Contents

Dinic.cc 1/27

```
// 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;
  }
};
```

MinCostMaxFlow.cc 2/27

```
// Implementation of min cost max flow algorithm
using adjacency
// matrix (Edmonds and Karp 1972).  This
implementation keeps track of
// forward and reverse edges separately (so you
can set cap[i][j] !=
// cap[j][i]).  For a regular max flow, set all
edge costs to 0.
//
// Running time, O(|V|^2) cost per augmentation
//     max flow:           O(|V|^3) augmentations
//     min cost max flow:  O(|V|^4 *
MAX_EDGE_COST) augmentations
//
// INPUT:
//     - graph, constructed using AddEdge()
//     - source
//     - sink
//
// OUTPUT:
//     - (maximum flow value, minimum cost value)
//     - To obtain the actual flow, look at
positive values only.

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

using namespace std;

typedef vector<int> VI;
typedef vector<VI> VVI;
typedef long long L;
typedef vector<L> VL;
typedef vector<VL> VVL;
typedef pair<int, int> PII;
typedef vector<PII> VPII;

const L INF = numeric_limits<L>::max() / 4;

struct MinCostMaxFlow {
  int N;
  VVL cap, flow, cost;
  VI found;
  VL dist, pi, width;
  VPII dad;

  MinCostMaxFlow(int N) :
    N(N), cap(N, VL(N)), flow(N, VL(N)), cost(N,
VL(N)),
    found(N), dist(N), pi(N), width(N), dad(N) {}

  void AddEdge(int from, int to, L cap, L cost) {
    this->cap[from][to] = cap;
    this->cost[from][to] = cost;
```

```
  }

  void Relax(int s, int k, L cap, L cost, int dir)
{
    L val = dist[s] + pi[s] - pi[k] + cost;
    if (cap && val < dist[k]) {
      dist[k] = val;
      dad[k] = make_pair(s, dir);
      width[k] = min(cap, width[s]);
    }
  }

  L Dijkstra(int s, int t) {
    fill(found.begin(), found.end(), false);
    fill(dist.begin(), dist.end(), INF);
    fill(width.begin(), width.end(), 0);
    dist[s] = 0;
    width[s] = INF;

    while (s != -1) {
      int best = -1;
      found[s] = true;
      for (int k = 0; k < N; k++) {
        if (found[k]) continue;
        Relax(s, k, cap[s][k] - flow[s][k],
cost[s][k], 1);
        Relax(s, k, flow[k][s], -cost[k][s], -1);
        if (best == -1 || dist[k] < dist[best])
best = k;
      }
      s = best;
    }

    for (int k = 0; k < N; k++)
      pi[k] = min(pi[k] + dist[k], INF);
    return width[t];
  }

  pair<L, L> GetMaxFlow(int s, int t) {
    L totflow = 0, totcost = 0;
    while (L amt = Dijkstra(s, t)) {
      totflow += amt;
      for (int x = t; x != s; x = dad[x].first) {
        if (dad[x].second == 1) {
          flow[dad[x].first][x] += amt;
          totcost += amt * cost[dad[x].first][x];
        } else {
          flow[x][dad[x].first] -= amt;
          totcost -= amt * cost[x][dad[x].first];
        }
      }
    }
    return make_pair(totflow, totcost);
  }
};
```

PushRelabel.cc 3/27

```
// 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;
  }
};
```

MinCostMatching.cc 4/27

```
//////////////////////////////////////////////////
/////////////////////////
// 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 5/27

```
// 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 6/27

```
// 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);
}
```

ConvexHull.cc 7/27

```
// Compute the 2D convex hull of a set of points
using the monotone chain
// algorithm.  Eliminate redundant points from the
hull if REMOVE_REDUNDANT is
// #defined.
//
// Running time: O(n log n)
//
//    INPUT:    a vector of input points, unordered.
```

```
//   OUTPUT:  a vector of points in the convex
hull, counterclockwise, starting
//            with bottommost/leftmost point

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

using namespace std;

#define REMOVE_REDUNDANT

typedef double T;
const T EPS = 1e-7;
struct PT {
  T x, y;
  PT() {}
  PT(T x, T y) : x(x), y(y) {}
  bool operator<(const PT &rhs) const { return
make_pair(y,x) < make_pair(rhs.y,rhs.x); }
  bool operator==(const PT &rhs) const { return
make_pair(y,x) == make_pair(rhs.y,rhs.x); }
};

T cross(PT p, PT q) { return p.x*q.y-p.y*q.x; }
T area2(PT a, PT b, PT c) { return cross(a,b) +
cross(b,c) + cross(c,a); }

#ifdef REMOVE_REDUNDANT
bool between(const PT &a, const PT &b, const PT
&c) {
  return (fabs(area2(a,b,c)) < EPS && (a.x-
b.x)*(c.x-b.x) <= 0 && (a.y-b.y)*(c.y-b.y) <= 0);
}
#endif

void ConvexHull(vector<PT> &pts) {
  sort(pts.begin(), pts.end());
  pts.erase(unique(pts.begin(), pts.end()),
pts.end());
  vector<PT> up, dn;
  for (int i = 0; i < pts.size(); i++) {
    while (up.size() > 1 && area2(up[up.size()-2],
up.back(), pts[i]) >= 0) up.pop_back();
    while (dn.size() > 1 && area2(dn[dn.size()-2],
dn.back(), pts[i]) <= 0) dn.pop_back();
    up.push_back(pts[i]);
    dn.push_back(pts[i]);
  }
  pts = dn;
  for (int i = (int) up.size() - 2; i >= 1; i--)
pts.push_back(up[i]);

#ifdef REMOVE_REDUNDANT
  if (pts.size() <= 2) return;
  dn.clear();
  dn.push_back(pts[0]);
  dn.push_back(pts[1]);
  for (int i = 2; i < pts.size(); i++) {
    if (between(dn[dn.size()-2], dn[dn.size()-1],
pts[i])) dn.pop_back();
    dn.push_back(pts[i]);
  }
  if (dn.size() >= 3 && between(dn.back(), dn[0],
dn[1])) {
    dn[0] = dn.back();
    dn.pop_back();
  }
  pts = dn;
#endif
}
```

Geometry.cc 8/27

```
// 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;
}
```

JavaGeometry.java 9/27

```
// 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)).

    }
}
```

Geom3D.java 10/27

```
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 11/27

```
// 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 12/27

```
// 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 13/27

```
// 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 14/27

```
// 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 nxn 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 15/27

```
#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 16/27

```
// 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;
}
```

FastDijkstra.cc 17/27

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

#include <queue>
#include <stdio.h>

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();
    if (p.second == t) break;
    Q.pop();

    int here = p.second;
    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;
}
```

SCC.cc 18/27

```
#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]);}
}
```

SuffixArray.cc 19/27

```
// 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;
  }
};

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;
}
```

BIT.cc 20/27

```
#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 21/27

```
//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 22/27

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

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

LongestIncreasingSubsequence.cc 23/27

```
// 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 24/27

```
// 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;
}
```

LogLan.java 25/27

```
// Code which demonstrates the use of Java's
regular expression libraries.
// This is a solution for
//
//   Loglan: a logical language
//   http://acm.uva.es/p/v1/134.html
//
// In this problem, we are given a regular
language, whose rules can be
// inferred directly from the code.  For each
sentence in the input, we must
// determine whether the sentence matches the
regular expression or not.  The
// code consists of (1) building the regular
expression (which is fairly
// complex) and (2) using the regex to match
sentences.

import java.util.*;
import java.util.regex.*;

public class LogLan {

    public static String BuildRegex (){
        String space = " +";

        String A = "([aeiou])";
        String C = "([a-z&&[^aeiou]])";
        String MOD = "(g" + A + ")";
        String BA = "(b" + A + ")";
        String DA = "(d" + A + ")";
        String LA = "(l" + A + ")";
        String NAM = "([a-z]*" + C + ")";
        String PREDA = "(" + C + C + A + C + A +
"|" + C + A + C + C + A + ")";
```

```
        String predstring = "(" + PREDA + "(" +
space + PREDA + ")*)";
        String predname = "(" + LA + space +
predstring + "|" + NAM + ")";
        String preds = "(" + predstring + "(" +
space + A + space + predstring + ")*)";
        String predclaim = "(" + predname + space +
BA + space + preds + "|" + DA + space +
            preds + ")";
        String verbpred = "(" + MOD + space +
predstring + ")";
        String statement = "(" + predname + space +
verbpred + space + predname + "|" +
            predname + space + verbpred + ")";
        String sentence = "(" + statement + "|" +
predclaim + ")";

        return "^" + sentence + "$";
    }

    public static void main (String args[]){

        String regex = BuildRegex();
        Pattern pattern = Pattern.compile (regex);

        Scanner s = new Scanner(System.in);
        while (true) {

            // In this problem, each sentence
consists of multiple lines, where the last
            // line is terminated by a period.  The
code below reads lines until
            // encountering a line whose final
character is a '.'.  Note the use of
            //
            //    s.length() to get length of
string
            //    s.charAt() to extract characters
from a Java string
            //    s.trim() to remove whitespace
from the beginning and end of Java string
            //
            // Other useful String manipulation
methods include
            //
            //    s.compareTo(t) < 0 if s < t,
lexicographically
            //    s.indexOf("apple") returns index
of first occurrence of "apple" in s
            //    s.lastIndexOf("apple") returns
index of last occurrence of "apple" in s
            //    s.replace(c,d) replaces
occurrences of character c with d
            //    s.startsWith("apple) returns
(s.indexOf("apple") == 0)
            //    s.toLowerCase() /
s.toUpperCase() returns a new lower/uppercased
string
            //
            //    Integer.parseInt(s) converts s
to an integer (32-bit)
            //    Long.parseLong(s) converts s to
a long (64-bit)
            //    Double.parseDouble(s) converts s
to a double

            String sentence = "";
            while (true){
                sentence = (sentence + " " +
s.nextLine()).trim();
                if (sentence.equals("#")) return;
                if
(sentence.charAt(sentence.length()-1) == '.')
break;
            }

            // now, we remove the period, and
match the regular expression

            String removed_period =
sentence.substring(0, sentence.length()-1).trim();
```

```
            if (pattern.matcher
(removed_period).find()){
                System.out.println ("Good");
            } else {
                System.out.println ("Bad!");
            }
        }
    }
}
```

Primes.cc 26/27

```
// O(sqrt(x)) Exhaustive Primality Test
#include <cmath>
#define EPS 1e-7
typedef long long LL;
bool IsPrimeSlow (LL x)
{
  if(x<=1) return false;
  if(x<=3) return true;
  if (!(x%2) || !(x%3)) return false;
  LL s=(LL)(sqrt((double)(x))+EPS);
  for(LL i=5;i<=s;i+=6)
  {
    if (!(x%i) || !(x%(i+2))) return false;
  }
  return true;
}
// Primes less than 1000:
//      2     3     5     7    11    13    17
19    23    29    31    37
//     41    43    47    53    59    61    67
71    73    79    83    89
//     97   101   103   107   109   113   127
131   137   139   149   151
//    157   163   167   173   179   181   191
193   197   199   211   223
//    227   229   233   239   241   251   257
263   269   271   277   281
//    283   293   307   311   313   317   331
337   347   349   353   359
//    367   373   379   383   389   397   401
409   419   421   431   433
//    439   443   449   457   461   463   467
479   487   491   499   503
//    509   521   523   541   547   557   563
569   571   577   587   593
//    599   601   607   613   617   619   631
641   643   647   653   659
//    661   673   677   683   691   701   709
719   727   733   739   743
//    751   757   761   769   773   787   797
809   811   821   823   827
//    829   839   853   857   859   863   877
881   883   887   907   911
//    919   929   937   941   947   953   967
971   977   983   991   997

// 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.
```

KMP.cpp 27/27

```
/*
Searches for the string w in the string s (of length k). Returns the
0-based index of the first match (k if no match is found). Algorithm
runs in O(k) time.
*/

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

using namespace std;

typedef vector<int> VI;

void buildTable(string& w, VI& t)
{
  t = VI(w.length());
  int i = 2, j = 0;
  t[0] = -1; t[1] = 0;

  while(i < w.length())
  {
    if(w[i-1] == w[j]) { t[i] = j+1; i++; j++; }
    else if(j > 0) j = t[j];
    else { t[i] = 0; i++; }
  }
}

int KMP(string& s, string& w)
{
  int m = 0, i = 0;
  VI t;

  buildTable(w, t);
  while(m+i < s.length())
  {
    if(w[i] == s[m+i])
    {
      i++;
      if(i == w.length()) return m;
    }
    else
    {
      m += i-t[i];
      if(i > 0) i = t[i];
    }
  }
  return s.length();
}

int main()
{
  string a = (string) "The example above illustrates the general technique for assembling "+
    "the table with a minimum of fuss. The principle is that of the overall search: "+
    "most of the work was already done in getting to the current position, so very "+
```

```
    "little needs to be done in leaving it. The 
only minor complication is that the "+
    "logic which is correct late in the string 
erroneously gives non-proper "+
    "substrings at the beginning. This 
necessitates some initialization code.";

  string b = "table";

  int p = KMP(a, b);
  cout << p << ": " << a.substr(p, b.length()) << 
" " << b << endl;
}
```