LalCheetah - IIT Kanpur ICPC Team Notebook (2016-17)

Contents

1 Combinatorial optimization

1.1 Dinic's Algo for Sparse max-flow

```
// 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

```
// 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 "template.h"

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
```

1.3 Min cost Bipartite Matching

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

#include "template.h"

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

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

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

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

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

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

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

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

    int j = 0;
    while (true) {

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

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

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

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

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

    mated++;
  }

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

  return value;
}
```

1.4 Min cost Max Flow

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

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

using namespace std;

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

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

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

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

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

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

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

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

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

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

// BEGIN CUT
// The following code solves UVA problem #10594: Data Flow

int main() {
  int N, M;

  while (scanf("%d%d", &N, &M) == 2) {
    VVL v(M, VL(3));
    for (int i = 0; i < M; i++)
      scanf("%Ld%Ld%Ld", &v[i][0], &v[i][1], &v[i][2]);
    L D, K;
    scanf("%Ld%Ld", &D, &K);

    MinCostMaxFlow mcmf(N+1);
    for (int i = 0; i < M; i++) {
      mcmf.AddEdge(int(v[i][0]), int(v[i][1]), K, v[i][2]);
      mcmf.AddEdge(int(v[i][1]), int(v[i][0]), K, v[i][2]);
    }
    mcmf.AddEdge(0, 1, D, 0);

    pair<L, L> res = mcmf.GetMaxFlow(0, N);

    if (res.first == D) {
      printf("%Ld\n", res.second);
    } else {
      printf("Impossible.\n");
    }
  }

  return 0;
}

// END CUT
```

2 Geometry

2.1 Convex hull

```cpp
// 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;
}
```

2.3 Delaunay Triangulation

```
// Slow but simple Delaunay triangulation. Does
    not handle
// degenerate cases (from O'Rourke, Computational
    Geometry in C)
//
// Generates a Tiangulation DT(P) such that no
    point in P in
// inside any Triangle in DT(P)
//
// Running time: O(n^4)
//
// INPUT:    x[] = x-coordinates
//           y[] = y-coordinates
//
// OUTPUT:   triples = a vector containing m
    triples of indices
//                   corresponding to triangle
    vertices
#include "template.h"

typedef double T;

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

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

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

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

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

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

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

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;
}
```

3.2 Euclid and Fermat's Theorem

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  // expected: 23 105
  //           11 12
  int v1[3]={3,5,7}, v2[3]={2,3,2};
  pii ret = chinese_remainder_theorem(vi(v1, v1
      +3), vi(v2, v2+3));
  cout << ret.first << " " << ret.second << endl;
  int v3[2]={4,6}, v4[2]={3,5};
  ret = chinese_remainder_theorem(vi(v3, v3+2),
      vi(v4, v4+2));
  cout << ret.first << " " << ret.second << endl;

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

3.3 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';
}
```

3.4 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;
  }
}
```

3.5 Simplex Algorithm, Linear Programming

```
// Two-phase simplex algorithm for solving linear
     programs of
// the form (c^T is c Transpose)
//
//     maximize      c^T x
//     subject to    Ax <= b
//                   x >= 0
//
// INPUT: A -- an m x n matrix
//        b -- an m-dimensional vector
//        c -- an n-dimensional vector
//        x -- a vector where the optimal
     solution will be stored
//
// OUTPUT: value of the optimal solution (
     infinity if unbounded
//         above, nan if infeasible)
//
// To use this code, create an LPSolver object
     with A, b, and c as
// arguments.  Then, call Solve(x).

#include "template.h"

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

const DOUBLE EPS = 1e-9;

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

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

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

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

```
    }
  }

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

int main() {

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

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

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

  cerr << "VALUE: " << value << endl; // VALUE:
      1.29032
  cerr << "SOLUTION:"; // SOLUTION: 1.74194
      0.451613 1
  for (size_t i = 0; i < x.size(); i++) cerr << "
      " << x[i];
  cerr << endl;
  return 0;
}
```

3.6 Constraint Satisfaction Problem

```
// Constraint satisfaction problems
// TODO doesn't compiles

#include "template.h"

#define DONE   -1
#define FAILED -2

typedef vector<int> vi;
typedef vector<vi> vvi;
typedef vector<vvi> vvvi;

typedef set<int> SI;

// Lists of assigned/unassigned variables.
vi assigned_vars;
SI unassigned_vars;

// For each variable, a list of reductions (each
    of which a list of eliminated
// variables)
vvvi reductions;

// For each variable, a list of the variables
    whose domains it reduced in
// forward-checking.
vvi forward_mods;

// need to implement ----------------------------
int Value(int var);

void SetValue(int var, int value);
void ClearValue(int var);

int DomainSize(int var);
void ResetDomain(int var);
void AddValue(int var, int value);
void RemoveValue(int var, int value);

int NextVar() {
  if ( unassigned_vars.empty() ) return DONE;

  // could also do most constrained...
  int var = *unassigned_vars.begin();
  return var;
}

int Initialize() {
  // setup here
  return NextVar();
}
// ------------------------- end -- need to
    implement

void UpdateCurrentDomain(int var) {
  ResetDomain(var);
  for (int i = 0; i < reductions[var].size(); i
      ++) {
    vector<int>& red = reductions[var][i];
    for (int j = 0; j < red.size(); j++) {
      RemoveValue(var, red[j]);
    }
  }
}

void UndoReductions(int var) {
  for (int i = 0; i < forward_mods[var].size(); i
      ++) {
    int other_var = forward_mods[var][i];
    vi& red = reductions[other_var].back();
    for (int j = 0; j < red.size(); j++) {
      AddValue(other_var, red[j]);
    }
    reductions[other_var].pop_back();
  }
  forward_mods[var].clear();
}

bool ForwardCheck(int var, int other_var) {
  vector<int> red;

  foreach value in current_domain(other_var) {
    SetValue(other_var, value);
    if ( !Consistent(var, other_var) ) {
      red.push_back(value);
      RemoveValue(other_var, value);
    }
    ClearValue(other_var);
  }
  if ( !red.empty() ) {
    reductions[other_var].push_back(red);
    forward_mods[var].push_back(other_var);
  }

  return DomainSize(other_var) != 0;
}

pair<int, bool> Unlabel(int var) {
  assigned_vars.pop_back();
  unassigned_vars.insert(var);

  UndoReductions(var);
  UpdateCurrentDomain(var);

  if ( assigned_vars.empty() ) return make_pair(
      FAILED, true);

  int prev_var = assigned_vars.back();
  RemoveValue(prev_var, Value(prev_var));
  ClearValue(prev_var);
  if ( DomainSize(prev_var) == 0 ) {
    return make_pair(prev_var, false);
  } else {
    return make_pair(prev_var, true);
  }
}


pair<int, bool> Label(int var) {
  unassigned_vars.erase(var);
  assigned_vars.push_back(var);

  bool consistent;
  foreach value in current_domain(var) {
    SetValue(var, value);
    consistent = true;
    for (int j=0; j<unassigned_vars.size(); j++)
        {
      int other_var = unassigned_vars[j];
      if ( !ForwardCheck(var, other_var) ) {
        RemoveValue(var, value);
        consistent = false;
        UndoReductions(var);
        ClearValue(var);
        break;
      }
    }
    if ( consistent ) return (NextVar(), true);
  }
  return make_pair(var, false);
}


void BacktrackSearch(int num_var) {
  // (next variable to mess with, whether current
      state is consistent)
  pair<int, bool> var_consistent = make_pair(
      Initialize(), true);
  while ( true ) {
    if ( var_consistent.second ) var_consistent =
```

```
      Label(var_consistent.first);
    else var_consistent = Unlabel(var_consistent.
      first);

    if ( var_consistent.first == DONE ) return;
      // solution found
    if ( var_consistent.first == FAILED ) return;
      // no solution
  }
}
```

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 "template.h"
const int INF = 2000000000;

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 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.

// TODO Optimization required

#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);
}
```

4.3 Union-find set(aka DSU)

```
#include "template.h"

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;
}
```

4.4 Strongly connected components

```
#include "template.h"
#define MAXE 1000000
#define MAXV 100000
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];    // Stack, stk[0] stores size
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]);}
}

int main() {return 0;}
```

4.5 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;
}
```

4.6 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;
}
```

4.7 Eulerian Path Algo

```
#include "template.h"

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

struct Edge {
  int next_vertex;
  iter reverse_edge;

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

const int max_vertices = 10;
// int num_vertices = 6;
list<Edge> adj[max_vertices];   // adjacency list

vector<int> path;

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

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

int main() {
  int total=0, start_vertex = 0;
  rep(i, max_vertices)
    if(adj[i].size()&1)
        // if the size is odd then increment '
        total'
      total++, start_vertex=i;          // put
          the starting vertex as an odd degree
          vertex
  if(total==0||total==2) {
      // necessary and sufficient condition to
      check the existence of an EC
    find_path(start_vertex);
    rep(i, path.size()) cout << path[i] << " ";
  }
  else
    cout << "No Eulerian Circuit\n";
  return 0;
}
```

4.8 FloydWarshall's Algorithm

```
#include "template.h"

typedef double T;
```

```
typedef vector<T> vt;
typedef vector<vt> vvt;
typedef vector<vi> vvi;

// This function runs the Floyd-Warshall
    algorithm for all-pairs
// shortest paths.  Also handles negative edge
    weights.  Returns true
// if a negative weight cycle is found.
//
// Running time: O(|V|^3)
//
//   INPUT:  Alist[i][j] = Alisteight of edge
    from i to j
//   OUTPUT: Alist[i][j] = shortest path from i
    to j
//           prev[i][j] = node before j on the
    best path starting at i

bool FloydWarshall (vvt &Alist, vvi &prev) {
  int n = Alist.size();
  prev = vvi(n, vi(n, -1));

  for (int k = 0; k < n; k++){
    for (int i = 0; i < n; i++){
      for (int j = 0; j < n; j++){
        if (Alist[i][j] > Alist[i][k] + Alist[k][
            j]){
          Alist[i][j] = Alist[i][k] + Alist[k][j
              ];
          prev[i][j] = k;
        }
      }
    }
  }

  // check for negative weight cycles
  for(int i=0;i<n;i++)
    if (Alist[i][i] < 0) return false;
  return true;
}
```

4.9 Prim's Algo in $\mathcal{O}(n^2)$ time

```
#include "template.h"

// This function runs Prim's algorithm for
    constructing minimum
// weight spanning trees.
//
// Running time: O(|V|^2)
//
//   INPUT: w[i][j] = cost of edge from i to j
//
//   NOTE: Make sure that w[i][j] is nonnegative
    and
//   symmetric.  Missing edges should be given -1
    weight.
//
//   OUTPUT:  edges = list of pair<int,int> in
    minimum
//            spanning tree return total weight
    of tree

typedef double T;
typedef vector<T> vt;
typedef vector<vt> vvt;
typedef vector<vi> vvi;

T Prim (const vvt &w, vii &edges){
  int n = w.size();
  vi found (n);
  vi prev (n, -1);
  vt dist (n, 1000000000);
  int here = 0;
  dist[here] = 0;

  while (here != -1){
    found[here] = true;
    int best = -1;
    for (int k = 0; k < n; k++) if (!found[k]){
      if (w[here][k] != -1 && dist[k] > w[here][k
          ]){
        dist[k] = w[here][k];
        prev[k] = here;
      }
      if (best == -1 || dist[k] < dist[best])
          best = k;
    }
    here = best;
  }

  T tot_weight = 0;
  for (int i = 0; i < n; i++) if (prev[i] != -1){
    edges.push_back (make_pair (prev[i], i));
    tot_weight += w[prev[i]][i];
  }
  return tot_weight;
}
```

4.10 MST for a directed graph

```
/*  Edmond's Algorithm for finding an aborescence
 *  Produces an aborescence (directed analog of a
      minimum
 *  spaning tree) of least weight in O(m*n) time
 */

#include "template.h"
#define sz size()
#define D(x) if(1) cout << __LINE__ <<" "<< #x "
    = " << (x) << endl;
#define D2(x,y) if(1) cout << __LINE__ <<" "<< #x
      " = " << (x) \
  <<", " << #y " = " << (y) << endl;
typedef vector<vi> vvi;
#define SZ(x) ((x).size())
int N;

vi match;
vi vis;

void couple(int n, int m) { match[n]=m; match[m]=
    n; }

// returns true if something interesting has been
    found, thus a
// augmenting path or a blossom (if blossom is
    non-empty).
// the dfs returns true from the moment the stem
    of the flower is
// reached and thus the base of the blossom is an
    unmatched node.
// blossom should be empty when dfs is called and
// contains the nodes of the blossom when a
    blossom is found.
bool dfs(int n, vvi &conn, vi &blossom) {
  vis[n]=0;
  rep(i, N) {
    if(conn[n][i]) {
      if(vis[i]==-1) {
        vis[i]=1;
        if(match[i]==-1 || dfs(match[i], conn,
            blossom)) { couple(n,i); return true;
             }
      }
      if(vis[i]==0 || SZ(blossom)) {  // found
          flower
        blossom.pb(i); blossom.pb(n);
        if(n==blossom[0]) { match[n]=-1; return
            true; }
        return false;
      }
    }
  }
  return false;
}

// search for an augmenting path.
// if a blossom is found build a new graph (
    newconn) where the
// (free) blossom is shrunken to a single node
    and recurse.
// if a augmenting path is found it has already
    been augmented
// except if the augmented path ended on the
    shrunken blossom.
// in this case the matching should be updated
    along the appropriate
// direction of the blossom.
bool augment(vvi &conn) {
  rep(m, N) {
    if(match[m]==-1) {
      vi blossom;
      vis=vi(N,-1);
      if(!dfs(m, conn, blossom)) continue;
      if(SZ(blossom)==0) return true; //
          augmenting path found

      // blossom is found so build shrunken graph
      int base=blossom[0], S=SZ(blossom);
      vvi newconn=conn;
      rep1(i, 1, S-1) rep(j, N) newconn[base][j]=
          newconn[j][base]|=conn[blossom[i]][j];
      rep1(i, 1, S-1) rep(j, N) newconn[blossom[i
          ]][j]=newconn[j][blossom[i]]=0;
      newconn[base][base]=0; // is now the new
          graph
      if(!augment(newconn)) return false;
      int n=match[base];
      D(base);
      // if n!=-1 the augmenting path ended on
          this blossom

      if(n!=-1) rep(i, S) if(conn[blossom[i]][n])
           {
        couple(blossom[i], n);
        if(i&1) for(int j=i+1; j<S; j+=2) couple(
            blossom[j],blossom[j+1]);
        else for(int j=0; j<i; j+=2) couple(
            blossom[j],blossom[j+1]);
        break;
      }
      return true;
    }
  }
  return false;
}

int edmonds(vvi &conn) {  //conn is the Adjacency
```

```
    matrix
  int res=0;
  match=vi(N,-1);
  while(augment(conn)) res++;
  return res;
}

int main() {
  vvi conn(10, vi(10, 0));  // Adjacency matrix
#define addEdge(x,y) conn[x][y]=conn[y][x] = 1;
  addEdge(1,2);
  addEdge(2,3);
  addEdge(2,5);
  addEdge(5,3);
  addEdge(3,4);
  addEdge(5,6);
  N = conn.size();
  D(edmonds(conn));
  return 0;
}
```

4.11 Maximum Matching in a Bipartite graphss

```
// Hopcraft-Karp Algo for finding Maximum
    Biparitie
// Matching using Augmenting paths

#include "template.h"

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

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

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

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

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

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

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

int main() {
  init(2, 2);
  addEdge(0, 0); addEdge(0, 1); addEdge(1, 1);
  cout << (2 == maxMatching()) << endl;
}
```

4.12 Articulation Pt/Bridge in a Graph

```
#include "template.h"

// Array u acts as visited bool array, d stores
    DFN No., low
// stores lowest DFN no reachable, par stores
    parent node's DFN no.

int gl = 0;
const int N = 10010;
int u[N],d[N],low[N],par[N];
vi G[N];
void dfs1(int node,int dep){ //find dfs_num and
    dfs_low
  u[node]=1;
  d[node]=dep;low[node]=dep;
  for(int i = 0; i < G[node].size(); i++){
    int it = G[node][i];
    if(!u[it]){
      par[it]=node;
      dfs1(it,dep+1);
      low[node]=min(low[node],low[it]);
      /*if(low[it] > d[node] ){
          node-it is cut edge/bridge
      }*/
      /*
      if(low[it] >= d[node] && (par[node]!=-1 ||
          sz(G[node]) > 2)){
          node is cut vertex/articulation point
      }
      */
    }else if(par[node]!=it) low[node]=min(low[
        node],low[it]);
    else par[node]=-1;
  }
}
int main(){
  return 0;
}
```

4.13 Closest Pair of points in a 2D Plane

```
#include "template.h"
const int MAXN = 4;
struct pt {
  int x, y, id;
};
// comparison on basis of x coordinate
inline bool cmp_x (const pt & a, const pt & b) {
  return a.x < b.x || a.x == b.x && a.y < b.y;
}
// comparison on basis of y coordinate

inline bool cmp_y (const pt & a, const pt & b) {
  return a.y < b.y;
}
// a for storing points
pt a[MAXN];
double mindist;
int ansa, ansb;

inline void upd_ans (const pt & a, const pt & b)
    {
  double dist = sqrt ((a.x-b.x)*(a.x-b.x) + (a.y-
      b.y)*(a.y-b.y) + .0);
  if (dist < mindist)
    mindist = dist,  ansa = a.id,  ansb = b.id;
}
// the basic recursive function
void rec (int l, int r) {
  if (r - l <= 3) {
    for (int i=l; i<=r; ++i)
      for (int j=i+1; j<=r; ++j)
        upd_ans (a[i], a[j]);
    sort (a+l, a+r+1, &cmp_y);
    return;
  }

  int m = (l + r) >> 1;
  int midx = a[m].x;
  rec (l, m),  rec (m+1, r);
  static pt t[MAXN];
  merge (a+l, a+m+1, a+m+1, a+r+1, t, &cmp_y);
  copy (t, t+r-l+1, a+l);

  int tsz = 0;
  for (int i=l; i<=r; ++i)
    if (abs (a[i].x - midx) < mindist) {
      for (int j=tsz-1; j>=0 && a[i].y - t[j].y <
          mindist; --j)
        upd_ans (a[i], t[j]);
      t[tsz++] = a[i];
    }
}

int main(){
  int n= 4;
```

```
    mindist = 1E20; //final answer is stored in
        mindist
    sort (a, a+n, &cmp_x);
    rec (0, n-1);
    cout<<mindist<<"\n";
    return 0;
  }
```

5 Data structures

5.1 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;
}
```

5.2 BIT for 2-D plane questions

```
/* Bit used as 2-D structure for a handling
    update/range
queries in a matrix in $\log^2{n}$ time */

#include "template.h"
int bit[M][M], n;
int sum( int x, int y ){
    int ret = 0;
    while( x > 0 ){
        int yy = y; while( yy > 0 ) ret += bit[x
            ][yy], yy -= yy & -yy;
        x -= (x & -x);
    }
    return ret ;
}
void update(int x , int y , int val){
    int y1;
    while (x <= n){
        y1 = y;
        while (y1 <= n){ bit[x][y1] += val; y1 +=
             (y1 & -y1); }
        x += (x & -x);
    }
}
```

5.3 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;
}
```

5.4 Segment tree class for range minima query

```
#include "template.h"

template<typename T>
struct segTree {
  T Tree[4*Lim];
  T combine(int l, int r) {
    T ret;
    ret=min(l, r);  // TODO
    return ret;
  }
  void buildST(int Node, int a, int b) {
    if (a==b)
      Tree[Node]=0; // TODO
    else if (a<b) {
      int left=Node<<1, right=(Node<<1)|1, mid=(a
          +b)>>1;
      buildST(left, a, mid); buildST(right, mid
          +1, b);
      Tree[Node]=combine(Tree[left], Tree[right])
          ;
    }
  }
  void buildST(int Node, int a, int b, vi Arr) {
    if (a==b)
      Tree[Node]=Arr[a];
    else if (a<b) {
      int left=Node<<1, mid=(a+b)>>1, right=(Node
          <<1)|1;
      buildST(left, a, mid, Arr); buildST(right,
          mid+1, b, Arr);
      Tree[Node]=combine(Tree[left], Tree[right])
          ;
    }
  }
  T query(int Node, int a, int b, int S, int E) {
```

```
    if (E < a || b < S) return 0;   // TODO
    else if (a==b) return Tree[Node];
    int left=Node<<1, mid=(a+b)>>1, right=(Node
        <<1)|1;
    return combine(query(left, a, mid, S, E),
        query(right, mid+1, b, S, E));
  }
  void update(int Node, int a, int b, int val,
      int I1, int I2) {
    if (I2 < a || b < I1) return;
    if (I1 <=a && b <= I2) return void(Tree[Node
        ]=val);   // TODO
    int left=Node<<1, mid=(a+b)>>1, right=(Node
        <<1)|1;
    update(left, a, mid, val, I1, I2), update(
        right, mid+1, b, val, I1, I2);
    Tree[Node]=combine(Tree[left], Tree[right]);
  }
};

int main() {return 0;}
```

5.5 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;
    }
  }
}
```

5.6 Range minima query in O(1) tiime using lookup matrix

```
/* matrix structure for finding the range minima
    in O(1) time using O(n)  log(n)) space */
#include "template.h"
#define better(a,b) A[a]<A[b]?(a):(b)
int A[100100], H[1100][1100];   //A is the Array
    and H is the lookup matrix
int make_dp(int n) { // N log N
    rep(i,n) H[i][0]=i;
    for(int l=0,k; (k=1<<l) < n; l++) for(int i
        =0;i+k<n;i++)
    H[i][l+1] = better(H[i][l], H[i+k][l]);
}
int query_dp(int a, int b) {
    int l = __lg(b-a);
    return better(H[a][l], H[b-(1<<l)+1][l]);
}
```

6 String Manipulation

6.1 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;
}
```

6.2 Suffix array

```cpp
// 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
```

6.3 Aho Corasick Structure for string matching

```cpp
#include "template.h"

#define NC 26        // No of characters
#define NP 10005
#define M 100005
#define MM 500005    // Max no of states

// b stores the strings in dictionary, g stores
    the trie using states,
// a is the query string, lenb stores length of
    strings in b
// output stores the index of word which end at
    the corresponding state
// f stores the blue edge (largest suffix of
    current word), pre is useless!
// marked represent that this word occurs in
    string a

char a[M];
char b[NP][105];
int nb, cnt[NP], lenb[NP], alen;
int g[MM][NC], ng, f[MM], marked[MM];
int output[MM], pre[MM];

#define init(x) {rep(_i,NC)g[x][_i] = -1; f[x]=
    marked[x]=0; output[x]=pre[x]=-1; }

void match() {
  ng = 0;
  init( 0 );
  // part 1 - building trie
  rep(i,nb) {
    cnt[i] = 0;
    int state = 0, j = 0;
    while(g[state][b[i][j]] != -1 && j < lenb[i])
        state = g[state][b[i][j]], j++;
    while( j < lenb[i] ) {
      g[state][ b[i][j] ] = ++ng;
      state = ng;
      init( ng );
      ++j;
    }
    // if( ng >= MM ) { cerr <<"i am dying"<<endl
        ; while(1);} // suicide
    output[ state ] = i;
  }
  // part 2 - building failure function
  queue< int > q;
  rep(i,NC) if( g[0][i] != -1 ) q.push( g[0][i] )
      ;
  while( !q.empty() ) {
    int r = q.front(); q.pop();
    rep(i,NC) if( g[r][i] != -1 ) {
      int s = g[r][i];
      q.push( s );
      int state = f[r];
      while( g[state][i] == -1 && state ) state =
          f[state];
      f[s] = g[state][i] == -1 ? 0 : g[state][i];
    }
  }
  // final smash
  int state = 0;
  rep(i,alen) {
    while( g[state][a[i]] == -1 ) {
      state = f[state];
      if( !state ) break;
    }
    state = g[state][a[i]] == -1 ? 0 : g[state][a
        [i]];
    if( state && output[ state ] != -1 ) marked[
        state ] ++;
```

```
    }
    // counting
    rep(i,ng+1) if( i && marked[i] ) {
      int s = i;
      while( s != 0 ) cnt[ output[s] ] += marked[i
          ], s = f[s];
    }
  }
```

6.4 Tries Structure for storing strings

```
#include "template.h"
typedef struct Trie{
  int words, prefixes;  //only proper prefixes(
      words not included)
  // bool isleaf; //for only checking words not
      counting prefix or words
  struct Trie * edges[26];
  Trie(){
    words = 0;  prefixes = 0;
    rep(i,26)
      edges[i] = NULL;
  }
} Trie;
Trie * root;
void addword(Trie * node, string a){
  rep(i,a.size()){
    if(node->edges[a[i] - 'a'] == NULL)
      node->edges[a[i] - 'a'] = new Trie();
    node = node->edges[a[i] - 'a'];
    node->prefixes++;
  }
  // node->isleaf = true;
  node->prefixes--;
  node->words++;
}

int count_words(Trie * node, string a){
  rep(i,a.size()){
    if(node->edges[a[i] - 'a'] == NULL)
      return 0;
    node = node->edges[a[i] - 'a'];
  }
  return node->words;
}

int count_prefixes(Trie * node, string a){
  rep(i,a.size()){
    if(node->edges[a[i] - 'a'] == NULL)
      return 0;
    node = node->edges[a[i] - 'a'];
  }
  return node->prefixes;
}

// bool find(Trie * node, string a){
//  rep(i,a.size()){
//    if(node->edges[a[i] - 'a'] == NULL)
//      return false;
//    node = node->edges[a[i] - 'a'];
//  }
//  return node->isleaf;
// }

int main(){
  root = new Trie();
  rep(i,26)
    if(root->edges[i] != NULL)
      cout<<(char)('a' + i);
  return 0;
}
```

6.5 Treaps Data structure implementation

```
#include "template.h"
const int N = 100 * 1000;
struct node { int value, weight, ch[2], size; } T
    [ N+10 ] ; int nodes;
#define Child(x,c) T[x].ch[c]
#define Value(x) T[x].value
#define Weight(x) T[x].weight
#define Size(x) T[x].size
#define Left Child(x,0)
#define Right Child(x,1)
int update(int x) { if(!x)return 0; Size(x) = 1+
    Size(Left)+Size(Right); return x; }
int newnode(int value, int prio) {
  T[++nodes]=(node){value,prio,0,0};
  return update(nodes);
}
void split(int x, int by, int &L, int &R)
{
  if(!x) { L=R=0; }
  else if(Value(x) < Value(by)) { split(Right,by,
      Right,R); update(L=x); }
  else { split(Left,by,L,Left); update(R=x); }
}
int merge(int L, int R)
{
  if(!L) return R; if(!R) return L;
  if(Weight(L)<Weight(R)) { Child(L,1) = merge(
      Child(L,1), R); return update(L);}
  else { Child(R,0) = merge( L, Child(R, 0));
      return update(R); }
}
int insert(int x, int n)
{
  if(!x) { return update(n); }
  if(Weight(n)<=Weight(x)) {split(x,n,Child(n,0),
      Child(n,1)); return update(n);}
  else if(Value(n) < Value(x)) Left=insert(Left,n
      ); else Right=insert(Right,n);
  return update(x);
}
int del(int x, int value)
{
  if(!x) return 0;
  if(value == Value(x)) { int q = merge(Left,
      Right); return update(q); }
  if(value < Value(x)) Left = del(Left,value);
      else Right = del(Right, value);
  return update(x);
}
int find_GE(int x, int value) {
  int ret=0;
  while(x) { if(Value(x)==value)return x;
    if(Value(x)>value) ret=x, x=Left; else x=
        Right; }
  return ret;
}
int find(int x, int value) {
  for(; x; x=Child(x,Value(x)<value)) if(Value(x)
      ==value)return x;
    return 0;
}
int findmin(int x) { for(;Left;x=Left); return x;
    }
int findmax(int x) { for(;Right;x=Right); return
    x; }
int findkth(int x, int k) {
  while(x) {
    if(k<=Size(Left)) x=Left;
    else if(k==Size(Left)+1)return x;
    else { k-=Size(Left)+1; x=Right; }
  }
}
int queryrangekth(int &x, int a1, int a2, int k)
    {
  a1 = find(x, a1); a2 = find(x, a2);
  assert(a1 && a2);
  int a,b,c; split(x,a1,a,b); split(b,a2,b,c);
  int ret = findkth(b,k);
  x = merge(a, merge(b,c));
  return Value(ret);
}
int main(){
  return 0;
}
```

6.6 Z's Algorithm, KMP's Bro

```
/* Z algorithm for matching substrings. KMP's
    Brother */

vector<int> zfunction(char *s) {
  int N = strlen(s), a=0, b=0;
  vector<int> z(N, N);
  for (int i = 1; i < N; i++) {
    int k = i<b ? min(b-i, z[i-a]) : 0;
    while (i+k < N && s[i+k]==s[k]) ++k;
    z[i] = k;
    if (i+k > b) { a=i; b=i+k; }
  }
  return z;
}

Definition:
z[i] = max {k: s[i..i+k-1]=s[0..k-1]}
```

7 Miscellaneous

7.1 C++ template

```
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace __gnu_pbds;
using namespace std;
template <typename T>
using ordered_set = tree<T, null_type, less<T>,
    rb_tree_tag,
        tree_order_statistics_node_update>;

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 <vi> vvi;
typedef vector <ll> vl;
typedef pair <int, int> pii;
typedef pair <ll, ll> pll;

#define F first
#define S second
#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 all(v) v.begin(), v.end()
#define fast ios_base::sync_with_stdio(false),cin
    .tie(0),cout.tie(0);

int main () {
        ordered_set<int> X;
        X.insert(1); X.insert(2); X.insert(4); X.
            insert(8); X.insert(16);
        cout << *X.find_by_order(5) << '\n';
        cout << X.order_of_key(4) << '\n';
        return 0;
}
```

7.2 C++ input/output

```
#include "template.h"

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

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

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

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

7.3 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;
}
```

7.4 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;
  }
}
```

7.5 Gauss Jordan

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

#include "template.h"
using namespace std;

typedef double T;
typedef vector<T> vt;
typedef vector<vt> vvt;

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

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

    T c = 1.0 / a[pk][pk];
    det *= a[pk][pk];
    a[pk][pk] = 1.0;
    for (int p = 0; p < n; p++) a[pk][p] *= c;
    for (int p = 0; p < m; p++) b[pk][p] *= c;
    for (int p = 0; p < n; p++) if (p != pk) {
      c = a[p][pk];
      a[p][pk] = 0;
      for (int q = 0; q < n; q++) a[p][q] -= a[pk
          ][q] * c;
      for (int q = 0; q < m; q++) b[p][q] -= b[pk
          ][q] * c;
    }
  }
  for (int p = n-1; p >= 0; p--) if (irow[p] !=
      icol[p]) {
    for (int k = 0; k < n; k++) swap(a[k][irow[p
        ]], a[k][icol[p]]);
  }

  return det;
}

int main() {
  vvt a(100), b(100);
  double det = GaussJordan(a, b);
}
```

7.6 Miller-Rabin Primality Test

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

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;
}
```

7.7 Binary Search

```
#include "template.h"
bool works(int m){
  // if Array[m] is the value to be searched,
      return true
  // else return false
}

// The property is increasing
int BinarySearch(int l, int h){
  int m;
  while(l <= h){ m = (l+h) / 2; if(works(m))  l=m
      +1; else h=m-1; }
  return l-1;
}
// The property is decreasing
// while(l <= h){ m = (l+h) / 2; if(works(m))  h=
    m-1; else l=m+1; }
// return h+1;
```

7.8 Playing with dates

```
// Routines for performing computations on dates.
      In these routines,
// months are expressed as integers from 1 to 12,
    days are expressed
// as integers from 1 to 31, and years are
    expressed as 4-digit
// integers.
#include "template.h"
string dayOfWeek[] = {"Mon", "Tue", "Wed", "Thu",
     "Fri", "Sat", "Sun"};

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

```
// converts integer (Julian day number) to
    Gregorian date: month/day/year
void intToDate (int jd, int &m, int &d, int &y){
  int x, n, i, j;
  x = jd + 68569;
  n = 4 * x / 146097;
  x -= (146097 * n + 3) / 4;
  i = (4000 * (x + 1)) / 1461001;
  x -= 1461 * i / 4 - 31;
  j = 80 * x / 2447;
  d = x - 2447 * j / 80;
  x = j / 11;
  m = j + 2 - 12 * x;
  y = 100 * (n - 49) + i + x;
}

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

7.9 Hashing

```
#include "template.h"

const int N = 1e5;

// fhash[i] stores hash of s from s[0] to s[i],
    bhash stores hash
// for s[i] to s[n-1], calcFhash/CalcBhash,
    calculate hash from
// s[l] to s[r] in forward/backward direction

struct HASH{
  pii fhash[N],bhash[N];
  pii p[N],ip[N];
  string s;
  int n;
  HASH(string str){
    s = str; n = s.size();
  }
  void init(){
    p[0] = ip[0] = mp(1,1);
    rep1(i,1,N-1){
      p[i].F = 31LL * p[i-1].F % Mod;
      p[i].S = 37LL * p[i-1].S % Mod;
      ip[i].F = 129032259LL * ip[i-1].F % Mod;
      ip[i].S = 621621626LL * ip[i-1].S % Mod;
    }
  }
  void infHash(){
    rep1(i,0,n-1){
      fhash[i].F = (1LL * s[i] * p[i].F + ( (i) ?
           fhash[i-1].F : 0 ) ) % Mod;
      fhash[i].S = (1LL * s[i] * p[i].S + ( (i) ?
           fhash[i-1].S : 0 ) ) % Mod;
    }
  }
  void inbHash(){
    rep2(i,n-1,0){
      bhash[i].F = (1LL * s[i] * p[n-i-1].F + ( (
          i<n-1) ? bhash[i+1].F : 0)) % Mod;
      bhash[i].S = (1LL * s[i] * p[n-i-1].S + ( (
          i<n-1) ? bhash[i+1].S : 0)) % Mod;
    }
  }
  pii CalcFhash(int l,int r){
    if(l > r)return mp(0,0);
    pii ret;
    ret.F = 1LL * (fhash[r].F - ((l)?fhash[l-1].F
        :0) + Mod) * ip[l].F % Mod;
    ret.S = 1LL * (fhash[r].S - ((l)?fhash[l-1].S
        :0) + Mod) * ip[l].S % Mod;
    return ret;
  }
  pii CalcBhash(int l,int r){
    if(l > r)return mp(0,0);
    pii ret;
    ret.F = 1LL * (bhash[l].F - ((r<n-1)?bhash[r
        +1].F:0) + Mod)* ip[n-1-r].F % Mod;
    ret.S = 1LL * (bhash[l].S - ((r<n-1)?bhash[r
        +1].S:0) + Mod)* ip[n-1-r].S % Mod;
    return ret;
  }
};

int main() {return 0;}
```

7.10 Mobius function

```
void MOB(int n){
  vector<int> mob(n);
  for(int i = 1; i < n ; ++i)mob[i] = 1;
  for(int i = 2; i < N ; i++){
    if(pf[i] == i){
      if(1LL * i * i < N){
        for(int j = i*i ; j < N ; j += (i*i) ){
          mob[j] = 0;
        }
      }
      for(int j = i ; j < N ; j+=i){
        mob[j] *= -1;
      }
    }
  }
}
```

7.11 Mo's Algorithm

```
// Algorithm for sorting the quries in an order
    which
// minimizes the time required from O(n^2) to O((
    n + Q)sqrt(n))
// + QlogQ This is done by sorting the queries in
// order of range on which they are performed
// We store the queries and sort them using the
    compare
// function cmp. Also we need to make an add
    function to
// calculate the value of range (l,r+1) from
    value of range
// (l,r) and (l+1,r) from the value of (l,r), and
     a remove
// function to calculate the value of (l-1, r)
    from the value
// of (l,r) and (l,r-1) from the value of (l,r)
    in constant time

// S is the max integer number which is less than
     sqrt(N);
int S = (int)(sqrt(N)); // Here see if you want
    ll
bool cmp(Query A, Query B)
{
  if (A.l / S    = B.l / S) return A.l / S < B.l
      / S;
  return A.r > B.r;
}
```

8 Theory

Combinatorics

Sums

$\sum_{k=0}^{n} k = n(n+1)/2$ $\quad\binom{n}{k} = \frac{n!}{(n-k)!k!}$

$\sum_{k=a}^{b} k = (a+b)(b-a+1)/2$ $\quad\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$

$\sum_{k=0}^{n} k^2 = n(n+1)(2n+1)/6$ $\quad\binom{n+1}{k} = \frac{n+1}{n-k+1}\binom{n}{k}$

$\sum_{k=0}^{n} k^3 = n^2(n+1)^2/4$ $\quad\binom{n}{k+1} = \frac{n-k}{k+1}\binom{n}{k}$

$\sum_{k=0}^{n} k^4 = (6n^5 + 15n^4 + 10n^3 - n)/30$ $\quad\binom{n}{k} = \frac{n}{n-k}\binom{n-1}{k}$

$\sum_{k=0}^{n} k^5 = (2n^6 + 6n^5 + 5n^4 - n^2)/12$ $\quad\binom{n}{k} = \frac{n-k+1}{k}\binom{n}{k-1}$

$\sum_{k=0}^{n} x^k = (x^{n+1} - 1)/(x-1)$ $\quad 12! \approx 2^{28.8}$

$\sum_{k=0}^{n} kx^k = (x - (n+1)x^{n+1} + nx^{n+2})/(x-1)^2$ $\quad 20! \approx 2^{61.1}$

$1 + x + x^2 + \cdots = 1/(1-x)$

Binomial coefficients

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1												
1	1	1											
2	1	2	1										
3	1	3	3	1									
4	1	4	6	4	1								
5	1	5	10	10	5	1							
6	1	6	15	20	15	6	1						
7	1	7	21	35	35	21	7	1					
8	1	8	28	56	70	56	28	8	1				
9	1	9	36	84	126	126	84	36	9	1			
10	1	10	45	120	210	252	210	120	45	10	1		
11	1	11	55	165	330	462	462	330	165	55	11	1	
12	1	12	66	220	495	792	924	792	495	220	66	12	1
	0	1	2	3	4	5	6	7	8	9	10	11	12

Number of ways to pick a multiset of size k from n elements: $\binom{n+k-1}{k}$
Number of n-tuples of non-negative integers with sum s: $\binom{s+n-1}{n-1}$, at most s: $\binom{s+n}{n}$
Number of n-tuples of positive integers with sum s: $\binom{s-1}{n-1}$
Number of lattice paths from $(0,0)$ to (a,b), restricted to east and north steps: $\binom{a+b}{a}$

Multinomial theorem. $(a_1 + \cdots + a_k)^n = \sum \binom{n}{n_1,\ldots,n_k} a_1^{n_1} \ldots a_k^{n_k}$, where $n_i \geq 0$ and $\sum n_i = n$.

$$\binom{n}{n_1,\ldots,n_k} = M(n_1,\ldots,n_k) = \frac{n!}{n_1!\ldots n_k!}$$

$$M(a,\ldots,b,c,\ldots) = M(a+\cdots+b,c,\ldots)M(a,\ldots,b)$$

Catalan numbers. $C_n = \frac{1}{n+1}\binom{2n}{n}$. $\quad C_0 = 1$, $C_n = \sum_{i=0}^{n-1} C_i C_{n-1-i}$. $\quad C_{n+1} = C_n \frac{4n+2}{n+2}$.
$C_0, C_1, \ldots = 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, \ldots$
C_n is the number of: properly nested sequences of n pairs of parentheses; rooted ordered binary trees with $n+1$ leaves; triangulations of a convex $(n+2)$-gon.

Derangements. Number of permutations of $n = 0, 1, 2, \ldots$ elements without fixed points is $1, 0, 1, 2, 9, 44, 265, 1854, 14833, \ldots$ Recurrence: $D_n = (n-1)(D_{n-1} + D_{n-2}) = nD_{n-1} + (-1)^n$. Corollary: number of permutations with exactly k fixed points is $\binom{n}{k} D_{n-k}$.

Stirling numbers of 1^{st} kind. $s_{n,k}$ is $(-1)^{n-k}$ times the number of permutations of n elements with exactly k permutation cycles. $|s_{n,k}| = |s_{n-1,k-1}| + (n-1)|s_{n-1,k}|$. $\sum_{k=0}^{n} s_{n,k}\, x^k = x^{\underline{n}}$

Stirling numbers of 2^{nd} kind. $S_{n,k}$ is the number of ways to partition a set of n elements into exactly k non-empty subsets. $S_{n,k} = S_{n-1,k-1} + kS_{n-1,k}$. $S_{n,1} = S_{n,n} = 1$. $x^n = \sum_{k=0}^{n} S_{n,k}\, x^{\underline{k}}$

Bell numbers. B_n is the number of partitions of n elements. $B_0, \ldots = 1, 1, 2, 5, 15, 52, 203, \ldots$
$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k = \sum_{k=1}^{n} S_{n,k}$. Bell triangle: $B_r = a_{r,1} = a_{r-1,r-1}$, $a_{r,c} = a_{r-1,c-1} + a_{r,c-1}$.

Bernoulli numbers. $\sum_{k=0}^{m-1} k^n = \frac{1}{n+1} \sum_{k=0}^{n} \binom{n+1}{k} B_k m^{n+1-k}$.
$\sum_{j=0}^{m} \binom{m+1}{j} B_j = 0$. $\quad B_0 = 1$, $B_1 = -\frac{1}{2}$. $B_n = 0$, for all odd $n \neq 1$.

Eulerian numbers. $E(n,k)$ is the number of permutations with exactly k descents ($i : \pi_i < \pi_{i+1}$) / ascents ($\pi_i > \pi_{i+1}$) / excedances ($\pi_i > i$) / $k+1$ weak excedances ($\pi_i \geq i$).
Formula: $E(n,k) = (k+1)E(n-1,k) + (n-k)E(n-1,k-1)$. $\quad x^n = \sum_{k=0}^{n-1} E(n,k)\binom{x+k}{n}$.

Burnside's lemma. The number of orbits under group G's action on set X:
$|X/G| = \frac{1}{|G|} \sum_{g \in G} |X_g|$, where $X_g = \{x \in X : g(x) = x\}$. ("Average number of fixed points.")
Let $w(x)$ be weight of x's orbit. Sum of all orbits' weights: $\sum_{o \in X/G} w(o) = \frac{1}{|G|} \sum_{g \in G} \sum_{x \in X_g} w(x)$.

Number Theory

Linear diophantine equation. $ax + by = c$. Let $d = \gcd(a,b)$. A solution exists iff $d|c$. If (x_0, y_0) is any solution, then all solutions are given by $(x,y) = (x_0 + \frac{b}{d}t, y_0 - \frac{a}{d}t)$, $t \in \mathbb{Z}$. To find some solution (x_0, y_0), use extended GCD to solve $ax_0 + by_0 = d = \gcd(a,b)$, and multiply its solutions by $\frac{c}{d}$.

Linear diophantine equation in n variables: $a_1x_1 + \cdots + a_nx_n = c$ has solutions iff $\gcd(a_1, \ldots, a_n)|c$. To find some solution, let $b = \gcd(a_2, \ldots, a_n)$, solve $a_1x_1 + by = c$, and iterate with $a_2x_2 + \cdots = y$.

Extended GCD

```
// Finds g = gcd(a,b) and x, y such that ax+by=g.
// Bounds: |x|<=b+1, |y|<=a+1.
void gcdext(int &g, int &x, int &y, int a, int b)
{ if (b == 0) { g = a; x = 1; y = 0; }
  else        { gcdext(g, y, x, b, a % b); y = y - (a / b) * x; } }
```

Multiplicative inverse of a modulo m: x in $ax + my = 1$, or $a^{\phi(m)-1} \pmod{m}$.

Chinese Remainder Theorem. System $x \equiv a_i \pmod{m_i}$ for $i = 1, \ldots, n$, with pairwise relatively-prime m_i has a unique solution modulo $M = m_1 m_2 \ldots m_n$: $x = a_1 b_1 \frac{M}{m_1} + \cdots + a_n b_n \frac{M}{m_n} \pmod{M}$, where b_i is modular inverse of $\frac{M}{m_i}$ modulo m_i.

System $x \equiv a \pmod m$, $x \equiv b \pmod n$ has solutions iff $a \equiv b \pmod g$, where $g = \gcd(m,n)$. The solution is unique modulo $L = \frac{mn}{g}$, and equals: $x \equiv a + T(b-a)m/g \equiv b + S(a-b)n/g \pmod L$, where S and T are integer solutions of $mT + nS = \gcd(m,n)$.

Prime-counting function. $\pi(n) = |\{p \le n : p \text{ is prime}\}|$. $n/\ln(n) < \pi(n) < 1.3n/\ln(n)$. $\pi(1000) = 168$, $\pi(10^6) = 78498$, $\pi(10^9) = 50\,847\,534$. n-th prime $\approx n \ln n$.

Miller-Rabin's primality test. Given $n = 2^r s + 1$ with odd s, and a random integer $1 < a < n$.
If $a^s \equiv 1 \pmod n$ or $a^{2^j s} \equiv -1 \pmod n$ for some $0 \le j \le r-1$, then n is a probable prime. With bases 2, 7 and 61, the test indentifies all composites below 2^{32}. Probability of failure for a random a is at most 1/4.

Pollard-ρ. Choose random x_1, and let $x_{i+1} = x_i^2 - 1 \pmod n$. Test $\gcd(n, x_{2^k+i} - x_{2^k})$ as possible n's factors for $k = 0, 1, \ldots$ Expected time to find a factor: $O(\sqrt{m})$, where m is smallest prime power in n's factorization. That's $O(n^{1/4})$ if you check $n = p^k$ as a special case before factorization.

Fermat primes. A Fermat prime is a prime of form $2^{2^n} + 1$. The only known Fermat primes are 3, 5, 17, 257, 65537. A number of form $2^n + 1$ is prime only if it is a Fermat prime.

Perfect numbers. $n > 1$ is called perfect if it equals sum of its proper divisors and 1. Even n is perfect iff $n = 2^{p-1}(2^p - 1)$ and $2^p - 1$ is prime (Mersenne's). No odd perfect numbers are yet found.

Carmichael numbers. A positive composite n is a Carmichael number ($a^{n-1} \equiv 1 \pmod n$ for all a: $\gcd(a,n) = 1$), iff n is square-free, and for all prime divisors p of n, $p - 1$ divides $n - 1$.

Number/sum of divisors. $\tau(p_1^{a_1} \ldots p_k^{a_k}) = \prod_{j=1}^k (a_j + 1)$. $\sigma(p_1^{a_1} \ldots p_k^{a_k}) = \prod_{j=1}^k \frac{p_j^{a_j+1} - 1}{p_j - 1}$.

Euler's phi function. $\phi(n) = |\{m \in \mathbb{N}, m \le n, \gcd(m,n) = 1\}|$.
$\phi(mn) = \frac{\phi(m)\phi(n)\gcd(m,n)}{\phi(\gcd(m,n))}$. $\phi(p^a) = p^{a-1}(p-1)$. $\sum_{d|n} \phi(d) = \sum_{d|n} \phi(\frac{n}{d}) = n$.

Euler's theorem. $a^{\phi(n)} \equiv 1 \pmod n$, if $\gcd(a,n) = 1$.

Wilson's theorem. p is prime iff $(p-1)! \equiv -1 \pmod p$.

Mobius function. $\mu(1) = 1$. $\mu(n) = 0$, if n is not squarefree. $\mu(n) = (-1)^s$, if n is the product of s distinct primes. Let f, F be functions on positive integers. If for all $n \in N$, $F(n) = \sum_{d|n} f(d)$, then $f(n) = \sum_{d|n} \mu(d) F(\frac{n}{d})$, and vice versa. $\phi(n) = \sum_{d|n} \mu(d) \frac{n}{d}$. $\sum_{d|n} \mu(d) = 1$.
If f is multiplicative, then $\sum_{d|n} \mu(d) f(d) = \prod_{p|n} (1 - f(p))$, $\sum_{d|n} \mu(d)^2 f(d) = \prod_{p|n} (1 + f(p))$.

Legendre symbol. If p is an odd prime, $a \in \mathbb{Z}$, then $\left(\frac{a}{p}\right)$ equals 0, if $p|a$; 1 if a is a quadratic residue modulo p; and -1 otherwise. Euler's criterion: $\left(\frac{a}{p}\right) = a^{\left(\frac{p-1}{2}\right)} \pmod p$.

Jacobi symbol. If $n = p_1^{a_1} \cdots p_k^{a_k}$ is odd, then $\left(\frac{a}{n}\right) = \prod_{i=1}^k \left(\frac{a}{p_i}\right)^{k_i}$.

Primitive roots. If the order of g modulo m (min $n > 0$: $g^n \equiv 1 \pmod m$) is $\phi(m)$, then g is called a primitive root. If Z_m has a primitive root, then it has $\phi(\phi(m))$ distinct primitive roots. Z_m has a primitive root iff m is one of 2, 4, p^k, $2p^k$, where p is an odd prime. If Z_m has a primitive root g, then for all a coprime to m, there exists unique integer $i = \text{ind}_g(a)$ modulo $\phi(m)$, such that $g^i \equiv a \pmod m$. $\text{ind}_g(a)$ has logarithm-like properties: $\text{ind}(1) = 0$, $\text{ind}(ab) = \text{ind}(a) + \text{ind}(b)$.

If p is prime and a is not divisible by p, then congruence $x^n \equiv a \pmod p$ has $\gcd(n, p-1)$ solutions if $a^{(p-1)/\gcd(n,p-1)} \equiv 1 \pmod p$, and no solutions otherwise. (Proof sketch: let g be a primitive root, and $g^i \equiv a \pmod p$, $g^u \equiv x \pmod p$. $x^n \equiv a \pmod p$ iff $g^{nu} \equiv g^i \pmod p$ iff $nu \equiv i \pmod p$.)

Discrete logarithm problem. Find x from $a^x \equiv b \pmod m$. Can be solved in $O(\sqrt{m})$ time and space with a meet-in-the-middle trick. Let $n = \lceil \sqrt{m} \rceil$, and $x = ny - z$. Equation becomes $a^{ny} \equiv ba^z \pmod m$. Precompute all values that the RHS can take for $z = 0, 1, \ldots, n-1$, and brute force y on the LHS, each time checking whether there's a corresponding value for RHS.

Pythagorean triples. Integer solutions of $x^2 + y^2 = z^2$ All relatively prime triples are given by: $x = 2mn, y = m^2 - n^2, z = m^2 + n^2$ where $m > n, \gcd(m,n) = 1$ and $m \not\equiv n \pmod 2$. All other triples are multiples of these. Equation $x^2 + y^2 = 2z^2$ is equivalent to $(\frac{x+y}{2})^2 + (\frac{x-y}{2})^2 = z^2$.

Postage stamps/McNuggets problem. Let a, b be relatively-prime integers. There are exactly $\frac{1}{2}(a-1)(b-1)$ numbers *not* of form $ax + by$ $(x, y \ge 0)$, and the largest is $(a-1)(b-1) - 1 = ab - a - b$.

Fermat's two-squares theorem. Odd prime p can be represented as a sum of two squares iff $p \equiv 1 \pmod 4$. A product of two sums of two squares is a sum of two squares. Thus, n is a sum of two squares iff every prime of form $p = 4k + 3$ occurs an even number of times in n's factorization.

RSA. Let p and q be random distinct large primes, $n = pq$. Choose a small odd integer e, relatively prime to $\phi(n) = (p-1)(q-1)$, and let $d = e^{-1} \pmod{\phi(n)}$. Pairs (e, n) and (d, n) are the public and secret keys, respectively. Encryption is done by raising a message $M \in Z_n$ to the power e or d, modulo n.

String Algorithms

Burrows-Wheeler inverse transform. Let $B[1..n]$ be the input (last column of sorted matrix of string's rotations.) Get the first column, $A[1..n]$, by sorting B. For each k-th occurence of a character c at index i in A, let $next[i]$ be the index of corresponding k-th occurence of c in B. The r-th fow of the matrix is $A[r]$, $A[next[r]]$, $A[next[next[r]]]$, ...

Huffman's algorithm. Start with a forest, consisting of isolated vertices. Repeatedly merge two trees with the lowest weights.

Graph Theory

Euler's theorem. For any planar graph, $V - E + F = 1 + C$, where V is the number of graph's vertices, E is the number of edges, F is the number of faces in graph's planar drawing, and C is the number of connected components. Corollary: $V - E + F = 2$ for a 3D polyhedron.

Vertex covers and independent sets. Let M, C, I be a max matching, a min vertex cover, and a max independent set. Then $|M| \le |C| = N - |I|$, with equality for bipartite graphs. Complement of an MVC is always a MIS, and vice versa. Given a bipartite graph with partitions (A, B), build a network: connect source to A, and B to sink with edges of capacities, equal to the corresponding nodes' weights, or 1 in the unweighted case. Set capacities of the original graph's edges to the infinity. Let (S, T) be a minimum s-t cut. Then a maximum(-weighted) independent set is $I = (A \cap S) \cup (B \cap T)$, and a minimum(-weighted) vertex cover is $C = (A \cap T) \cup (B \cap S)$.

Matrix-tree theorem. Let matrix $T = [t_{ij}]$, where t_{ij} is the number of multiedges between i and j, for $i \neq j$, and $t_{ii} = -\deg_i$. Number of spanning trees of a graph is equal to the determinant of a matrix obtained by deleting any k-th row and k-th column from T.

Euler tours. Euler tour in an undirected graph exists iff the graph is connected and each vertex has an even degree. Euler tour in a directed graph exists iff in-degree of each vertex equals its out-degree, and underlying undirected graph is connected. Construction:

```
doit(u):
  for each edge e = (u, v) in E, do: erase e, doit(v)
  prepend u to the list of vertices in the tour
```

Stable marriages problem. While there is a free man m: let w be the most-preferred woman to whom he has not yet proposed, and propose m to w. If w is free, or is engaged to someone whom she prefers less than m, match m with w, else deny proposal.

Stoer-Wagner's min-cut algorithm. Start from a set A containing an arbitrary vertex. While $A \neq V$, add to A the most tightly connected vertex ($z \notin A$ such that $\sum_{x \in A} w(x, z)$ is maximized.) Store cut-of-the-phase (the cut between the last added vertex and rest of the graph), and merge the two vertices added last. Repeat until the graph is contracted to a single vertex. Minimum cut is one of the cuts-of-the-phase.

Tarjan's offline LCA algorithm. (Based on DFS and union-find structure.)

```
DFS(x):
  ancestor[Find(x)] = x
  for all children y of x:
     DFS(y); Union(x, y); ancestor[Find(x)] = x
  seen[x] = true
  for all queries {x, y}:
    if seen[y] then output "LCA(x, y) is ancestor[Find(y)]"
```

Strongly-connected components. Kosaraju's algorithm.

1. Let G^T be a transpose G (graph with reversed edges.)
1. Call DFS(G^T) to compute finishing times $f[u]$ for each vertex u.
3. For each vertex u, in the order of decreasing $f[u]$, perform DFS(G, u).
4. Each tree in the 3rd step's DFS forest is a separate SCC.

2-SAT. Build an implication graph with 2 vertices for each variable – for the variable and its inverse; for each clause $x \vee y$ add edges $(\overline{x}, y)$ and $(\overline{y}, x)$. The formula is satisfiable iff x and $\overline{x}$ are in distinct SCCs, for all x. To find a satisfiable assignment, consider the graph's SCCs in topological order from sinks to sources (i.e. Kosaraju's last step), assigning 'true' to all variables of the current SCC (if it hasn't been previously assigned 'false'), and 'false' to all inverses.

Randomized algorithm for non-bipartite matching. Let G be a simple undirected graph with even $|V(G)|$. Build a matrix A, which for each edge $(u, v) \in E(G)$ has $A_{i,j} = x_{i,j}$, $A_{j,i} = -x_{i,j}$, and is zero elsewhere. Tutte's theorem: G has a perfect matching iff $\det G$ (a multivariate polynomial) is identically zero. Testing the latter can be done by computing the determinant for a few random values of $x_{i,j}$'s over some field. (e.g. Z_p for a sufficiently large prime p)

Prufer code of a tree. Label vertices with integers 1 to n. Repeatedly remove the leaf with the smallest label, and output its only neighbor's label, until only one edge remains. The sequence has length $n - 2$. Two isomorphic trees have the same sequence, and every sequence of integers from 1 and n corresponds to a tree. Corollary: the number of labelled trees with n vertices is n^{n-2}.

Erdos-Gallai theorem. A sequence of integers $\{d_1, d_2, \ldots, d_n\}$, with $n - 1 \geq d_1 \geq d_2 \geq \cdots \geq d_n \geq 0$ is a degree sequence of some undirected simple graph iff $\sum d_i$ is even and $d_1 + \cdots + d_k \leq k(k-1) + \sum_{i=k+1}^{n} \min(k, d_i)$ for all $k = 1, 2, \ldots, n-1$.

Games

Grundy numbers. For a two-player, normal-play (last to move wins) game on a graph (V, E): $G(x) = \operatorname{mex}(\{G(y) : (x, y) \in E\})$, where $\operatorname{mex}(S) = \min\{n \geq 0 : n \notin S\}$. x is losing iff $G(x) = 0$.

Sums of games.

- *Player chooses a game and makes a move in it.* Grundy number of a position is xor of grundy numbers of positions in summed games.
- *Player chooses a non-empty subset of games (possibly, all) and makes moves in all of them.* A position is losing iff each game is in a losing position.
- *Player chooses a proper subset of games (not empty and not all), and makes moves in all chosen ones.* A position is losing iff grundy numbers of all games are equal.
- *Player must move in all games, and loses if can't move in some game.* A position is losing if any of the games is in a losing position.

Misère Nim. A position with pile sizes $a_1, a_2, \ldots, a_n \geq 1$, not all equal to 1, is losing iff $a_1 \oplus a_2 \oplus \cdots \oplus a_n = 0$ (like in normal nim.) A position with n piles of size 1 is losing iff n is *odd*.

Bit tricks

Clearing the lowest 1 bit: `x & (x - 1)`, all trailing 1's: `x & (x + 1)`
Setting the lowest 0 bit: `x | (x + 1)`
Enumerating subsets of a bitmask m:

```
x=0; do { ...; x=(x+1+~m)&m; } while (x!=0);
```

`__builtin_ctz`/`__builtin_clz` returns the number of trailing/leading zero bits.
`__builtin_popcount(unsigned x)` counts 1-bits (slower than table lookups).
For 64-bit unsigned integer type, use the suffix '`ll`', i.e. `__builtin_popcountll`.

Math

Stirling's approximation $z! = \Gamma(z+1) = \sqrt{2\pi}\, z^{z+1/2}\, e^{-z}(1 + \frac{1}{12z} + \frac{1}{288z^2} - \frac{139}{51840z^3} + \ldots)$

Taylor series. $f(x) = f(a) + \frac{x-a}{1!} f'(a) + \frac{(x-a)^2}{2!} f^{(2)}(a) + \cdots + \frac{(x-a)^n}{n!} f^{(n)}(a) + \ldots$.

$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$

$\ln x = 2(a + \frac{a^3}{3} + \frac{a^5}{5} + \ldots)$, where $a = \frac{x-1}{x+1}$. $\ln x^2 = 2 \ln x$.

$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \ldots$, $\arctan x = \arctan c + \arctan \frac{x-c}{1+xc}$ (e.g c=.2)

$\pi = 4 \arctan 1$, $\pi = 6 \arcsin \frac{1}{2}$