Al-Baath University ICPC Team Notebook (2018)

Contents

1 Combinatorial optimization

1.1 Sparse max-flow

```cpp
// Adjacency list implementation of Dinic's
    blocking flow algorithm.
// This is very fast in practice, and only loses
     to push-relabel flow.
//
// Running time:
//     O(|V|^2 |E|)
//
// INPUT:
//     - graph, constructed using AddEdge()
//     - source and sink
//
// OUTPUT:
//     - maximum flow value
//     - To obtain actual flow values, look at
    edges with capacity > 0
//       (zero capacity edges are residual edges
    ).

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

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

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

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

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

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

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

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

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

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

// END CUT
```

1.2 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
```

1.3 Push-relabel max-flow

```
// Adjacency list implementation of FIFO push
    relabel maximum flow
// with the gap relabeling heuristic.  This
    implementation is
// significantly faster than straight Ford-
    Fulkerson.  It solves
// random problems with 10000 vertices and
    1000000 edges in a few
// seconds, though it is possible to construct
    test cases that
// achieve the worst-case.
//
// Running time:
//     O(|V|^3)
//
// INPUT:
//     - graph, constructed using AddEdge()
//     - source
//     - sink
//
// OUTPUT:
//     - maximum flow value
//     - To obtain the actual flow values, look
    at all edges with
//       capacity > 0 (zero capacity edges are
    residual edges).

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

using namespace std;

typedef long long LL;

struct Edge {
  int from, to, cap, flow, index;
  Edge(int from, int to, int cap, int flow, int
      index) :
    from(from), to(to), cap(cap), flow(flow),
        index(index) {}
};

struct PushRelabel {
  int N;
  vector<vector<Edge> > G;
  vector<LL> excess;
  vector<int> dist, active, count;
  queue<int> Q;
```

```
PushRelabel(int N) : N(N), G(N), excess(N),
    dist(N), active(N), count(2*N) {}

void AddEdge(int from, int to, int cap) {
  G[from].push_back(Edge(from, to, cap, 0, G[
      to].size()));
  if (from == to) G[from].back().index++;
  G[to].push_back(Edge(to, from, 0, 0, G[from
      ].size() - 1));
}

void Enqueue(int v) {
  if (!active[v] && excess[v] > 0) { active[v]
       = true; Q.push(v); }
}

void Push(Edge &e) {
  int amt = int(min(excess[e.from], LL(e.cap -
       e.flow)));
  if (dist[e.from] <= dist[e.to] || amt == 0)
      return;
  e.flow += amt;
  G[e.to][e.index].flow -= amt;
  excess[e.to] += amt;
  excess[e.from] -= amt;
  Enqueue(e.to);
}

void Gap(int k) {
  for (int v = 0; v < N; v++) {
    if (dist[v] < k) continue;
    count[dist[v]]--;
    dist[v] = max(dist[v], N+1);
    count[dist[v]]++;
    Enqueue(v);
  }
}

void Relabel(int v) {
  count[dist[v]]--;
  dist[v] = 2*N;
  for (int i = 0; i < G[v].size(); i++)
    if (G[v][i].cap - G[v][i].flow > 0)
      dist[v] = min(dist[v], dist[G[v][i].to]
          + 1);
  count[dist[v]]++;
  Enqueue(v);
}

void Discharge(int v) {
  for (int i = 0; excess[v] > 0 && i < G[v].
      size(); i++) Push(G[v][i]);
  if (excess[v] > 0) {
    if (count[dist[v]] == 1)
      Gap(dist[v]);
    else
      Relabel(v);
  }
}

LL GetMaxFlow(int s, int t) {
  count[0] = N-1;
  count[N] = 1;
  dist[s] = N;
  active[s] = active[t] = true;
  for (int i = 0; i < G[s].size(); i++) {
    excess[s] += G[s][i].cap;
    Push(G[s][i]);
  }

  while (!Q.empty()) {
    int v = Q.front();
    Q.pop();
    active[v] = false;
    Discharge(v);
  }

  LL totflow = 0;
  for (int i = 0; i < G[s].size(); i++)
      totflow += G[s][i].flow;
  return totflow;
 }
};

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

int main() {
  int n, m;
  scanf("%d%d", &n, &m);

  PushRelabel pr(n);
  for (int i = 0; i < m; i++) {
   int a, b, c;
    scanf("%d%d%d", &a, &b, &c);
    if (a == b) continue;
    pr.AddEdge(a-1, b-1, c);
    pr.AddEdge(b-1, a-1, c);
  }
  printf("%Ld\n", pr.GetMaxFlow(0, n-1));
  return 0;
}

// END CUT
```

1.4 Min-cost 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 <algorithm>
#include <cstdio>
#include <cmath>
#include <vector>

using namespace std;

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

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

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

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

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

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

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

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

    int j = 0;
    while (true) {

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

```
        if (j == -1 || dist[k] < dist[j]) j = k;
      }
      seen[j] = 1;

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

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

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

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

    mated++;
  }

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

  return value;
}
```

1.5 Max bipartite matchine

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

#include <vector>

using namespace std;

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

bool FindMatch(int i, const VVI &w, VI &mr, VI &
    mc, VI &seen) {
  for (int j = 0; j < w[i].size(); j++) {
    if (w[i][j] && !seen[j]) {
      seen[j] = true;
      if (mc[j] < 0 || FindMatch(mc[j], w, mr,
          mc, seen)) {
        mr[i] = j;
        mc[j] = i;
        return true;
      }
    }
  }
  return false;
}

int BipartiteMatching(const VVI &w, VI &mr, VI &
    mc) {
  mr = VI(w.size(), -1);
  mc = VI(w[0].size(), -1);

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

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

using namespace std;

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

const int INF = 1000000000;

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

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

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

2 Geometry

2.1 Convex hull

```
// Compute the 2D convex hull of a set of points
    using the monotone chain
// algorithm.  Eliminate redundant points from
    the hull if REMOVE_REDUNDANT is
// #defined.
//
```

```
// Running time: O(n log n)
//
//   INPUT:   a vector of input points,
    unordered.
//   OUTPUT:  a vector of points in the convex
    hull, counterclockwise, starting
//            with bottommost/leftmost point

#include <cstdio>
#include <cassert>
#include <vector>
#include <algorithm>
#include <cmath>
// BEGIN CUT
#include <map>
// END CUT

using namespace std;

#define REMOVE_REDUNDANT

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

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

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

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

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

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

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

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

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

// END CUT
```

2.2 Miscellaneous geometry

```
// C++ routines for computational geometry.
// pi = 3.1415926535 8979323846 2643383279
    5028841971 6939937510 5820974944
#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) {
  return 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 3D geometry

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

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

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

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

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

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

2.4 3D geometry -Ashley

```
struct point3 {
double x, y, z;
point3(double x=0, double y=0, double z=0):x(x),
     y(y),z(z){}
point3 operator+(point3 p)const ?{ return point3
     (x + p.x, y
+ p.y, z + p.z); }
point3 operator*(double k)const { return point3(
     k*x, k*y,
k*z); }
point3 operator-(point3 p)const ?{ return *this
     + (p*-1.0);}
point3 operator/(double k)const { return *this
     *(1.0/k); }
double norm() { return x*x + y*y + z*z; }
double abs() { return sqrt(norm()); }
point3 normalize() { return *this/this->abs(); }
};
// dot product
double dot(point3 a, point3 b) {
return a.x*b.x + a.y*b.y + a.z*b.z;
}
// cross product
point3 cross(point3 a, point3 b) {
return point3(a.y*b.z - b.y*a.z, b.x*a.z - a.x*b
     .z, a.x*b.y
- b.x*a.y);
}
struct line {
point3 a, b;
```

```
line(point3 A=point3(), point3 B=point3()) : a(A
    ), b(B) {}
// Direction unit vector a -> b
point3 dir() { return (b - a).normalize(); }
};
// Returns closest point on an infinite line u
    to the point p
point3 cpoint_iline(line u, point3 p) {
point3 ud = u.dir();
return u.a - ud*dot(u.a - p, ud);
}
// Returns Shortest distance between two
    infinite lines u and v
double dist_ilines(line u, line v) {
return dot(v.a - u.a, cross(u.dir(), v.dir()).
    normalize());
}
// Finds the closest point on infinite line u to
     infinite line v
// Note: if (uv*uv - uu*vv) is zero then the
    lines are parallel
// and such a single closest point does not
    exist. Check for
// this if needed.
point3 cpoint_ilines(line u, line v) {
point3 ud = u.dir(); point3 vd = v.dir();
double uu = dot(ud, ud), vv = dot(vd, vd), uv =
    dot(ud, vd);
double t = dot(u.a, ud) - dot(v.a, ud); t *= vv;
t -= uv*(dot(u.a, vd) - dot(v.a, vd));
t /= (uv*uv - uu*vv);
return u.a + ud*t;
}
// Closest point on a line segment u to a given
    point p
point3 cpoint_lineseg(line u, point3 p) {
point3 ud = u.b - u.a; double s = dot(u.a - p,
ud)/ud.norm();
if (s < -1.0) return u.b;
if (s > ?0.0) return u.a;
return u.a - ud*s;
}
struct plane {
point3 n, p;
plane(point3 ni = point3(), point3 pi = point3()
    ) : n(ni),
p(pi) {}
plane(point3 a, point3 b, point3 c) : n(cross(b-
    a, ca).normalize()), p(a) {}
//Value of d for the equation ax + by + cz + d =
     0
double d() { return -dot(n, p); }
};
// Closest point on a plane u to a given point p
point3 cpoint_plane(plane u, point3 p) {
return p - u.n*(dot(u.n, p) + u.d());
}
// Point of intersection of an infinite line v
    and a plane u.
// Note: if dot(u.n, vd) == 0 then the line and
    plane do not
// intersect at a single point. Check for this
    if needed.
point3 iline_isect_plane(plane u, line v) {
point3 vd = v.dir();
return v.a - vd*((dot(u.n, v.a) + u.d())/dot(u.n
    , vd));
}
// Infinite line of intersection between two
    planes u and v.
// Note: if dot(v.n, uvu) == 0 then the planes
    do not intersect
// at a line. Check for this case if it is
    needed.
line isect_planes(plane u, plane v) {
point3 o = u.n*-u.d(), uv = cross(u.n, v.n);
point3 uvu = cross(uv, u.n);
point3 a = o - uvu*((dot(v.n, o) + v.d())/(dot(v
    .n,
uvu)*uvu.norm()));
return line(a, a + uv);
}
// Returns great circle distance (lat[-90,90],
    long[-180,180])
double greatcircle(double lt1, double lo1,
    double lt2, double
lo2, double r) {
double a = M_PI*(lt1/180.0), b = M_PI*(lt2
    /180.0);
double c = M_PI*((lo2-lo1)/180.0);
return r*acos(sin(a)*sin(b) + cos(a)*cos(b)*cos(
    c));
}
// Rotates point p around directed line a->b
    with angle 'theta'
point3 rotate(point3 a, point3 b, point3 p,
    double theta) {
point3 o = cpoint_iline(line(a,b),p);
point3 perp = cross(b-a,p-o);
return o+perp*sin(theta)+(p-o)*cos(theta);
}
```

2.5 Circles

```
// Returns whether they form a circle or not.
// 'center' and 'r' contain the circle if there
    is one
bool get_circle(point p1, point p2, point p3,
    point &center,
double &r) {
double g = 2*imag(conj(p2-p1)*(p3-p2));
if (abs(g) < eps) return false;
center = p1*(norm(p3)-norm(p2));
center += p2*(norm(p1)-norm(p3));
center += p3*(norm(p2)-norm(p1));
center /= point(0, g); r = abs(p1-center);
return true;
}



// Returns number of circles that are tangent to
     all three lines
// 'cirs' has all possible circles with radius >
     0
// It has zero circles when two of them are
    coincide
// It has two circles when only two of them are
    parallel
// It has four circles when they form a triangle
    . In this case
// first circle is incircle. Next circles are ex
    -circles tangent
// to edge a,b,c of triangle respectively.
int get_circle(point a1, point a2, point b1,
    point b2, point c1,
point c2, vector<circle> &cirs) {
point a,b,c;
int sa=line_line_inter(a1,a2,b1,b2,c);
int sb=line_line_inter(b1,b2,c1,c2,a);
int sc=line_line_inter(c1,c2,a1,a2,b);
if(sa==-1 || sb==-1 || sc==-1)
return 0;
if(sa+sb+sc==0)
return 0;
if(sb==0) {
swap(a1,c1);
swap(a2,c2);
}
if(sc==0) {
swap(b1,c1);
swap(b2,c2);
}
sa=line_line_inter(a1,a2,b1,b2,c);
line_line_inter(b1,b2,c1,c2,a);
line_line_inter(c1,c2,a1,a2,b);
if(sa==0) {
point v1 = polar(1.0,(arg(a2-a1)+arg(a-b))/2)+b;
point v2 = polar(1.0,(arg(a1-a2)+arg(a-b))/2)+b;
point v3 = polar(1.0,(arg(b2-b1)+arg(a-b))/2)+a;
point v4 = polar(1.0,(arg(b1-b2)+arg(a-b))/2)+a;

point p;
if(line_line_inter(b,v1,a,v3,p)==0)
swap(v3,v4);
line_line_inter(b,v1,a,v3,p);
circle c1,c2;
c1.c = p;
line_line_inter(b,v2,a,v4,p);
c2.c = p;
c1.r = c2.r = abs(((a1-b1)/(b2-b1)).imag()*abs(
    b2-
b1))/2;
cirs.push_back(c1);
cirs.push_back(c2);
} else {
if(abs(a-b)<eps)
return 0;
point bisec1[4][2];
point bisec2[4][2];
bisec1[0][0]=polar(1.0,(arg(c-a)+arg(b-a))/2);
bisec1[0][1]=a;
bisec2[0][0]=polar(1.0,(arg(c-b)+arg(a-b))/2);
bisec2[0][1]=b;
bisec1[1][0]=polar(1.0,(arg(c-a)+arg(b-a))/2);
bisec1[1][1]=a;
bisec2[1][0]=polar(1.0,(arg(c-b)+arg(b-a))/2);
bisec2[1][1]=b;
bisec1[2][0]=polar(1.0,(arg(a-b)+arg(c-b))/2);
bisec1[2][1]=b;
bisec2[2][0]=polar(1.0,(arg(a-c)+arg(c-b))/2);
bisec2[2][1]=c;
bisec1[3][0]=polar(1.0,(arg(b-c)+arg(a-c))/2);
bisec1[3][1]=b;
bisec2[3][0]=polar(1.0,(arg(b-a)+arg(a-c))/2);
bisec2[3][1]=c;
for(int i=0;i<4;i++) {
point p;
line_line_inter(bisec1[i][1],bisec1[i][1]+bisec1
    [i]
[0],bisec2[i][1],bisec2[i][1]+bisec2[i][0],p);
circle c1;

c1.c = p;
c1.r = abs(((p-a)/(b-a)).imag())*abs(b-a);
```

```
cirs.push_back(c1);
}
}
return cirs.size();
}


// Returns number of circles that pass through point a and b and
// are tangent to the line c-d
// 'ans' has all possible circles with radius > 0
int get_circle(point a, point b, point c, point d,
vector<circle> &ans) {
point pa = (a+b)/2.0;
point pb = (b-a)*point(0,1)+pa;
vector<point> ta;
parabola_line_inter(a,c,d,pa,pb,ta);
for(int i=0;i<ta.size();i++)
ans.push_back(circle(ta[i],abs(a-ta[i])));
return ans.size();
}


// Returns number of circles that pass through point p and are
// tangent to the lines a-b and c-d
// 'ans' has all possible circles with radius greater than zero
int get_circle(point p, point a, point b, point c, point d,
vector<circle> &ans) {
point inter;
int st = line_line_inter(a,b,c,d,inter);
if(st==-1) return 0;
d-=c;
b-=a;
vector<point> ta;
if(st==0) {
point pa = point(0,imag((a-c)/d)/2)*d+c;
point pb = b+pa;
parabola_line_inter(p,a,a+b,pa,pb,ta);
} else {
if(abs(inter-p)>eps) {
point bi;
bi = polar(1.0,(arg(b)+arg(d))/2)+inter;
vector<point> temp;
parabola_line_inter(p,a,a+b,inter,bi,temp);
ta.insert(ta.end(),temp.begin(),temp.end());
temp.clear();
bi = polar(1.0,(arg(b)+arg(d)+M_PI)/2)+inter;
parabola_line_inter(p,a,a+b,inter,bi,temp);
ta.insert(ta.end(),temp.begin(),temp.end());
}
}
for(int i=0;i<ta.size();i++)
ans.push_back(circle(ta[i],abs(p-ta[i])));
return ans.size();
}
```

2.6 Parabola Circle Intersection

```
// Find intersection of the line d-e and the parabola that
// is defined by point 'p' and line a-b
// Returns the number of intersections
// 'ans' has intersection points
int parabola_line_inter(point p, point a, point b, point d,
point e, vector<point> &ans) {
b = b-a;
p/=b; a/=b; d/=b; e/=b;
a-=p; d-=p; e-=p;
point n = (e-d)*point(0,1);
double c = -dot(n,e);
if(abs(n.imag())<eps) {
if(abs(a.imag())>eps) {
double x = -c/n.real();
ans.push_back(point(x,a.imag()/2-x*x/(2*a.imag())));
}
} else {
double aa = 1;
double bb = -2*a.imag()*n.real()/n.imag();
double cc = -2*a.imag()*c/n.imag()-a.imag()*a.imag();
double delta = bb*bb-4*aa*cc;
if(delta>-eps) {
if(delta<0)
delta = 0;
delta = sqrt(delta);
double x = (-bb+delta)/(2*aa);
ans.push_back(point(x,(-c-n.real()*x)/n.imag()));
if(delta>eps) {
double x = (-bb-delta)/(2*aa);
ans.push_back(point(x,(-cn.real()*x)/n.imag()));
}
}
}
for(int i=0;i<ans.size();i++)
ans[i]=(ans[i]+p)*b;
return ans.size();
}
```

2.7 Fegla Geometry

```
int segmentLatticePointsCount(int x1, int y1, int x2, int y2) {
  return abs(__gcd(x1 - x2, y1 - y2)) + 1;
}
int picksTheorm(int a, int b) {
  return a - b / 2 + 1;
}
void polygonCut(const vector<point>& p, const point&a, const point&b, vector<point>& res) {
  res.clear();
  for (int i = 0; i < sz(p); i++) {
    int j = (i + 1) % sz(p);
    bool in1 = cross(vec(a,b),vec(a,p[i])) > EPS;
    bool in2 = cross(vec(a,b),vec(a,p[j])) > EPS;
    if (in1) res.push_back(p[i]);
    if (in1 ^ in2) {
        point r;
        intersect(a, b, p[i], p[j], r);
        res.push_back(r);
    }
  }
}
// assume that both are anti-clockwise
void convexPolygonIntersect(const vector<point>& p, const vector<point>& q, vector<point>& res) {
  res = q;
  for (int i = 0; i < sz(p); i++) {
    int j = (i + 1) % sz(p);
    vector<point> temp;
    polygonCut(res, p[i], p[j], temp);
    res = temp;
    if (res.empty()) return;
  }
}
```

3 Numerical algorithms

3.1 Number theory (modular, Chinese remainder, linear Diophantine)

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

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

using namespace std;

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

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

// computes gcd(a,b)
int gcd(int a, int b) {
        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: 95 451
        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
        PII ret = chinese_remainder_theorem(VI({ 3, 5, 7 }), VI({ 2, 3, 2 }));
        cout << ret.first << " " << ret.second << endl;
        ret = chinese_remainder_theorem(VI({ 4, 6 }), VI({ 3, 5 }));
        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;
}
```

3.2 Systems of linear equations, matrix inverse, determinant

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

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

using namespace std;

const double EPS = 1e-10;

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

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

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

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

```
        c = a[p][pk];
        a[p][pk] = 0;
        for (int q = 0; q < n; q++) a[p][q] -= a[
            pk][q] * c;
        for (int q = 0; q < m; q++) b[p][q] -= b[
            pk][q] * c;
      }
    }

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

    return det;
  }

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

    double det = GaussJordan(a, b);

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

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

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

3.3 Reduced row echelon form, matrix rank

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

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

  using namespace std;

  const double EPSILON = 1e-10;

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

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

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

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

    int rank = rref(a);

    // expected: 3
    cout << "Rank: " << rank << endl;

    // expected: 1 0 0 1
    //           0 1 0 3
    //           0 0 1 -3
    //           0 0 0 3.10862e-15
    //           0 0 0 2.22045e-15
    cout << "rref: " << endl;
    for (int i = 0; i < 5; i++) {
      for (int j = 0; j < 4; j++)
        cout << a[i][j] << ' ';
      cout << endl;
    }
  }
```

3.4 Number Theory Essentials

```
  // Sieve
  ll _sieve_size;
  bitset<10000010> bs;
  vi primes;
  void sieve(ll upperbound) {
      // create list of primes in [0..upperbound]
      _sieve_size = upperbound + 1;
      bs.set();bs[0] = bs[1] = 0;
      for (ll i = 2; i <= _sieve_size; i++) if (bs
          [i]) {
          for (ll j = i * i; j <= _sieve_size; j
              += i) bs[j] = 0;
          primes.push_back((int)i);
      }
  }
  bool isPrime(ll N) {
      if (N <= _sieve_size) return bs[N];
      for (int i = 0; i < (int)primes.size(); i++)
          if (N % primes[i] == 0) return false;
      return true;
  }

  // Prime Factors
  vi primeFactors(ll N) {
      vi factors;
      ll PF_idx = 0, PF = primes[PF_idx];
      while (PF * PF <= N) {
          while (N % PF == 0) { N /= PF; factors.
              push_back(PF); }
          PF = primes[++PF_idx];
      }
      if (N != 1) factors.push_back(N);
      return factors;
  }

  // NumDiv
  ll numDiv(ll N) {
      ll PF_idx = 0, PF = primes[PF_idx], ans = 1;
      while (PF * PF <= N) {
          ll power = 0;
          while (N % PF == 0) { N /= PF; power++;
              }
          ans *= (power + 1);
          PF = primes[++PF_idx];
      }
      if (N != 1) ans *= 2;
      return ans;
  }

  // SumDiv
  ll sumDiv(ll N) {
      ll PF_idx = 0, PF = primes[PF_idx], ans = 1;
      while (PF * PF <= N) {
          ll power = 0;
          while (N % PF == 0) { N /= PF; power++;
              }
```

```
        ans *= ((ll)pow((double)PF, power + 1.0)
            - 1) / (PF - 1);
        PF = primes[++PF_idx];
    }
    if (N != 1) ans *= ((ll)pow((double)N, 2.0)
        - 1) / (N - 1);
    return ans;
}

// EulerPhi
ll EulerPhi(ll N) {
    ll PF_idx = 0, PF = primes[PF_idx], ans = N;
    while (PF * PF <= N) {
        if (N % PF == 0) ans -= ans / PF;
        while (N % PF == 0) N /= PF;
        PF = primes[++PF_idx];
    }
    if (N != 1) ans -= ans / N;
    return ans;
}
```

4 Graph algorithms

4.1 Eulerian path

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

struct Edge
{
        int next_vertex;
        iter reverse_edge;

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

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

vector<int> path;

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

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

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;
}

//range update source http://petr-mitrichev.
    blogspot.com/2013/05/fenwick-tree-range-
    updates.html
int dataMul[N+1];
int dataAdd[N+1]
void internalUpdate(int at, int mul, int add) {
    while (at <= N) {
        dataMul[at] += mul;
        dataAdd[at] += add;
        at |= (at + 1);
    }
}

void update(int left, int right, int by) {
    internalUpdate(left, by, -by * (left - 1));
    internalUpdate(right, -by, by * right);
}

 int query(int at) {
    int mul = 0;
    int add = 0;
    int start = at;
    while (at >= 0) {
        mul += dataMul[at];
        add += dataAdd[at];
        at = (at & (at + 1)) - 1;
    }
    return mul * start + add;
}
```

6 Miscellaneous

6.1 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 <iostream>
#include <string>

using namespace std;

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

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

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

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

// converts integer (Julian day number) to day
    of week
string intToDay (int jd){
```

```
  return dayOfWeek[jd % 7];
}

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

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

6.2 Prime numbers

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

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

6.3 C++ input/output

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

using namespace std;

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

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

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

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

6.4 Knuth-Morris-Pratt

```
/*
Finds all occurrences of the pattern string p
    within the
text string t. Running time is O(n + m), where n
    and m
are the lengths of p and t, respecitvely.
*/

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

using namespace std;

typedef vector<int> VI;

void buildPi(string& p, VI& pi)
{
  pi = VI(p.length());
  int k = -2;
  for(int i = 0; i < p.length(); i++) {
    while(k >= -1 && p[k+1] != p[i])
      k = (k == -1) ? -2 : pi[k];
    pi[i] = ++k;
  }
}

int KMP(string& t, string& p)
{
  VI pi;
  buildPi(p, pi);
  int k = -1;
  for(int i = 0; i < t.length(); i++) {
    while(k >= -1 && p[k+1] != t[i])
      k = (k == -1) ? -2 : pi[k];
    k++;
    if(k == p.length() - 1) {
      // p matches t[i-m+1, ..., i]
      cout << "matched at index " << i-k << ": "
          ;
      cout << t.substr(i-k, p.length()) << endl;
      k = (k == -1) ? -2 : pi[k];
    }
  }
  return 0;
}

int main()
{
  string a = "AABAACAADAABAABA", b = "AABA";
  KMP(a, b); // expected matches at: 0, 9, 12
  return 0;
}
```

6.5 Latitude/longitude

```
/*
Converts from rectangular coordinates to
    latitude/longitude and vice
versa. Uses degrees (not radians).
```

```
*/

#include <iostream>
#include <cmath>

using namespace std;

struct ll
{
  double r, lat, lon;
};

struct rect
{
  double x, y, z;
};

ll convert(rect& P)
{
  ll Q;
  Q.r = sqrt(P.x*P.x+P.y*P.y+P.z*P.z);
  Q.lat = 180/M_PI*asin(P.z/Q.r);
  Q.lon = 180/M_PI*acos(P.x/sqrt(P.x*P.x+P.y*P.y
      ));

  return Q;
}

rect convert(ll& Q)
{
  rect P;
  P.x = Q.r*cos(Q.lon*M_PI/180)*cos(Q.lat*M_PI
      /180);
  P.y = Q.r*sin(Q.lon*M_PI/180)*cos(Q.lat*M_PI
      /180);
  P.z = Q.r*sin(Q.lat*M_PI/180);

  return P;
}

int main()
{
  rect A;
  ll B;

  A.x = -1.0; A.y = 2.0; A.z = -3.0;

  B = convert(A);
  cout << B.r << " " << B.lat << " " << B.lon <<
       endl;

  A = convert(B);
  cout << A.x << " " << A.y << " " << A.z <<
      endl;
}
```

6.6 Dates (Java)

```
// Example of using Java's built-in date
    calculation routines

import java.text.SimpleDateFormat;
import java.util.*;

public class Dates {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        SimpleDateFormat sdf = new
            SimpleDateFormat("M/d/yyyy");
        while (true) {
            int n = s.nextInt();
            if (n == 0) break;
            GregorianCalendar c = new
                GregorianCalendar(n, Calendar.
                JANUARY, 1);
            while (c.get(Calendar.DAY_OF_WEEK)
                 != Calendar.SATURDAY)
                c.add(Calendar.DAY_OF_YEAR, 1);
            for (int i = 0; i < 12; i++) {
                System.out.println(sdf.format(c.
                    getTime()));
                while (c.get(Calendar.MONTH) ==
                    i) c.add(Calendar.
                    DAY_OF_YEAR, 7);
            }
        }
    }
}
```

7 Hussain

7.1 Adaptive Simpson

```
//
// Adaptive Simpson's Rule (Wikipedia Article)
//
dbl adaptiveSimpsons(dbl (*f)(dbl),   // ptr to
    function
   dbl a, dbl b,  // interval [a,b]
   dbl epsilon,  // error tolerance
   int maxRecursionDepth) {   // recursion cap
  dbl c = (a + b)/2, h = b - a;
  dbl fa = f(a), fb = f(b), fc = f(c);
  dbl S = (h/6)*(fa + 4*fc + fb);
  return adaptiveSimpsonsAux(f, a, b, epsilon, S
      , fa, fb, fc, maxRecursionDepth);
}

//
// Recursive auxiliary function for
    adaptiveSimpsons() function below
//
dbl adaptiveSimpsonsAux(dbl (*f)(dbl), dbl a,
    dbl b, dbl epsilon,
    dbl S, dbl fa, dbl fb, dbl fc, int bottom) {
  dbl c = (a + b)/2, h = b - a;
  dbl d = (a + c)/2, e = (c + b)/2;
  dbl fd = f(d), fe = f(e);
  dbl Sleft = (h/12)*(fa + 4*fd + fc);
  dbl Sright = (h/12)*(fc + 4*fe + fb);
  dbl S2 = Sleft + Sright;
  if (bottom <= 0 || fabs(S2 - S) <= 15*epsilon)
          // magic 15 comes from error analysis
    return S2 + (S2 - S)/15;
  return adaptiveSimpsonsAux(f, a, c, epsilon/2,
       Sleft,  fa, fc, fd, bottom-1) +
         adaptiveSimpsonsAux(f, c, b, epsilon/2,
             Sright, fc, fb, fe, bottom-1);
}

int main(){
// compute integral of sin(x)
// from 0 to 2 and store it in
// the new variable I
 float I = adaptiveSimpsons(sin, 0, 2, 0.001,
     100);
 printf("I = %lf\n",I); // print the result
 return 0;
}
```

7.2 Binomial Coeff (constant N)

```
C[0] = 1
for (int k = 0; k < n; ++ k)
    C[k+1] = (C[k] * (n-k)) / (k+1)
// C[i] = C(n,i)
```

7.3 Generate (x,y) pairs s.t. x AND y=y

```
for(int x = 1; x <= n; x++)
    for(int y = x; y; y = (y-1)&x)
        cout<<y<<endl;
```

7.4 Index of LSB

```
int msb(unsigned x) {
union { double a; int b[2]; };
a = x;
return (b[1] >> 20) - 1023;
}
```

8 Malek

8.1 Finding bridges in graph

```
int dfslow[N];
int dfsnum[N];
int dfscnt = 1;
vector<int> adj[N];
void dfs(int u, int p) {
  dfslow[u] = dfsnum[u] = dfscnt++;
  for (int i = 0; i < adj[u].size(); i++) {
    int v = adj[u][i];
    if (!dfsnum[v]) {
      dfs(v, u);
      if(dfslow[v]>dfsnum[u]){
        //it's a bridge
      }
      dfslow[u]=min(dfslow[u],dfslow[v]);
    }else if(v!=p){
      //back edge
      dfslow[u]=min(dfslow[u],dfsnum[v]);
    }
  }
}
```

8.2 LCA(Sparse Table) and Centroid Decomposition

```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef vector<int> vi;


#define lp(i,n) for(int i=0;i<(int)n;i++)
#define lp1(i,n)for(int i=1;i<=(int)n;i++)
const int N = 1e5 + 5;
const int LOGN = 20;
vi adj[N];
int dp[LOGN][N]; //sparse
int level[N];
int n;
int cs; //composition size
int sub[N];
bool cen[N];
int par[N];
int ans[N];
void dfs1(int u) {
  lp(i,adj[u].size())
  {
    int v = adj[u][i];
    if (v != dp[0][u]) {
      level[v] = level[u] + 1;
      dp[0][v] = u;
      dfs1(v);
    }
  }
}
void sparse() {
  dfs1(0);
  lp1(i,LOGN-1)
  {
    lp(j,n)
    {
      dp[i][j] = dp[i - 1][dp[i - 1][j]];
    }
  }

}
int lca(int a, int b) {
  if (level[a] > level[b])
    swap(a, b);
  int dif = level[b] - level[a];
  lp(i,LOGN)
  {
    if (dif & (1 << i))
      b = dp[i][b];
  }
  if (a == b)
    return b;
  for (int i = LOGN - 1; i >= 0; i--) {
    if (dp[i][a] != dp[i][b])
      a = dp[i][a],b=dp[i][b];
  }
  return dp[0][a];
}
int dist(int a, int b) {
  return level[a] + level[b] - 2 * level[lca(a,
      b)];
}
/*----rot&decay--------*/
void dfssub(int u, int p) {
  sub[u] = 1;
  cs++;
  lp(i,adj[u].size())
  {
    int v = adj[u][i];
    if (!cen[v] && v != p)
      dfssub(v, u), sub[u] += sub[v];
  }
}
int dfscen(int u, int p) {
  lp(i,adj[u].size())
  {
    int v = adj[u][i];
    if (!cen[v] && v != p&&sub[v]>cs/2)
      return dfscen(v, u);
  }
  return u;
}
void decomp(int root, int p) {
  cs = 0;
  dfssub(root, p);
  int centroid = dfscen(root, p);
//  cout<<centroid<<endl;
  cen[centroid] = 1;
  par[centroid] = p;
  lp(i,adj[centroid].size())
  {
    if (!cen[adj[centroid][i]])
      decomp(adj[centroid][i], centroid);
  }

}
void update(int u) {
  int x = u;
  while (x != -1) {
    ans[x] = min(ans[x], dist(u, x));
    x = par[x];
  }

}
int query(int u) {
  int x = u;
  int mn = ans[u];
  while (x != -1) {
    mn = min(mn, ans[x] + dist(u, x));
    x = par[x];
  }
  return mn;
}
int main() {
  int m;
  sii(n, m);
  {
    int x,y;
    lp(i,n-1)
    {
      sii(x,y);
      x--;y--;
      adj[x].push_back(y);
      adj[y].push_back(x);
    }
  }
  sparse();
  decomp(0, -1);
  lp(i,n)ans[i]=1e9;
  update(0);
  while(m--){
//    int x,y;
//    cin>>x>>y;
//    cout<<dist(x,y)<<endl;
    int t,u;
    sii(t,u);
    u--;
    if(t==1)update(u);
    else printf("%d\n",query(u));
  }

}
```

8.3 Convex Hull Trick

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

bool query_flag = false;
struct line {
  long long m, c;
  mutable function<const line*()> succ;
  bool operator<(const line& o) const {
    if (!query_flag) return m < o.m;
    const line* s = succ();
    if (!s) return false;
    return (c - s->c) < (s->m - m) * o.m;
  }
};

struct maximum_hull : multiset<line> {
  bool bad(iterator y) {
    auto x = (y == begin()) ? end() : prev(y), z
        = next(y);
    if (x == end() && z == end()) return false;
    else if (x == end()) return y->m == z->m &&
        y->c <= z->c;
    else if (z == end()) return y->m == x->m &&
        y->c <= x->c;
    else return (x->c - y->c) * (z->m - y->m) >=
          (y->c - z->c) * (y->m - x->m);
  }
  void insert_line(const long long& m, const
      long long& c) {
    auto y = insert({ m, c, nullptr });
    y->succ = [=] { return next(y) == end() ?
        nullptr : &*next(y); };
    if (bad(y)) { erase(y); return; }
    iterator z;
    while ((z = next(y)) != end() && bad(z))
        erase(z);
    while (y != begin() && bad(z = prev(y)))
        erase(z);
  }
  long long eval(long long x) {
    if (empty()) return numeric_limits<ll>::min
        ();
    query_flag = true;
    auto l = *lower_bound({ x, 0, nullptr });
    query_flag = false;
    return l.m * x + l.c;
  }
};
struct minimum_hull: maximum_hull {
    void insert_line(ll a, ll b) {
        maximum_hull::insert_line(-a, -b);
    }
    ll eval(ll x) {
```

```
        return -maximum_hull::eval(x);
    }
};
```

8.4 Convex Hull Trick (stack)

```
int pointer; //Keeps track of the best line from
    previous query
vector<long long> M; //Holds the slopes of the
    lines in the envelope
vector<long long> B; //Holds the y-intercepts of
    the lines in the envelope
//Returns true if either line l1 or line l3 is
    always better than line l2
bool bad(int l1,int l2,int l3)
{
        /*
        intersection(l1,l2) has x-coordinate (b1
            -b2)/(m2-m1)
        intersection(l1,l3) has x-coordinate (b1
            -b3)/(m3-m1)
        set the former greater than the latter,
            and cross-multiply to
        eliminate division
        */
        return (B[l3]-B[l1])*(M[l1]-M[l2])<(B[l2
            ]-B[l1])*(M[l1]-M[l3]);
}
//Adds a new line (with lowest slope) to the
    structure
void add(long long m,long long b)
{
        //First, let's add it to the end
        M.push_back(m);
        B.push_back(b);
        //If the penultimate is now made
            irrelevant between the
            antepenultimate
        //and the ultimate, remove it. Repeat as
            many times as necessary
        while (M.size()>=3&&bad(M.size()-3,M.
            size()-2,M.size()-1))
        {
                M.erase(M.end()-2);
                B.erase(B.end()-2);
        }
}
//Returns the minimum y-coordinate of any
    intersection between a given vertical
//line and the lower envelope
long long query(long long x)
{
        //If we removed what was the best line
            for the previous query, then the
        //newly inserted line is now the best
            for that query
        if (pointer>=M.size())
                pointer=M.size()-1;
        //Any better line must be to the right,
            since query values are
        //non-decreasing
        while (pointer<M.size()-1&&
          M[pointer+1]*x+B[pointer+1]<M[pointer
              ]*x+B[pointer])
                pointer++;
        return M[pointer]*x+B[pointer];
}
```

9 Marsil

9.1 2D geomtry using Complex

```
src: http://codeforces.com/blog/entry/22175
Functions using std::complex
1)  Vector addition: a + b
2)  Scalar multiplication: r * a
3)  Dot product: (conj(a) * b).x
4)  Cross product: (conj(a) * b).y
5)  notice: conj(a) * b = (ax*bx + ay*by) + i (
     ax*by     ay*bx)
6)  Squared distance: norm(a - b)
7)  Euclidean distance: abs(a - b)
8)  Angle of elevation: arg(b - a)
9)  Slope of line (a, b): tan(arg(b - a))
10) Polar to cartesian: polar(r, theta)
11) Cartesian to polar: point(abs(p), arg(p))
12) Rotation about the origin: a * polar(1.0,
     theta)
13) Rotation about pivot p: (a-p) * polar(1.0,
     theta) + p
14) Angle ABC: abs(remainder(arg(a-b) - arg(c-b)
     , 2.0 * M_PI))
      remainder normalizes the angle to be
           between [-PI, PI]. Thus, we can get
           the positive non-reflex angle by
           taking its abs value.
15) Project p onto vector v: v * dot(p, v) /
     norm(v);
16) Project p onto line (a, b): a + (b - a) *
     dot(p - a, b - a) / norm(b - a)
17) Reflect p across line (a, b): a + conj((p -
     a) / (b - a)) * (b - a)
18) Intersection of line (a, b) and (p, q):
19)
point intersection(point a, point b, point p,
     point q) {
  double c1 = cross(p - a, b - a), c2 = cross(q
       - a, b - a);
  return (c1 * q - c2 * p) / (c1 - c2); //
       undefined if parallel
}

Drawbacks:
Using std::complex is very advantageous, but it
     has one disadvantage: you can't use std::
     cin or scanf. Also, if we macro x and y, we
      can't use them as variables. But that's
     rather minor, don't you think?
EDIT: Credits to Zlobober for pointing out that
     std::complex has issues with integral data
     types. The library will work for simple
     arithmetic like vector addition and such,
     but not for polar or abs. It will compile
     but there will be some errors in
     correctness! The tip then is to rely on the
      library only if you're using floating
     point data all throughout.
```

9.2 bottom up lasy segment tree

```
template<typename T, typename U> struct
     seg_tree_lazy {
    int S, H;

    T zero;
    vector<T> value;

    U noop;
    vector<bool> dirty;
    vector<U> prop;

    seg_tree_lazy<T, U>(int _S, T _zero = T(), U
         _noop = U()) {
        zero = _zero, noop = _noop;
        for (S = 1, H = 1; S < _S; ) S *= 2, H
             ++;

        value.resize(2*S, zero);
        dirty.resize(2*S, false);
        prop.resize(2*S, noop);
    }

    void set_leaves(vector<T> &leaves) {
        copy(leaves.begin(), leaves.end(), value
             .begin() + S);

        for (int i = S - 1; i > 0; i--)
            value[i] = value[2 * i] + value[2 *
                 i + 1];
    }

    void apply(int i, U &update) {
        value[i] = update(value[i]);
        if(i < S) {
            prop[i] = prop[i] + update;
            dirty[i] = true;
        }
    }

    void rebuild(int i) {
        for (int l = i/2; l; l /= 2) {
            T combined = value[2*l] + value[2*l
                 +1];
            value[l] = prop[l](combined);
        }
    }

    void propagate(int i) {
        for (int h = H; h > 0; h--) {
            int l = i >> h;

            if (dirty[l]) {
                apply(2*l, prop[l]);
                apply(2*l+1, prop[l]);

                prop[l] = noop;
                dirty[l] = false;
            }
        }
    }

    void upd(int i, int j, U update) {
        i += S, j += S;
        propagate(i), propagate(j);
```

```
        for (int l = i, r = j; l <= r; l /= 2, r
             /= 2) {
            if((l&1) == 1) apply(l++, update);
            if((r&1) == 0) apply(r--, update);
        }

        rebuild(i), rebuild(j);
    }

    T query(int i, int j){
        i += S, j += S;
        propagate(i), propagate(j);

        T res_left = zero, res_right = zero;
        for(; i <= j; i /= 2, j /= 2){
            if((i&1) == 1) res_left = res_left +
                 value[i++];
            if((j&1) == 0) res_right = value[j
                --] + res_right;
        }
        return res_left + res_right;
    }
};
/*
As an example, let's see how to use it to
     support the follow operations:

    Type 1: Add amount V to the values in range
         [L, R].
    Type 2: Reset the values in range [L, R] to
         value V.
    Type 3: Query for the sum of the values in
         range [L, R].
*/
//The T struct would look like this:

struct node {
    int sum, width;

    node operator+(const node &n) {
        return { sum + n.sum, width + n.width };
    }
};
//And the U struct would look like this:

struct update {
    bool type; // 0 for add, 1 for reset
    int value;

    node operator()(const node &n) {
        if (type) return { n.width * value, n.
            width };
        else return { n.sum + n.width * value, n
            .width };
    }

    update operator+(const update &u) {
        if (u.type) return u;
        return { type, value + u.value };
    }
};
```

9.3 Ordered Statistics Tree

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace __gnu_pbds;

typedef tree<
double,
int,
less<double>,
rb_tree_tag,
tree_order_statistics_node_update> map_t;


typedef tree<
int,
null_type,
less<int>,
rb_tree_tag,
tree_order_statistics_node_update> ordered_set;

int main(){
    map_t a;
    a[0] = 1;
    a[2] = 1;
    a[5] = 1;
    cout << a.find_by_order(1)->first <<endl;
    cout << a.order_of_key(-5) << endl;
}
```

10 Laws

Triangle

$inradius = \sqrt{\frac{(s-a)(s-b)(s-c)}{s}}$

$exradius = \sqrt{\frac{s(s-b)(s-c)}{(s-a)}}$

Sphere

$V = \frac{4}{3}\pi r^3$, $SA = 4\pi r^2$

Spherical cap

$V = \frac{\pi h^2}{3}\left(3r - h\right)$, $SA = 2\pi rh$

Cone/Pyramid [1]

$V = \frac{1}{3}Bh$, $SA = B + \frac{1}{2}c\ell$

Circular truncated cone

$V = \frac{1}{3}\pi\left(r_1^2 + r_1 r_2 + r_2^2\right)$

Lateral Area: $F = \pi\left(r_1 + r_2\right)\sqrt{\left(r_1 - r_2\right)^2 + h^2}$

Surface Area: $SA = F + \pi\left(r_1^2 + r_2^2\right)$

Truncated Pyramid [2]

$V = \frac{1}{6}\left(ab + (a + c) \times (b + d) + cd\right)$

[1] B is the area of the base, h is the height,while ℓ is the slant height(Cone only).

[2] a and c are parallel, just like b and d.