```
////////////////////////////////////////////////////////////////////////
// Min cost bipartite matching via shortest augmenting paths
//
// This is an O(n^3) implementation of a shortest augmenting path
// algorithm for finding min cost perfect matchings in dense
// graphs.  In practice, it solves 1000x1000 problems in around 1
// second.
//
//   cost[i][j] = cost for pairing left node i with right node j
//   Lmate[i] = index of right node that left node i pairs with
//   Rmate[j] = index of left node that right node j pairs with
//
// The values in cost[i][j] may be positive or negative.  To perform
// maximization, simply negate the cost[][] matrix.
////////////////////////////////////////////////////////////////////////

#include <algorithm>
#include <cstdio>
#include <cmath>
#include <vector>

using namespace std;

typedef vector<double> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;

double MinCostMatching(const VVD &cost, VI &Lmate, VI &Rmate) {
  int n = int(cost.size());

  // construct dual feasible solution
  VD u(n);
  VD v(n);
  for (int i = 0; i < n; i++) {
    u[i] = cost[i][0];
    for (int j = 1; j < n; j++) u[i] = min(u[i], cost[i][j]);
  }
  for (int j = 0; j < n; j++) {
    v[j] = cost[0][j] - u[0];
    for (int i = 1; i < n; i++) v[j] = min(v[j], cost[i][j] - u[i]);
  }

  // construct primal solution satisfying complementary slackness
  Lmate = VI(n, -1);
  Rmate = VI(n, -1);
  int mated = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      if (Rmate[j] != -1) continue;
      if (fabs(cost[i][j] - u[i] - v[j]) < 1e-10) {
        Lmate[i] = j;
        Rmate[j] = i;
        mated++;
        break;
      }
    }
  }

  VD dist(n);
  VI dad(n);
  VI seen(n);

  // repeat until primal solution is feasible
  while (mated < n) {

    // find an unmatched left node
    int s = 0;
    while (Lmate[s] != -1) s++;

    // initialize Dijkstra
    fill(dad.begin(), dad.end(), -1);
    fill(seen.begin(), seen.end(), 0);
    for (int k = 0; k < n; k++)
      dist[k] = cost[s][k] - u[s] - v[k];

    int j = 0;
    while (true) {

      // find closest
      j = -1;
      for (int k = 0; k < n; k++) {
        if (seen[k]) continue;
        if (j == -1 || dist[k] < dist[j]) j = k;
      }
      seen[j] = 1;

      // termination condition
      if (Rmate[j] == -1) break;

      // relax neighbors
      const int i = Rmate[j];
      for (int k = 0; k < n; k++) {
        if (seen[k]) continue;
        const double new_dist = dist[j] + cost[i][k] - u[i] - v[k];
        if (dist[k] > new_dist) {
          dist[k] = new_dist;
          dad[k] = j;
        }
      }
    }

    // update dual variables
    for (int k = 0; k < n; k++) {
      if (k == j || !seen[k]) continue;
      const int i = Rmate[k];
      v[k] += dist[k] - dist[j];
      u[i] -= dist[k] - dist[j];
    }
    u[s] += dist[j];

    // augment along path
    while (dad[j] >= 0) {
      const int d = dad[j];
      Rmate[j] = Rmate[d];
      Lmate[Rmate[j]] = j;
      j = d;
    }
    Rmate[j] = s;
    Lmate[s] = j;

    mated++;
  }
```

```
  double value = 0;
  for (int i = 0; i < n; i++)
    value += cost[i][Lmate[i]];

  return value;
}
```

MaxBipartiteMatching.cc 5/34

```
// This code performs maximum bipartite matching.
//
// Running time: O(|E| |V|) -- often much faster in practice
//
//   INPUT: w[i][j] = edge between row node i and column node j
//   OUTPUT: mr[i] = assignment for row node i, -1 if unassigned
//           mc[j] = assignment for column node j, -1 if unassigned
//           function returns number of matches made

#include <vector>

using namespace std;

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

bool FindMatch(int i, const VVI &w, VI &mr, VI &mc, VI &seen) {
  for (int j = 0; j < 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 BipartiteMatching(const VVI &w, VI &mr, VI &mc) {
  mr = VI(w.size(), -1);
  mc = VI(w[0].size(), -1);

  int ct = 0;
  for (int i = 0; i < w.size(); i++) {
    VI seen(w[0].size());
    if (FindMatch(i, w, mr, mc, seen)) ct++;
  }
  return ct;
}
```

MinCut.cc 6/34

```
// Adjacency matrix implementation of Stoer-Wagner min cut algorithm.
//
// Running time:
//     O(|V|^3)
//
// INPUT:
//     - graph, constructed using AddEdge()
//
// OUTPUT:
//     - (min cut value, nodes in half of min cut)

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

using namespace std;

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

const int INF = 1000000000;

pair<int, VI> GetMinCut(VVI &weights) {
  int N = weights.size();
  VI used(N), cut, best_cut;
  int best_weight = -1;

  for (int phase = N-1; phase >= 0; phase--) {
    VI w = weights[0];
    VI added = used;
    int prev, last = 0;
    for (int i = 0; i < phase; i++) {
      prev = last;
      last = -1;
      for (int j = 1; j < N; j++)
        if (!added[j] && (last == -1 || w[j] > w[last])) last = j;
      if (i == phase-1) {
        for (int j = 0; j < N; j++) weights[prev][j] += weights[last][j];
        for (int j = 0; j < N; j++) weights[j][prev] = weights[prev][j];
        used[last] = true;
        cut.push_back(last);
        if (best_weight == -1 || w[last] < best_weight) {
          best_cut = cut;
          best_weight = w[last];
        }
      } else {
        for (int j = 0; j < N; j++)
          w[j] += weights[last][j];
        added[last] = true;
      }
    }
  }
  return make_pair(best_weight, best_cut);
}
```

GraphCutInference.cc 7/34

```
// Special-purpose {0,1} combinatorial optimization solver for
// problems of the following by a reduction to graph cuts:
//
//        minimize         sum_i  psi_i(x[i])
//  x[1]...x[n] in {0,1}      + sum_{i < j}  phi_{ij}(x[i], x[j])
//
// where
//      psi_i : {0, 1} --> R
//   phi_{ij} : {0, 1} x {0, 1} --> R
//
// such that
//   phi_{ij}(0,0) + phi_{ij}(1,1) <= phi_{ij}(0,1) + phi_{ij}(1,0)  (*)
//
// This can also be used to solve maximization problems where the
// direction of the inequality in (*) is reversed.
//
```

```
// INPUT: phi -- a matrix such that phi[i][j][u][v] = phi_{ij}(u, v)
//        psi -- a matrix such that psi[i][u] = psi_i(u)
//        x -- a vector where the optimal solution will be stored
//
// OUTPUT: value of the optimal solution
//
// To use this code, create a GraphCutInference object, and call the
// DoInference() method.  To perform maximization instead of minimization,
// ensure that #define MAXIMIZATION is enabled.

#include <vector>
#include <iostream>

using namespace std;

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

const int INF = 1000000000;

// comment out following line for minimization
#define MAXIMIZATION

struct GraphCutInference {
  int N;
  VVI cap, flow;
  VI reached;

  int Augment(int s, int t, int a) {
    reached[s] = 1;
    if (s == t) return a;
    for (int k = 0; k < N; k++) {
      if (reached[k]) continue;
      if (int aa = min(a, cap[s][k] - flow[s][k])) {
        if (int b = Augment(k, t, aa)) {
          flow[s][k] += b;
          flow[k][s] -= b;
          return b;
        }
      }
    }
    return 0;
  }

  int GetMaxFlow(int s, int t) {
    N = cap.size();
    flow = VVI(N, VI(N));
    reached = VI(N);

    int totflow = 0;
    while (int amt = Augment(s, t, INF)) {
      totflow += amt;
      fill(reached.begin(), reached.end(), 0);
    }
    return totflow;
  }

  int DoInference(const VVVVI &phi, const VVI &psi, VI &x) {
    int M = phi.size();
    cap = VVI(M+2, VI(M+2));
    VI b(M);
    int c = 0;

    for (int i = 0; i < M; i++) {
      b[i] += psi[i][1] - psi[i][0];
      c += psi[i][0];
      for (int j = 0; j < i; j++)
        b[i] += phi[i][j][1][1] - phi[i][j][0][1];
      for (int j = i+1; j < M; j++) {
        cap[i][j] = phi[i][j][0][1] + phi[i][j][1][0] - phi[i][j][0][0] - phi[i]
[j][1][1];
        b[i] += phi[i][j][1][0] - phi[i][j][0][0];
        c += phi[i][j][0][0];
      }
    }

#ifdef MAXIMIZATION
    for (int i = 0; i < M; i++) {
      for (int j = i+1; j < M; j++)
        cap[i][j] *= -1;
      b[i] *= -1;
    }
    c *= -1;
#endif

    for (int i = 0; i < M; i++) {
      if (b[i] >= 0) {
        cap[M][i] = b[i];
      } else {
        cap[i][M+1] = -b[i];
        c += b[i];
      }
    }

    int score = GetMaxFlow(M, M+1);
    fill(reached.begin(), reached.end(), 0);
    Augment(M, M+1, INF);
    x = VI(M);
    for (int i = 0; i < M; i++) x[i] = reached[i] ? 0 : 1;
    score += c;
#ifdef MAXIMIZATION
    score *= -1;
#endif

    return score;
  }

};


Geometry.cc 9/34
// 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;
}
```

Geom3D.java 11/34

```
public class Geom3D {
  // distance from point (x, y, z) to plane aX + bY + cZ + d = 0
  public static double ptPlaneDist(double x, double y, double z,
      double a, double b, double c, double d) {
    return Math.abs(a*x + b*y + c*z + d) / Math.sqrt(a*a + b*b + c*c);
  }

  // distance between parallel planes aX + bY + cZ + d1 = 0 and
  // aX + bY + cZ + d2 = 0
  public static double planePlaneDist(double a, double b, double c,
      double d1, double d2) {
    return Math.abs(d1 - d2) / Math.sqrt(a*a + b*b + c*c);
  }

  // distance from point (px, py, pz) to line (x1, y1, z1)-(x2, y2, z2)
  // (or ray, or segment; in the case of the ray, the endpoint is the
  // first point)
  public static final int LINE = 0;
  public static final int SEGMENT = 1;
  public static final int RAY = 2;
  public static double ptLineDistSq(double x1, double y1, double z1,
      double x2, double y2, double z2, double px, double py, double pz,
      int type) {
    double pd2 = (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) + (z1-z2)*(z1-z2);

    double x, y, z;
    if (pd2 == 0) {
      x = x1;
      y = y1;
      z = z1;
    } else {
      double u = ((px-x1)*(x2-x1) + (py-y1)*(y2-y1) + (pz-z1)*(z2-z1)) / pd2;
      x = x1 + u * (x2 - x1);
      y = y1 + u * (y2 - y1);
      z = z1 + u * (z2 - z1);
      if (type != LINE && u < 0) {
        x = x1;
        y = y1;
        z = z1;
      }
      if (type == SEGMENT && u > 1.0) {
        x = x2;
        y = y2;
        z = z2;
      }
    }

    return (x-px)*(x-px) + (y-py)*(y-py) + (z-pz)*(z-pz);
  }

  public static double ptLineDist(double x1, double y1, double z1,
      double x2, double y2, double z2, double px, double py, double pz,
      int type) {
    return Math.sqrt(ptLineDistSq(x1, y1, z1, x2, y2, z2, px, py, pz, type));
  }
}
```

Delaunay.cc 12/34

```
// 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;
}
```

Euclid.cc 13/34

```
// 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) {
  int tmp;
  while(b){a%=b; tmp=a; a=b; b=tmp;}
  return a;
}

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

// returns d = 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 solutions;
  int d = extended_euclid(a, n, x, y);
  if (!(b%d)) {
    x = mod (x*(b/d), n);
    for (int i = 0; i < d; i++)
      solutions.push_back(mod(x + i*(n/d), n));
  }
  return solutions;
}

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

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

// Chinese remainder theorem: find z such that
// z % x[i] = a[i] for all i.  Note that the solution is
// unique modulo M = lcm_i (x[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 &x, const VI &a) {
  PII ret = make_pair(a[0], x[0]);
  for (int i = 1; i < x.size(); i++) {
    ret = chinese_remainder_theorem(ret.second, ret.first, x[i], a[i]);
    if (ret.second == -1) break;
  }
  return ret;
}

// computes x and y such that ax + by = c; on failure, x = y =-1
void linear_diophantine(int a, int b, int c, int &x, int &y) {
  int d = gcd(a,b);
  if (c%d) {
    x = y = -1;
  } else {
    x = c/d * mod_inverse(a/d, b/d);
    y = (c-a*x)/b;
  }
}

int main() {

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

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

  // expected: 95 45
  VI sols = modular_linear_equation_solver(14, 30, 100);
  for (int i = 0; i < (int) sols.size(); i++) cout << sols[i] << " ";
  cout << endl;

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

  // expected: 23 56
  //           11 12
  int xs[] = {3, 5, 7, 4, 6};
  int as[] = {2, 3, 2, 3, 5};
  PII ret = chinese_remainder_theorem(VI (xs, xs+3), VI(as, as+3));
  cout << ret.first << " " << ret.second << endl;
  ret = chinese_remainder_theorem (VI(xs+3, xs+5), VI(as+3, as+5));
  cout << ret.first << " " << ret.second << endl;

  // expected: 5 -15
  linear_diophantine(7, 2, 5, x, y);
  cout << x << " " << y << endl;

}
```

GaussJordan.cc 14/34

```
// Gauss-Jordan elimination with full pivoting.
//
// Uses:
//   (1) solving systems of linear equations (AX=B)
//   (2) inverting matrices (AX=I)
//   (3) computing determinants of square matrices
//
// Running time: O(n^3)
//
// INPUT:    a[][] = an nxn matrix
//           b[][] = an nxm matrix
//
// OUTPUT:   X      = an nxm matrix (stored in b[][])
//           A^{-1} = an nxn matrix (stored in a[][])
//           returns determinant of a[][]

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

using namespace std;

const double EPS = 1e-10;

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

T GaussJordan(VVT &a, VVT &b) {
  const int n = a.size();
  const int m = b[0].size();
  VI irow(n), icol(n), ipiv(n);
  T det = 1;

  for (int i = 0; i < n; i++) {
    int pj = -1, pk = -1;
    for (int j = 0; j < n; j++) if (!ipiv[j])
      for (int k = 0; k < n; k++) if (!ipiv[k])
        if (pj == -1 || fabs(a[j][k]) > fabs(a[pj][pk])) { pj = j; pk = k; }
    if (fabs(a[pj][pk]) < EPS) { cerr << "Matrix is singular." << endl; exit(0);
}
    ipiv[pk]++;
    swap(a[pj], a[pk]);
    swap(b[pj], b[pk]);
    if (pj != pk) det *= -1;
    irow[i] = pj;
    icol[i] = pk;

    T c = 1.0 / a[pk][pk];
    det *= a[pk][pk];
    a[pk][pk] = 1.0;
    for (int p = 0; p < n; p++) a[pk][p] *= c;
    for (int p = 0; p < m; p++) b[pk][p] *= c;
    for (int p = 0; p < n; p++) if (p != pk) {
      c = a[p][pk];
      a[p][pk] = 0;
      for (int q = 0; q < n; q++) a[p][q] -= a[pk][q] * c;
      for (int q = 0; q < m; q++) b[p][q] -= b[pk][q] * c;
    }
  }

  for (int p = n-1; p >= 0; p--) if (irow[p] != icol[p]) {
    for (int k = 0; k < n; k++) swap(a[k][irow[p]], a[k][icol[p]]);
  }

  return det;
}

int main() {
```

```
  const int n = 4;
  const int m = 2;
  double A[n][n] = { {1,2,3,4},{1,0,1,0},{5,3,2,4},{6,1,4,6} };
  double B[n][m] = { {1,2},{4,3},{5,6},{8,7} };
  VVT a(n), b(n);
  for (int i = 0; i < n; i++) {
    a[i] = VT(A[i], A[i] + n);
    b[i] = VT(B[i], B[i] + m);
  }

  double det = GaussJordan(a, b);

  // expected: 60
  cout << "Determinant: " << det << endl;

  // expected: -0.233333 0.166667 0.133333 0.0666667
  //           0.166667 0.166667 0.333333 -0.333333
  //           0.233333 0.833333 -0.133333 -0.0666667
  //           0.05 -0.75 -0.1 0.2
  cout << "Inverse: " << endl;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++)
      cout << a[i][j] << ' ';
    cout << endl;
  }

  // expected: 1.63333 1.3
  //           -0.166667 0.5
  //           2.36667 1.7
  //           -1.85 -1.35
  cout << "Solution: " << endl;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++)
      cout << b[i][j] << ' ';
    cout << endl;
  }
}
```

ReducedRowEchelonForm.cc 15/34

```
// Reduced row echelon form via Gauss-Jordan elimination
// with partial pivoting.  This can be used for computing
// the rank of a matrix.
//
// Running time: O(n^3)
//
// INPUT:    a[][] = an nxm matrix
//
// OUTPUT:   rref[][] = an nxm matrix (stored in a[][])
//           returns rank of a[][]

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

using namespace std;

const double EPSILON = 1e-10;

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

int rref(VVT &a) {
  int n = a.size();
  int m = a[0].size();
  int r = 0;
  for (int c = 0; c < m && r < n; c++) {
    int j = r;
    for (int i = r+1; i < n; i++)
      if (fabs(a[i][c]) > fabs(a[j][c])) j = i;
    if (fabs(a[j][c]) < EPSILON) continue;
    swap(a[j], a[r]);

    T s = 1.0 / a[r][c];
    for (int j = 0; j < m; j++) a[r][j] *= s;
    for (int i = 0; i < n; i++) if (i != r) {
      T t = a[i][c];
      for (int j = 0; j < m; j++) a[i][j] -= t * a[r][j];
    }
    r++;
  }
  return r;
}

int main(){
  const int n = 5;
  const int m = 4;
  double A[n][m] = { {16,2,3,13},{5,11,10,8},{9,7,6,12},{4,14,15,1},
{13,21,21,13} };
  VVT a(n);
  for (int i = 0; i < n; i++)
    a[i] = VT(A[i], A[i] + n);

  int rank = rref (a);

  // expected: 4
  cout << "Rank: " << rank << endl;

  // expected: 1 0 0 1
  //           0 1 0 3
  //           0 0 1 -3
  //           0 0 0 2.78206e-15
  //           0 0 0 3.22398e-15
  cout << "rref: " << endl;
  for (int i = 0; i < 5; i++){
    for (int j = 0; j < 4; j++)
      cout << a[i][j] << ' ';
    cout << endl;
  }

}
```

FFT_new.cpp 16/34

```
#include <cassert>
#include <cstdio>
#include <cmath>

struct cpx {
  cpx(){}
  cpx(double aa):a(aa){}
  cpx(double aa, double bb):a(aa),b(bb){}
  double a;
  double b;
  double modsq(void) const {
```

```
    return a * a + b * b;
  }
  cpx bar(void) const {
    return cpx(a, -b);
  }
};

cpx operator +(cpx a, cpx b) {
  return cpx(a.a + b.a, a.b + b.b);
}

cpx operator *(cpx a, cpx b) {
  return cpx(a.a * b.a - a.b * b.b, a.a * b.b + a.b * b.a);
}

cpx operator /(cpx a, cpx b) {
  cpx r = a * b.bar();
  return cpx(r.a / b.modsq(), r.b / b.modsq());
}

cpx EXP(double theta) {
  return cpx(cos(theta),sin(theta));
}

const double two_pi = 4 * acos(0);

// in:     input array
// out:    output array
// step:   {SET TO 1} (used internally)
// size:   length of the input/output {MUST BE A POWER OF 2}
// dir:    either plus or minus one (direction of the FFT)
// RESULT: out[k] = \sum_{j=0}^{size - 1} in[j] * exp(dir * 2pi * i * j * k /
size)
void FFT(cpx *in, cpx *out, int step, int size, int dir) {
  if(size < 1) return;
  if(size == 1) {
    out[0] = in[0];
    return;
  }
  FFT(in, out, step * 2, size / 2, dir);
  FFT(in + step, out + size / 2, step * 2, size / 2, dir);
  for(int i = 0 ; i < size / 2 ; i++) {
    cpx even = out[i];
    cpx odd = out[i + size / 2];
    out[i] = even + EXP(dir * two_pi * i / size) * odd;
    out[i + size / 2] = even + EXP(dir * two_pi * (i + size / 2) / size) * odd;
  }
}

// Usage:
// f[0...N-1] and g[0..N-1] are numbers
// Want to compute the convolution h, defined by
// h[n] = sum of f[k]g[n-k] (k = 0, ..., N-1).
// Here, the index is cyclic; f[-1] = f[N-1], f[-2] = f[N-2], etc.
// Let F[0...N-1] be FFT(f), and similarly, define G and H.
// The convolution theorem says H[n] = F[n]G[n] (element-wise product).
// To compute h[] in O(N log N) time, do the following:
//   1. Compute F and G (pass dir = 1 as the argument).
//   2. Get H by element-wise multiplying F and G.
//   3. Get h by taking the inverse FFT (use dir = -1 as the argument)
//      and *dividing by N*. DO NOT FORGET THIS SCALING FACTOR.

int main(void) {
  printf("If rows come in identical pairs, then everything works.\n");

  cpx a[8] = {0, 1, cpx(1,3), cpx(0,5), 1, 0, 2, 0};
  cpx b[8] = {1, cpx(0,-2), cpx(0,1), 3, -1, -3, 1, -2};
  cpx A[8];
  cpx B[8];
  FFT(a, A, 1, 8, 1);
  FFT(b, B, 1, 8, 1);

  for(int i = 0 ; i < 8 ; i++) {
    printf("%7.2lf%7.2lf", A[i].a, A[i].b);
  }
  printf("\n");
  for(int i = 0 ; i < 8 ; i++) {
    cpx Ai(0,0);
    for(int j = 0 ; j < 8 ; j++) {
      Ai = Ai + a[j] * EXP(j * i * two_pi / 8);
    }
    printf("%7.2lf%7.2lf", Ai.a, Ai.b);
  }
  printf("\n");

  cpx AB[8];
  for(int i = 0 ; i < 8 ; i++)
    AB[i] = A[i] * B[i];
  cpx aconvb[8];
  FFT(AB, aconvb, 1, 8, -1);
  for(int i = 0 ; i < 8 ; i++)
    aconvb[i] = aconvb[i] / 8;
  for(int i = 0 ; i < 8 ; i++) {
    printf("%7.2lf%7.2lf", aconvb[i].a, aconvb[i].b);
  }
  printf("\n");
  for(int i = 0 ; i < 8 ; i++) {
    cpx aconvbi(0,0);
    for(int j = 0 ; j < 8 ; j++) {
      aconvbi = aconvbi + a[j] * b[(8 + i - j) % 8];
    }
    printf("%7.2lf%7.2lf", aconvbi.a, aconvbi.b);
  }
  printf("\n");

  return 0;
}


Simplex.cc 17/34
// Two-phase simplex algorithm for solving linear programs of the form
//
//     maximize     c^T x
//     subject to   Ax <= b
//                  x >= 0
//
// INPUT: A -- an m x n matrix
//        b -- an m-dimensional vector
//        c -- an n-dimensional vector
//        x -- a vector where the optimal solution will be stored
//
// OUTPUT: value of the optimal solution (infinity if unbounded
//         above, nan if infeasible)
//
```

```
// To use this code, create an LPSolver object with A, b, and c as
// arguments.  Then, call Solve(x).

#include <iostream>
#include <iomanip>
#include <vector>
#include <cmath>
#include <limits>

using namespace std;

typedef long double DOUBLE;
typedef vector<DOUBLE> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;

const DOUBLE EPS = 1e-9;

struct LPSolver {
  int m, n;
  VI B, N;
  VVD D;

  LPSolver(const VVD &A, const VD &b, const VD &c) :
    m(b.size()), n(c.size()), N(n+1), B(m), D(m+2, VD(n+2)) {
    for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) D[i][j] = A[i][j];
    for (int i = 0; i < m; i++) { B[i] = n+i; D[i][n] = -1; D[i][n+1] = b[i]; }
    for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j]; }
    N[n] = -1; D[m+1][n] = 1;
  }

  void Pivot(int r, int s) {
    for (int i = 0; i < m+2; i++) if (i != r)
      for (int j = 0; j < n+2; j++) if (j != s)
        D[i][j] -= D[r][j] * D[i][s] / D[r][s];
    for (int j = 0; j < n+2; j++) if (j != s) D[r][j] /= D[r][s];
    for (int i = 0; i < m+2; i++) if (i != r) D[i][s] /= -D[r][s];
    D[r][s] = 1.0 / D[r][s];
    swap(B[r], N[s]);
  }

  bool Simplex(int phase) {
    int x = phase == 1 ? m+1 : m;
    while (true) {
      int s = -1;
      for (int j = 0; j <= n; j++) {
        if (phase == 2 && N[j] == -1) continue;
        if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] && N[j] < N[s]) s
= j;
      }
      if (D[x][s] >= -EPS) return true;
      int r = -1;
      for (int i = 0; i < m; i++) {
        if (D[i][s] <= 0) continue;
        if (r == -1 || D[i][n+1] / D[i][s] < D[r][n+1] / D[r][s] ||
            D[i][n+1] / D[i][s] == D[r][n+1] / D[r][s] && B[i] < B[r]) r = i;
      }
      if (r == -1) return false;
      Pivot(r, s);
    }
  }

  DOUBLE Solve(VD &x) {
    int r = 0;
    for (int i = 1; i < m; i++) if (D[i][n+1] < D[r][n+1]) r = i;
    if (D[r][n+1] <= -EPS) {
      Pivot(r, n);
      if (!Simplex(1) || D[m+1][n+1] < -EPS) return
-numeric_limits<DOUBLE>::infinity();
      for (int i = 0; i < m; i++) if (B[i] == -1) {
        int s = -1;
        for (int j = 0; j <= n; j++)
          if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s] && N[j] < N[s])
s = j;
        Pivot(i, s);
      }
    }
    if (!Simplex(2)) return numeric_limits<DOUBLE>::infinity();
    x = VD(n);
    for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n+1];
    return D[m][n+1];
  }
};

int main() {

  const int m = 4;
  const int n = 3;
  DOUBLE _A[m][n] = { { 6, -1, 0 }, { -1, -5, 0 }, { 1, 5, 1 }, { -1, -5, -1 }
  };
  DOUBLE _b[m] = { 10, -4, 5, -5 };
  DOUBLE _c[n] = { 1, -1, 0 };

  VVD A(m);
  VD b(_b, _b + m);
  VD c(_c, _c + n);
  for (int i = 0; i < m; i++) A[i] = VD(_A[i], _A[i] + n);

  LPSolver solver(A, b, c);
  VD x;
  DOUBLE value = solver.Solve(x);

  cerr << "VALUE: "<< value << endl;
  cerr << "SOLUTION:";
  for (size_t i = 0; i < x.size(); i++) cerr << " " << x[i];
  cerr << endl;
  return 0;
}
```

SCC.cc 19/34

```
#include<memory.h>
struct edge{int e, nxt;};
int V, E;
edge e[MAXE], er[MAXE];
int sp[MAXV], spr[MAXV];
int group_cnt, group_num[MAXV];
bool v[MAXV];
int stk[MAXV];
void fill_forward(int x) {
  int i;
  v[x]=true;
  for(i=sp[x];i;i=e[i].nxt) if(!v[e[i].e]) fill_forward(e[i].e);
```

```
  stk[++stk[0]]=x;
}
void fill_backward(int x) {
  int i;
  v[x]=false;
  group_num[x]=group_cnt;
  for(i=spr[x];i;i=er[i].nxt) if(v[er[i].e]) fill_backward(er[i].e);
}
void add_edge(int v1, int v2) //add edge v1->v2 {
  e [++E].e=v2; e [E].nxt=sp [v1]; sp [v1]=E;
  er[  E].e=v1; er[E].nxt=spr[v2]; spr[v2]=E;
}
void SCC() {
  int i;
  stk[0]=0;
  memset(v, false, sizeof(v));
  for(i=1;i<=V;i++) if(!v[i]) fill_forward(i);
  group_cnt=0;
  for(i=stk[0];i>=1;i--) if(v[stk[i]]){group_cnt++; fill_backward(stk[i]);}
}
```

EulerianPath.cc 20/34

```
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;
}
```

BIT.cc 22/34

```
#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;
}
```

UnionFind.cc 23/34

```
//union-find set: the vector/array contains the parent of each node
int find(vector <int>& C, int x){return (C[x]==x) ? x : C[x]=find(C, C[x]);}
//C++
int find(int x){return (C[x]==x)?x:C[x]=find(C[x]);} //C
```

KDTree.cc 24/34

```
// ----------------------------------------------------------------------------
// A straightforward, but probably sub-optimal KD-tree implmentation that's
// probably good enough for most things (current it's a 2D-tree)
//
//  - constructs from n points in O(n lg^2 n) time
//  - handles nearest-neighbor query in O(lg n) if points are well distributed
//  - worst case for nearest-neighbor may be linear in pathological case
//
// Sonny Chan, Stanford University, April 2009
// ----------------------------------------------------------------------------

#include <iostream>
#include <vector>
#include <limits>
#include <cstdlib>
```

```
using namespace std;

// number type for coordinates, and its maximum value
typedef long long ntype;
const ntype sentry = numeric_limits<ntype>::max();

// point structure for 2D-tree, can be extended to 3D
struct point {
    ntype x, y;
    point(ntype xx = 0, ntype yy = 0) : x(xx), y(yy) {}
};

bool operator==(const point &a, const point &b) {
    return a.x == b.x && a.y == b.y;
}

// sorts points on x-coordinate
bool on_x(const point &a, const point &b) {
    return a.x < b.x;
}

// sorts points on y-coordinate
bool on_y(const point &a, const point &b) {
    return a.y < b.y;
}

// squared distance between points
ntype pdist2(const point &a, const point &b) {
    ntype dx = a.x-b.x, dy = a.y-b.y;
    return dx*dx + dy*dy;
}

// bounding box for a set of points
struct bbox {
    ntype x0, x1, y0, y1;

    bbox() : x0(sentry), x1(-sentry), y0(sentry), y1(-sentry) {}

    // computes bounding box from a bunch of points
    void compute(const vector<point> &v) {
        for (int i = 0; i < v.size(); ++i) {
            x0 = min(x0, v[i].x);   x1 = max(x1, v[i].x);
            y0 = min(y0, v[i].y);   y1 = max(y1, v[i].y);
        }
    }

    // squared distance between a point and this bbox, 0 if inside
    ntype distance(const point &p) {
        if (p.x < x0) {
            if (p.y < y0)       return pdist2(point(x0, y0), p);
            else if (p.y > y1)  return pdist2(point(x0, y1), p);
            else                return pdist2(point(x0, p.y), p);
        }
        else if (p.x > x1) {
            if (p.y < y0)       return pdist2(point(x1, y0), p);
            else if (p.y > y1)  return pdist2(point(x1, y1), p);
            else                return pdist2(point(x1, p.y), p);
        }
        else {
            if (p.y < y0)       return pdist2(point(p.x, y0), p);
            else if (p.y > y1)  return pdist2(point(p.x, y1), p);
            else                return 0;
        }
    }
};

// stores a single node of the kd-tree, either internal or leaf
struct kdnode  {
    bool leaf;      // true if this is a leaf node (has one point)
    point pt;       // the single point of this is a leaf
    bbox bound;     // bounding box for set of points in children

    kdnode *first, *second; // two children of this kd-node

    kdnode() : leaf(false), first(0), second(0) {}
    ~kdnode() { if (first) delete first; if (second) delete second; }

    // intersect a point with this node (returns squared distance)
    ntype intersect(const point &p) {
        return bound.distance(p);
    }

    // recursively builds a kd-tree from a given cloud of points
    void construct(vector<point> &vp) {
        // compute bounding box for points at this node
        bound.compute(vp);

        // if we're down to one point, then we're a leaf node
        if (vp.size() == 1) {
            leaf = true;
            pt = vp[0];
        }
        else {
            // split on x if the bbox is wider than high (not best heuristic...)
            if (bound.x1-bound.x0 >= bound.y1-bound.y0)
                sort(vp.begin(), vp.end(), on_x);
            // otherwise split on y-coordinate
            else
                sort(vp.begin(), vp.end(), on_y);

            // divide by taking half the array for each child
            // (not best performance if many duplicates in the middle)
            int half = vp.size()/2;
            vector<point> vl(vp.begin(), vp.begin()+half);
            vector<point> vr(vp.begin()+half, vp.end());
            first = new kdnode();   first->construct(vl);
            second = new kdnode();  second->construct(vr);
        }
    }
};

// simple kd-tree class to hold the tree and handle queries
struct kdtree {
    kdnode *root;

    // constructs a kd-tree from a points (copied here, as it sorts them)
    kdtree(const vector<point> &vp) {
        vector<point> v(vp.begin(), vp.end());
        root = new kdnode();
        root->construct(v);
    }
    ~kdtree() { delete root; }
```

```
    // recursive search method returns squared distance to nearest point
    ntype search(kdnode *node, const point &p) {
        if (node->leaf) {
            // commented special case tells a point not to find itself
//            if (p == node->pt) return sentry;
//            else
              return pdist2(p, node->pt);
        }

        ntype bfirst = node->first->intersect(p);
        ntype bsecond = node->second->intersect(p);

        // choose the side with the closest bounding box to search first
        // (note that the other side is also searched if needed)
        if (bfirst < bsecond) {
            ntype best = search(node->first, p);
            if (bsecond < best)
                best = min(best, search(node->second, p));
            return best;
        }
        else {
            ntype best = search(node->second, p);
            if (bfirst < best)
                best = min(best, search(node->first, p));
            return best;
        }
    }

    // squared distance to the nearest
    ntype nearest(const point &p) {
        return search(root, p);
    }
};

// --------------------------------------------------------------------------
// some basic test code here

int main() {
    // generate some random points for a kd-tree
    vector<point> vp;
    for (int i = 0; i < 100000; ++i) {
        vp.push_back(point(rand()%100000, rand()%100000));
    }
    kdtree tree(vp);

    // query some points
    for (int i = 0; i < 10; ++i) {
        point q(rand()%100000, rand()%100000);
        cout << "Closest squared distance to (" << q.x << ", " << q.y << ")"
             << " is " << tree.nearest(q) << endl;
    }

    return 0;
}

// --------------------------------------------------------------------------
```

```
LCA.cc 26/34
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;
}
```

LongestIncreasingSubsequence.cc 27/34

```
// Given a list of numbers of length n, this routine extracts a
// longest increasing subsequence.
//
// Running time: O(n log n)
//
//   INPUT: a vector of integers
//   OUTPUT: a vector containing the longest increasing subsequence

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

using namespace std;

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

#define STRICTLY_INCREASNG

VI LongestIncreasingSubsequence(VI v) {
  VPII best;
  VI dad(v.size(), -1);

  for (int i = 0; i < v.size(); i++) {
#ifdef STRICTLY_INCREASNG
    PII item = make_pair(v[i], 0);
    VPII::iterator it = lower_bound(best.begin(), best.end(), item);
    item.second = i;
#else
    PII item = make_pair(v[i], i);
    VPII::iterator it = upper_bound(best.begin(), best.end(), item);
#endif
    if (it == best.end()) {
      dad[i] = (best.size() == 0 ? -1 : best.back().second);
      best.push_back(item);
    } else {
      dad[i] = dad[it->second];
      *it = item;
    }
  }

  VI ret;
  for (int i = best.back().second; i >= 0; i = dad[i])
    ret.push_back(v[i]);
  reverse(ret.begin(), ret.end());
  return ret;
}
```

Dates.cc 28/34

```
// Routines for performing computations on dates.  In these routines,
// months are expressed as integers from 1 to 12, days are expressed
// as integers from 1 to 31, and years are expressed as 4-digit
// integers.

#include <iostream>
#include <string>

using namespace std;

string dayOfWeek[] = {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"};

// converts Gregorian date to integer (Julian day number)
int dateToInt (int m, int d, int y){
  return
    1461 * (y + 4800 + (m - 14) / 12) / 4 +
    367 * (m - 2 - (m - 14) / 12 * 12) / 12 -
    3 * ((y + 4900 + (m - 14) / 12) / 100) / 4 +
    d - 32075;
}

// converts integer (Julian day number) to Gregorian date: month/day/year
void intToDate (int jd, int &m, int &d, int &y){
  int x, n, i, j;

  x = jd + 68569;
  n = 4 * x / 146097;
  x -= (146097 * n + 3) / 4;
  i = (4000 * (x + 1)) / 1461001;
  x -= 1461 * i / 4 - 31;
  j = 80 * x / 2447;
  d = x - 2447 * j / 80;
  x = j / 11;
  m = j + 2 - 12 * x;
  y = 100 * (n - 49) + i + x;
}

// converts integer (Julian day number) to day of week
string intToDay (int jd){
  return dayOfWeek[jd % 7];
}

int main (int argc, char **argv){
  int jd = dateToInt (3, 24, 2004);
  int m, d, y;
  intToDate (jd, m, d, y);
  string day = intToDay (jd);

  // expected output:
  //    2453089
  //    3/24/2004
  //    Wed
  cout << jd << endl
    << m << "/" << d << "/" << y << endl
    << day << endl;
}
```

```
// Other primes:
//    The largest prime smaller than 10 is 7.
//    The largest prime smaller than 100 is 97.
//    The largest prime smaller than 1000 is 997.
//    The largest prime smaller than 10000 is 9973.
//    The largest prime smaller than 100000 is 99991.
//    The largest prime smaller than 1000000 is 999983.
//    The largest prime smaller than 10000000 is 9999991.
//    The largest prime smaller than 100000000 is 99999989.
//    The largest prime smaller than 1000000000 is 999999937.
//    The largest prime smaller than 10000000000 is 9999999967.
//    The largest prime smaller than 100000000000 is 99999999977.
//    The largest prime smaller than 1000000000000 is 999999999989.
//    The largest prime smaller than 10000000000000 is 9999999999971.
//    The largest prime smaller than 100000000000000 is 99999999999973.
//    The largest prime smaller than 1000000000000000 is 999999999999989.
//    The largest prime smaller than 10000000000000000 is 9999999999999937.
//    The largest prime smaller than 100000000000000000 is 99999999999999997.
//    The largest prime smaller than 1000000000000000000 is 999999999999999989.


// SPLAY TREE

/* splay tree with kth element, number of elements less than k, insert, erase */
struct splay_tree // cannot have duplicates {
 struct node {
 node *C[2];
 node *P;
 int v, ss;

        node(int v) : v(v), ss(1), P(NULL) {memset(C, NULL, sizeof(C));}

        int d() {
            return (this == P->C[1]);
        }
        void setc(node *x, int d) {
            C[d] = x;
            if (x) x->P = this;
        }
        void upd() {
            ss = 1 + (C[0] ? C[0]->ss : 0) + (C[1] ? C[1]->ss : 0);
        }
    } *root;

    void rotate(node *x) {
        node *y = x->P;
        int d = x->d();

        if (y->P) y->P->setc(x, y->d());
        else root = x, x->P = NULL;

        y->setc(x->C[!d], d);
        x->setc(y, !d);

        y->upd();
        x->upd();
    }

    void splay(node *x) {
        if (!x) return;

        node *y;
        while (x->P) {
            if ((y = x->P)->P) {
                if (y->d() == x->d()) rotate(y);
                else rotate(x);
            }
            rotate(x);
        }
    }

    node *find(int v) {
        node *p, *x = root;
        while (x) {
            p = x;
            if (x->v == v) {
                splay(x);
                return x;
            }
            x = x->C[x->v < v];
        }
        splay(p);
        return p;
    }

    node *max(node *x) {
        if (!x) return x;
        while (x->C[1]) x = x->C[1];
        splay(x);
        return x;
    }

    node *min(node *x) {
        if (!x) return x;
        while (x->C[0]) x = x->C[0];
        splay(x);
        return x;
    }

    node *lower_bound(int v) {
        node *ans = NULL, *x = root;
        while (x) {
            if (x->v >= v)
                ans = x;
            if (x->v == v) break;
            x = x->C[x->v < v];
        }
        splay(ans);
        return ans;
    }

    void insert(int v) {
        if (!root) {
            root = new node(v);
        }
        else {
            node *x = lower_bound(v);
            if (!x) {
                node *y = new node(v);
                y->setc(root, 0);
                root = y;
```

```
            root->upd();
        }
        else {
            if (x->v == v) return;
            node *y = new node(v);
            y->setc(x->C[0], 0);
            x->setc(y, 0);

            y->upd();
            x->upd();
        }
    }
}

void erase(int v) {
    node *x = lower_bound(v);
    if (!x || x->v != v) return;

    splay_tree *lside = new splay_tree(), *rside = new splay_tree();
    lside->root = x->C[0]; if (lside->root) lside->root->P = NULL;
    rside->root = x->C[1]; if (rside->root) rside->root->P = NULL;
    delete x;

    if (!lside->root) {
        root = rside->root;
    }
    else if (!rside->root) {
        root = lside->root;
    }
    else {
        lside->max(lside->root);
        lside->root->setc(rside->root, 1);
        root = lside->root;
    }
    if (root) root->upd();
}

node *kth(int k) // 0-indexed {
    if (!root) return NULL;
    if (k < 0 || k >= root->ss) return NULL;

    node *x = root;
    int at = (x->C[0] ? x->C[0]->ss : 0);
    while (at != k) {
        if (at < k) {
            x = x->C[1];
            at += (x->C[0] ? x->C[0]->ss : 0) + 1;
        }
        else {
            x = x->C[0];
            at -= (x->C[1] ? x->C[1]->ss : 0) + 1;
        }
    }
    splay(x);
    return x;
}

int count(int v) // number < v {
    if (!root) return 0;

    node *x = lower_bound(v);
        if (!x) return root->ss;

        return (x->C[0] ? x->C[0]->ss : 0);
    }

    void print_at(node *at) {
        if (!at) return;
        cout << "( ";
        print_at(at->C[0]);
        cout << " ) [" << at->v << "] ( ";
        print_at(at->C[1]);
        cout << " )";
    }

    void print() {
        print_at(root);
        cout << endl;
    }
};

/* link-cut tree */
struct node {
    node *P, *C[2];
    bool flip;
    node() : flip(0), P(0) {memset(C, 0, sizeof(C));}

    int d() {
        return (this == P->C[1]);
    }
    void setc(node *x, int c) {
        C[c] = x;
        if (x) x->P = this;
        update();
    }
    bool isroot() {
        return (P == NULL || P->C[d()] != this);
    }
    void update() {
        if (flip) {
            flip = false;
            swap(C[0], C[1]);
            if (C[0]) C[0]->flip = !C[0]->flip;
            if (C[1]) C[1]->flip = !C[1]->flip;
        }
    }
};
void rotate(node *x) {
    node *y = x->P;
    y->update();
    x->update();
    int d = x->d();

    if (y->isroot()) x->P = y->P;
    else y->P->setc(x, y->d());

    y->setc(x->C[!d], d);
    x->setc(y, !d);

    x->update();
    y->update();
}
void splay(node *x) {
```

```
    while (!x->isroot()) {
        x->P->update();
        if (!x->P->isroot()) {
            x->P->P->update();
            if (x->P->d() == x->d()) rotate(x->P);
            else rotate(x);
        }
        x->P->update();
        rotate(x);
    }
    x->update();
}
void expose(node *x) {
    node *from = NULL;
    for (node *p=x; p; p=p->P) {
        splay(p);
        p->setc(from, 0);
        p->update();
        from = p;
    }
    splay(x);
}
void make_root(node *x) {
    expose(x);
    x->flip = !x->flip;
    x->update();
}
node *getroot(node *x) {
    expose(x);
    while (x->C[1]) {
        x = x->C[1];
        x->update();
    }
    expose(x);
    return x;
}
bool same(node *x, node *y) {
    return getroot(x) == getroot(y);
}
bool link(node *x, node *y) {
    if (getroot(x) == getroot(y))
        return false;
    make_root(x);
    x->P = y;
    return true;
}
bool cut(node *x, node *y) {
    make_root(x);
    expose(y);
    if (y->C[1] == x) {
        x->P = NULL;
        y->C[1] = NULL;
        return true;
    }
    return false;
}


// BBST
#include <bits/stdc++.h>
using namespace std;
struct Node {
  Node *left, *right;
  int val, depth;
  Node(int a) : val(a), depth(1), left(NULL), right(NULL) {}
};
int height(Node *x) {
  return x == NULL ? 0 : x->depth;
}
void updateHeight(Node *x) {
  x->depth = max(height(x->left), height(x->right)) + 1;
}
Node *leftRotate(Node *x) {
  Node *a = x->right, *b = x->right->left;
  x->right = b;
  a->left = x;
  updateHeight(a);
  updateHeight(x);
  return a;
}
Node *rightRotate(Node *x) {
  Node *a = x->left, *b = x->left->right;
  x->left = b;
  a->right = x;
  updateHeight(a);
  updateHeight(x);
  return a;
}
void insert(Node *&x, int a) {
  if (x == NULL) {
    x = new Node(a);
    return;
  }
  if (a == x->val) return; // Already in tree
  if (a < x->val) insert(x->left, a);
  else insert(x->right, a);
  int lefth = height(x->left), righth = height(x->right);
  if (lefth - righth >= 2) {
    if (height(x->left->left) < height(x->left->right))
      x->left = leftRotate(x->left);
    x = rightRotate(x);
  } else if (righth - lefth >= 2) {
    if (height(x->right->left) > height(x->right->right))
      x->right = rightRotate(x->right);
    x = leftRotate(x);
  } else {
    updateHeight(x);
  }
}
void printTree(Node *x) {
  if (x == NULL) return;
  printTree(x->left);
  printf("%d ", x->val);
  printTree(x->right);
}
int A[1000005];
int main() {
  int N = 1000000;
  for (int i = 0; i < N; ++i) A[i] = i;
  srand(time(NULL));
  random_shuffle(A, A + N);
  Node *root = NULL;
  for (int i = 0; i < N; ++i) insert(root, A[i]);
  printf("%d\n", root->depth);
```

```
}
// Convex Hull
#include <bits/stdc++.h>
using namespace std;
struct Point {
  int x, y;
} P[2005];
bool cmp(Point a, Point b) {
  if (a.x != b.x) return a.x < b.x;
  return a.y < b.y;
}
int area2(Point &a, Point &b, Point &c) {
  return (b.x - a.x) * (c.y - a.y) - (c.x - a.x) * (b.y - a.y);
}
int main() {
  // Get counter-clockwise convex hull, starting at leftmost bottommost point
  for (int i = 0; i < N; ++i) scanf("%d%d", &P[i].x, &P[i].y);
  sort(P, P + N, cmp);
  vector<Point> up, dn, hull;
  for (int i = 0; i < N; ++i) {
    while (up.size() > 1 && area2(up[up.size() - 2], up.back(), P[i]) >= 0)
up.pop_back();
    while (dn.size() > 1 && area2(dn[dn.size() - 2], dn.back(), P[i]) <= 0)
dn.pop_back();
    up.push_back(P[i]);
    dn.push_back(P[i]);
  }
  hull = up;
  for (int i = (int)dn.size() - 2; i > 0; --i)
    hull.push_back(dn[i]);
  reverse(hull.begin(), hull.end());
}

// Min Cost Flow
#include <bits/stdc++.h>
using namespace std;
const int INF = 1000000000;
int C[105][105], cap[105][105], N, SRC, SINK, dist[105], pi[105], cost[105]
[105], dad[105];
int match[105];
bool done[105];
bool path() {
  memset(done, 0, sizeof(done));
  for (int i = 0; i <= SINK; ++i) dist[i] = INF;
  dist[SRC] = 0;
  for (;;) {
    int x = -1;
    for (int i = 0; i <= SINK; ++i)
      if (!done[i] && (x == -1 || dist[i] < dist[x]))
        x = i;
    if (x == -1) break;
    done[x] = true;
    for (int i = 0; i <= SINK; ++i)
      if (!done[i] && cap[x][i] > 0) {
        int nd = dist[x] + cost[x][i];
        if (nd < dist[i]) {
          dist[i] = nd;
          dad[i] = x;
        }
      }
  }
  if (dist[SINK] == INF) return false;
  int x = SINK;
  while (x != SRC) {
    ++cap[x][dad[x]];
    --cap[dad[x]][x];
    x = dad[x];
  }
  for (int i = 0; i <= SINK; ++i)
    pi[i] = min(pi[i] + dist[i], INF);
  return true;
}
void minCostFlow(int G[105][105]) {
  memset(pi, 0, sizeof(pi));
  for (;;) {
    for (int i = 0; i <= SINK; ++i)
      for (int j = 0; j <= SINK; ++j)
        cost[i][j] = G[i][j] + pi[i] - pi[j];
    if (!path()) break;
  }
  for (int i = 0; i < N; ++i)
    for (int j = N; j < 2 * N; ++j)
      if (!cap[i][j]) {
        match[i] = j;
      }
}
int main() {
  int cn = 1;
  while (scanf("%d", &N) == 1) {
    if (!N) break;
    SRC = 2 * N;
    SINK = 2 * N + 1;
    memset(cap, 0, sizeof(cap));
    memset(C, 0, sizeof(C));
    for (int i = 0; i < N; ++i) {
      cap[SRC][i] = cap[N + i][SINK] = 1;
      for (int j = N; j < SRC; ++j) {
        scanf("%d", &C[i][j]);
        C[j][i] = -C[i][j];
        cap[i][j] = 1;
      }
    }
    minCostFlow(C);
  }
}
// Aho Corasick
#include <bits/stdc++.h>
using namespace std;
char S[100005], word[100005];
struct Node {
  int matchlen;
  Node *fail;
  Node *nex[26];
  Node() {
    matchlen = 0;
    memset(nex, 0, sizeof(nex));
    fail = NULL;
  }
} *root;
char chst[100005];
Node mems[100005];
Node *nodest[100005];
int n, memi;
int main() {
```

```
  int N;
  scanf("%s%d", S, &N);
  root = &mems[memi++];
  for (int i = 0; i < N; ++i) {
    scanf("%s", word);
    int len = strlen(word);
    Node *p = root;
    for (int j = 0; j < len; ++j) {
      int c = word[j] - 'a';
      if (p->nex[c] == NULL) {
        p->nex[c] = &mems[memi++];
      }
      p = p->nex[c];
    }
    p->matchlen = len;
  }
  queue<Node*> Q;
  for (int i = 0; i < 26; ++i)
    if (root->nex[i] == NULL) {
      root->nex[i] = root;
    } else {
      root->nex[i]->fail = root;
      Q.push(root->nex[i]);
    }
  while (!Q.empty()) {
    Node *p = Q.front();
    Q.pop();
    for (int i = 0; i < 26; ++i) {
      Node *v = p->fail;
      while (v->nex[i] == NULL) v = v->fail;
      v = v->nex[i];
      if (p->nex[i] != NULL) {
        p->nex[i]->fail = v;
        Q.push(p->nex[i]);
      } else {
        p->nex[i] = v;
      }
    }
  }
  int M = strlen(S);
  Node *p = root;
  nodest[n++] = p;
  for (int i = 0; i < M; ++i) {
    int c = S[i] - 'a';
    p = p->nex[c];
    if (p->matchlen) {
      n -= p->matchlen - 1;
      p = nodest[n - 1];
    } else {
      chst[n] = S[i];
      nodest[n++] = p;
    }
  }
  chst[n] = '\0';
  printf("%s\n", chst + 1);
}
// Suffix Array
#include <bits/stdc++.h>
using namespace std;
const int MAXP = 17;
char S[100005];
int N;
pair<pair<int, int>, int> T[100005];
void getSuffixArray(int rank[MAXP + 1][100005]) {
  for (int i = 0; i < N; ++i) rank[0][i] = S[i];
  for (int l = 1; l <= MAXP; ++l) {
    int len = 1 << (l - 1);
    for (int i = 0; i < N; ++i) {
      int nex = i + len < N ? rank[l - 1][i + len] : -1;
      T[i] = make_pair(make_pair(rank[l - 1][i], nex), i);
    }
    sort(T, T + N);
    for (int i = 0; i < N; ++i) {
      if (i && T[i].first == T[i - 1].first) rank[l][T[i].second] = rank[l][T[i
- 1].second];
      else rank[l][T[i].second] = i;
    }
  }
}
int getLcp(int a, int b, int rank[MAXP + 1][100005]) {
  int len = 0;
  for (int l = MAXP; l >= 0; --l) {
    if (a + len >= N || b + len >= N) break;
    if (rank[l][a + len] == rank[l][b + len])
      len += 1 << l;
  }
  return len;
}
int main() {
  while (scanf("%s", S) == 1) {
    if (S[0] == '0') break;
    N = strlen(S);
    getSuffixArray(sufrank);
  }
}


// DINIC
const int MAXV = 3000000;
const int MAXE = 2 * 30000000;
const int INF = 1000000005, CAPINF = 1000000005;

template <typename T> struct Dinic {
 int V, source, sink;
 int eind, eadj [MAXE], eprev [MAXE], elast [MAXV], start [MAXV];
 int front, back, q [MAXV], dist [MAXV];
 T ecap [MAXE];

 inline void init (int v) {
 V = v; eind = 0;
 memset (elast, -1, V * sizeof (int));
 }

 inline void addedge (int a, int b, T cap1, T cap2) {
 eadj [eind] = b; ecap [eind] = cap1;
 eprev [eind] = elast [a]; elast [a] = eind++;
 eadj [eind] = a; ecap [eind] = cap2;
 eprev [eind] = elast [b]; elast [b] = eind++;
 }

 bool bfs () {
 memset (dist, 63, V * sizeof (int));
 front = back = 0;
 q [back++] = source; dist [source] = 0;
```

```
 while (front < back) {
 int top = q [front++];

 for (int i = elast [top]; i != -1; i = eprev [i])
 if (ecap [i] > 0 && dist [top] + 1 < dist [eadj [i]]) {
 dist [eadj [i]] = dist [top] + 1;
 q [back++] = eadj [i];
 }
 }

 return dist [sink] < INF;
 }

    T dfs (int num, T pcap) {
        if (num == sink) return pcap;
        T total = 0;

        for (int &i = start [num]; i != -1; i = eprev [i])
            if (ecap [i] > 0 && dist [num] + 1 == dist [eadj [i]]) {
                T p = dfs (eadj [i], min (pcap, ecap [i]));
                ecap [i] -= p; ecap [i ^ 1] += p;
                pcap -= p; total += p;
                if (pcap == 0) break;
            }

        return total;
    }

    T flow (int _source, int _sink) {
        if (V == 0) return -1;

        source = _source; sink = _sink;
        T total = 0;
        while (bfs ()) {
            memcpy (start, elast, V * sizeof (int));
            total += dfs (source, CAPINF);
        }

        return total;
    }
};



/***** MIN-COST FLOW *****/

const int MAX = 1215;
const int INF = 1231231231;

vector<int> G[MAX];
int cap[MAX][MAX];
int cost[MAX][MAX];
int pi[MAX];
int dist[MAX];
int from[MAX];

int mcf(int src, int snk, int flow) {
  for(int i = 0; i < MAX; i++)
    for(int j = 0; j < MAX; j++)
      if(cap[i][j])
        cost[j][i] = -cost[i][j];
  memset(pi, 0, sizeof(pi));

  int cst = 0;
  for(int f = 0; f < flow; ) {
    for(int i = 0; i < MAX; i++) dist[i] = INF;
    memset(from, -1, sizeof(from));
    dist[src] = 0, from[src] = -2;
    priority_queue<pair<int, int> > q;
    q.push(make_pair(0, src));
    while(!q.empty()) {
      pair<int, int> pr = q.top(); q.pop();
      int best = pr.second;
      if(abs(pr.first + dist[best]) > 0) continue;
      for(int vi = 0; vi < G[best].size(); vi++) {
        int i = G[best][vi];
        if(cap[best][i] && dist[best] + cost[best][i] + pi[best] - pi[i] <
dist[i]) {
          dist[i] = dist[best] + cost[best][i] + pi[best] - pi[i];
          from[i] = best;
          q.push(make_pair(-dist[i], i));
        }
      }
    }
    if(from[snk] == -1) return -1;
    for(int i = 0; i < MAX; i++) if(from[i] == -1) pi[i] += dist[i];

    int aug_f = flow - f;
    for(int v = snk; v != src; v = from[v])
      aug_f = min(aug_f, cap[from[v]][v]);

    for(int v = snk; v != src; v = from[v]) {
      int u = from[v];
      cap[u][v] -= aug_f;
      cap[v][u] += aug_f;
      cst += aug_f * cost[u][v];
    }
    f += aug_f;
  }
  return cst;
}

void add_edge(int u, int v, int cp, int cst) {
  G[u].push_back(v);
  G[v].push_back(u);
  cap[u][v] = cp;
  cap[v][u] = 0;
  cost[u][v] = cst;
}



/***** FFT WITH PRIMES *****/

typedef long long ll;

const ll P = 2013265921; // 15*2^27+1
const ll ORDER = (1 << 27);
const ll ROOT = 440564289; // ORDER'th root of unity
const int MAX = (1 << 16);

ll omega[MAX];
```

```
ll power(ll b, ll e) {
  ll res = 1;
  while(e > 0) {
    if(e % 2 == 1) res = (res * b) % P;
    b = (b * b) % P;
    e /= 2;
  }
  return res;
}

void fft(vector<ll> &A, int n, bool inverse = false) {
  int N = (1 << n);
  ll root = power(ROOT, ORDER / N * (inverse ? (N - 1) : 1));
  omega[0] = 1;
  for(int i = 1; i < N; i++) omega[i] = (omega[i - 1] * root) % P;
  for(int i = 0; i < n; i++) {
    for(int j = 0; j < (1 << i); j++) {
      for(int k = 0; k < (1 << (n - i - 1)); k++) {
        int s = (j << (n - i)) + k;
        int t = s + (1 << (n - i - 1));
        ll w = omega[k << i];
        ll temp = A[s] + A[t];
        if(temp >= P) temp -= P;
        ll temp2 = A[s] - A[t] + P;
        A[t] = (w * temp2) % P;
        A[s] = temp;
      }
    }
  }
  for(int i = 0; i < N; i++) {
    int x = i, y = 0;
    for(int j = 0; j < n; j++) {
      y = y * 2 + x % 2;
      x /= 2;
    }
    if(i < y) swap(A[i], A[y]);
  }
  if(inverse) {
    ll inv = power(N, P - 2);
    for(int i = 0; i < N; i++) A[i] = (A[i] * inv) % P;
  }
}

vector<ll> conv(vector<ll> A, vector<ll> B) {
  int N = A.size() + B.size();
  int n = 1;
  while((1 << n) < N) n++;
  while(A.size() < (1 << n)) A.push_back(0);
  while(B.size() < (1 << n)) B.push_back(0);
  fft(A, n);
  fft(B, n);
  for(int i = 0; i < (1 << n); i++) A[i] = (A[i] * B[i]) % P;
  fft(A, n, true);
  return A;
}

int main() {
  vector<ll> A(8), B(8);
  for(int i = 0; i < 8; i++) cin >> A[i];
  for(int i = 0; i < 8; i++) cin >> B[i];
  A = conv(A, B);
  for(int i = 0; i < A.size(); i++) cout << A[i] << ' ';
  cout << endl;
}
```

```
/***** MANACHER'S ALGORITHM *****/

int pals[2*MAX-1]; // length of pal centered at s[i] is at [2*i]
void find_pals(const string& S) {
  pals[0] = 1, pals[1] = 0;
  for(int d, i = 1; i+2 < 2 * S.size(); i += d) {
    int& p = pals[i];
    int left = (i-p-1)/2, right = (i+p+1)/2;
    while(0 <= left && right < S.size() && S[left] == S[right]) {
      left--;
      right++;
      p += 2;
    }
    for(d = 1; pals[i-d] < p-d; d++)
      pals[i+d] = pals[i-d];
    pals[i+d] = p-d;
  }
  pals[2*(S.size()-1)] = 1;
}
```

```
/***** RANK SEGTREE *****/

const int MAX = (1 << 21);
const int OFFSET = (1 << 20);

struct RankSegTree {
  int seg[2 * MAX];
  RankSegTree() {
    memset(seg, 0, sizeof(seg));
  }
  void insert(int v,
 int i) {
    for(i += MAX; i > 0; i /= 2) seg[i] += v;
  }
  int rank(int k) {
    int p;
    for(p = 1; p < MAX; ) {
      //cout << 2*p << ' ' << seg[2*p] << endl;
      if(seg[2 * p] < k) {
        k -= seg[2 * p];
        p = 2 * p + 1;
      }
      else {
        p = 2 * p;
      }
    }
    return p - MAX;
  }
};
```

```
// STRONGLY CONNECTED COMPONENTS

const int MAX = 100100;
```

```
struct SCC {
  int N, cnt, cmpt;
  int num[MAX], low[MAX], ans[MAX];
  vector<int> G[MAX];
  vector<int> S; // stack
  bool on_stack[MAX];

  void reset() {
    for(int i = 0; i < MAX; i++) G[i].clear();
  }
  void strong(int u) {
    if(num[u] != 0) return;
    num[u] = low[u] = ++cnt;
    S.push_back(u);
    on_stack[u] = true;
    for(int i = 0; i < G[u].size(); i++) {
      int v = G[u][i];
      strong(v);
    }
    for(int i = 0; i < G[u].size(); i++) {
      int v = G[u][i];
      if(on_stack[v]) low[u] = min(low[u], low[v]);
    }
    if(num[u] == low[u]) {
      while(!S.empty()) {
        int x = S.back();
        S.pop_back();
        on_stack[x] = false;
        ans[x] = cmpt;
        if(x == u) break;
      }
      cmpt++;
    }
  }
  void scc() {
    memset(num, 0, sizeof(num));
    memset(low, 0, sizeof(low));
    memset(ans, 0, sizeof(ans));
    memset(on_stack, 0, sizeof(on_stack));
    S.clear();
    cnt = 0;
    cmpt = 0;
    for(int i = 1; i <= N; i++) strong(i);
  }
};
```

```
// KMP

const int MAX = 100005;

int T[MAX];
void build_table(string& W) {
  int pos = 2, cnd = 0;
  T[0] = -1, T[1] = 0;
  while(pos < W.size()) {
    if(W[pos - 1] == W[cnd])
      cnd++, T[pos] = cnd, pos++;
    else if(cnd > 0)
      cnd = T[cnd];
    else
      T[pos] = 0, pos++;
  }
}

bool full[MAX];
void kmp_search(string& S, string& W) {
  memset(full, 0, sizeof(full));
  int m = 0, i = 0;
  while(m + i < S.size()) {
    if(i == W.size() - 1) full[m] = true;
    if(W[i] == S[m + i])
      i++;
    else {
      if(T[i] > -1)
        m += i - T[i], i = T[i];
      else
        i = 0, m++;
    }
  }
}

string S, W;

int main() {
  cin >> S >> W;
  S += "###";
  W += '$';
  build_table(W);
  kmp_search(S, W);
}
```

```
// find negative cycle

int neg_cycle() {
  //for(int i = 0; i < N; i++) { for(int j = 0; j < N; j++) printf("%d/%d ",
cost[i][j], cap[i][j]); printf("\n"); }
  int dist[122];
  int from[122];
  memset(from, -1, sizeof(from));
  for(int i = 0; i < N; i++) dist[i] = inf;
  dist[0] = 0;
  int neg_cycle_at = -1;
  for(int t = 0; t <= N; t++)
    for(int u = 0; u < N; u++)
      for(int v = 0; v < N; v++)
        if(cap[u][v] && dist[u] + cost[u][v] < dist[v]) {
          if(t == N)
            neg_cycle_at = v;
          dist[v] = dist[u] + cost[u][v];
          from[v] = u;
        }
  if(neg_cycle_at == -1) return 0;

  bool vis[122];
  memset(vis, 0, sizeof(vis));
  int neg_cycle_start = -1;
  for(int v = neg_cycle_at; ; v = from[v]) {
    vis[v] = true;
```

```
    int u = from[v];
    //printf("%d -> %d\n", u, v);
    if(vis[u]) {
      neg_cycle_start = u;
      break;
    }
  }

  int v = neg_cycle_start;
  int len = 0;
  do {
    int u = from[v];
    //printf("cycle: %d -> %d\n", u, v);
    cap[u][v]--;
    cap[v][u]++;
    len += cost[u][v];
    v = from[v];
  }
  while(v != neg_cycle_start);
  assert(len < 0);
  //printf("len = %d\n", len);
  return len;
}


// lazy propagation segtree

const int MAX = (1 << 23);
struct SegTree {
  int seg[2 * MAX];
  int delta[2 * MAX];
  void init() {
    memset(seg, 0, sizeof(seg));
    memset(delta, 0, sizeof(delta));
  }
  void propagate_up(int p) {
    seg[p] = max(seg[p * 2], seg[p * 2 + 1]);
  }
  void propagate_down(int p) {
    if(p >= MAX) return;
    seg[p * 2] += delta[p];
    delta[p * 2] += delta[p];
    seg[p * 2 + 1] += delta[p];
    delta[p * 2 + 1] += delta[p];
    delta[p] = 0;
  }
  int lookup2(int l, int r, int p, int a, int b) {
    if(a >= r || b <= l) return 0;
    if(l <= a && b <= r) return seg[p];
    propagate_down(p);
    int m = (a + b) / 2;
    return max(lookup2(l, r, p * 2, a, m), lookup2(l, r, p * 2 + 1, m, b));
  }
  int lookup(int l, int r) {
    return lookup2(l, r + 1, 1, 0, MAX);
  }
  void insert2(int v, int l, int r, int p, int a, int b) {
    if(a >= r || b <= l) return;
    if(l <= a && b <= r) {
      seg[p] += v;
      delta[p] += v;
    }
    else {
      propagate_down(p);
      int m = (a + b) / 2;
      insert2(v, l, r, p * 2, a, m);
      insert2(v, l, r, p * 2 + 1, m, b);
      propagate_up(p);
    }
  }
  void insert(int v, int l, int r) {
    //printf("insert %d in [%d, %d]\n", v, l, r);
    insert2(v, l, r + 1, 1, 0, MAX);
  }
};


typedef long long ll;
const int MAX = 1000100;
const ll base = 2, invbase = 500000004, mod = 1000000007;

int N;
char S[MAX];

struct Hash {
  ll key[256];
  ll power[MAX];
  ll inv[MAX];
  ll hash[MAX];
  void init() {
    for(int i = 0; i < 256; i++)
      key[i] = rand() % mod;
    power[0] = inv[0] = 1;
    for(int i = 1; i < N; i++) {
      power[i] = (power[i - 1] * base) % mod;
      inv[i] = (inv[i - 1] * invbase) % mod;
    }
    hash[0] = (power[0] * key[S[0]]) % mod;
    for(int i = 1; i < N; i++)
      hash[i] = (hash[i - 1] + power[i] * key[S[i]]) % mod;
  }
  ll get(ll start, ll len) {
    return (inv[start] * (hash[start + len - 1] - (start == 0 ? 0LL : hash[start
- 1]) + mod)) % mod;
  }
};


struct edge {
  int v, id;
  edge(int v2, int id2) {
    v = v2, id = id2;
  }
};

const int MAX = 300300;
struct EulerTour {
  vector<edge> G[MAX];
  bool used[MAX];
  vector<int> tour;
  void euler_tour(int v) {
    for(int i = 0; i < G[v].size(); i++) {
      //cout << v << " -> " << G[v][i].v << endl;
      if(used[G[v][i].id]) continue;
```

```
      used[G[v][i].id] = true;
      euler_tour(G[v][i].v);
    }
    tour.push_back(v);
  }
};


// MATRIX POWER

#define SIZE 105

ll mult(ll A[SIZE][SIZE], ll B[SIZE][SIZE], ll C[SIZE][SIZE]) {
  memset(C, 0, sizeof(C));
  for(int i = 0; i < SIZE; i++)
    for(int j = 0; j < SIZE; j++)
      for(int k = 0; k < SIZE; k++)
        C[i][j] = (C[i][j] + A[i][k] * B[k][j]) % mod;
}
void identity(ll I[SIZE][SIZE]) {
  memset(I, 0, sizeof(I));
  for(int i = 0; i < SIZE;i ++) I[i][i] = 1;
}
ll power(ll A[SIZE][SIZE], ll C[SIZE][SIZE], int e) {
  if(e == 0) identity(C);
  else {
    ll temp[SIZE][SIZE];
    memset(temp, 0, sizeof(temp));
    power(A, temp, e / 2);
    if(e % 2) {
      ll temp2[SIZE][SIZE];
      memset(temp2, 0, sizeof(temp2));
      mult(temp, temp, temp2);
      mult(temp2, A, C);
    }
    else mult(temp, temp, C);
  }
}


// SUFFIX ARRAY

int n;
char s[MaxN + 1];

int sa[MaxN];
int rank[MaxN];
int height[MaxN];

inline bool sa_init_equal(int *y, int k, int i, int j)
{
  return y[i] == y[j] && y[(i + k) % n] == y[(j + k) % n];
}

void sa_init()
{
  int *x = rank, *y = height;
  static int w[MaxN];
  fill(w, w + NLetter, 0);
  for (int i = 0; i < n; i++)
    w[(int)s[i]]++;
  for (int i = 1; i < NLetter; i++)
    w[i] += w[i - 1];
  for (int i = n - 1; i >= 0; i--)
    sa[--w[(int)s[i]]] = i;

  int nR = 0;
  for (int i = 0; i < n; i++)
  {
    if (i == 0 || s[sa[i]] != s[sa[i - 1]])
      nR++;
    y[sa[i]] = nR - 1;
  }
  for (int k = 1; k < n && nR < n; k <<= 1, swap(x, y))
  {
    int len = 0;
    for (int i = 0; i < n; i++)
      x[len++] = (sa[i] + n - k) % n;
    fill(w, w + nR, 0);
    for (int i = 0; i < n; i++)
      w[y[x[i]]]++;
    for (int i = 1; i < nR; i++)
      w[i] += w[i - 1];
    for (int i = n - 1; i >= 0; i--)
      sa[--w[y[x[i]]]] = x[i];

    nR = 0;
    for (int i = 0; i < n; i++)
    {
      if (i == 0 || !sa_init_equal(y, k, sa[i], sa[i - 1]))
        nR++;
      x[sa[i]] = nR - 1;
    }
  }
}
```