University of Calgary

T E A M   N O T E B O O K

ACM International Collegiate Programming Contest
2004 World Finals

Sonny Chan ◆ Alex Fink ◆ Kelly Poon

Our Magic Incantations List

When Choosing a Problem

- Find out which balloons are the popular ones!
- Pick one with a nice, ***clean solution*** that you are totally convinced ***will work*** to do first.

Before Designing Your Solution

- Highlight the important information on the problem statement – input bounds, special rules, formatting, etc.
- Look for code in this notebook that you can use!
- Convince yourself that your algorithm will run with time to spare on the biggest input.
- Create several ***test cases*** that you will use, especially for ***special or boundary cases***.

Prior to Submitting

- Check ***maximum*** input, ***zero*** input, and other ***degenerate*** test cases.
- Cross check with team mates' supplementary test cases.
- Read the problem ***output specification*** one more time – your program's output behaviour is fresh in your mind.
- Does your program work with ***negative*** numbers?
- Make sure that your program is reading from an appropriate ***input file***.
- Check all ***variable initialisation***, ***array bounds***, and ***loop variables*** (`i` vs `j`, `m` vs `n`, etc.).
- Finally, run a `diff` on the provided sample output and your program's output.
- And don't forget to submit your solution under the ***correct problem number!***

After Submitting

- Immediately ***print a copy*** of your source.
- Staple the solution to the problem statement and keep them safe. Do not lose them!

If It Doesn't Work…

- Remember that a ***run-time error*** can be ***division by zero***.
- If the solution is not complex, allow a team mate to start the problem afresh.
- Don't waste a lot of time – it's not shameful to ***simply give up!!!***

Remember to ***HAVE FUN!!!***

Table of Contents

Graph Algorithms Summary

Undirected Graphs

Cycle Detection	Depth-First Search	Linear
Simple Connectivity	Depth-First Search	Linear
Two-Colouring, Odd Cycle	DFS Parity	Linear
Bridges, Biconnectivity	DFS Back Edges	Linear

Directed Graphs

Cycle Detection	DFS Back Edges	Linear
Transitive Closure	Warshall's Algorithm	V^3
Topological Sort	Reverse Postorder DFS	Linear
DAG Transitive Closure	DFS with DP, 'or' the rows	Linear
Strong Connectivity	Kosaraju's Algorithm (reverse, postorder, then DFS)	Linear
DAG Longest Path	Topological Sort with DP	Linear

Minimum Spanning Trees

MST on Dense Graph	Prim's Algorithm	V^2
MST on Sparse Graph	Kruskal's Algorithm	E lg E

Shortest Paths

Non-negative Weights	Dijkstra's Algorithm	V^2
Acyclic Shortest/Longest	Topological Sort with Relax	Linear
All-Pairs Shortest Paths	Floyd's Algorithm	V^3
Negative Cycle Detection	Bellman-Ford	VE

Network Flow

Bipartite Match	Ford-Fulkerson Algorithm	VE
Edge Connectivity	Minimum Flow from vertex of smallest degree to all others	E^2
Assignment	Mincost Maxflow	?
Mail Carrier	Mincost Maxflow	?

Hard Problems

Hamiltonian Path/Tour	Exhaustive Search	V!
General Shortest Path	Exhaustive Search	V!
Even Cycle Detection	No Nice Algorithm Known	?
Graph Isomorphism	?	?

```
//----------------------------------------------------------------------------
// GRAPH ALGORITHMS ON ADJACENCY MATRIX GRAPH
//
// This file contains a graph class that uses an adjacency matrix
// representation, and implementations of the following algorithms:
//    - Breadth First Search for counting connected components
//    - Prim's algorithm for minimum spanning trees
//    - Dijkstra's algorithm for shortest paths
//    - Floyd's algorithm for all pairs shortest paths
//
// Author:  Sonny Chan
// Date:    November 10, 2003
//----------------------------------------------------------------------------

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

using namespace std;

//----------------------------------------------------------------------------
// graph class with adjacency matrix representation
//    - template parameter T is edge type (eg. int, double, or edge struct)

template <class T>
struct graph {

   int vertices;                 // number of vertices in the graph

   vector< vector<T> > d;        // distance matrix
   vector< vector<bool> > c;     // connectivity matrix (true if edge exists)

   vector<bool> present;         // indicates which vertices actually exist

   vector< vector<T> > spd;      // shortest paths adjacency matrix
   vector< vector<int> > spp;    // shortest paths tree for all vertices
   vector< vector<bool> > tcc;   // transitive closure connectivity matrix

   // constructor takes size as (maximum) number of vertices
   graph(int size = 1, bool allv = true) : vertices(size)
   {
      // initialise vertex existence vector
      present = vector<bool>(size, allv);

      // create connectivity and distance matrices
      d = vector< vector<T> >(size, vector<T>(size));
      c = vector< vector<bool> >(size, vector<bool>(size, false));

      // create shortest path and transitive closure matrices
      spd = d;
      tcc = c;
      spp = vector< vector<int> >(size, vector<int>(size, -1));
   }

   // adds a directed edge between vertices a and b with weight w
   void add(int a, int b, const T &w)
   {
      present[a] = present[b] = true;
      c[a][b] = true;
      d[a][b] = w;
   }

// adds an undirected edge between vertices a and b with weight w
void addu(int a, int b, const T &w)
{
   add(a, b, w);
   add(b, a, w);
}

// computes and returns the degree of vertex v
//    (V complexity)
int degree(int v)
{
   return count(c[v].begin(), c[v].end(), true);
}

// counts the number of connected components in the graph
//    (V^2 complexity, worst case)
int components()
{
   // component numbering of the vertices
   vector<int> cn(vertices, -1);
   int ct = 0;

   for (int v = 0; v < vertices; ++v) {

      // check if vertex v is not present or has been processed already
      if (!present[v] || cn[v] != -1)
         continue;

      // do a breadth first search on this component
      queue<int> q;
      q.push(v);
      cn[v] = ct;
      while (!q.empty()) {
         int x = q.front(); q.pop();
         for (int w = 0; w < vertices; ++w)
            if (cn[w] == -1 && c[x][w]) {
               q.push(w);
               cn[w] = ct;
            }
      }

      // increment the component number
      ++ct;
   }

   // return number of components found
   return ct;
}

// computes the minimum spanning tree using Prim's algorithm
// assumes the graph is connected, and will return V-1 edges of the tree
//    (V^2 complexity)
vector< pair<int, int> > prim()
{
   vector< pair<int, int> > edges;

   int s = -1;
   for (int i = 0; i < vertices && s == -1; ++i)
      if (present[i]) s = i;
   if (s == -1) return edges;

   vector<bool> seen(vertices, false); // seen[v] true if weight[v] is valid
   vector<bool> used(vertices, false); // used[v] true if v added to mst
```

```
      vector<int> mst(vertices, -1);      // the minimum spanning tree
      vector<T> weight(vertices, T());    // fringe vector

      // set algorithm to start on vertex s
      seen[s] = true;

      // loop on all connected vertices
      for (;;) {

         // find closest connected vertex to add to mst
         int v = -1;
         T lo;
         for (int i = 0; i < vertices; ++i) {
            if (seen[i] && !used[i]) {
               if (v == -1 || weight[i] < lo) {
                  v = i;
                  lo = weight[i];
               }
            }
         }
         if (v == -1) break;

         used[v] = true;

         // update the fringe vertices
         for (int i = 0; i < vertices; ++i) {
            if (c[v][i] && !used[i]) {
               T cost = d[v][i];
               if (!seen[i] || cost < weight[i]) {
                  seen[i] = true;
                  weight[i] = cost;
                  mst[i] = v;
               }
            }
         }
      }

      // extract the edges from the mst
      for (int i = 0; i < vertices; ++i)
         if (mst[i] != -1)
            edges.push_back(pair<int,int>(i, mst[i]));

      return edges;
   }

   // computes the shortest paths vector from source s using Dijkstra's
   // algorithm, then returns the length of the shortest path from s to t
   //    (V^2 complexity)
   T dijkstra(int s, int t)
   {
      vector<bool> seen(vertices, false); // seen[v] true if weight[v] is valid
      vector<bool> used(vertices, false); // used[v] true if relaxed on v
      vector<int> path(vertices, -1);     // the shortest path tree for s
      vector<T> weight(vertices, T());    // shortest distance vector for s

      // set algorithm to start on vertex s
      seen[s] = true;
      path[s] = s;

      // loop on all connected vertices for relaxing
      for (;;) {

         // find closest connected vertex to relax on
         int v = -1;
         T lo;
         for (int i = 0; i < vertices; ++i) {
            if (seen[i] && !used[i]) {
               if (v == -1 || weight[i] < lo) {
                  v = i;
                  lo = weight[i];
               }
            }
         }
         if (v == -1) break;

         used[v] = true;

         // relax on vertex v
         for (int i = 0; i < vertices; ++i) {
            if (c[v][i]) {
               T cost = weight[v] + d[v][i];
               if (!seen[i] || cost < weight[i]) {
                  seen[i] = true;
                  weight[i] = cost;
                  path[i] = v;
               }
            }
         }
      }

      // update the transitive closure and shortest pathes matrices for s
      tcc[s] = seen;
      spd[s] = weight;
      spp[s] = path;

      // return the shortest distance from s to t
      return weight[t];
   }

   // traverses the shortest paths tree for s to find the shortest path to t
   vector<int> shortestpath(int s, int t)
   {
      vector<int> p;
      if (spp[s][t] != -1) {
         for (int v = t; v != s; v = spp[s][v])
            p.push_back(v);
         p.push_back(s);
      }
      reverse(p.begin(), p.end());
      return p;
   }

   // computes all pairs shortest paths using Floyd's algorithm
   //    (V^3 complexity)
   void floyd() {

      // set the transitive closure and shortest path distance matrices
      tcc = c;
      spd = d;

      for (int i = 0; i < vertices; ++i)
         for (int s = 0; s < vertices; ++s)
            if (tcc[s][i])
               for (int t = 0; t < vertices; ++t)
                  if (tcc[i][t]) {

                     // if there's not an existing path from s to t,
                     // create a new one
                  if (!tcc[s][t]) {
                     tcc[s][t] = true;
                     spd[s][t] = spd[s][i] + spd[i][t];
                  }
                  // otherwise update the cost
                  else
                     spd[s][t] = min(spd[s][t], spd[s][i] + spd[i][t]);

               }
   }
};

//----------------------------------------------------------------------------
// small test bed that does nothing but generate a random graph

int main()
{
   const int v = 10;

   graph<int> g(v, false);

   for (int i = 0; i < 12; ++i)
      g.addu(rand()%v, rand()%v, rand()%10);

   cout << "Number of connected components:" << endl;
   cout << g.components() << endl << endl;

   // test out Dijkstra's shortest path alrogithm
   int s = rand()%v;
   int t = rand()%v;
   cout << "Shortest path between " << s << " and " << t << ':' << endl;
   int length = g.dijkstra(s, t);
   vector<int> path = g.shortestpath(s, t);
   copy(path.begin(), path.end(), ostream_iterator<int>(cout, " "));
   if (g.tcc[s][t])
      cout << endl << "Length: " << length << endl;
   else
      cout << endl << "Unreachable!" << endl;
   cout << endl;

   // test out Floyd's algorithm for transitive closure
   cout << "The graph's transitive closure:" << endl;
   g.floyd();
   for (int i = 0; i < g.vertices; ++i) {
      for (int j = 0; j < g.vertices; ++j)
         if (g.tcc[i][j]) cout << setw(4) << g.spd[i][j];
         else cout << setw(4) << '-';
      cout << endl;
   }
   cout << endl;

   return 0;
}
//----------------------------------------------------------------------------
```

```
//----------------------------------------------------------------------------
// UNION-FIND ALGORITHM FOR CONNECTIVITY
//
// This file mainly features the union-find algorithm for testing for
// connectivity/reachability and keeping track of connected components.
// A graph structure with adjacency list representation is also present to
// demonstrate the use of the union-find algorithm in finding a minimum
// spanning tree using Kruskal's algorithm.
//
// Author:  Sonny Chan
// Date:    March 12, 2004
//----------------------------------------------------------------------------

#include <iostream>
#include <vector>
#include <map>
#include <algorithm>
#include <iterator>
#include <cstdlib>
#include <ctime>

using namespace std;

//----------------------------------------------------------------------------
// union find data structure for keeping track of connectivity
struct ufind {

   int n;                  // number of vertices or elements
   int components;         // number of connected components
   vector<int> c;          // array representation of parent-link tree

   // constructor takes size of structure as argument
   ufind(int size) : n(size), components(size)
   {
      c = vector<int>(n);
      for (int i = 0; i < n; ++i) c[i] = i;
   }

   // finds the root of element a
   int root(int a)
   {
      int r;
      for (r = a; r != c[r]; r = c[r]) c[r] = c[c[r]];
      return r;
   }

   // joins elements a and b
   int join(int a, int b)
   {
      int p = root(a);
      int q = root(b);

      if (p == q) return p;

      --components;
      int r = min(p, q);
      int s = max(p, q);
      c[s] = r;
      return r;
   }

   // answers a query as to whether elements a and b are connected
   bool connected(int a, int b)
   {
      return root(a) == root(b);
   }

};

//----------------------------------------------------------------------------
// adjacency lists undirected graph structure for minimum spanning tree

struct graph {

   // edge structure with
   struct edge {
      int s, t, cost;
      edge(int a = -1, int b = -1, int c = 0) : s(a), t(b), cost(c) {}
   };

   int n;                        // number of vertices
   vector< map<int, int> > g;    // sparse upper triangle of connectivity

   // constructor takes the number of vertices in the graph, numbered 0 to n
   graph(int vertices = 1) : n(vertices)
   {
      g = vector< map<int, int> >(n);
   }

   // adds an undirected edge to the graph
   void add(int a, int b, int c = 1)
   {
      g[min(a, b)][max(a, b)] = c;
   }

   // computes a minimum spanning tree of the graph using Kruskal's algorithm
   //    (E log E complexity)
   vector<edge> kruskal()
   {
      vector<edge> e;
      for (int i = 0; i < n; ++i)
         for (map<int, int>::iterator jt = g[i].begin(); jt != g[i].end(); ++jt)
            e.push_back(edge(i, jt->first, jt->second));

      sort(e.begin(), e.end());

      ufind uf(n);
      vector<edge> mst;
      for (vector<edge>::iterator it = e.begin(); it != e.end(); ++it) {
         if (!uf.connected(it->s, it->t)) {
            mst.push_back(*it);
            uf.join(it->s, it->t);
            if (uf.components == 1) break;
         }
      }

      return mst;
   }
};

// comparison operator to sort edges according to cost
bool operator<(const graph::edge &a, const graph::edge &b)
{
   return a.cost < b.cost;
}

// insertion operator to output edges
ostream &operator<<(ostream &stream, const graph::edge &e)
{
   stream << '(' << e.s << '-' << e.t << ", " << e.cost << ')';
   return stream;
}

//----------------------------------------------------------------------------
// small test bed that uses union-find and the MST graph

int main()
{
   // try it out on Sedgewick's MST example graph
   cout << "Finding minimum spanning tree for Sedgewick's graph..." << endl;
   graph sedgewick(8);
   sedgewick.add(0, 1, 32);
   sedgewick.add(0, 2, 29);
   sedgewick.add(0, 5, 60);
   sedgewick.add(0, 6, 51);
   sedgewick.add(0, 7, 31);
   sedgewick.add(1, 7, 21);
   sedgewick.add(3, 4, 34);
   sedgewick.add(3, 5, 18);
   sedgewick.add(4, 5, 40);
   sedgewick.add(4, 6, 51);
   sedgewick.add(4, 7, 46);
   sedgewick.add(6, 7, 25);
   cout << "\tMinimum spanning tree: ";
   vector<graph::edge> v = sedgewick.kruskal();
   copy(v.begin(), v.end(), ostream_iterator<graph::edge>(cout, " "));
   cout << endl << endl;

   // try it out on a random graph of size 20
   cout << "Finding MST for random graph with 20 vertices..." << endl;
   srand(time(0));
   graph gr20(20);
   ufind uf20(20);
   for (int i = 0; i < 20; ++i) {
      int n = rand()%5;
      for (int j = 0; j < n; ++j) {
         int v = rand()%20;
         gr20.add(i, v, rand()%100);
         uf20.join(i, v);
      }
   }
   cout << "\tGraph has " << uf20.components << " connected components." <<
endl;
   cout << "\tMinimum spanning tree: ";
   vector<graph::edge> w = gr20.kruskal();
   copy(w.begin(), w.end(), ostream_iterator<graph::edge>(cout, " "));
   cout << endl << endl;

   return 0;
}
//----------------------------------------------------------------------------
```

```cpp
//----------------------------------------------------------------------------
// FORD-FULKERSON ALGORITHM FOR MAXIMUM FLOW
//
// This file contains a network structure which implements the Ford-Fulkerson
// method of augmenting paths to find a maxflow.  The augmenting path search
// method is simply a depth-first search in this implementation -- a better
// search priority may be needed if you're doing a heavy flow application.
//
// Author:  Sonny Chan
// Date:    March 19, 2004
//----------------------------------------------------------------------------

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

using namespace std;

//----------------------------------------------------------------------------
// graph structure to implement Ford-Fulkerson for maximum flow

struct graph {

   int n;                        // number of vertices in the graph
   int hicap;                    // highest capacity edge
   vector< vector<int> > g;      // capacity matrix
   vector< bool > seen;          // for marking verices in search

   // constructs a graph of size vertices
   graph(int size = 1) : n(size), hicap(0)
   {
      g = vector< vector<int> >(n, vector<int>(n, 0));
      seen = vector<bool>(n, false);
   }

   // adds a directed edge to the graph
   void add(int s, int t, int c = 1)
   {
      g[s][t] = c;
      hicap = max(hicap, c);
   }

   // use a depth-first search to find an augmenting path from s to sink,
   // and augment along the path on the backward recursion
   int augment(int s, int sink, int cap)
   {
      if (s == sink) return cap;

      seen[s] = true;
      vector<int> &adj = g[s];

      for (int t = 0; t < n; ++t) {
         if (!seen[t] && adj[t] > 0) {
            if (int c = augment(t, sink, min(cap, adj[t]))) {
               g[s][t] -= c;
               g[t][s] += c;
               return c;
            }
         }
      }

      return 0;
   }

   // finds a maximum flow using the Ford-Fulkerson algorithm
   int maxflow(int source, int sink)
   {
      int total = 0;
      fill(seen.begin(), seen.end(), false);
      while (int flow = augment(source, sink, hicap)) {
         total += flow;
         fill(seen.begin(), seen.end(), false);
      }

      // At this point, the maxflow is found, and g is the residual network.
      // To recover the flow, take the original network and subtract g.

      return total;
   }

};

//----------------------------------------------------------------------------
// small test bed to test out the implementation

int main()
{
   // test it out on the small sedgewick graph
   graph sedgewick(6);
   sedgewick.add(0, 1, 2);
   sedgewick.add(0, 2, 3);
   sedgewick.add(1, 3, 3);
   sedgewick.add(1, 4, 1);
   sedgewick.add(2, 3, 1);
   sedgewick.add(2, 4, 1);
   sedgewick.add(3, 5, 2);
   sedgewick.add(4, 5, 3);
   cout << "Maxflow on Sedgewick graph: " << sedgewick.maxflow(0, 5) << endl;

   return 0;
}
//----------------------------------------------------------------------------
```

```cpp
//---------------------------------------------------------------------------
// NETWORK SIMPLEX ALGORITHM FOR MINCOST MAXFLOW
//
// This file contains a network structure which implements a version of the
// network simplex algorithm for finding a mincost maxflow.  Beware that
// this code is not fully tested nor optimised to my liking!
//
// Author:  Sonny Chan
// Date:    March 12, 2004
//---------------------------------------------------------------------------

#include <iostream>
#include <iomanip>
#include <vector>
#include <list>
#include <algorithm>

using namespace std;

// sentries for the maximum cost and flow for the whole network
const double maxcost = 1e15;
const double maxcap = 1e15;

//---------------------------------------------------------------------------
// edge structure to represent a flow edge in the network

struct edge {
   int v, w;
   double cost;
   double capacity;
   double flow;

   edge(int a = 0, int b = 0, double c = 0.0, double k = 0.0)
      : v(a), w(b), cost(c), capacity(k), flow(0.0) {}

   int other(int a)
      { return v == a ? w : v; }

   double costto(int a)
      { return v == a ? -cost : cost; }

   double capacityto(int a)
      { return v == a ? flow : capacity - flow; }

   void addflowto(int a, double f)
      { flow += (v == a ? -f : f); }
};

ostream &operator<<(ostream &stream, const edge &e)
{
   stream << e.v << '-' << e.w;
   return stream;
}

//---------------------------------------------------------------------------
// network structure to implement network simplex for mincost maxflow

struct network {

   typedef list<edge>::iterator edgep;
   list<edge> elist;             // list of all edges in the network

   vector< vector<edgep> > g;    // network connectivity as adjacency list
   int vertices;                 // number for vertices in the network
```

```cpp
   int source, sink;             // the source and sink vertices

   vector<edgep> stree;          // spanning tree for network simplex
   vector<int> mark;             // marking to help traverse spanning tree
   int valid;                    // value of next valid mark

   vector<double> phi;           // vertex potentials

   bool backwards;               // tracks whether we're pushing flow backward
   int augmentations;            // counts the total number of augmentations

   // constructor takes size of network as argument
   network(int size = 1) : vertices(size), valid(0)
   {
      g = vector< vector<edgep> >(size);

      stree = vector<edgep>(size);
      mark = vector<int>(size, 0);

      phi = vector<double>(size, 0.0);
   }

   // adds a directed edge from v to w with cost c and capacity k
   void add(int v, int w, double c, double k)
   {
      elist.push_back(edge(v, w, c, k));
      g[v].push_back(--elist.end());
      g[w].push_back(--elist.end());
   }

   // adds an undirected edge between v and w with cost c and capacity k
   void addu(int v, int w, double c, double k)
   {
      add(v, w, c, k);
      add(w, v, c, k);
   }

   int st(int v)
   {
      const edgep &ep = stree[v];
      return v == ep->v ? ep->w : ep->v;
   }

   // recursively builds a spanning tree
   void buildst(int v)
   {
      mark[v] = valid;
      for (vector<edgep>::iterator it = g[v].begin(); it != g[v].end(); ++it) {
         int w = (*it)->w;
         if (mark[w] != valid) {
            stree[w] = *it;
            buildst(w);
         }
      }
   }

   // recursively calculates vertex potentials
   double potential(int v)
   {
      if (mark[v] == valid) return phi[v];
      phi[v] = potential(st(v)) - stree[v]->costto(v);
      return phi[v];
   }
```

```
// calculates potentials for the vertices
void calculatep()
{
   ++valid;
   mark[sink] = valid;
   for (int v = 0; v < vertices; ++v)
      if (mark[v] != valid)
         potential(v);
}

// computes the lowest common ancestor of two vertices v and w
int lca(int v, int w)
{
   ++valid;
   mark[v] = mark[w] = valid;
   while (v != w) {
      if (v != sink) v = st(v);
      if (v != sink && mark[v] == valid) return v;
      mark[v] = valid;
      if (w != sink) w = st(w);
      if (w != sink && mark[w] == valid) return w;
      mark[w] = valid;
   }
   return v;
}

// augments along a negative cycle by adding edge x to the spanning tree
// returns an empty/full edge to remove from the tree
edgep augment(const edgep &x)
{
   int v = x->v, w = x->w;
   if (backwards) swap(v, w);

   int r = lca(v, w);
   double d = x->capacityto(w);

   for (int u = w; u != r; u = st(u))
      d = min(d, stree[u]->capacityto(st(u)));
   for (int u = v; u != r; u = st(u))
      d = min(d, stree[u]->capacityto(u));

   x->addflowto(w, d);
   edgep e = x;

   for (int u = w; u != r; u = st(u)) {
      stree[u]->addflowto(st(u), d);
      if (stree[u]->capacityto(st(u)) == 0.0)
         e = stree[u];
   }
   for (int u = v; u != r; u = st(u)) {
      stree[u]->addflowto(u, d);
      if (stree[u]->capacityto(u) == 0.0)
         e = stree[u];
   }

   return e;
}

// tests if vertex b is on the path from a to c in the spanning tree
bool onpath(int a, int b, int c)
{
   for (int i = a; i != c; i = st(i))
      if (i == b) return true;
   return false;
}

// reverses all the links in the spanning tree from vertex u to x
void reverse(int u, int x)
{
   if (u == x) return;
   edgep e = stree[u];
   int j;
   for (int i = st(u); ; i = j) {
      edgep y = stree[i];
      j = st(i);
      stree[i] = e;
      e = y;
      if (i == x) break;
   }
}

// updates the spanning tree by removing edge w and adding edge y
void update(const edgep &w, const edgep &y)
{
   if (w == y) return;

   int u = y->w, v = y->v, x = w->w;
   if (stree[x] != w) x = w->v;

   int r = lca(u, v);

   if (onpath(u, x, r)) {
      reverse(u, x);
      stree[u] = y;
      return;
   }
   if (onpath(v, x, r)) {
      reverse(v, x);
      stree[v] = y;
      return;
   }
}

// calculates the reduced cost of an edge e from vertex v
double reduced(const edgep &e, int v)
{
   double r = e->cost + phi[e->w] - phi[e->v];
   return (e->v == v ? r : -r);
}

// finds the best eligible edge for augmenting
edgep besteligible()
{
   edgep x;
   double small = maxcost;
   for (int v = 0; v < vertices; ++v) {
      for (vector<edgep>::iterator it = g[v].begin();
         it != g[v].end(); ++it)
      {
         const edgep &e = *it;
         if (e->capacityto(e->other(v)) > 0.0)
            if (e->capacityto(v) == 0.0)
               if (reduced(e, v) < small) {
                  x = e;
                  small = reduced(e, v);
                  backwards = e->v != v;
               }
      }
      }
      return x;
   }

   // calculates the total cost of the flow
   double totalcost()
   {
      double sum = 0.0;
      for (list<edge>::iterator it = elist.begin(); it != elist.end(); ++it)
         if (it->flow > 0.0) {
            sum += it->cost * it->flow;
         }
      return sum;
   }

   // calculates a mincost maxflow from source s to sink t
   void mincost_maxflow(int s, int t)
   {
      source = s;
      sink = t;

      // add a dummy edge from source to sink
      add(s, t, maxcost, maxcap);
      edgep dummy = --elist.end();
      dummy->addflowto(t, maxcap);

      // build initial spanning tree using recursive DFS
      elist.push_back(edge(t, t));

      ++valid;
      stree[t] = --elist.end();
      mark[t] = valid;
      stree[s] = dummy;
      buildst(s);

      augmentations = 0;
      for ( ; ; ++valid) {
         // calculate vertex potentials
         calculatep();

         // find best eligible edge
         edgep e = besteligible();

         // check for no more eligible edges
         double rcost = reduced(e, (backwards ? e->w : e->v));
         if (rcost == 0.0) break;

         // augment on e
         ++augmentations;
         edgep r = augment(e);

         update(r, e);
      }

      // remove the dummy edges
      elist.pop_back();
      elist.pop_back();

   }

   void printst()
   {
      cout << "Spanning tree:" << endl;
      for (int i = 0; i < vertices; ++i) {
         cout << i << ": " << i;
         for (int j = st(i); ; j = st(j)) {
            cout << '-' << j;
            if (j == st(j)) break;
         }
         cout << endl;
      }
   }

   void printflow()
   {
      cout << "Network flow is:" << endl;
      for (list<edge>::iterator it = elist.begin(); it != elist.end(); ++it)
         if (it->flow > 0.0) {
            cout << '\t' << *it << ' ' << it->cost << ' ' << it->flow
                 << ' ' << it->cost * it->flow << endl;
         }
   }
};

//---------------------------------------------------------------------------
// a somewhat inadequate test bed to try to test this crazy algorithm

int main()
{
   // try it on Sedgewick's sample weighted network
   cout << "Finding mincost maxflow on Sedgewick's sample network..." << endl;
   network rs(6);
   rs.add(0, 1, 3, 3);
   rs.add(0, 2, 1, 3);
   rs.add(1, 3, 1, 2);
   rs.add(1, 4, 1, 2);
   rs.add(2, 3, 4, 1);
   rs.add(2, 4, 2, 2);
   rs.add(3, 5, 2, 2);
   rs.add(4, 5, 1, 2);
   rs.mincost_maxflow(0, 5);
   rs.printflow();
   cout << "Total cost: " << rs.totalcost() << endl;
   cout << "Total augmentations: " << rs.augmentations << endl;
   cout << endl;

   // try it on a weighted bipartite match problem
   cout << "Finding best weighted bipartite match for 7-7 matching..." << endl;
   int iq[14] = {  70, 80, 90,100,105,110,115,110, 85,115,105, 80, 75,120 };
   int ht[14] = { 145,155,165,175,156,158,160,175,170,170,149,155,179,168 };
   network match(16);
   for (int i = 0; i < 7; ++i) {
      match.add(14, i, 0, 1);
      match.add(i+7, 15, 0, 1);
      for (int j = 7; j < 14; ++j)
         match.add(i, j, abs(iq[i]-iq[j]) + abs(ht[i]-ht[j]), 1);
   }
   match.mincost_maxflow(14, 15);
   match.printflow();
   cout << "Total cost: " << match.totalcost() << endl;
   cout << "Total augmentations: " << match.augmentations << endl;
   cout << endl;

   return 0;
}
//---------------------------------------------------------------------------
```

```
//---------------------------------------------------------------------------
// MAXIMUM CLIQUE AND GRAPH COLOURING ALGORITHMS
//
// This file contains optimised implementations of maximum clique and graph
// colouring algorithms for a graph structure with adjacency matrix
// representation.
//
// Author:  Sonny Chan
// Date:    March 12, 2004
//---------------------------------------------------------------------------

#include <iostream>
#include <vector>
#include <set>
#include <map>
#include <algorithm>
#include <iterator>

using namespace std;

//---------------------------------------------------------------------------
// graph class with adjacency matrix for maximum clique and graph colouring

struct graph {

   // vertex structure with vertex number and degree
   struct vertex {
      int v, d;
      vertex(int vn = 0, int deg = 0) : v(vn), d(deg) {}
   };

   int n;                        // number of vertices
   vector< vector<bool> > c;     // connectivity matrix
   vector< set<int> > g;         // sparse upper triangle of above

   vector< vertex > vinfo;       // maps a new vertex number to original
   map< int, int > vmap;         // maps original vertex number to sorted

   vector< int > mark;           // marking for each vertex for search
   vector< int > cv;             // contains the vertices for our clique
   int bestk;                    // max clique or best colouring found so far

   vector< vector<int> > forbidden; // marks forbidden colours for each vertex
   vector< int > colouring;         // contains a colouring of the graph

   int evaluations;                 // number of function evaluations

   // constructor takes the number of vertices in the graph, numbered 0 to n
   graph(int vertices = 1) : n(vertices)
   {
      c = vector< vector<bool> >(n, vector<bool>(n, false));
      g = vector< set<int> >(n);
   }

   // adds an undirected edge to the graph
   void add(int a, int b)
   {
      c[a][b] = true;
      c[b][a] = true;
   }

   // sorts the vertices in descending order of degree
   void sortv()
   {
      vinfo.clear();
      for (int v = 0; v < n; ++v) {
         int degree = count(c[v].begin(), c[v].end(), true);
         vinfo.push_back(vertex(v, degree));
      }
      sort(vinfo.begin(), vinfo.end());

      vmap.clear();
      for (int i = 0; i < n; ++i) {
         vmap[vinfo[i].v] = i;
         g[i].clear();
      }

      // create the upper triangle of connectivity using adjacency lists
      for (int i = 0; i < n; ++i) {
         for (int j = i+1; j < n; ++j)
            if (c[i][j]) {
               int a = vmap[i];
               int b = vmap[j];
               g[min(a, b)].insert(max(a, b));
            }
      }
   }

   // finds the largest clique with vertex v (start with m=1)
   void clique(int v, int m, vector<int> &st)
   {
      ++evaluations;

      st.push_back(v);

      if (m > bestk) { bestk = m; cv = st; }

      set<int> &adj = g[v];

      int tally = 0;
      for (set<int>::iterator it = adj.begin(); it != adj.end(); ++it)
         if (mark[*it] == m-1 && vinfo[*it].d >= bestk-1) {
            ++mark[*it];
            ++tally;
         }

      for (set<int>::iterator it = adj.begin(); it != adj.end(); ++it) {
         if (mark[*it] == m) {
            if (tally >= bestk-m) clique(*it, m+1, st);
            --mark[*it];
            --tally;
         }
      }

      st.pop_back();
   }

   // find a maximum clique in the graph
   int maxclique()
   {
      sortv();

      evaluations = 0;
      bestk = 0;
      mark = vector<int>(n, 0);
      vector<int> st;
      for (int v = 0; v < n; ++v) {
         if (v + bestk >= n) break;
         if (vinfo[v+bestk].d < bestk) break;
         clique(v, 1, st);
      }

      // transform the clique vertices to their originals
      for (vector<int>::iterator it = cv.begin(); it != cv.end(); ++it)
         *it = vinfo[*it].v;

      return bestk;
   }

   bool colour(int v, int k)
   {
      ++evaluations;

      if (v == n) return true;

      for (int c = 0; c < min(k, v+1); ++c) {
         if (forbidden[v][c] == 0) {
            set<int> &adj = g[v];

            for (set<int>::iterator it = adj.begin(); it != adj.end(); ++it)
               ++forbidden[*it][c];

            colouring[v] = c;
            if (colour(v+1, k)) return true;

            for (set<int>::iterator it = adj.begin(); it != adj.end(); ++it)
               --forbidden[*it][c];
         }
      }

      return false;
   }

   // find a minimum colouring of the graph
   int mincolouring()
   {
      sortv();

      evaluations = 0;
      bestk = 0;
      colouring = vector<int>(n, -1);

      // linear search on k... we assume it's small
      for (int k = 1; k <= n; ++k) {
         forbidden = vector< vector<int> >(n, vector<int>(k, 0));
         if (colour(0, k)) {
            bestk = k;
            break;
         }
      }

      // transform the colouring to their original (unsorted) vertex indices
      vector<int> tc(n);
      for (int i = 0; i < n; ++i)
         tc[vinfo[i].v] = colouring[i];
      colouring.swap(tc);

      return bestk;
   }

};
```

```
// comparator to sort vertices according to degree
bool operator<(const graph::vertex &a, const graph::vertex &b)
{
   return a.d > b.d;
}

//----------------------------------------------------------------------------
// test bed with fixed and random graphs

int main()
{
   // try it out on a 10-clique graph with 12 vertices
   cout << "Finding maximum clique for graph with k10..." << endl;
   graph gk10(12);
   for (int i = 1; i <= 10; ++i)
      for (int j = i+1; j <= 10; ++j)
         gk10.add(i, j);
   gk10.add(0, 9);
   gk10.add(11, 5);
   cout << "\tClique size:       " << gk10.maxclique() << endl;
   cout << "\tTotal evaluations: " << gk10.evaluations << endl;
   cout << "\tClique vertices:   ";
   copy(gk10.cv.begin(), gk10.cv.end(), ostream_iterator<int>(cout, " "));
   cout << endl << endl;
   cout << "Finding best colouring for graph with k10..." << endl;
   cout << "\tColours used:      " << gk10.mincolouring() << endl;
   cout << "\tTotal evaluations: " << gk10.evaluations << endl;
   cout << "\tVertex colouring:  ";
   copy(gk10.colouring.begin(), gk10.colouring.end(),
      ostream_iterator<int>(cout, " "));

   // try it out on the Coxeter graph
   cout << "Finding maximum clique in the complement Coxeter graph..." << endl;
   graph coxeter(28);
   for (int i = 0; i < 7; ++i) {
      coxeter.add(i, i+7);
      coxeter.add(i, i+14);
      coxeter.add(i, i+21);
      coxeter.add(i+7,  (i+1)%7 + 7);
      coxeter.add(i+14, (i+2)%7 + 14);
      coxeter.add(i+21, (i+3)%7 + 21);
   }
   // complement the graph
   for (int i = 0; i < 28; ++i)
      for (int j = 0; j < 28; ++j)
         if (i != j) coxeter.c[i][j] = !coxeter.c[i][j];
   cout << "\tClique size:       " << coxeter.maxclique() << endl;
   cout << "\tTotal evaluations: " << coxeter.evaluations << endl;
   cout << "\tClique vertices:   ";
   copy(coxeter.cv.begin(), coxeter.cv.end(), ostream_iterator<int>(cout, " "));
   cout << endl << endl;

   return 0;
}
//----------------------------------------------------------------------------
```

```
//---------------------------------------------------------------------------
// POWERS OF TWO AND COMBINATIONS
//
// This file includes nifty bit manipulation algorithms to calculate:
//    - power of 2 floor and ceiling for integer
//    - determining whether or not an integer is a power of 2
//    - next integer with same amout of 1 bits (snoob)
//
// These algorithms are courtesy of "Hacker's Delight" by Henry S. Warren Jr.
// (Addison-Wesley 2003 ISBN 0-201-91465-4)
//
// Author:  Sonny Chan
// Date:    November 12, 2003
//---------------------------------------------------------------------------

#include <iostream>

using namespace std;

//---------------------------------------------------------------------------
// integer power of 2 floor function
unsigned int p2floor(unsigned int x)
{
   x = x | (x >> 1);
   x = x | (x >> 2);
   x = x | (x >> 4);
   x = x | (x >> 8);
   x = x | (x >> 16);
   return x - (x >> 1);
}

// integer power of 2 ceiling function
unsigned int p2ceiling(unsigned int x)
{
   x -= 1;
   x = x | (x >> 1);
   x = x | (x >> 2);
   x = x | (x >> 4);
   x = x | (x >> 8);
   x = x | (x >> 16);
   return x + 1;
}

// determines whether an integer is a power of 2
bool ispower2(unsigned int x)
{
   return p2floor(x) == p2ceiling(x);
}

//---------------------------------------------------------------------------
// calculates the next integer with the same number of 1-bits
unsigned int snoob(unsigned int x)
{
   unsigned int smallest, ripple, ones;
                                    // x = xxx0 1111 0000
   smallest = x & -x;               //     0000 0001 0000
   ripple = x + smallest;           //     xxx1 0000 0000
   ones = x ^ ripple;               //     0001 1111 0000
   ones = (ones >> 2) / smallest;   //     0000 0000 0111
   return ripple | ones;            //     xxx1 0000 0111
}
//---------------------------------------------------------------------------
```

```
/****************************************************************************
BINARY SEARCH
Submitted March 21, 2004 by Kelly Poon
Original source courtesy of The University of Alberta
****************************************************************************/

/* Returns non-zero if x is found, and zero otherwise.  If x is found, then
A[index] = x.  If not, then index is the place x should be inserted into A. */
int bin_search(int *A, int n, int x, int *index){
   int low, up, mid;

   if (n <= 0 || x < A[0]) { *index = 0; return 0; }
   if (A[n-1] < x) { *index = n; return 0; }
   if (x == A[n-1]) { *index = n-1; return 1; }
   for(low = 0, up = n-1; low + 1 < up;){
     mid = (low+up)/2;
     if (A[mid] <= x) low = mid;
     else up = mid;
   }
   if (A[low] == x) { *index = low; return 1; }
   else { *index = up; return 0; }
}
```

```
/****************************************************************************
GEOMETRY ROUTINES
Submitted March 21, 2004 by Alex Fink
Original source courtesy of The University of Alberta
****************************************************************************/

#define EPS 1E-8
#define SQR(x) ((x)*(x))
#define SGN(x) ((x)<0?-1:1)

typedef struct {
  double x, y;
} Point;

typedef struct {
  Point o;
  double r;
} Circle;

/* distance squared */
double dist2(Point a, Point b) {
  return SQR(a.x-b.x) + SQR(a.y-b.y);
}

/* distance */
double dist_2d(Point a, Point b){
  return sqrt(SQR(a.x-b.x)+SQR(a.y-b.y));
}
```

```c
/* which side of a line */
enum {LEFT, RIGHT, CL};

int pt_leftright(Point a, Point b, Point p)
{
  double res;

  res = (p.x - a.x)*(b.y - a.y) -
        (p.y - a.y)*(b.x - a.x);

  if (fabs(res) < EPSILON)
    return CL;
  else if (res > 0.0)
    return RIGHT;
  return LEFT;
}

/* angle */
double angle2d(Point a, Point b, Point c){
  double dx1 = a.x - b.x, dy1 = a.y - b.y;
  double dx2 = c.x - b.x, dy2 = c.y - b.y;
  double dot = dx1 * dx2 + dy1 * dy2;
  double l1 = sqrt(SQR(dx1)+SQR(dy1));
  double l2 = sqrt(SQR(dx2)+SQR(dy2));

  return acos(dot / (l1*l2));
}

/* ax+by=c equation of line */
typedef struct{
  double a, b, c;
} Line;

Line pt2line(Point a, Point b){
  double dx = a.x-b.x, dy = a.y-b.y;
  double len = sqrt(SQR(dx)+SQR(dy));
  Line res;

  if(dy < 0){
    dy *= -1;
    dx *= -1;
  }
  res.a = dy/len;
  res.b = -dx/len;
  res.c = res.a*a.x + res.b*a.y;
  return res;
}

/* closest point to c on line ab */
Point closest_pt_iline(Point a, Point b, Point c) {
  Point p;
  double dp;

  b.x -= a.x;
  b.y -= a.y;
  dp = (b.x*(c.x-a.x) + b.y*(c.y-a.y)) / (SQR(b.x)+SQR(b.y));
  p.x = b.x*dp + a.x;
  p.y = b.y*dp + a.y;
  return p;
}

/* closest point to c on segment ab */
Point closest_pt_lineseg(Point a, Point b, Point c) {
  Point p;
  double dp;

  b.x -= a.x;
  b.y -= a.y;
  if (fabs(b.x) < EPS && fabs(b.y) < EPS) return a;
  dp = (b.x*(c.x-a.x) + b.y*(c.y-a.y))/(SQR(b.x)+SQR(b.y));
  if (dp > 1) dp = 1;
  if (dp < 0) dp = 0;
  p.x = b.x*dp + a.x;
  p.y = b.y*dp + a.y;
  return p;
}

/* distance from p to line ab */
double dist_iline(Point a, Point b, Point p){
  return fabs(((a.y-p.y)*(b.x-a.x)-
               (a.x-p.x)*(b.y-a.y))
              /dist_2d(a,b));
}

/* reflection of c across ab */
Point reflect(Point a, Point b, Point c) {
  Point d, p;

  d = closest_pt_iline(a,b,c);
  p.x = 2.0*d.x - c.x;
  p.y = 2.0*d.y - c.y;
  return p;
}

/* rotation of p around o */
Point rotate_2d(Point p, Point o, double theta){
  double m[2][2];
  Point r;

  m[0][0] = m[1][1] = cos(theta);
  m[0][1] = -sin(theta);
  m[1][0] = -m[0][1];
  p.x -= o.x;
  p.y -= o.y;
  r.x = m[0][0] * p.x + m[0][1] * p.y + o.x;
  r.y = m[1][0] * p.x + m[1][1] * p.y + o.y;
  if(fabs(r.x) < EPS) r.x = 0;
  if(fabs(r.y) < EPS) r.y = 0;
  return r;
}

/* intersection of lines */
int isect_iline(Point a, Point b, Point c, Point d, Point *p){
  double r, denom, num1;

  num1  = (a.y - c.y) * (d.x - c.x) - (a.x - c.x) * (d.y - c.y);
  denom = (b.x - a.x) * (d.y - c.y) - (b.y - a.y) * (d.x - c.x);

  if (fabs(denom) >= EPS) {
    r = num1 / denom;
    p->x = a.x + r*(b.x - a.x);
    p->y = a.y + r*(b.y - a.y);
    return 1;
  }
  if (fabs(num1) >= EPS) return 0;
  return -1;
}
```

```
/* intersection of segments */
int intersect_line(Point a, Point b, Point c, Point d, Point *p){
  Point t; double r, s, denom, num1, num2;

  num1  = (a.y - c.y)*(d.x - c.x) - (a.x - c.x)*(d.y - c.y);
  num2  = (a.y - c.y)*(b.x - a.x) - (a.x - c.x)*(b.y - a.y);
  denom = (b.x - a.x)*(d.y - c.y) - (b.y - a.y)*(d.x - c.x);

  if (fabs(denom) >= EPS) {
    r = num1 / denom;
    s = num2 / denom;
    if (0-EPS <= r && r <= 1+EPS &&
        0-EPS <= s && s <= 1+EPS) {
      p->x = a.x + r*(b.x - a.x);
      p->y = a.y + r*(b.y - a.y);
      return 1;
    }
    return 0;
  }
  if (fabs(num1) >= EPS) return 0;
  if (a.x > b.x || (a.x == b.x && a.y > b.y)) { t = a; a = b; b = t; }
  if (c.x > d.x || (c.x == d.x && c.y > d.y)) { t = c; c = d; d = t; }
  if (a.x == b.x) {
    if (b.y == c.y) {
      *p = b; return 1;
    } else if (a.y == d.y) {
      *p = a; return 1;
    } else if (b.y < c.y || d.y < a.y)
      return 0;
  } else {
    if (b.x == c.x) {
      *p = b; return 1;
    } else if (a.x == d.x) {
      *p = a; return 1;
    } else if (b.x < c.x || d.x < a.x)
      return 0;
  }
  return -1;
}

/* triangle area */
double area_tri(Point a, Point b, Point c){
  double area;

  area = (b.x-a.x) * (c.y-a.y)
        -(b.y-a.y) * (c.x-a.x);
  return (fabs(area))/2;
}

/* triangle area -- Heron's formula */
double area_heron(double a, double b, double c){
  double s = (a+b+c)/2.0;
  if(a > s || b > s || c > s) return -1;
  return sqrt(s*(s-a)*(s-b)*(s-c));
}

/* signed polygon area (counterclockwise positive) */
double area_poly(Point *p, int n){
  double sum = 0;
  int i, j;

  for(i = n-1, j = 0; j < n; i = j++)
    sum += p[i].x * p[j].y -
           p[i].y * p[j].x;
  return sum/2.0;
}

/* point in polygon */
#define BOUNDARY 1 // what to return for boundary points

int pt_in_poly(Point *p, int n, Point a) {
  int i, j, c = 0;

  for (i = 0, j = n-1; i < n; j = i++) {
    if (dist_2d(p[i],a)+dist_2d(p[j],a)-dist_2d(p[i],p[j]) < EPS)
      return BOUNDARY;
    if ((((p[i].y<=a.y) && (a.y<p[j].y)) ||
         ((p[j].y<=a.y) && (a.y<p[i].y))) &&
        (a.x < (p[j].x-p[i].x) * (a.y - p[i].y)
               / (p[j].y-p[i].y) + p[i].x)) c = !c;
  }
  return c;
}

/* centroid */
Point centroid(Point *p, int n) {
  double area, sum;
  Point c;
  int i, j;

  c.x = c.y = sum = 0.0;
  for(i = n-1, j = 0; j < n; i = j++) {
    sum += area = p[i].x * p[j].y - p[i].y * p[j].x;
    c.x += (p[i].x + p[j].x)*area;
    c.y += (p[i].y + p[j].y)*area;
  }
  sum *= 3.0;
  c.x /= sum;
  c.y /= sum;
  return c;
}

/* Pick's theorem */
void lat_poly_pick(Point *p, int n, long long *I, long long *B){
  int i, j, dx, dy;
  double A = fabs(area_poly(p, n));

  *B = 0;
  for(i = n-1, j = 0; j < n; i = j++){
    dx = abs(p[i].x - p[j].x);
    dy = abs(p[i].y - p[j].y);
    *B += gcd(dx,dy);
  }
  *I = A+1-*B/2.0;
}

/* convex hull (counterclockwise, minimum size) */
Point *P0;

enum {CCW, CW, CL};

int cross_prod(Point *p1, Point *p2, Point *p0)
{
  double res, x1, x2, y1, y2;

  x1 = p1->x - p0->x;
  x2 = p2->x - p0->x;
  y1 = p1->y - p0->y;
```

```c
  y2 = p2->y - p0->y;

  res = x1*y2 - x2*y1;

  if (fabs(res) < EPSILON)
    return CL;
  else if (res > 0.0)
    return CW;
  else
    return CCW;
}

int polar_cmp(Point *p1, Point *p2)
{
  int res;
  double d, x1, x2, y1, y2;

  res = cross_prod(p1, p2, P0);

  if (res == CW)
    return -1;
  else if (res == CCW)
    return 1;
  else {
    x1 = p1->x - P0->x;
    x2 = p2->x - P0->x;
    y1 = p1->y - P0->y;
    y2 = p2->y - P0->y;

    d = ((x1*x1) + (y1*y1)) - ((x2*x2) + (y2*y2));

    if (fabs(d) < EPSILON)
      return 0;
    else if (d < 0.0)
      return -1;
    else
      return 1;
  }
}

int convex_hull(Point *poly, int n, Point *hull)
{
  int i, min, h;

  if (n < 1)
    return 0;

  min = 0;
  P0 = &hull[0];
  *P0 = poly[0];

  for (i = 1; i < n; i++) {
    if ((poly[i].y < P0->y) ||
        ((poly[i].y == P0->y) && (poly[i].x < P0->x))) {
      min = i;
      *P0 = poly[i];
    }
  }

  poly[min] = poly[0];
  poly[0] = *P0;
  h = 1;

  if (n == 1)
    return h;

  qsort(poly+1, n-1, sizeof(poly[1]),
        (int (*)(const void *, const void *))polar_cmp);

  for (i = 1; i < n; i++) {
    if ((fabs(poly[i].x - hull[0].x) > EPSILON) ||
        (fabs(poly[i].y - hull[0].y) > EPSILON))  {
      break;
    }
  }

  if (i == n)
    return h;

  hull[h++] = poly[i++];

  for ( ; i < n; i++) {
    while ((h > 1) &&
           (cross_prod(&poly[i], &hull[h-1], &hull[h-2]) != CCW)) {
      h--;
    }

    hull[h++] = poly[i];
  }

  return h;
}
/* end of convex hull */

/* circle through 3 points */
int circle(Point p1, Point p2, Point p3, Point *center, double *r)
{
  double a,b,c,d,e,f,g;

  a = p2.x - p1.x;  b = p2.y - p1.y;
  c = p3.x - p1.x;  d = p3.y - p1.y;
  e = a*(p1.x + p2.x) + b*(p1.y + p2.y);
  f = c*(p1.x + p3.x) + d*(p1.y + p3.y);
  g = 2.0*(a*(p3.y - p2.y) - b*(p3.x - p2.x));
  if (fabs(g) < EPS) return 0;
  center->x = (d*e - b*f) / g;
  center->y = (a*f - c*e) / g;
  *r = sqrt((p1.x-center->x)*(p1.x-center->x) +
            (p1.y-center->y)*(p1.y-center->y));
  return 1;
}

/* tangents from point p to circle c, r */
void circ_tangents(Point c, double r, Point p, Point *a, Point *b) {
  double perp, para, tmp = dist2(p,c);

  para = r*r/tmp;
  perp = r*sqrt(tmp-r*r)/tmp;

  a->x = c.x + (p.x-c.x)*para - (p.y-c.y)*perp;
  a->y = c.y + (p.y-c.y)*para + (p.x-c.x)*perp;
  b->x = c.x + (p.x-c.x)*para + (p.y-c.y)*perp;
  b->y = c.y + (p.y-c.y)*para - (p.x-c.x)*perp;
}

/* intersection of circle and line */
int circ_iline_isect(Circle c, Point a, Point b,
                      Point *r1, Point *r2){
```

```
  double dx = b.x-a.x, dy = b.y-a.y;
  double sdr = SQR(dx)+SQR(dy), dr = sqrt(sdr);
  double D,disc,x,y;

  a.x -= c.o.x; a.y -= c.o.y;
  b.x -= c.o.x; b.y -= c.o.y;
  D = a.x*b.y - b.x*a.y;
  disc = SQR(c.r*dr)-SQR(D);

  if(disc < 0) return 0;
  x = SGN(dy)*dx*sqrt(disc);
  y = fabs(dy)*sqrt(disc);
  r1->x = (D*dy + x)/sdr + c.o.x;
  r2->x = (D*dy - x)/sdr + c.o.x;
  r1->y = (-D*dx + y)/sdr + c.o.y;
  r2->y = (-D*dx - y)/sdr + c.o.y;
  return disc == 0 ? 1 : 2;
}

/* intersection of circle and segment */
int circ_lineseg_isect(Circle c, Point a, Point b,
                        Point *r1, Point *r2){
  double d = dist_2d(a,b);
  int res = circ_iline_isect(c,a,b,r1,r2);

  if(res == 2 && dist_2d(a,*r2)+dist_2d(*r2,b) != d) res--;
  if(res >= 1 && dist_2d(a,*r1)+dist_2d(*r1,b) != d){
    *r1 = *r2;
    res--;
  }
  return res;
}

/* intersection of circles */
enum int_t {NONE=0, ONE, TWO, AEQUALSB, AINB, BINA,
            AINB_TANGENT, BINA_TANGENT};

Point rotate_2d(Point p, Point o, double theta){
  double m[2][2];
  Point r;

  m[0][0] = m[1][1] = cos(theta);
  m[0][1] = -sin(theta);
  m[1][0] = -m[0][1];
  p.x -= o.x;
  p.y -= o.y;
  r.x = m[0][0] * p.x + m[0][1] * p.y + o.x;
  r.y = m[1][0] * p.x + m[1][1] * p.y + o.y;
  if(fabs(r.x) < EPS) r.x = 0;
  if(fabs(r.y) < EPS) r.y = 0;
  return r;
}

int CIType (Circle A, Circle B) {
  double distance, dx = A.o.x - B.o.x, dy = A.o.y - B.o.y;

  distance = sqrt(dx*dx + dy*dy);

  if (distance < EPS && fabs(A.r-B.r) < EPS) return AEQUALSB;
  if (fabs(distance - (A.r + B.r)) < EPS) return ONE;
  if (distance > A.r + B.r) return NONE;
  if (distance + A.r <= B.r) {
    if (B.r - (distance+A.r) < EPS) return AINB_TANGENT;
    return AINB;
  }
  if (distance + B.r <= A.r) {
    if (A.r - (distance+B.r) < EPS) return BINA_TANGENT;
    return BINA;
  }
  return TWO;
}

int CIPoints (Circle A, Circle B, Point *s, Point *t) {
  double dx = B.o.x-A.o.x, dy = B.o.y-A.o.y;
  double dA, d, c, a;
  int type;

  type = CIType(A, B);

  d = sqrt(dx*dx + dy*dy);

  switch (type) {
  case AINB_TANGENT:
    s->x = B.o.x + (B.r/d)*-dx;
    s->y = B.o.y + (B.r/d)*-dy;
    return 1;

  case BINA_TANGENT: case ONE:
    s->x = A.o.x + (A.r/d)*dx;
    s->y = A.o.y + (A.r/d)*dy;
    return 1;

  case TWO:
    c = atan2(dy, dx);

    a = sqrt(4*SQR(d)*SQR(A.r) - SQR(SQR(d)-SQR(B.r)+SQR(A.r))) /d;

    dA = (SQR(d) - SQR(B.r) + SQR(A.r)) / (2*d);

    t->x = s->x = dA + A.o.x;
    s->y = a/2 + A.o.y;
    t->y = -a/2 + A.o.y;

    /* Rotate these points */
    *s = rotate_2d(*s, A.o, c);
    *t = rotate_2d(*t, A.o, c);

    return 2;

  default:
    return type;
  }
}
/* end of intersection of circles */
```

```
/*****************************************************************************
3D GEOMETRY ROUTINES
Submitted March 21, 2004 by Kelly Poon
Original source courtesy of The University of Alberta
*****************************************************************************/

#include <math.h>
#define EPS 1E-8
#define pt(a) &(a.x), &(a.y), &(a.z)

struct Point{
  double x, y, z;
  Point(){};
  Point(double xi, double yi, double zi){x = xi; y = yi; z = zi;}
};

Point operator + (const Point& a, const Point& b) {
  return Point(a.x + b.x, a.y + b.y, a.z + b.z);
}

Point operator * (double k, const Point& a) {
  return Point(k*a.x, k*a.y, k*a.z);
}

Point operator - (const Point& a, const Point& b) {
  return Point(a.x - b.x, a.y - b.y, a.z - b.z);
}

Point operator * (Point a, double k) {
  return (k*a);
}

Point operator / (Point a, double k) {
  return (1.0/k)*a;
}

double dot(const Point& a, const Point& b) {
  return a.x*b.x + a.y*b.y + a.z*b.z;
}

Point cross(const Point& a, const Point& b) {
  return Point(a.y*b.z-b.y*a.z, b.x*a.z-a.x*b.z, a.x*b.y-b.x*a.y);
}

double length2(const Point& a) {
  return dot(a,a);
}

double length(const Point& a) {
  return sqrt(dot(a,a));
}

Point closest_pt_iline(const Point& a, const Point& b, const Point& p) {
  double along = dot(b-a,p-a)/length2(b-a);
  return (b-a)*along + a;
}

Point closest_pt_seg(const Point& a, const Point& b, const Point& p) {
  double along;

  if (length2(b-a) < EPS) return a;
  along = dot(b-a,p-a)/length2(b-a);
  if (along < 0) along = 0;
  if (along > 1) along = 1;
  return (b-a)*along + a;
}

/* plane represented by a normal and a point on plane */
Point closest_pt_plane(const Point& norm, const Point& a, const Point& p) {
  Point res = cross(cross(norm,p-a),norm);
  if (length2(res) < EPS) return a;
  return res*dot(res,p-a)/length2(res);
}

/* plane represented by three points */
Point closest_pt_plane(const Point& a, const Point& b, const Point& c, const
Point& p) {
  Point norm;

  norm = cross(b-a,c-a);
  /*assert(length2(norm) > EPS);*/  // collinearity
  return closest_pt_plane(norm,a,p);
}

/* returns number of intersections and the intersections*/
int sphere_iline_isect(const Point& c, double r, const Point& a, const Point& b,
                       Point *p, Point *q) {
  Point vec, mid = closest_pt_iline(a,b,c);

  if (length2(c-mid) > r*r) return 0;
  vec = (a-b)*sqrt((r*r - length2(c-mid))/length2(a-b));
  *p = mid + vec;
  *q = mid - vec;
  return ((length2(vec) > EPS) ? 2 : 1);
}

/* project point p to the plane defined by a, b and c */
Point to_plane(const Point& a, const Point& b, const Point& c, const Point& p) {
  Point norm, ydir, xdir, res;

  norm = cross(b-a,c-a);
  /*assert(length2(norm) > EPS);*/  // collinearity
  xdir = (b-a)/length(b-a); // create orthonormal vectors
  ydir = cross(norm,xdir);
  ydir = ydir/length(ydir);
  res.x = dot(p-a,xdir);
  res.y = dot(p-a,ydir);
  res.z = 0;
  return res;
}

/* given two lines in 3D space, find distance of closest approach */
double line_line_dist(const Point& a, const Point& b, const Point& c, const
Point& d) {
  Point perp = cross(b-a,d-c);

  if (length2(perp) < EPS)  /* parallel */
    perp = cross(b-a,cross(b-a,c-a));
  if (length2(perp) < EPS) return 0; /* coincident */

  return fabs(dot(a-c,perp))/length(perp);
}

/* same as line_line_dist, but returns the points of closest approach */
double closest_approach(const Point& a, const Point& b, const Point& c, const
Point& d,
                                                   Point *p, Point *q) {
  double s = dot(d-c,b-a), t = dot(a-c,d-c);
  double num, den, tmp;

  den = length2(b-a)*length2(d-c) - s*s;
  num = t*s - dot(a-c,b-a)*length2(d-c);
  if (fabs(den) < EPS) { /* parallel */
    *p = a;
    *q = (d-c)*t/length2(d-c) + c;
    if (fabs(s) < EPS) *q = a;  /* coincident */
  } else {  /* skew */
    tmp = num/den;
    *p = a + (b-a)*tmp;
    *q = c + (d-c)*(t + s*tmp)/length2(d-c);
  }
  return length(*p-*q);
}

/* is the point p on the infinite line ab? */
int on_iline(const Point& a, const Point& b, const Point& p) {
  return (length2(p-closest_pt_iline(a,b,p)) < EPS);
}

/* is the point p on the segment ab? */
int on_seg(const Point& a, const Point& b, const Point& p) {
  return (length(a-p) + length(p-b) - length(a-b) < EPS);
}

/* Given a plane and a line ab, determine if the two intersect,
   and if so, find the single point of intersection */
int plane_iline_isect(const Point& norm, const Point& ori, const Point& a, const
Point& b, Point *p) {
  double along, den = dot(norm,b-a);

  if (fabs(den) < EPS) { /* parallel */
    if (length2(cross(ori-a,b-a)) < EPS) return -1; /* coincident */
    return 0;  /* non-intersecting */
  }
  along = dot(norm,ori-a)/den;

  /* if you want to intersect a plane with a finite segment,
     check that (along <= 1 && along => 0) */
  *p = a + along*(b-a);
  return 1;
}
```

```
/* triangulate.h  - triangulates a polygon in O(n^2) time */
/*                  (note: fails on degenerate case of 3 collinear points) */
#include <list>
#include <vector>

using namespace std;

#define EPS 1e-8
#define ORDER 1 /* 1: cw, -1: ccw */

struct Point {
  double x, y;
};
struct Triangle {
  Point p[3];
};

/* classifies p as either being -1 left of, 1 right of or 0 on the line ab. */
int leftRight(Point &a, Point &b, Point &p){
  double res = ((b.x - a.x)*(p.y - a.y) - (p.x - a.x)*(b.y - a.y));
  if (res > EPS) return -1;
  else if (res < -EPS) return 1;
  return 0;
}

/* returns non-0 if b in the sequence a->b->c is concave, 0 for convex. */
int isConcave(Point &a, Point &b, Point &c){
  return (ORDER*leftRight(a, b, c) <= 0);
}

/* returns non-zero if point p is located on or inside the triangle <a b c>. */
int isInsideTriangle(Point &a, Point &b, Point &c, Point &p){
  int r1 = leftRight(a, b, p);
  int r2 = leftRight(b, c, p);
  int r3 = leftRight(c, a, p);
  return ((ORDER*r1 >= 0) && (ORDER*r2 >= 0) && (ORDER*r3 >= 0));
}

/* P - n cw-ordered points of a polygon (n>=3, P modified during function
   T - n-2 triangles, returns the triangulation of P */
void triangulate(list<Point> &P, vector<Triangle> &T){
  list<Point>::iterator a, b, c, q;
  Triangle t;

  T.clear();
  if (P.size() < 3) return;

  for (a=b=P.begin(), c=++b, ++c; c != P.end(); a=b, c=++b, ++c) {
    if (!isConcave(*a, *b, *c)) {
      for (q = P.begin(); q != P.end(); q++) {
        if (q == a) { ++q; ++q; continue; }
        if (isInsideTriangle(*a, *b, *c, *q)) break;
      }
      if (q == P.end()) {
        t.p[0] = *a; t.p[1] = *b; t.p[2] = *c;
        T.push_back(t);
        P.erase(b);
        b = a;
        if (b != P.begin()) b--;
      }
    }
  }
}
```

```c
/******************************************************************************
LONGEST INCREASING SUBSEQUENCE
Submitted March 21, 2004 by Kelly Poon
Original source courtesy of University of Alberta
******************************************************************************/

#include <stdlib.h>

/* Given an array of size n, asc_seq returns the length
   of the longest ascending subsequence, as well as one
   of the subsequences in S.*/
int asc_seq(int *A, int n, int *S){
  int *m, *seq, i, k, low, up, mid, start;

  m   = (int*)malloc((n+1) * sizeof(int));
  seq = (int*)malloc(n * sizeof(int));
  /* assert(m && seq); */

  for (i = 0; i < n; i++) seq[i] = -1;
  m[1] = start = 0;
  for (k = i = 1; i < n; i++) {
    if (A[i] >= A[m[k]]) {
      seq[i] = m[k++];
      start = m[k] = i;
    } else if (A[i] < A[m[1]]) {
      m[1] = i;
    } else {
      /* assert(A[m[1]] <= A[c] && A[c] < A[m[k]]); */
      low = 1;
      up = k;
      while (low != up-1) {
         mid = (low+up)/2;
         if(A[m[mid]] <= A[i]) low = mid;
         else up = mid;
      }
      seq[i] = m[low];
      m[up] = i;
    }
  }
  for (i = k-1; i >= 0; i--) {
    S[i] = A[start];
    start = seq[start];
  }
  free(m); free(seq);
  return k;
}
```

```c
/******************************************************************************
LONGEST COMMON SUBSEQUENCE AND EDIT DISTANCE
Submitted March 23, 2004 by Kelly Poon
Original source courtesy of University of Alberta
******************************************************************************/

#include <stdlib.h>

#define MAXN 20
#define Atype int
#define max(x,y) (((x)>(y))?(x):(y))

int LCS(Atype *A, int n, Atype *B, int m, Atype *s)
{
  int L[MAXN+1][MAXN+1];
  int i, j, k;

  for(i = n; i >= 0; i--) for(j = m; j >= 0; j--){
    if(i == n || j == m){
      L[i][j] = 0;
    } else if(A[i] == B[j]){
      L[i][j] = 1 + L[i+1][j+1];
    } else {
      L[i][j] = max(L[i+1][j], L[i][j+1]);
    }
  }

  /* The following is not needed if you are not interested in
     returning a longest common subsequence */
  k = 0;
  i = j = 0;
  while(i < n && j < m){
    if(A[i] == B[j]){
      s[k++] = A[i++];
      j++;
    } else if(L[i+1][j] > L[i][j+1]){
      i++;
    } else if(L[i+1][j] < L[i][j+1]){
      j++;
    } else {
      /* tie breaking conditions here*/
      j++;
    }
  }
  return L[0][0];
}


/*** EDIT DISTANCE CODE ***/

#include <string.h>

#define MAXN 90

char move[MAXN][MAXN];  /* Type of command used */
int  g[MAXN][MAXN];      /* Cost of changes */

int editDistance(char *src, char *dst, int replace, int insert, int delete){
  int i, j, l1, l2;

  l1 = strlen(src);
  l2 = strlen(dst);
  for(j = 0; j <= l1; j++){
    g[0][j] = j;
    move[0][j] = 'D';
  }

  for(i = 1; i <= l2; i++){
    g[i][0] = i;
    move[i][0] = 'I';

    for(j = 1; j <= l1; j++){
      g[i][j] = g[i-1][j-1]+replace;
      move[i][j] = 'R';

      if(g[i-1][j]+insert < g[i][j]){
        g[i][j] = g[i-1][j]+insert;
        move[i][j] = 'I';
      }

      if(g[i][j-1]+delete < g[i][j]){
        g[i][j] = g[i][j-1]+delete;
        move[i][j] = 'D';
      }

      if(src[j-1] == dst[i-1] && g[i-1][j-1] < g[i][j]){
        g[i][j] = g[i-1][j-1];
        move[i][j] = 'N';
      }
    }
  }
  return g[l2][l1];
}

int counter;
void PathRecovery(int x, int y, int *delta, char *src, char *dst){
  int ndelta;

  if(x == 0 && y == 0){
    *delta = 0;
    return;
  }
  else {
    switch(move[x][y]){
    case 'R':
      PathRecovery(x-1,y-1,&ndelta,src,dst);
      *delta = ndelta;
      printf("%d Replace %d,%c\n", counter++, y+ndelta, dst[x-1]);
      break;
    case 'I':
      PathRecovery(x-1,y,&ndelta,src,dst);
      *delta = ndelta+1;
      printf("%d Insert %d,%c\n", counter++, y+ndelta+1, dst[x-1]);
      break;
    case 'D':
      PathRecovery(x,y-1,&ndelta,src,dst);
      *delta = ndelta-1;
      printf("%d Delete %d\n", counter++, y+ndelta);
      break;
    case 'N':
      PathRecovery(x-1,y-1,&ndelta,src,dst);
      *delta = ndelta;
      break;
    }
  }
}
```

```
/* matrix.h - contains matrix and vector maths
   NOTE: be careful using homogenous coords */
#include <vector>
#include <math.h>

using namespace std;

#ifndef MATRIX_H
#define MATRIX_H

#define Matrix vector < vector<double> >
#define Vector vector<double>

bool ludcmp(Matrix& a, vector<int>& indx, double& d);
void lubksb(const Matrix& a, const vector<int>& indx, Vector& b);

Vector operator+(const Vector& v1, const Vector& v2){
    Vector ret(v1.size());
    for(int i = 0; i < v1.size(); i++) ret[i] = v1[i] + v2[i];
    return ret;
}
Vector operator-(const Vector& v1, const Vector& v2){
    Vector ret(v1.size());
    for(int i = 0; i < v1.size(); i++) ret[i] = v1[i] - v2[i];
    return ret;
}
Vector operator*(const double d, const Vector& v){
    Vector ret(v.size());
    for(int i = 0; i < v.size(); i++) ret[i] = d*v[i];
    return ret;
}
double dot(const Vector& v1, const Vector& v2){
    double ret = 0.0;
    for(int i = 0; i < v1.size(); i++) ret += v1[i]*v2[i];
    return ret;
}
double length(const Vector& v){
    return sqrt(dot(v,v));
}
Matrix operator+(const Matrix& m1, const Matrix& m2){
    Matrix ret(m1.size(), Vector(m1[0].size()));
    for(int i = 0; i < m1.size(); i++)
        for(int j = 0; j < m1[0].size(); j++)
            ret[i][j] = m1[i][j] + m2[i][j];
    return ret;
}
Matrix operator-(const Matrix& m1, const Matrix& m2){
    Matrix ret(m1.size(), Vector(m1[0].size()));
    for(int i = 0; i < m1.size(); i++)
        for(int j = 0; j < m1[0].size(); j++)
            ret[i][j] = m1[i][j] - m2[i][j];
    return ret;
}
Matrix operator*(const double s, const Matrix& m){
    Matrix ret(m.size(), Vector(m[0].size()));
    for(int i = 0; i < m.size(); i++)
        for(int j = 0; j < m[0].size(); j++)
            ret[i][j] = s*m[i][j];
    return ret;
}
Matrix operator*(const Matrix& m1, const Matrix& m2){
    Matrix ret(m1.size(), Vector(m2[0].size(), 0.0));
    for(int r = 0; r < m1.size(); r++)
        for(int c = 0; c < m2[0].size(); c++)
```

```
            for(int i = 0; i < m2.size(); i++)
                ret[r][c] += m1[r][i]*m2[i][c];
    return ret;
}
Vector operator*(const Matrix& m, const Vector& v){
    Vector ret(m.size(), 0.0);
    for(int r = 0; r < m.size(); r++)
        for(int c = 0; c < m[0].size(); c++)
            ret[r] += m[r][c]*v[c];
    return ret;
}
#include <iostream>
Matrix id(int N){
    Matrix ret(N, Vector(N, 0.0));
    for(int i = 0; i < N; i++)
        ret[i][i] = 1.0;
    return ret;
}

/* inverts m (assumes m is square) */
Matrix inverse(const Matrix& m){
    int N = m.size();
    Matrix mT = m, inv(N, Vector(N, 0.0));
    double d;
    vector<int> indx(N);

    ludcmp(mT, indx, d);
    for(int j = 0; j < N; j++){
        Vector col(N, 0.0); col[j] = 1.0;
        lubksb(mT, indx, col);
        for(int i = 0; i < N; i++) inv[i][j] = col[i];
    }

    return inv;
}

/* returns the determinant of m, an NxN matrix in O(N^3) */
double determinant(const Matrix& m){
    int N = m.size();
    Matrix mT = m;
    double d;
    vector<int> indx(N);

    ludcmp(mT, indx, d);
    for(int j = 0; j < N; j++) d*= mT[j][j];

    return d;
}

/* return the solution, x, to Ax = b (assumes A is NxN and b is N) */
Vector solve(const Matrix& A, const Vector& b){
    int N = A.size();
    Matrix aT = A;
    Vector x = b;
    double d;
    vector<int> indx(N);

    ludcmp(aT, indx, d);
    lubksb(aT, indx, x);

    return x;
}

#endif
```

```
/* lu.h - LU Decomposition */

#include "matrix.h"
#define TINY 1.0e-20

/* replaces A with the LU decomposition of A rowwise permutation of A
   indx records the row permutation effected by pivoting
   d is 1.0 if n interchanges is even, else -1 */
bool ludcmp(Matrix& A, vector<int>& indx, double& d)
{
    int i, j, k, imax = 0, n = A.size();
    double big, dum, sum, temp;
    vector<double> vv(n+1);

    d = 1.0;
    for(i = 0; i < n; i++){
        big = 0.0;
        for(j = 0;j < n; j++)
            if((temp = fabs(A[i][j])) > big) big = temp;
        if(big == 0.0) return false; /* singular matrix */
        vv[i] = 1.0/big;
    }
    for(j = 0; j < n; j++){
        for(i = 0; i < j; i++){
            sum = A[i][j];
            for(k = 0; k < i; k++) sum -= A[i][k]*A[k][j];
            A[i][j] = sum;
        }
        big = 0.0;
        for(i = j; i < n; i++){
            sum = A[i][j];
            for(k = 0; k < j; k++) sum -=A[i][k]*A[k][j];
            A[i][j] = sum;
            if((dum = vv[i]*fabs(sum)) >= big){
                big = dum;
                imax = i;
            }
        }
        if(j != imax){
            for(k = 0; k < n; k++){
                dum = A[imax][k];
                A[imax][k] = A[j][k];
                A[j][k] = dum;
            }
            d = -d;
            vv[imax] = vv[j];
        }
        indx[j] = imax;
        if(A[j][j] == 0.0) A[j][j]=TINY;
        if(j != n-1){
            dum = 1.0/(A[j][j]);
            for(i = j+1; i < n; i++) A[i][j] *= dum;
        }
    }

    return true;
}

/* solves Ax=b (returns x in b) */
void lubksb(const Matrix& A, const vector<int>& indx, Vector& b)
{
    int i, ip, j, ii=0, n = A.size();
    double sum;
    for(i = 0; i < n; i++){
        ip = indx[i];
        sum = b[ip];
        b[ip] = b[i];
        if(ii) for(j = ii-1; j < i; j++) sum -= A[i][j]*b[j];
        else if(sum) ii = i+1;
        b[i] = sum;
    }
    for(i = n-1; i >= 0; i--){
        sum = b[i];
        for(j = i+1; j < n; j++) sum -= A[i][j]*b[j];
        b[i] = sum/A[i][i];
    }
}
```

Here is a potentially useful excerpt from *3D Game Engine Design* by David H. Eberly (Morgan Kaufmann, 2001):

Angle-Axis to Rotation Matrix

Any standard computer graphics text discusses the relationship between an angle and axis of rotation and the rotation matrix, although the constructions can be varied. A useful one is given here. If θ is the angle of rotation and $\vec{U}$ is the unit-length axis of rotation, then the corresponding rotation matrix is

$$R = I + (\sin\theta)S + (1-\cos\theta)S^2,$$

where I is the identity matrix and

$$S = \begin{bmatrix} 0 & -u_2 & u_1 \\ u_2 & 0 & -u_0 \\ -u_1 & u_0 & 0 \end{bmatrix}$$

a skew-symmetric matrix. For $\theta > 0$, the rotation represent a counterclockwise rotation about the axis. The sense of clockwise or counterclockwise is based on looking at the plane with normal $\vec{U}$ from the side of the plane to which the normal points. Note that $S\vec{V} = \vec{U} \times \vec{V}$ and

$$R\vec{V} = \vec{V} + (\sin\theta)\vec{U}\times\vec{V} + (1-\cos\theta)\vec{U}\times(\vec{U}\times\vec{V}).$$

```
/* rotations.h - makes rotation matrices */
#include "matrix.h"
#include <math.h>

/* rotations about main axes */
Matrix rotX(double angle){
    double cosa = cos(angle), sina = sin(angle);
    Matrix ret = id(4);
    ret[1][1] = cosa; ret[1][2] = -sina;
    ret[2][1] = sina; ret[2][2] = cosa;
    return ret;
}
Matrix rotY(double angle){
    double cosa = cos(angle), sina = sin(angle);
    Matrix ret = id(4);
    ret[0][0] = cosa; ret[0][2] = sina;
    ret[2][0] = -sina; ret[2][2] = cosa;
    return ret;
}
Matrix rotZ(double angle){
    double cosa = cos(angle), sina = sin(angle);
    Matrix ret = id(4);
    ret[0][0] = cosa; ret[0][1] = -sina;
    ret[1][0] = sina; ret[1][1] = cosa;
    return ret;
}

/* rotation about arbitrary axis (flattens to z) */
Matrix rot(double angle, Vector axis){
    double u = axis[0], v = axis[1], w = axis[2];
    double u2 = u*u, v2 = v*v, w2 = w*w, len2 = u2 + v2 + w2, len = sqrt(len2);
    double cosa = cos(angle), sina = sin(angle);

    Matrix ret = id(4);
    ret[0][0] = (u2+(v2+w2)*cosa)/len2;
    ret[0][1] = (u*v*(1-cosa)-w*len*sina)/len2;
    ret[0][3] = (u*w*(1-cosa)+v*len*sina)/len2;
    ret[1][0] = (u*v*(1-cosa)+w*len*sina)/len2;
    ret[1][1] = (v2+(u2+w2)*cosa)/len2;
    ret[1][2] = (v*w*(1-cosa)-u*len*sina)/len2;
    ret[2][0] = (u*w*(1-cosa) - v*len*sina)/len2;
    ret[2][1] = (v*w*(1-cosa)+ u*len*sina)/len2;
    ret[2][2] = (w2+(u2+v2)*cosa)/len2;

    return ret;
}

/* rotation about the axis parallel to axis that goes through point */
Matrix rot(double angle, Vector axis, Vector point){
    double u = axis[0], v = axis[1], w = axis[2];
    double a = point[0], b = point[1], c = point[2];
    double u2 = u*u, v2 = v*v, w2 = w*w, len2 = u2 + v2 + w2, len = sqrt(len2);
    double cosa = cos(angle), sina = sin(angle);

    Matrix ret = rot(angle, axis);
    ret[0][3] = (a*(v2+w2)-u*(b*v-c*w)+(u*(b*v+c*w)-a*(v2+w2))*cosa+(b*w-
c*v)*len*sina)/len2;
    ret[1][3] = (b*(u2+w2)-v*(a*u+c*w)+(v*(a*u+c*w)-b*(u2+w2))*cosa+(c*u-
a*w)*len*sina)/len2;
    ret[2][3] = (c*(u2+v2)-w*(a*u+b*v)+(w*(a*u+b*v)-c*(u2+v2))*cosa+(a*v-
b*u)*len*sina)/len2;

    return ret;
}
```

```c
/*******************************************************************************
TABLE OF PRIMES, EXTENDED EUCLIDEAN ALGORITHM, CHINESE REMAINDER THEOREM
    AND CUBIC EQUATION SOLVER
Author:  Alex Fink, with some original source form Howard Cheng
Date:    March 22, 2004
*******************************************************************************/

int primes[6543];

void generate_primes(int max) {
  int i, j, k = 1;
  primes[0] = 2;
  for(i = 3; i < max; i += 2) {
    for(j = 0; j < k; j++)
      if (!(i%primes[j]))
         break;
    if (j == k)
      primes[k++] = i;
  }
  primes[k++] = 0;
}

void print_table() {
  int i;
  printf("{");
  for(i = 0; primes[i]; i++)
    printf("%d,%s", primes[i], 12==i%13?"\n":"");
  printf("0}\n");
}

int least_factor(int n) {
  int i;
  for(i = 0; primes[i]; i++)
    if (!(n%primes[i]))
      return primes[i];
  return n;
}

int extended_euclid(int a, int b, int *x, int *y) {
  int d, t;
  if (!b) {
    *x = 1;
    *y = 0;
    return a;
  }
  d = extended_euclid(b, a%b, &t, x);
  *y = t - *x*(a/b);
  return d;
}

/* Chinese remainder theorem for x % m[i] = a[i] */
int cra(int n, int *m, int *a){
  int x, i, k, prod, temp;
  int *gamma, *v;

  gamma = (int *)malloc(n*sizeof(int));
  v     = (int *)malloc(n*sizeof(int));

  /* compute inverses */
  for (k = 1; k < n; k++) {
    prod = m[0] % m[k];
    for (i = 1; i < k; i++) {
      prod = (prod * m[i]) % m[k];
    }
    extended_euclid(prod, m[k], gamma+k, &temp);
    gamma[k] %= m[k];
    if (gamma[k] < 0) {
      gamma[k] += m[k];
    }
  }

  /* compute coefficients */
  v[0] = a[0];
  for (k = 1; k < n; k++) {
    temp = v[k-1];
    for (i = k-2; i >= 0; i--) {
      temp = (temp * m[i] + v[i]) % m[k];
      if (temp < 0) {
        temp += m[k];
      }
    }
    v[k] = ((a[k] - temp) * gamma[k]) % m[k];
    if (v[k] < 0) {
      v[k] += m[k];
    }
  }

  /* convert from mixed-radix representation */
  x = v[n-1];
  for (k = n-2; k >= 0; k--) {
    x = x * m[k] + v[k];
  }
  free(gamma);
  free(v);
  return x;
}

/* solve a cubic equation */
typedef struct{
  int n;        /* Number of solutions */
  double x[3];  /* Solutions */
} Result;

double PI; // PI = acos(-1);

Result solve_cubic(double a, double b, double c, double d){
  Result s;
  long double a1 = b/a, a2 = c/a, a3 = d/a;
  long double q = (a1*a1 - 3*a2)/9.0, sq = -2*sqrt(q);
  long double r = (2*a1*a1*a1 - 9*a1*a2 + 27*a3)/54.0;
  double z = r*r-q*q*q;
  double theta;

  if(z <= 0){
    s.n = 3;
    theta = acos(r/sqrt(q*q*q));
    s.x[0] = sq*cos(theta/3.0) - a1/3.0;
    s.x[1] = sq*cos((theta+2.0*PI)/3.0) - a1/3.0;
    s.x[2] = sq*cos((theta+4.0*PI)/3.0) - a1/3.0;
  } else {
    s.n = 1;
    s.x[0] = pow(sqrt(z)+fabs(r),1/3.0);
    s.x[0] += q/s.x[0];
    s.x[0] *= (r < 0) ? 1 : -1;
    s.x[0] -= a1/3.0;
  }
  return s;
}
```

Jabberwocky!

Lewis Carroll

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogroves,
And the mome raths outgrabe.

"Beware the Jabberock, my son!
The jaws that bite, the claws that catch!
Beware the Jubjub bird, and shun
The frumious Bandersnatch!"

He took his vorpal blade in hand;
Long time the manxome foe he sought-
So rested he by the Tumtum tree,
And stood a while in thought

And, as in uffish thought he stood,
The jabberwock, with eyes of flame,
Came whiffling through the tulgey wood,
And burbled as it came!

One, two! One, two! And through and through
The vorpal blade went snicker-snack!
He left it dead, and with its head
He went galumphing back.

"And hast thou slain the Jabberwock?
Come to my arms, my beamish boy!
O frabjous day! Callooh, Callay!"
He chortled in his joy.

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogroves,
And the mome raths outgrabe.