i++) bb.push_back(i < b.size() ? COMPLEX(b[i], 0) : 0);

    VC AA = DFT(aa, false);
    VC BB = DFT(bb, false);
    VC CC;
    for (size_t i = 0; i < AA.size(); i++) CC.push_back(AA[i] * BB[i]);
    VC cc = DFT(CC, true);

    VD c;
    for (int i = 0; i < a.size() + b.size() - 1; i++) c.push_back(cc[i].real());
    return c;
  }

};

int main() {
  double a[] = {1, 3, 4, 5, 7};
  double b[] = {2, 4, 6};

  FFT fft;
  VD c = fft.Convolution(VD(a, a + 5), VD(b, b + 3));

  // expected output: 2 10 26 44 58 58 42
  for (int i = 0; i < c.size(); i++) cerr << c[i] << " ";
  cerr << endl;

  return 0;
}
```

5 Flows

5.1 Ford Fulkerson

```
///--------------------- Ford-Fulkerson O(MaxFlow * |E|) ---------------------

struct OutEdge {
        int to, cap, rIdx;
        OutEdge ( ) { }
        OutEdge(int to, int cap, int rIdx) :
                to(to), cap(cap), rIdx(rIdx) { }
};

struct Network
{
        vector<vector<OutEdge> > out;
        vector<bool> seen;

        int sink;
        int augment ( int i, const int cur ) {
                if ( i == sink ) return cur;
                if ( seen[i] ) return false;
                seen[i] = true;
                int ans;
                for ( OutEdge& e : out[i] )
                        if ( e.cap > 0 && ( ans = augment(e.to,min(cur,e.cap)) ) ) {
                                e.cap -= ans;
                                out[e.to][e.rIdx].cap += ans;
                                return ans;
                        }
                return 0;
        }

        int maxflow ( int source, int _sink ) {
                sink = _sink;
                int curflow = 0, aug;
                while ( true ) {
                        fill ( seen.begin(), seen.end(), false );
                        aug = augment(source,INT_MAX);
                        if ( aug == 0 ) break;
                        curflow += aug;
                }
                return curflow;
        }

        void addEdge ( int fr, int to, int c ) {
                assert ( fr != to );
                out[fr].push_back(OutEdge(to, c, out[to].size()));
                out[to].push_back(OutEdge(fr, 0, out[fr].size() - 1));
        }

        Network(int n) {
                out.assign(n, vector<OutEdge>());
                seen.resize(n);
        }
};

// Adjacency list implementation of FIFO push relabel maximum flow
// with the gap relabeling heuristic.  This implementation is
// significantly faster than straight Ford-Fulkerson.  It solves
// random problems with 10000 vertices and 1000000 edges in a few
// seconds, though it is possible to construct test cases that
// achieve the worst-case.
//
// Running time:
//     O(|V|^3)
//
// INPUT:
//     - graph, constructed using AddEdge()
//     - source
//     - sink
//
// OUTPUT:
//     - maximum flow value
//     - To obtain the actual flow values, look at all edges with
//       capacity > 0 (zero capacity edges are residual edges).

#include <cmath>
#include <vector>
#include <iostream>
#include <queue>

using namespace std;

typedef long long LL;

struct Edge {
  int from, to, cap, flow, index;
  Edge(int from, int to, int cap, int flow, int index) :
    from(from), to(to), cap(cap), flow(flow), index(index) {}
};

struct PushRelabel {
  int N;
  vector<vector<Edge> > G;
  vector<LL> excess;
  vector<int> dist, active, count;
  queue<int> Q;

  PushRelabel(int N) : N(N), G(N), excess(N), dist(N), active(N), count(2*N) {}

  void AddEdge(int from, int to, int cap) {
    G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
    if (from == to) G[from].back().index++;
    G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1));
  }

  void Enqueue(int v) {
    if (!active[v] && excess[v] > 0) { active[v] = true; Q.push(v); }
  }

  void Push(Edge &e) {
    int amt = int(min(excess[e.from], LL(e.cap - e.flow)));
    if (dist[e.from] <= dist[e.to] || amt == 0) return;
    e.flow += amt;
    G[e.to][e.index].flow -= amt;
    excess[e.to] += amt;
    excess[e.from] -= amt;
    Enqueue(e.to);
  }

  void Gap(int k) {
    for (int v = 0; v < N; v++) {
      if (dist[v] < k) continue;
      count[dist[v]]--;
      dist[v] = max(dist[v], N+1);
      count[dist[v]]++;
      Enqueue(v);
    }
  }

  void Relabel(int v) {
    count[dist[v]]--;
    dist[v] = 2*N;
    for (int i = 0; i < G[v].size(); i++)
      if (G[v][i].cap - G[v][i].flow > 0)
        dist[v] = min(dist[v], dist[G[v][i].to] + 1);
    count[dist[v]]++;
    Enqueue(v);
  }

  void Discharge(int v) {
```

```
    for (int i = 0; excess[v] > 0 && i < G[v].size(); i++) Push(G[v][i]);
    if (excess[v] > 0) {
      if (count[dist[v]] == 1)
        Gap(dist[v]);
      else
        Relabel(v);
    }
  }

  LL GetMaxFlow(int s, int t) {
    count[0] = N-1;
    count[N] = 1;
    dist[s] = N;
    active[s] = active[t] = true;
    for (int i = 0; i < G[s].size(); i++) {
      excess[s] += G[s][i].cap;
      Push(G[s][i]);
    }

    while (!Q.empty()) {
      int v = Q.front();
      Q.pop();
      active[v] = false;
      Discharge(v);
    }

    LL totflow = 0;
    for (int i = 0; i < G[s].size(); i++) totflow += G[s][i].flow;
    return totflow;
  }
};

// Adjacency list implementation of Dinic's blocking flow algorithm.
// This is very fast in practice, and only loses to push-relabel flow.
//
// Running time:
//     O(|V|^2 |E|)
//
// INPUT:
//     - graph, constructed using AddEdge()
//     - source
//     - sink
//
// OUTPUT:
//     - maximum flow value
//     - To obtain the actual flow values, look at all edges with
//       capacity > 0 (zero capacity edges are residual edges).

#include <cmath>
#include <vector>
#include <iostream>
#include <queue>

using namespace std;

const int INF = 2000000000;

struct Edge {
  int from, to, cap, flow, index;
  Edge(int from, int to, int cap, int flow, int index) :
    from(from), to(to), cap(cap), flow(flow), index(index) {}
};

struct Dinic {
  int N;
  vector<vector<Edge> > G;
  vector<Edge *> dad;
  vector<int> Q;

  Dinic(int N) : N(N), G(N), dad(N), Q(N) {}

  void AddEdge(int from, int to, int cap) {
    G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
    if (from == to) G[from].back().index++;
    G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1));
  }

  long long BlockingFlow(int s, int t) {
    fill(dad.begin(), dad.end(), (Edge *) NULL);
    dad[s] = &G[0][0] - 1;

    int head = 0, tail = 0;
    Q[tail++] = s;
    while (head < tail) {
      int x = Q[head++];
      for (int i = 0; i < G[x].size(); i++) {
        Edge &e = G[x][i];
        if (!dad[e.to] && e.cap - e.flow > 0) {
          dad[e.to] = &G[x][i];
          Q[tail++] = e.to;
        }
      }
    }
    if (!dad[t]) return 0;

    long long totflow = 0;
    for (int i = 0; i < G[t].size(); i++) {
      Edge *start = &G[G[t][i].to][G[t][i].index];
      int amt = INF;
      for (Edge *e = start; amt && e != dad[s]; e = dad[e->from]) {
        if (!e) { amt = 0; break; }
        amt = min(amt, e->cap - e->flow);
      }
      if (amt == 0) continue;
      for (Edge *e = start; amt && e != dad[s]; e = dad[e->from]) {
        e->flow += amt;
        G[e->to][e->index].flow -= amt;
      }
      totflow += amt;
    }
    return totflow;
  }

  long long GetMaxFlow(int s, int t) {
    long long totflow = 0;
    while (long long flow = BlockingFlow(s, t))
      totflow += flow;
    return totflow;
  }
};
```

5.2 Edmons Karp Min cut

```
///------------------- Edmonds Karp with MinCut O(|V|*|E|^2) -------------------

struct Network {

        vector<Edge> G[TAM];
        int from[TAM], n;
        bool color[TAM];

        // Call flood (source) to color one node
        // component of min cut.
        void flood ( int node ) {
                if ( color[node] ) return;
                color[node] = true;
                for ( const Edge& e : G[node] )
                        if ( e.cap > 0 )
                                flood ( e.to );
        }

        int maxFlow ( int A, int B )
        {
                int flow = 0;
                while ( 1 ) {
                        memset ( from, -1, sizeof(from) );

                        queue<int> q;
                        q.push ( A );
                        from[A] = -2;
                        for ( int i; !q.empty(); q.pop() ) {
                                i = q.front();
                                for ( Edge& e : G[i] )
                                        if ( from[e.to] == -1 && e.cap ) {
                                                from[e.to] = e.invIdx;
                                                q.push ( e.to );
                                        }
                        }

                        if ( from[B] == -1 ) break;

                        int aug = INF_CAP;
                        for ( int i = B, j; i != A; i = j ) {
                                j = G[i][from[i]].to;
                                aug = min ( aug, G[j][ G[i][from[i]].invIdx ].cap );
                        }

                        for ( int i = B, j; i != A; i = j ) {
                                j = G[i][from[i]].to;
                                G[j][ G[i][from[i]].invIdx ].cap -= aug;
                                G[i][from[i]].cap += aug;
                        }

                        flow += aug;
                }
```

```
		return flow;
	}

	void addNonDirEdge ( int a, int b, int c ) {
		assert ( a != b );
		G[a].push_back ( Edge(b,c,G[b].size()) );
		G[b].push_back ( Edge(a,c,G[a].size()-1) );
	}

	void addDirEdge ( int a, int b, int c ) {
		assert ( a != b );
		G[a].push_back ( Edge(b,c,G[b].size()) );
		G[b].push_back ( Edge(a,0,G[a].size()-1) );
	}

	void clear ( int _n ) {
		n = _n;
		memset ( color, false, n );
		for ( int i = 0; i < n; ++i )
			G[i].clear();
	}
} netw;
```

5.3 Hopcroft karp's maximum bipartite matching

```
///------ Hopcroft Karp - Maximum Bipartite Matching O( sqrt(|V|) * |E| ) ------

namespace hopcroftKarp {

	const int MAXN1 = 50000;
	const int MAXN2 = 50000;
	const int MAXM = 150000;

	int n1, n2, edges, last[MAXN1], prev[MAXM], head[MAXM];
	int matching[MAXN2], dist[MAXN1], Q[MAXN1];
	bool used[MAXN1], vis[MAXN1];

	void init(int _n1, int _n2) {
		n1 = _n1;
		n2 = _n2;
		edges = 0;
		fill(last, last + n1, -1);
	}

	void addEdge(int u, int v) {
		head[edges] = v;
		prev[edges] = last[u];
		last[u] = edges++;
	}

	void bfs() {
		fill(dist, dist + n1, -1);
		int sizeQ = 0;
		for (int u = 0; u < n1; ++u) {
			if (!used[u]) {
				Q[sizeQ++] = u;
				dist[u] = 0;
			}
		}
		for (int i = 0; i < sizeQ; i++) {
			int u1 = Q[i];
			for (int e = last[u1]; e >= 0; e = prev[e]) {
				int u2 = matching[head[e]];
				if (u2 >= 0 && dist[u2] < 0) {
					dist[u2] = dist[u1] + 1;
					Q[sizeQ++] = u2;
				}
			}
		}
	}

	bool dfs(int u1) {
		vis[u1] = true;
		for (int e = last[u1]; e >= 0; e = prev[e]) {
			int v = head[e];
			int u2 = matching[v];
			if (u2 < 0 || !vis[u2] && dist[u2] == dist[u1] + 1 && dfs(u2)) {
				matching[v] = u1;
				used[u1] = true;
				return true;
			}
		}
		return false;
	}

	int maxMatching() {
		fill(used, used + n1, false);
		fill(matching, matching + n2, -1);
		for (int res = 0;;) {
			bfs();
			fill(vis, vis + n1, false);
			int f = 0;
			for (int u = 0; u < n1; ++u)
				if (!used[u] && dfs(u))
					++f;
			if (!f)
				return res;
			res += f;
		}
	}
};
```

5.4 Maxmium bipartite matching (short but slower)

```
///------------------ Maximum Bipartite Matching O(|V|*|E|) ------------------

bool findMatch(int i, const VVI &w, VI &mr, VI &mc, VI &seen) {
    for (int j = 0; j < int(w[i].size()); j++) {
		if (w[i][j] && !seen[j]) {
			seen[j] = true;
			if (mc[j] < 0 || findMatch(mc[j], w, mr, mc, seen)) {
				mr[i] = j; mc[j] = i;
				return true;
			}
		}
	}
	return false;
}

int maxBipartiteMatching(const VVI &w ) {
	if ( w.empty() || w[0].empty() ) return 0;
	VI mr(w.size(),-1), mc(w[0].size(), -1);
	int ct = 0;
	for (int i = 0; i < int(w.size()); i++) {
		VI seen(w[0].size());
		if (findMatch(i, w, mr, mc, seen)) ct++;
	}
	return ct;
}
```

6 Math

6.1 general math tricks

```
 long square(long n){ return n*n;}

 int fastPow(long x, long n){
     if(n == 0)
        return 1;

     if(n % 2 == 0)
        return square(fastPow(x, n/2));

     return x * (fastPow(x, n - 1));
 }

/* LCM */
```

```
int LCM(int m, n){return (m*n)/__gcd(m, n); }


int main(){

    /* n es impar?*/
    odd = ((n & 1)? true : false);

    /*como saber si un numero es una potencia de 2*/
    power_of_2 = ((v & (v-1)) == 0);

    /*contar trailing 0's de una mascara */
    __builtin_ctz(n);

    /*contar 1's de una mascara*/
    __builtin_popcount(n);

    /*quitar el elemento j de la m scara*/

    mask &= ~(1<<j);

    /*revisar si el elemento j del arreglo esta en la m scara ( si es 0 el resultado es porque no
         est )*/
    int t = mask & (1<<j);

    /*Obtener el bit menos significativo*/
     t = mask & -mask

    /*encender todos los n primeros bits de la m scara*/

    mask = (1<<n) - 1;

    /*iterar sobre cada uno de los subsets de un subset y*/
    for(int x = y; x>0; x = (y & (x-1)) )
}
```

6.2 Miller Rabin's primality test

```
// Randomized Primality Test (Miller-Rabin):
//   Error rate: 2^(-TRIAL)
//   Almost constant time. srand is needed

#include <stdlib.h>
#define EPS 1e-7

typedef long long LL;

LL ModularMultiplication(LL a, LL b, LL m)
{
        LL ret=0, c=a;
        while(b)
        {
                if(b&1) ret=(ret+c)%m;
                b>>=1; c=(c+c)%m;
        }
        return ret;
}
LL ModularExponentiation(LL a, LL n, LL m)
{
        LL ret=1, c=a;
        while(n)
        {
                if(n&1) ret=ModularMultiplication(ret, c, m);
                n>>=1; c=ModularMultiplication(c, c, m);
        }
        return ret;
}
bool Witness(LL a, LL n)
{
        LL u=n-1;
  int t=0;
        while(!(u&1)){u>>=1; t++;}
        LL x0=ModularExponentiation(a, u, n), x1;
        for(int i=1;i<=t;i++)
        {
                x1=ModularMultiplication(x0, x0, n);
                if(x1==1 && x0!=1 && x0!=n-1) return true;
                x0=x1;
        }
        if(x0!=1) return true;
        return false;
}
LL Random(LL n)
{
  LL ret=rand(); ret*=32768;
        ret+=rand(); ret*=32768;
        ret+=rand(); ret*=32768;
        ret+=rand();
  return ret%n;
}
bool IsPrimeFast(LL n, int TRIAL)
{
  while(TRIAL--)
  {
    LL a=Random(n-2)+1;
    if(Witness(a, n)) return false;
  }
  return true;
}
```

6.3 Pollard rho

```
#include<bits/stdc++.h>
#include<time.h>

#define show(x) cout << #x << " = " << x << endl;

using namespace std;

typedef long long ll;
typedef pair<ll, ll> ii;
typedef pair<double, ii> iii;

const int MAX = 200005;
const double EPS = 1e-5;
const int INF = INT_MAX;

//modular multiplication for really big numbers
ll mul(ll a, ll b, ll mod) {
    ll ret = 0;
    for(a %= mod, b %= mod; b != 0;
        b >>= 1, a <<= 1, a = a >= mod ? a - mod : a) {
        if (b&1) {
            ret += a;
            if (ret >= mod) ret -= mod;
        }
    }
    return ret;
}

ll fpow(ll a, ll b, ll MOD) {
    ll ans = 1LL;
    while(b > 0) {
        if(b&1) ans = mul(ans, a, MOD);
        a = mul(a, a, MOD);
        b >>= 1LL;
    }
    return ans;
}

const int rounds = 6;
// Checks if a number is prime with prob 1 - 1 / (2 ^ it)
bool miller_rabin(ll n) {
    if(n == 2 || n == 3) return true;
    if(n < 2 || (n&1) == 0) return false;
    for(int i = 0; i < rounds; i++) {
        int a = rand()%(n-4)+2;
        if(fpow(a, n-1, n) != 1)
            return false;
    }
    return true;
}

// if n is prime , check with miller rabin (n^(1/4)) and check return != n and  != 1
ll pollard_rho(ll n, ll c) {
    ll x = 2, y = 2, i = 1, k = 2, d;
    while (true) {
        x = (mul(x, x, n) + c);
        if (x >= n)     x -= n;
        d = __gcd(x - y, n);
        if (d > 1) return d;
        if (++i == k) y = x, k <<= 1;
    }
    return n;
}

//return factorization of a big number
```

```
void factorize(ll n, vector<ll> &f) {
    if(n == 1) return;
    if (miller_rabin(n)) {
        f.push_back(n);
        return;
    }
    ll d = n;
    for (int i = 2; d == n; i++)
        d = pollard_rho(n, i);
    factorize(d, f);
    factorize(n/d, f);
}
```

6.4 number theory general

```
// This is a collection of useful code for solving problems that
// involve modular linear equations.  Note that all of the
// algorithms described here work on nonnegative integers.

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

typedef vector<int> VI;
typedef pair<int, int> PII;

// return a % b (positive value)
int mod(int a, int b) {
	return ((a%b) + b) % b;
}

// computes gcd(a,b)
int gcd(int a, int b) {
	while (b) { int t = a%b; a = b; b = t; }
	return a;
}

// computes lcm(a,b)
int lcm(int a, int b) {
	return a / gcd(a, b)*b;
}

// (a^b) mod m via successive squaring
int powermod(int a, int b, int m)
{
	int ret = 1;
	while (b)
	{
		if (b & 1) ret = mod(ret*a, m);
		a = mod(a*a, m);
		b >>= 1;
	}
	return ret;
}

// returns g = gcd(a, b); finds x, y such that d = ax + by
int extended_euclid(int a, int b, int &x, int &y) {
	int xx = y = 0;
	int yy = x = 1;
	while (b) {
		int q = a / b;
		int t = b; b = a%b; a = t;
		t = xx; xx = x - q*xx; x = t;
		t = yy; yy = y - q*yy; y = t;
	}
	return a;
}

// finds all solutions to ax = b (mod n)
VI modular_linear_equation_solver(int a, int b, int n) {
	int x, y;
	VI ret;
	int g = extended_euclid(a, n, x, y);
	if (!(b%g)) {
		x = mod(x*(b / g), n);
		for (int i = 0; i < g; i++)
			ret.push_back(mod(x + i*(n / g), n));
	}
	return ret;
}

// computes b such that ab = 1 (mod n), returns -1 on failure
int mod_inverse(int a, int n) {
	int x, y;
	int g = extended_euclid(a, n, x, y);
	if (g > 1) return -1;
	return mod(x, n);
}

// Chinese remainder theorem (special case): find z such that
// z % m1 = r1, z % m2 = r2.  Here, z is unique modulo M = lcm(m1, m2).
// Return (z, M).  On failure, M = -1.
PII chinese_remainder_theorem(int m1, int r1, int m2, int r2) {
	int s, t;
	int g = extended_euclid(m1, m2, s, t);
	if (r1%g != r2%g) return make_pair(0, -1);
	return make_pair(mod(s*r2*m1 + t*r1*m2, m1*m2) / g, m1*m2 / g);
}

// Chinese remainder theorem: find z such that
// z % m[i] = r[i] for all i.  Note that the solution is
// unique modulo M = lcm_i (m[i]).  Return (z, M). On
// failure, M = -1. Note that we do not require the a[i]'s
// to be relatively prime.
PII chinese_remainder_theorem(const VI &m, const VI &r) {
	PII ret = make_pair(r[0], m[0]);
	for (int i = 1; i < m.size(); i++) {
		ret = chinese_remainder_theorem(ret.second, ret.first, m[i], r[i]);
		if (ret.second == -1) break;
	}
	return ret;
}

// computes x and y such that ax + by = c
// returns whether the solution exists
bool linear_diophantine(int a, int b, int c, int &x, int &y) {
	if (!a && !b)
	{
		if (c) return false;
		x = 0; y = 0;
		return true;
	}
	if (!a)
	{
		if (c % b) return false;
		x = 0; y = c / b;
		return true;
	}
	if (!b)
	{
		if (c % a) return false;
		x = c / a; y = 0;
		return true;
	}
	int g = gcd(a, b);
	if (c % g) return false;
	x = c / g * mod_inverse(a / g, b / g);
	y = (c - a*x) / b;
	return true;
}

int main() {
	// expected: 2
	cout << gcd(14, 30) << endl;

	// expected: 2 -2 1
	int x, y;
	int g = extended_euclid(14, 30, x, y);
	cout << g << " " << x << " " << y << endl;

	// expected: 95 451
	VI sols = modular_linear_equation_solver(14, 30, 100);
	for (int i = 0; i < sols.size(); i++) cout << sols[i] << " ";
	cout << endl;

	// expected: 8
	cout << mod_inverse(8, 9) << endl;

	// expected: 23 105
	//           11 12
	PII ret = chinese_remainder_theorem(VI({ 3, 5, 7 }), VI({ 2, 3, 2 }));
	cout << ret.first << " " << ret.second << endl;
	ret = chinese_remainder_theorem(VI({ 4, 6 }), VI({ 3, 5 }));
	cout << ret.first << " " << ret.second << endl;

	// expected: 5 -15
	if (!linear_diophantine(7, 2, 5, x, y)) cout << "ERROR" << endl;
	cout << x << " " << y << endl;
	return 0;
}
```

7 Miscellaneous

7.1 c++ ios tricks

```cpp
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    // Ouput a specific number of digits past the decimal point,
    // in this case 5
    cout.setf(ios::fixed); cout << setprecision(5);
    cout << 100.0/7.0 << endl;
    cout.unsetf(ios::fixed);

    // Output the decimal point and trailing zeros
    cout.setf(ios::showpoint);
    cout << 100.0 << endl;
    cout.unsetf(ios::showpoint);

    // Output a '+' before positive values
    cout.setf(ios::showpos);
    cout << 100 << " " << -100 << endl;
    cout.unsetf(ios::showpos);

    // Output numerical values in hexadecimal
    cout << hex << 100 << " " << 1000 << " " << 10000 << dec << endl;
}
```

7.2 java IO template and iterative binary search

```java
import java.io.OutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.PrintWriter;
import java.util.StringTokenizer;
import java.io.IOException;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.InputStream;

public class Main {
    public static void main(String[] args) {
        InputStream inputStream = System.in;
        OutputStream outputStream = System.out;
        InputReader in = new InputReader(inputStream);
        PrintWriter out = new PrintWriter(outputStream);
        TaskC solver = new TaskC();
        solver.solve(1, in, out);
        out.close();
    }

    static class TaskC {
        private static final int ITERATIONS = 500;
        public void solve(int testNumber, InputReader in, PrintWriter out) {
            int n = in.nextInt();
            //Iterative binary search
            double l = 0.0, h = 1e17;
            for (int i = 0; i < ITERATIONS; i++) {
                double mid = (l + h) / 2.0;
                if (can(mid, a, b, p))
                    l = mid;
                else
                    h = mid;
            }
            double ans = l;
        }
    }

    static class InputReader {
        public BufferedReader reader;
        public StringTokenizer tokenizer;

        public InputReader(InputStream stream) {
            reader = new BufferedReader(new InputStreamReader(stream), 32768);
            tokenizer = null;
        }

        public String next() {
            while (tokenizer == null || !tokenizer.hasMoreTokens()) {
                try {
                    tokenizer = new StringTokenizer(reader.readLine());
                } catch (IOException e) {
                    throw new RuntimeException(e);
                }
            }
            return tokenizer.nextToken();
        }

        public int nextInt() {
            return Integer.parseInt(next());
        }

        public double nextDouble() {
            return Double.parseDouble(next());
        }

    }
}
```