LalCheetah ICPC Team Notebook (2016-17)

Contents

1 Combinatorial optimization

1.1 Sparse max-flow(aka Modified Ford Fulkerson)

```cpp
// 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 and sink
//
// OUTPUT:
//     - maximum flow value
//     - To obtain actual flow values, look at edges with capacity > 0
//       (zero capacity edges are residual edges).

#include<cstdio>
#include<vector>
#include<queue>
using namespace std;
typedef long long LL;

struct Edge {
  int u, v;
  LL cap, flow;
  Edge() {}
  Edge(int u, int v, LL cap): u(u), v(v), cap(cap), flow(0) {}
};

struct Dinic {
  int N;
  vector<Edge> E;
  vector<vector<int>> g;
  vector<int> d, pt;

  Dinic(int N): N(N), E(0), g(N), d(N), pt(N) {}

  void AddEdge(int u, int v, LL cap) {
    if (u != v) {
      E.emplace_back(Edge(u, v, cap));
      g[u].emplace_back(E.size() - 1);
      E.emplace_back(Edge(v, u, 0));
      g[v].emplace_back(E.size() - 1);
    }
  }

  bool BFS(int S, int T) {
    queue<int> q({S});
    fill(d.begin(), d.end(), N + 1);
    d[S] = 0;
    while(!q.empty()) {
      int u = q.front(); q.pop();
      if (u == T) break;
      for (int k: g[u]) {
        Edge &e = E[k];
        if (e.flow < e.cap && d[e.v] > d[e.u] + 1) {
          d[e.v] = d[e.u] + 1;
          q.emplace(e.v);
        }
      }
    }
    return d[T] != N + 1;
  }

  LL DFS(int u, int T, LL flow = -1) {
    if (u == T || flow == 0) return flow;
    for (int &i = pt[u]; i < g[u].size(); ++i) {
      Edge &e = E[g[u][i]];
      Edge &oe = E[g[u][i]^1];
      if (d[e.v] == d[e.u] + 1) {
        LL amt = e.cap - e.flow;
        if (flow != -1 && amt > flow) amt = flow;
        if (LL pushed = DFS(e.v, T, amt)) {
          e.flow += pushed;
          oe.flow -= pushed;
          return pushed;
        }
      }
    }
    return 0;
  }

  LL MaxFlow(int S, int T) {
    LL total = 0;
    while (BFS(S, T)) {
      fill(pt.begin(), pt.end(), 0);
      while (LL flow = DFS(S, T))
        total += flow;
    }
    return total;
  }
};

// BEGIN CUT
// The following code solves SPOJ problem #4110: Fast Maximum Flow (FASTFLOW)

int main()
{
  int N, E;
  scanf("%d%d", &N, &E);
  Dinic dinic(N);
  for(int i = 0; i < E; i++)
  {
    int u, v;
    LL cap;
    scanf("%d%d%lld", &u, &v, &cap);
    dinic.AddEdge(u - 1, v - 1, cap);
    dinic.AddEdge(v - 1, u - 1, cap);
  }
  printf("%lld\n", dinic.MaxFlow(0, N - 1));
  return 0;
}

// END CUT
```

1.2 Global min-cut

```cpp
// 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);
}

// BEGIN CUT
// The following code solves UVA problem #10989: Bomb, Divide and Conquer
int main() {
  int N;
  cin >> N;
  for (int i = 0; i < N; i++) {
    int n, m;
    cin >> n >> m;
    VVI weights(n, VI(n));
    for (int j = 0; j < m; j++) {
      int a, b, c;
      cin >> a >> b >> c;
      weights[a-1][b-1] = weights[b-1][a-1] = c;
    }
```

```
    pair<int, VI> res = GetMinCut(weights);
    cout << "Case #" << i+1 << ": " << res.first << endl;
  }
}
// END CUT
```

2 Geometry

2.1 Convex hull

```
// 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>
// BEGIN CUT
#include <map>
// END CUT

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
}

// BEGIN CUT
// The following code solves SPOJ problem #26: Build the
     Fence (BSHEEP)

int main() {
  int t;
  scanf("%d", &t);
  for (int caseno = 0; caseno < t; caseno++) {
    int n;
    scanf("%d", &n);
    vector<PT> v(n);
    for (int i = 0; i < n; i++) scanf("%lf%lf", &v[i].x, &v
        [i].y);
    vector<PT> h(v);
    map<PT,int> index;
    for (int i = n-1; i >= 0; i--) index[v[i]] = i+1;
    ConvexHull(h);

    double len = 0;
    for (int i = 0; i < h.size(); i++) {
      double dx = h[i].x - h[(i+1)%h.size()].x;
      double dy = h[i].y - h[(i+1)%h.size()].y;
      len += sqrt(dx*dx+dy*dy);
    }

    if (caseno > 0) printf("\n");
    printf("%.2f\n", len);
    for (int i = 0; i < h.size(); i++) {
      if (i > 0) printf(" ");
      printf("%d", index[h[i]]);
    }
    printf("\n");
  }
}

// END CUT
```

2.2 Miscellaneous geometry

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

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

using namespace std;

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

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

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

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

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

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

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

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

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

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

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

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

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

// determine if point is in a possibly non-convex polygon (
     by William
// Randolph Franklin); returns 1 for strictly interior
     points, 0 for
// strictly exterior points, and 0 or 1 for the remaining
     points.
// Note that it is possible to convert this into an *exact*
     test using
// integer arithmetic by taking care of the division
```

```
    appropriately
// (making sure to deal with signs properly) and then by
    writing exact
// tests for checking point on polygon boundary
bool PointInPolygon(const vector<PT> &p, PT q) {
  bool c = 0;
  for (int i = 0; i < p.size(); i++){
    int j = (i+1)%p.size();
    if ((p[i].y <= q.y && q.y < p[j].y ||
      p[j].y <= q.y && q.y < p[i].y) &&
      q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) / (
          p[j].y - p[i].y))
      c = !c;
  }
  return c;
}

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

// compute intersection of line through points a and b with
// circle centered at c with radius r > 0
vector<PT> CircleLineIntersection(PT a, PT b, PT c, double
    r) {
  vector<PT> ret;
  b = b-a;
  a = a-c;
  double A = dot(b, b);
  double B = dot(a, b);
  double C = dot(a, a) - r*r;
  double D = B*B - A*C;
  if (D < -EPS) return ret;
  ret.push_back(c+a+b*(-B+sqrt(D+EPS))/A);
  if (D > EPS)
    ret.push_back(c+a+b*(-B-sqrt(D))/A);
  return ret;
}

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

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

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

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

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

int main() {

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  return 0;
}
```

3 Numerical algorithms

3.1 Fast Fourier transform

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

struct cpx
{
  cpx(){}
  cpx(double aa):a(aa),b(0){}
  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;
}
```

4 Graph algorithms

4.1 Fast Dijkstra's algorithm

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

#include <queue>
#include <cstdio>

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

int main() {

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

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

                for (vector<PII>::iterator it = edges[here
                    ].begin(); it != edges[here].end(); it
                    ++) {
                        if (dist[here] + it->first < dist[
                            it->second]) {
                                dist[it->second] = dist[
                                    here] + it->first;
                                dad[it->second] = here;
                                Q.push(make_pair(dist[it->
                                    second], it->second));
                        }
                }
        }

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

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

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

4.2 Strongly connected components

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

4.3 Bellman Ford's algorithm

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

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

using namespace std;

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

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

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

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

  return true;
}
```

4.4 Minimum Spanning Tree: Kruskal

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

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

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

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

T Kruskal(vii Alist[], int n){
  T weight = 0;

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

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

  rep(i, n)
    rep(j, Alist[i].size()) {
      edge e;
      e.u = i, e.v = Alist[i][j].F, e.d = Alist[i][j].S;
      E.push(e);
    }

  while(T.size() < n-1 && !E.empty()) {
    edge cur = E.top(); E.pop();
    int uc = find(C, cur.u), vc = find(C, cur.v);
    if(uc != vc) {
      T.push_back(cur); weight += cur.d;
      if(R[uc] > R[vc])
        C[vc] = uc;
      else if(R[vc] > R[uc])
        C[uc] = vc;
      else
        C[vc] = uc; R[uc]++;
    }
  }
  return weight;
}

int main() {
  int n;
  cin >> n;
  vii Alist[Lim];
  cout << Kruskal(Alist, n) << endl;
}
```

5 Data structures

5.1 Suffix array

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

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

using namespace std;

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

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

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

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

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

#else
// END CUT
int main() {

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

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

5.2 Binary Indexed Tree

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

#define LOGSZ 17

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

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

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

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

5.3 Union-find set(aka DSU)

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

5.4 Lowest common ancestor

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

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

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

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

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

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

    if(p == q)
        return p;

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

    return A[p][0];
}

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

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

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

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

    // precompute L
    DFS(root, 0);


    return 0;
}
```

5.5 Segment tree for range minima query

```
#include "template.h"
/*
    Segment Tree for Range Minima Query, Can be modified
         easily for
    other cases.

    Deal Everything in one based indexing
*/

ll Arr[Lim], Tree[4*Lim];

void buildTree(int Node, int a, int b) {
    if(a == b) {
        Tree[Node]=Arr[a];
    } else if (a < b) {
        int mid=(a+b)>>1, left=Node<<1;
        int right=left|1;
        buildTree(left, a, mid);
        buildTree(right, mid+1, b);
        Tree[Node] = min(Tree[left], Tree[right]);
    }
}

void updateTree(int Node, ll value, int a, int b, int index
    ) {
    if (a > index || b < index) {

    } else if (a == b) {
        Tree[Node] = value;
        Arr[index] = value;
    } else if (a <= index && b >= index) {
        int mid=(a+b)>>1, left=Node<<1;
        int right=left|1;
        updateTree(left, index, value, a, mid);
        updateTree(right, index, value, mid+1, b);
        Tree[Node]=min(Tree[left], Tree[right]);
    }
 }

ll queryTree(int Node, int start, int end, int a, int b) {
    int mid=(a+b)>>1, left=Node<<1;
    int right=left|1;
    ll Ans = Inf;
    if (start <= a && b <= end) {
        return Tree[Node];
    } else {
        if(mid >= start)
            Ans = queryTree(left, start, end, a, mid);

        if(mid < end)
            Ans = min(Ans, queryTree(right, start, end, mid
                 +1, b));

        return Ans;
    }
}
```

5.6 Lazy Propogation for Range update and Query

```
#include "template.h"

/*
  A lazy tree implementation of Range Updation & Range
       Query
*/

ll Arr[Lim], Tree[4*Lim], lazy[4*Lim];

void build_tree(int Node, int a, int b) {
  // Do not forget to clear lazy Array before calling build

  if(a == b) {
    Tree[Node] = Arr[a];
  } else if (a < b) {
    int mid = (a+b)>>1, left=Node<<1, right=left|1;
    build_tree(left, a, mid); build_tree(right, mid+1, b);
    Tree[Node] = Tree[left]+Tree[right];
  }
}

void Propogate(int Node, int a, int b) {
  int left=Node<<1, right=left|1;
  Tree[Node]+=lazy[Node]*(b-a+1);
  if(a != b) {
    lazy[left]+=lazy[Node];
    lazy[right]+=lazy[Node];
  }
  lazy[Node] = 0;
}

void update_tree (int Node, int start, int end, ll value,
     int a, int b) {
  int mid=(a+b)>>1, left=Node<<1, right=left|1;
  if(lazy[Node] != 0)
    Propogate(Node, a, b);
```

```
    if(a > b || a > end || b < start) {
      return;
    } else {
      if(start <= a && b <= end) {
        if (a != b) {
          lazy[left] += value;
          lazy[right] += value;
        }
        Tree[Node] += value * (b - a + 1);
      } else {
        update_tree(left, start, end, value, a, mid);
        update_tree(right, start, end, value, mid+1, b);
        Tree[Node]=Tree[left]+Tree[right];
      }
    }
  }

  ll query(int Node, int start, int end, int a, int b) {
    int mid=(a+b)>>1, left=Node<<1, right=left|1;
    if(lazy[Node] != 0)
      Propogate(Node, a, b);

    if (a > b || a > end || b < start) {
      return 0;
    } else {
      ll Sum1, Sum2;
      if (start <= a && b <= end) {
        return Tree[Node];
      } else {
        Sum1 = query(left, start, end, a, mid);
        Sum2 = query(right, start, end, mid + 1, b);
        return Sum1+Sum2;
      }
    }
  }
```

6 Miscellaneous

6.1 C++ template

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

const long long Mod = 1e9 + 7;
const long long Inf = 1e18;
const long long Lim = 1e5 + 1e3;
const double eps = 1e-10;

typedef long long ll;
typedef vector <int> vi;
typedef vector <ll> vl;
typedef pair <int, int> pii;
typedef pair <ll, ll> pll;
typedef vector <pii> vii;
typedef vector <pll> vll;

#define F first
#define S second
#define uint unsigned int
#define mp make_pair
#define pb push_back
#define pi 2*acos(0.0)
#define rep2(i,b,a) for(ll i = (ll)b, _a = (ll)a; i >= _a;
     i--)
#define rep1(i,a,b) for(ll i = (ll)a, _b = (ll)b; i <= _b;
     i++)
#define rep(i,n) for(ll i = 0, _n = (ll)n ; i < _n ; i++)
#define mem(a,val) memset(a,val,sizeof(a))
#define fast ios_base::sync_with_stdio(false),cin.tie(0),
     cout.tie(0);
```

6.2 C++ input/output

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

using namespace std;

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

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

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

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

6.3 Sieve for Prime Numbers

```
#include "template.h"
/*
    isPrime stores the largest prime number which divides
         the index
    vector primeNum contains all the prime numbers
*/

vi primeNum;
int isPrime[Lim];

void pop_isPrime(int limit) {
    mem(isPrime, 0);
    rep1(i, 2, limit) {
        if (isPrime[i])
            continue;

        if (i <= (int)(sqrt(limit)+10))
            for(ll j = i*i; j <= limit; j += i)
                isPrime[j] = i;

        primeNum.pb(i);
        isPrime[i]=i;
    }
}

int main() {
    fast;
    pop_isPrime(500);
    rep1(i, 1, 500)
        cout << i << ' ' << isPrime[i] << '\n';
}
```

6.4 Longest increasing subsequence

```
// 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;
}
```

6.5 Knuth-Morris-Pratt

```
/*
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;
}
```

6.6 Topological sort (C++)

```
// This function uses performs a non-recursive topological sort.
//
// Running time: O(|V|^2).  If you use adjacency lists (vector<map<int> >),
//               the running time is reduced to O(|E|).
//
//   INPUT:   w[i][j] = 1 if i should come before j, 0 otherwise
//   OUTPUT:  a permutation of 0,...,n-1 (stored in a vector)
//            which represents an ordering of the nodes which
//            is consistent with w
//
// If no ordering is possible, false is returned.

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

using namespace std;

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

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

bool TopologicalSort (const VVI &w, VI &order){
  int n = w.size();
  VI parents (n);
  queue<int> q;
  order.clear();

  for (int i = 0; i < n; i++){
    for (int j = 0; j < n; j++)
      if (w[j][i]) parents[i]++;
      if (parents[i] == 0) q.push (i);
  }

  while (q.size() > 0){
    int i = q.front();
    q.pop();
    order.push_back (i);
    for (int j = 0; j < n; j++) if (w[i][j]){
      parents[j]--;
      if (parents[j] == 0) q.push (j);
    }
  }

  return (order.size() == n);
}
```

6.7 Fast exponentiation

```
/*
Uses powers of two to exponentiate numbers and matrices. Calculates
n^k in O(log(k)) time when n is a number. If A is an n x n matrix,
calculates A^k in O(n^3*log(k)) time.
*/

#include <iostream>
#include <vector>

using namespace std;

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

T power(T x, int k) {
  T ret = 1;

  while(k) {
    if(k & 1) ret *= x;
    k >>= 1; x *= x;
  }
  return ret;
}

VVT multiply(VVT& A, VVT& B) {
  int n = A.size(), m = A[0].size(), k = B[0].size();
  VVT C(n, VT(k, 0));

  for(int i = 0; i < n; i++)
    for(int j = 0; j < k; j++)
      for(int l = 0; l < m; l++)
        C[i][j] += A[i][l] * B[l][j];

  return C;
}

VVT power(VVT& A, int k) {
  int n = A.size();
  VVT ret(n, VT(n)), B = A;
  for(int i = 0; i < n; i++) ret[i][i]=1;

  while(k) {
    if(k & 1) ret = multiply(ret, B);
    k >>= 1; B = multiply(B, B);
  }
  return ret;
}

int main()
{
  /* Expected Output:
     2.37^48 = 9.72569e+17

     376 264 285 220 265
     550 376 529 285 484
     484 265 376 264 285
     285 220 265 156 264
     529 285 484 265 376 */
  double n = 2.37;
  int k = 48;

  cout << n << "^" << k << " = " << power(n, k) << endl;

  double At[5][5] = {
    { 0, 0, 1, 0, 0 },
    { 1, 0, 0, 1, 0 },
    { 0, 0, 0, 0, 1 },
    { 1, 0, 0, 0, 0 },
    { 0, 1, 0, 0, 0 } };

  vector <vector <double> > A(5, vector <double>(5));
  for(int i = 0; i < 5; i++)
    for(int j = 0; j < 5; j++)
      A[i][j] = At[i][j];

  vector <vector <double> > Ap = power(A, k);

  cout << endl;
  for(int i = 0; i < 5; i++) {
    for(int j = 0; j < 5; j++)
      cout << Ap[i][j] << " ";
    cout << endl;
  }
}
```

6.8 Longest common subsequence

```
/*
Calculates the length of the longest common subsequence of two vectors.
Backtracks to find a single subsequence or all subsequences. Runs in
O(m*n) time except for finding all longest common subsequences, which
may be slow depending on how many there are.
*/

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

using namespace std;

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

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

void backtrack(VVI& dp, VT& res, VT& A, VT& B, int i, int j)
{
  if(!i || !j) return;
  if(A[i-1] == B[j-1]) { res.push_back(A[i-1]); backtrack(dp, res, A, B, i-1, j-1); }
  else
  {
    if(dp[i][j-1] >= dp[i-1][j]) backtrack(dp, res, A, B, i, j-1);
    else backtrack(dp, res, A, B, i-1, j);
  }
}

void backtrackall(VVI& dp, set<VT>& res, VT& A, VT& B, int i, int j)
{
  if(!i || !j) { res.insert(VI()); return; }
  if(A[i-1] == B[j-1])
  {
    set<VT> tempres;
    backtrackall(dp, tempres, A, B, i-1, j-1);
    for(set<VT>::iterator it=tempres.begin(); it!=tempres.end(); it++)
    {
      VT temp = *it;
      temp.push_back(A[i-1]);
      res.insert(temp);
    }
  }
  else
  {
    if(dp[i][j-1] >= dp[i-1][j]) backtrackall(dp, res, A, B, i, j-1);
    if(dp[i][j-1] <= dp[i-1][j]) backtrackall(dp, res, A, B, i-1, j);
  }
}

VT LCS(VT& A, VT& B)
{
  VVI dp;
  int n = A.size(), m = B.size();
  dp.resize(n+1);
  for(int i=0; i<=n; i++) dp[i].resize(m+1, 0);

  for(int i=1; i<=n; i++)
    for(int j=1; j<=m; j++)
    {
      if(A[i-1] == B[j-1]) dp[i][j] = dp[i-1][j-1]+1;
      else dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
    }

  VT res;
  backtrack(dp, res, A, B, n, m);
```

```
  reverse(res.begin(), res.end());
  return res;
}

set<VT> LCSall(VT& A, VT& B)
{
  VVI dp;
  int n = A.size(), m = B.size();
  dp.resize(n+1);
  for(int i=0; i<=n; i++) dp[i].resize(m+1, 0);
  for(int i=1; i<=n; i++)
    for(int j=1; j<=m; j++)
    {
      if(A[i-1] == B[j-1]) dp[i][j] = dp[i-1][j-1]+1;
      else dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
    }
  set<VT> res;
  backtrackall(dp, res, A, B, n, m);
  return res;
}

int main()
{
  int a[] = { 0, 5, 5, 2, 1, 4, 2, 3 }, b[] = { 5, 2, 4, 3,
      2, 1, 2, 1, 3 };
  VI A = VI(a, a+8), B = VI(b, b+9);
  VI C = LCS(A, B);

  for(int i=0; i<C.size(); i++) cout << C[i] << " ";
  cout << endl << endl;

  set <VI> D = LCSall(A, B);
  for(set<VI>::iterator it = D.begin(); it != D.end(); it
      ++)
  {
    for(int i=0; i<(*it).size(); i++) cout << (*it)[i] << "
        ";
    cout << endl;
  }
}
```

6.9 Miller-Rabin Primality Test (C)

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

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

typedef long long LL;

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