

Stanford University ICPC Team Notebook (2015-16)

Contents

1 Combinatorial optimization

1.1 Dense max-flow

// Adjacency matrix implementation of Dinic's blocking flow algorithm.

```
// Running time:
// O(VV)
// INPUT:
// - graph, constructed using AddEdge()
// - source
// - sink
// OUTPUT:
// - maximum flow 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;
const int INF = 10000000000;

struct MaxFlow {
    int N;
    VVI cap, flow;
    VI dad, Q;

    MaxFlow(int N) :
        N(N), cap(N, VI(N)), flow(N, VI(N)), dad(N, Q(N)) {}

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

    int BlockingFlow(int s, int t) {
        fill(dad.begin(), dad.end(), -1);
        dad[s] = -2;

        int head = 0, tail = 0;
        Q[tail++] = s;
        while (head < tail) {
            int x = Q[head++];
            for (int i = 0; i < N; i++) {
                if (dad[i] == -1 && cap[x][i] - flow[x][i] > 0) {
                    dad[i] = x;
                    Q[tail++] = i;
                }
            }
        }

        if (dad[t] == -1) return 0;

        int totflow = 0;
        for (int i = 0; i < N; i++) {
            if (dad[i] == -1) continue;

```

```
int amt = cap[i][t] - flow[i][t];
for (int j = i; amt && j != s; j = dad[j])
    amt = min(amt, cap[dad[j]][j] - flow[dad[j]][j]);
if (amt == 0) continue;
flow[i][t] += amt;
flow[t][i] -= amt;
for (int j = i; j != s; j = dad[j]) {
    flow[dad[j]][j] += amt;
    flow[j][dad[j]] -= amt;
}
totflow += amt;
}

return totflow;
}

int GetMaxFlow(int source, int sink) {
    int totflow = 0;
    while (int flow = BlockingFlow(source, sink))
        totflow += flow;
    return totflow;
};

int main() {
    MaxFlow mf(5);
    mf.AddEdge(0, 1, 3);
    mf.AddEdge(0, 2, 4);
    mf.AddEdge(0, 3, 5);
    mf.AddEdge(0, 4, 5);
    mf.AddEdge(1, 2, 2);
    mf.AddEdge(2, 3, 4);
    mf.AddEdge(2, 4, 1);
    mf.AddEdge(3, 4, 10);

    // should print out "15"
    cout << mf.GetMaxFlow(0, 4) << endl;
}

// BEGIN CUT
// The following code solves SPOJ problem #203: Potholders (POTHOLES)
#ifdef COMMENT
int main() {
    int t;
    cin >> t;
    for (int i = 0; i < t; i++) {
        int n;
        cin >> n;
        MaxFlow mf(n);
        for (int j = 0; j < n-1; j++) {
            int m;
            cin >> m;
            for (int k = 0; k < m; k++) {
                int p;
                cin >> p;
                p--;
                int cap = (j == 0 || p == n-1) ? 1 : INF;
                mf.AddEdge(j, p, cap);
            }
        }
        cout << mf.GetMaxFlow(0, n-1) << endl;
    }
    return 0;
}
#endif
// END CUT

```

1.2 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|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<stdio>
#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(u, v, cap);
            g[u].emplace_back(E.size() - 1);
            E.emplace_back(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]
            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%d", &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
```

1.3 Min-cost max-flow

```
// 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 cost per augmentation
//   max flow: O(V augmentations
//   min cost max flow: O(V * 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(k, s, 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 < N; 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

```

1.4 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 100000 edges in a few
// seconds, though it is possible to construct test cases that
// achieve the worst-case.
//
// Running time:
// O(V
//
// 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>

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;
}

```

1.6 Max bipartite machine

```

// 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;
}

}

}

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

```

1.8 Graph cut inference

```

// Special-purpose {0,1} combinatorial optimization solver for
// problems of the following by a reduction to graph cuts:
//
// minimize      sum_i psi_i(x[i])
// x[1]...x[n] in {0,1}      + sum_{i < j} phi_{ij}(x[i], x[j])
//
// where
//   psi_i : {0, 1} --> R
//   phi_{ij} : {0, 1} x {0, 1} --> R
//
// such that
//   phi_{ij}(0,0) + phi_{ij}(1,1) <= phi_{ij}(0,1) + phi_{ij}(1,0)
//   (*)
//
// This can also be used to solve maximization problems where the
// direction of the inequality in (*) is reversed.
//
// INPUT: phi -- a matrix such that phi[i][j][u][v] = phi_{ij}(u, v)
//        psi -- a matrix such that psi[i][u] = psi_i(u)
//        x -- a vector where the optimal solution will be stored
//
// OUTPUT: value of the optimal solution
//
// To use this code, create a GraphCutInference object, and call the
// DoInference() method. To perform maximization instead of
// minimization,
// ensure that #define MAXIMIZATION is enabled.
//
#include <vector>
#include <iostream>
using namespace std;
typedef vector<int> VI;
typedef vector<VI> VVI;
typedef vector<VVI> VVVI;
typedef vector<VVVI> VVVVI;
const int INF = 1000000000;

```

1.7 Global min-cut

Adjacency matrix implementation of Stoer-Wagner min cut algorithm.

```

// Running time:
// O(V^4)
//
// 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;
            }
        }
    }
}

```

```

// comment out following line for minimization
#define MAXIMIZATION

struct GraphCutInference {
    int N;
    VVI cap, flow;
    VI reached;

    int Augment(int s, int t, int a) {
        reached[s] = 1;
        if (s == t) return a;
        for (int k = 0; k < N; k++) {
            if (reached[k]) continue;
            if (int aa = min(a, cap[s][k] - flow[s][k])) {
                flow[s][k] += b;
                flow[k][s] -= b;
                return b;
            }
        }
    }

    return 0;
}

int GetMaxFlow(int s, int t) {
    N = cap.size();
    flow = VVI(N, VI(N));
    reached = VI(N);

    int totflow = 0;
    while (int amt = Augment(s, t, INF)) {
        totflow += amt;
        fill(reached.begin(), reached.end(), 0);
    }
    return totflow;
}

int DoInference(const VVVI &phi, const VVI &psi, VI &x) {
    int M = phi.size();
    cap = VVI(M+2, VI(M+2));
    VI b(M);
    int c = 0;

    for (int i = 0; i < M; i++) {
        b[i] += psi[i][1] - psi[i][0];
        c += psi[i][0];
        for (int j = 0; j < i; j++)
            b[i] += phi[i][j][1][1] - phi[i][j][j][0][1];
        for (int j = i+1; j < M; j++) {
            cap[i][j] = phi[i][j][0][1] + phi[i][j][1][1][0] - phi[i][j][j][0][0] - phi[i][j][1][1];
            b[i] += phi[i][j][1][1][0] - phi[i][j][j][0][0];
            c += phi[i][j][0][0];
        }
    }

    #ifndef MAXIMIZATION
    for (int i = 0; i < M; i++) {
        for (int j = i+1; j < M; j++)
            cap[i][j] *= -1;
        b[i] *= -1;
    }
    c *= -1;
    #endif

    for (int i = 0; i < M; i++) {
        if (b[i] >= 0) {
            cap[M][i] = b[i];
        } else {
            cap[i][M+1] = -b[i];
            c += b[i];
        }
    }

    int score = GetMaxFlow(M, M+1);
    fill(reached.begin(), reached.end(), 0);
    Augment(M, M+1, INF);
    x = VI(M);
    for (int i = 0; i < M; i++) x[i] = reached[i] ? 0 : 1;
    score += c;
    #ifndef MAXIMIZATION
    score *= -1;
    #endif
    return score;
}

int main() {
    // solver for "Cat vs. Dog" from NWERC 2008

    int numcases;
    cin >> numcases;
    for (int caseno = 0; caseno < numcases; caseno++) {
        int c, d, v;
        cin >> c >> d >> v;

        VVVI phi(c+d, VVI(c+d, VVI(2, VI(2))));
        VVI psi(c+d, VI(2));
        for (int i = 0; i < v; i++) {
            char p, q;
            int u, v;
            cin >> p >> u >> q >> v;
            u--; v--;
            if (p == 'C') {
                phi[u][c+v][0][0]++;
                phi[c+v][u][0][0]++;
            } else {
                phi[v][c+u][1][1]++;
                phi[c+u][v][1][1]++;
            }
        }

        GraphCutInference graph;
        VI x;
        cout << graph.DoInference(phi, psi, x) << endl;
    }
    return 0;
}

```

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
//
// 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(const PT p, PT q) { return p.x*q.x+p.y*q.y; }
double dist2(const PT p, PT q) { return dot(p-q,p-q); }
double cross(const 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(const PT p) { return PT(-p.y, p.x); }
PT RotateCW90(const PT p) { return PT(p.y, -p.x); }
PT RotateCCW(const 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(const 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(const 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(const 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(const PT a, PT b, PT c, PT d) {
    return fabs(cross(b-a, c-d)) < EPS;
}

bool LinesCollinear(const 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(const 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(const PT a, PT b, PT c, PT d) {
    b=b-a; d=d-c; 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(const PT a, PT b, PT c) {
    b=(a+b)/2;
    c=(a+c)/2;
    return ComputeLineIntersection(b, b+RotateCW90(a-b), c, c+RotateCW90(b-a));
}

// 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) <
            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(const PT a, PT b, PT c, double r) {
    vector<PT> ret;
    b = b-a;
    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(const PT a, PT b, double r, double R) {

```

```

vector<PT> ret;
double d = sqrt(dist2(a, b));
if (d > r+R || dmin(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].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 == 1 || 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, 1.1666666)
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 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;
}

```

2.4 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));
        }
    }
}

```

2.5 Slow Delaunay triangulation

// Slow but simple Delaunay triangulation. Does not handle

```

// degenerate cases (from O'Rourke, Computational Geometry in C)
//
// Running time: O(n)
//
// INPUT:  x[] = x-coordinates
//         y[] = y-coordinates
//
// OUTPUT: triples = a vector containing m triples of indices
//               corresponding to triangle vertices
//
#include<vector>
using namespace std;
typedef double T;
struct triple {
    int i, j, k;
    triple() {}
    triple(int i, int j, int k) : i(i), j(j), k(k) {}
};

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

int main()
{
    T xs[] = {0, 0, 1, 0, 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;
}

```

```

// In this example, we read an input file containing three lines, each
// containing an even number of doubles, separated by commas. The
// first two
// lines represent the coordinates of two polygons, given in
// counterclockwise
// (or clockwise) order, which we will call "A" and "B". The last
// line
// contains a list of points, p[1], p[2], ...
//
// Our goal is to determine:
// (1) whether B - A is a single closed shape (as opposed to
// multiple shapes)
// (2) the area of B - A
// (3) whether each p[i] is in the interior of B - A
//
// INPUT:
// 0 0 10 0 0 10
// 0 0 10 10 10 0
// 8 6
// 5 1
//
// OUTPUT:
// The area is singular.
// The area is 25.0
// Point belongs to the area.
// Point does not belong to the area.
//
import java.util.*;
import java.awt.geom.*;
import java.io.*;

public class JavaGeometry {
    // make an array of doubles from a string
    static double[] readPoints(String s) {
        String[] arr = s.trim().split("\\s+");
        double[] ret = new double[arr.length];
        for (int i = 0; i < arr.length; i++) ret[i] = Double.
            parseDouble(arr[i]);
        return ret;
    }

    // make an Area object from the coordinates of a polygon
    static Area makeArea(double[] pts) {
        Path2D.Double p = new Path2D.Double();
        p.moveTo(pts[0], pts[1]);
        for (int i = 2; i < pts.length; i += 2) p.lineTo(pts[i], pts[i
            +1]);
        p.closePath();
        return new Area(p);
    }

    // compute area of polygon
    static double computePolygonArea(ArrayList<Point2D.Double> points)
    {
        Point2D.Double[] pts = points.toArray(new Point2D.Double[
            points.size()]);
        double area = 0;
        for (int i = 0; i < pts.length; i++) {
            int j = (i+1) % pts.length;
            area += pts[i].x * pts[j].y - pts[j].x * pts[i].y;
        }
        return Math.abs(area)/2;
    }

    // compute the area of an Area object containing several disjoint
    // polygons
    static double computeArea(Area area) {
        double totArea = 0;
        PathIterator iter = area.getPathIterator(null);
        ArrayList<Point2D.Double> points = new ArrayList<Point2D.

```

2.6 Java geometry

```

Double>();
while (!iter.isDone()) {
    double[] buffer = new double[6];
    switch (iter.currentSegment(buffer)) {
        case PathIterator.SEG_MOVETO:
        case PathIterator.SEG_LINETO:
            points.add(new Point2D.Double(buffer[0], buffer[1]));
            break;
        case PathIterator.SEG_CLOSE:
            totArea += computePolygonArea(points);
            points.clear();
            break;
    }
    iter.next();
}
return totArea;
}

// notice that the main() throws an Exception -- necessary to
// avoid wrapping the Scanner object for file reading in a
// try { ... } catch block.
public static void main(String args[]) throws Exception {
    Scanner scanner = new Scanner(new File("input.txt"));
    // also,
    // Scanner scanner = new Scanner (System.in);

    double[] pointsA = readPoints(scanner.nextLine());
    double[] pointsB = readPoints(scanner.nextLine());
    Area areaA = makeArea(pointsA);
    Area areaB = makeArea(pointsB);
    areaB.subtract(areaA);
    // also,
    // areaB.exclusiveOr (areaA);
    // areaB.add (areaA);
    // areaB.intersect (areaA);

    // (1) determine whether B - A is a single closed shape (as
    // opposed to multiple shapes)
    boolean issingle = areaB.isSingular();
    // also,
    // areaB.isEmpty();

    if (issingle)
        System.out.println("The area is singular.");
    else
        System.out.println("The area is not singular.");

    // (2) compute the area of B - A
    System.out.println("The area is " + computeArea(areaB) + ".");

    // (3) determine whether each p[i] is in the interior of B - A
    while (scanner.hasNextDouble()) {
        double x = scanner.nextDouble();
        assert(scanner.hasNextDouble());
        double y = scanner.nextDouble();

        if (areaB.contains(x,y)) {
            System.out.println ("Point belongs to the area.");
        } else {
            System.out.println ("Point does not belong to the area
            .");
        }
    }

    // Finally, some useful things we didn't use in this example:
    //
    // Ellipse2D.Double ellipse = new Ellipse2D.Double (double x
    // , double y,
    // , double w
    // , double h);

```

```

//
//      creates an ellipse inscribed in box with bottom-left
//      corner (x,y)
//      and upper-right corner (x+y,w+h)
//
//      Rectangle2D.Double rect = new Rectangle2D.Double (double
//      x, double y,
//      w, double h);
//
//      creates a box with bottom-left corner (x,y) and upper-
//      right
//      corner (x+y,w+h)
//
//      Each of these can be embedded in an Area object (e.g., new
//      Area (rect)).
    }
}

```

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>
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;
}

// (ab) 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));
    }
}

}

// 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 Prime numbers

```

// O(sqrt(x)) Exhaustive Primality Test
#include <cmath>
#define EPS 1e-7
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;
}

// O(n) fast generate prime number list
const ll limit = 1000000; // prime number upper bound
bool prime[limit+1]; // bool of sequence is prime or not
vector<ll> primes; // list of prime in order

void generateprimes() {
    for (ll i = 0; i < limit; i++)
        prime[i] = false;
    prime[2] = true;
    prime[3] = true;
    for (ll x = 1; x * x < limit; x++) {
        for (ll y = 1; y * y < limit; y++) {
            ll n = (4 * x * x) + (y * y);
            if (n <= limit && (n % 12 == 1 || n % 12 == 5))
                prime[n] = true;
            n = (3 * x * x) + (y * y);
            if (n <= limit && n % 12 == 7)
                prime[n] = true;
            n = (3 * x * x) - (y * y);
            if (x > y && n <= limit && n % 12 == 11)

```

```

        prime[n] = true;
    }
    for (ll r = 5; r * r < limit; r++) {
        if (prime[r]) {
            for (ll i = r * r; i < limit; i += r * r)
                prime[i] = false;
        }
        for (ll i=2; i<limit; i++) if (prime[i]) primes.push_back(i);
    }
    // O(nlog(n)) return number of coprime pair in set [a,b] [c,d]
    ll coprime(ll a,b,c,d) {
        N = max(b, d);
        int mu[N];
        for (ll i=0; i<=N; i++) mu[i] = 1;
        for (auto p : primes) {
            for (ll i=1; i*p <= N; i++) {
                mu[i*p] *= -1;
            }
            ll pp = p*p;
            for (ll i=1; i*pp <= N; i++) {
                mu[i*pp] = 0;
            }
        }
        ll sum = 0;
        for (ll i=1; i<=N; i++) {
            sum += mu[i] * (b/i - (a-1)/i) * (d/i - (c-1)/i);
        }
        return ll;
    }
}

```

```

// 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 9991.
// The largest prime smaller than 1000000 is 99983.
// The largest prime smaller than 10000000 is 999991.
// The largest prime smaller than 100000000 is 99999937.
// 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 99999999989.
// The largest prime smaller than 10000000000000 is 999999999971.
// The largest prime smaller than 100000000000000 is
999999999993.
// The largest prime smaller than 1000000000000000 is
9999999999989.
// The largest prime smaller than 10000000000000000 is
99999999999937.
// The largest prime smaller than 100000000000000000 is
99999999999997.
// The largest prime smaller than 1000000000000000000 is
999999999999989.
```

3.3 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)
//
// INPUT:  a[][] = an nxn matrix
//          b[][] = an nxm matrix
//
// OUTPUT:  X      = an nxm matrix (stored in b[][])
//          A      = 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(m);
    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 0.066667
// 0.166667 0.166667 0.333333 -0.333333
// 0.233333 0.833333 -0.133333 -0.066667
// 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;
}
}
```

3.4 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
//
// 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;
    }
}

```

3.5 Fast Fourier transform

```

#include <cassert>
#include <stdio>
#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;
}

```

3.6 Simplex algorithm

```

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

using namespace std;

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

const DOUBLE EPS = 1e-9;

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

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

    void Pivot(int r, int s) {
        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 i = 0; i < 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;
}

```

4 Graph algorithms

4.1 Bellman-Ford shortest paths with negative edge weights

```
// 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(VV)
//
// 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;
}
```

4.2 Dijkstra and Floyd's algorithm

```
#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;

// This function runs Dijkstra's algorithm for single source
// shortest paths. No negative cycles allowed!
//
// Running time: O(VV)
//
// 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
//
void Dijkstra (const VVT &w, VT &dist, VI &prev, int start) {
    int n = w.size();
    VI found (n);
    prev = VI(n, -1);
    dist = VT(n, 1000000000);
    dist[start] = 0;

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

// 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(VV)
//
// INPUT: w[i][j] = weight of edge from i to j
// OUTPUT: w[i][j] = shortest path from i to j
//         prev[i][j] = node before j on the best path starting at i
//
bool FloydWarshall (VVT &w, VVI &prev) {
    int n = w.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 (w[i][j] > w[i][k] + w[k][j]) {
                    w[i][j] = w[i][k] + w[k][j];
                    prev[i][j] = k;
                }
            }
        }
    }

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

4.3 Fast Dijkstra's algorithm

```
// Implementation of Dijkstra's algorithm using adjacency lists
// and priority queue for efficiency.
```

*/

```

// Running time: O(|E| log |V|)

#include <queue>
#include <stdio>
using namespace std;
const int INF = 2000000000;
typedef pair<int, int> PII;

int main() {
    int N, s, t;
    scanf("%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

```

4.4 Strongly connected components

```

#include <memory.h>
struct edge {int e, nxt;};
int V, E;
edge e[MAXE], er[MAXE];
int sp[MAXV], spr[MAXV];
int group_cnt, group_num[MAXV];
bool v[MAXV];
int stk[MAXV];
void fill_forward(int x)
{
    int i;
    v[x] = true;
    for (i = sp[x]; i != 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 != 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]);}
}

// Tarjan's SCC Algorithm
int n, m;
struct Node {vector<int> adj;};
Node graph[MAX_N];
stack<int> Stack;
bool onStack[MAX_N];
int Indices;
int Index[MAX_N];
int LowLink[MAX_N];
int component[MAX_N];
int numComponents;

void tarjanDFS(int i)
{
    Index[i] = ++Indices;
    LowLink[i] = Indices;
    Stack.push(i); onStack[i] = true;
    for (int j = 0; j < graph[i].adj.size(); j++) {
        int w = graph[i].adj[j];
        if (Index[w] == 0) {
            tarjanDFS(w);
            LowLink[i] = min(LowLink[i], LowLink[w]);
        } else if (onStack[w]) {
            LowLink[i] = min(LowLink[i], Index[w]);
        }
    }
}

```

4.6 Kruskal's algorithm

```

/*
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  $O(E \cdot \log(E))$  time.
*/
#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>
using namespace std;

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(vector<vector<T>>& w)
{
    int n = w.size();
    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;
    for(int i=0; i<n; i++)
        for(int j=i+1; j<n; j++)
            if(w[i][j] >= 0)
            {
                edge e;
                e.u = i; e.v = j; e.d = w[i][j];
                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[uc] > R[vc]) C[uc] = vc;
        }
    }
}

```

```

}
if (LowLink[i] == Index[i]) {
    int w = 0;
    do {
        w = Stack.top(); Stack.pop();
        component[w] = numComponents;
        onStack[w] = false;
    } while (i != w && !Stack.empty());
    numComponents++;
}
}

void Tarjan()
{
    Indices = 0;
    while (!Stack.empty()) Stack.pop();
    for (int i=n; i>0; i--) onStack[i] = LowLink[i] = Index[i] = 0;
    numComponents = 0;
    for (int i=n; i>0; i--) if (Index[i] == 0) tarjanDFS(i);
}

// add edge i to j
// graph[i].adj.push_back(j);

```

4.5 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;
}

```

```

    else { C[vc] = uc; R[uc]++; }
}
return weight;
}

int main()
{
    int wa[6][6] = {
        { 0, -1, 2, -1, 7, -1 },
        { -1, 0, -1, 2, -1, -1 },
        { 2, -1, 0, -1, 8, 6 },
        { -1, 2, -1, 0, -1, -1 },
        { 7, -1, 8, -1, 0, 4 },
        { -1, -1, 6, -1, 4, 0 };
    vector <vector<int>> w(6, vector<int>(6));
    for(int i=0; i<6; i++)
        for(int j=0; j<6; j++)
            w[i][j] = wa[i][j];
    cout << Kruskal(w) << endl;
    cin >> wa[0][0];
}

```

4.7 Minimum spanning trees

```

// This function runs Prim's algorithm for constructing minimum
// weight spanning trees.
//
// Running time: O(|V|
//
// 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
//
#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;
typedef pair<int,int> PII;
typedef vector<PII> VPII;
T Prim (const VVT &w, VPII &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;
}

int main(){
    int ww[5][5] = {
        {0, 400, 400, 300, 600},
        {400, 0, 3, -1, 7},
        {400, 3, 0, 2, 0},
        {300, -1, 2, 0, 5},
        {600, 7, 0, 5, 0}
    };
    VVT w(5, VT(5));
    for (int i = 0; i < 5; i++)
        for (int j = 0; j < 5; j++)
            w[i][j] = ww[i][j];
    // expected: 305
    // 2 1
    // 3 2
    // 0 3
    // 2 4
    VPII edges;
    cout << Prim (w, edges) << endl;
    for (int i = 0; i < edges.size(); i++)
        cout << edges[i].first << " " << edges[i].second << endl;
}

```

4.8 Topological sort

```

// This function uses performs a non-recursive topological sort.
//
// Running time: O(|V|. 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
//          a permutation of 0,...,n-1 (stored in a vector)
//          which represents an ordering of the nodes which
//          is consistent with w
//
// If no ordering is possible, false is returned.
#include <iostream>
#include <queue>
#include <cmath>
#include <vector>
using namespace std;
typedef double T;
typedef vector<T> VT;

```

```
typedef vector<VT> VVT;
typedef vector<int> VI;
typedef vector<VI> VVI;

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

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

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

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

5 Data structures

5.1 Suffix array

```
// Suffix array construction in  $O(L \log L)$  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>>,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
    //
    for (int i = 0; i < v.size(); i++) cout << v[i] << " ";
    cout << endl;
    cout << suffix.LongestCommonPrefix(0, 2) << endl;
}

// BEGIN CUT
//endif
// END CUT
```

5.2 Binary Indexed Tree

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

#define LOGSZ 17

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

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

```

    }
}

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

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

5.3 Union-find set

#include <iostream>
#include <vector>
using namespace std;
struct UnionFind {
    vector<int> C;
    UnionFind(int n) : C(n) { for (int i = 0; i < n; i++) C[i] = i; }
    int find(int x) { return (C[x] == x) ? x : C[x] = find(C[x]); }
    void merge(int x, int y) { C[find(x)] = find(y); }
};

int main()
{
    int n = 5;
    UnionFind uf(n);
    uf.merge(0, 2);
    uf.merge(1, 0);
    uf.merge(3, 4);
    for (int i = 0; i < n; i++) cout << i << " " << uf.find(i) << endl;
    return 0;
}

5.4 KD-tree

// -----
// A straightforward, but probably sub-optimal KD-tree implementation
// that's probably good enough for most things (current it's a
// 2D-tree)
//
// - constructs from n points in O(n lg n) time
// - handles nearest-neighbor query in O(lg n) if points are well
//   distributed
// - worst case for nearest-neighbor may be linear in pathological
//   case
//
// Sonny Chan, Stanford University, April 2009
// -----

#include <iostream>
#include <vector>
#include <limits>
#include <cstdlib>

using namespace std;

// number type for coordinates, and its maximum value
typedef long long ntype;
const ntype sentry = numeric_limits<ntype>::max();

// point structure for 2D-tree, can be extended to 3D
struct point {
    ntype x, y;
    point(ntype xx = 0, ntype yy = 0) : x(xx), y(yy) {}
};

bool operator==(const point &a, const point &b)
{
    return a.x == b.x && a.y == b.y;
}

// sorts points on x-coordinate
bool on_x(const point &a, const point &b)
{
    return a.x < b.x;
}

// sorts points on y-coordinate
bool on_y(const point &a, const point &b)
{
    return a.y < b.y;
}

// squared distance between points
ntype pdist2(const point &a, const point &b)
{
    ntype dx = a.x-b.x, dy = a.y-b.y;
    return dx*dx + dy*dy;
}

// bounding box for a set of points
struct bbox
{
    ntype x0, x1, y0, y1;
    bbox() : x0(sentry), x1(-sentry), y0(sentry), y1(-sentry) {}

    // computes bounding box from a bunch of points
    void compute(const vector<point> &v) {
        for (int i = 0; i < v.size(); ++i) {
            x0 = min(x0, v[i].x); x1 = max(x1, v[i].x);
            y0 = min(y0, v[i].y); y1 = max(y1, v[i].y);
        }
    }

    // squared distance between a point and this bbox, 0 if inside
    ntype distance(const point &p) {
        if (p.x < x0) {
            return pdist2(point(x0, y0), p);
        } else if (p.y > y1) {
            return pdist2(point(x0, y1), p);
        } else {
            return pdist2(point(x0, p.y), p);
        }
    } else if (p.x > x1) {
        return pdist2(point(x1, y0), p);
    } else if (p.y < y0) {
        return pdist2(point(x1, y1), p);
    } else {
        return pdist2(point(x1, p.y), p);
    }
} else {
    if (p.y < y0) {
        return pdist2(point(p.x, y0), p);
    } else if (p.y > y1) {
        return pdist2(point(p.x, y1), p);
    }
}
}

```

```

    }
    else
        return 0;
}

// stores a single node of the kd-tree, either internal or leaf
struct kdnode
{
    bool leaf; // true if this is a leaf node (has one point)
    point pt; // the single point of this is a leaf
    bbox bound; // bounding box for set of points in children

    kdnode *first, *second; // two children of this kd-node

    kdnode() : leaf(false), first(0), second(0) {}
    kdnode() { if (first) delete first; if (second) delete second; }

    // intersect a point with this node (returns squared distance)
    ntype intersect(const point &p) {
        return bound.distance(p);
    }

    // recursively builds a kd-tree from a given cloud of points
    void construct(vector<point> &vp)
    {
        // compute bounding box for points at this node
        bound.compute(vp);

        // if we're down to one point, then we're a leaf node
        if (vp.size() == 1) {
            leaf = true;
            pt = vp[0];
        }
        else {
            // split on x if the bbox is wider than high (not best
            // heuristic...)
            if (bound.x1-bound.x0 >= bound.y1-bound.y0)
                sort(vp.begin(), vp.end(), on_x);
            // otherwise split on y-coordinate
            else
                sort(vp.begin(), vp.end(), on_y);

            // divide by taking half the array for each child
            // (not best performance if many duplicates in the middle)
            int half = vp.size()/2;
            vector<point> vl(vp.begin(), vp.begin()+half);
            vector<point> vr(vp.begin()+half, vp.end());
            first = new kdnode(); first->construct(vl);
            second = new kdnode(); second->construct(vr);
        }
    }

    // simple kd-tree class to hold the tree and handle queries
    struct kdtree
    {
        kdnode *root;

        // constructs a kd-tree from a points (copied here, as it sorts
        // them)
        kdtree(const vector<point> &vp) {
            vector<point> v(vp.begin(), vp.end());
            root = new kdnode();
            root->construct(v);
        }

        kdtree() { delete root; }

        // recursive search method returns squared distance to nearest
        // point
        ntype search(kdnode *node, const point &p)
        {
            if (node->leaf) {
                // commented special case tells a point not to find itself
                if (p == node->pt) return sentry;
                else
                    return pdist2(p, node->pt);
            }

            ntype bfirst = node->first->intersect(p);
            ntype bsecond = node->second->intersect(p);

            // choose the side with the closest bounding box to search
            // first
            // (note that the other side is also searched if needed)
            if (bfirst < bsecond) {
                ntype best = search(node->first, p);
                if (bsecond < best)
                    best = min(best, search(node->second, p));
                return best;
            }
            else {
                ntype best = search(node->second, p);
                if (bfirst < best)
                    best = min(best, search(node->first, p));
                return best;
            }
        }

        // squared distance to the nearest
        ntype nearest(const point &p) {
            return search(root, p);
        }
    };
};

// some basic test code here

int main()
{
    // generate some random points for a kd-tree
    vector<point> vp;
    for (int i = 0; i < 100000; ++i) {
        vp.push_back(point(rand()%100000, rand()%100000));
    }
    kdtree tree(vp);

    // query some points
    for (int i = 0; i < 10; ++i) {
        point q(rand()%100000, rand()%100000);
        cout << "Closest squared distance to (" << q.x << ", " << q.y
        << ")\n";
        << " is " << tree.nearest(q) << endl;
    }

    return 0;
}

```

5.5 Splay tree

```

#include <cstdio>
#include <algorithm>
using namespace std;

const int N_MAX = 130010;
const int oo = 0x3f3f3f3f;
struct Node

```

```

    if (x == y->ch[1])
        rotate(y, 0), rotate(x, 0);
    else
        rotate(x, 1), rotate(x, 0);
    }
}
update(x);
}

void select(int k, Node *fa)
{
    Node *now = root;
    while(1)
    {
        pushDown(now);
        int tmp = now->ch[0]->size + 1;
        if (tmp == k)
            break;
        else if (tmp < k)
            now = now->ch[1], k -= tmp;
        else
            now = now->ch[0];
    }
    splay(now, fa);
}

Node *makeTree(Node *p, int l, int r)
{
    if (l > r)
        return null;
    int mid = (l + r) / 2;
    Node *x = allocNode(mid);
    x->pre = p;
    x->ch[0] = makeTree(x, l, mid - 1);
    x->ch[1] = makeTree(x, mid + 1, r);
    update(x);
    return x;
}

int main()
{
    int n, m;
    null = allocNode(0);
    null->size = 0;
    root = allocNode(0);
    root->ch[1] = allocNode(oo);
    root->ch[1]->pre = root;
    update(root);

    scanf("%d%d", &n, &m);
    root->ch[1]->ch[0] = makeTree(root->ch[1], 1, n);
    splay(root->ch[1]->ch[0], null);

    while (m --)
    {
        int a, b;
        scanf("%d%d", &a, &b);
        a ++, b ++;
        select(a - 1, null);
        select(b + 1, root);
        makeTurned(root->ch[1]->ch[0]);
    }

    for (int i = 1; i <= n; i ++ )
    {
        select(i + 1, null);
        printf("%d ", root->val);
    }
}

```

```

{
    Node *ch[2], *pre;
    int val, size;
    bool isTurned;
    nodePool[N_MAX], *null, *root;

    Node *allocNode(int val)
    {
        static int freePos = 0;
        Node *x = &nodePool[freePos ++];
        x->val = val, x->isTurned = false;
        x->ch[0] = x->ch[1] = x->pre = null;
        x->size = 1;
        return x;
    }

    inline void update(Node *x)
    {
        x->size = x->ch[0]->size + x->ch[1]->size + 1;
    }

    inline void makeTurned(Node *x)
    {
        if (x == null)
            return;
        swap(x->ch[0], x->ch[1]);
        x->isTurned ^= 1;
    }

    inline void pushDown(Node *x)
    {
        if (x->isTurned)
        {
            makeTurned(x->ch[0]);
            makeTurned(x->ch[1]);
            x->isTurned ^= 1;
        }
    }

    inline void rotate(Node *x, int c)
    {
        Node *y = x->pre;
        x->pre = y->pre;
        if (y->pre != null)
            y->pre->ch[y == y->pre->ch[1]] = x;
        y->ch[!c] = x->ch[c];
        if (x->ch[c] != null)
            x->ch[c]->pre = y;
        x->ch[c] = y, y->pre = x;
        update(y);
        if (y == root)
            root = x;
    }

    void splay(Node *x, Node *p)
    {
        while (x->pre != p)
        {
            if (x->pre->pre == p)
                rotate(x, x == x->pre->ch[0]);
            else
            {
                Node *y = x->pre, *z = y->pre;
                if (y == z->ch[0])
                {
                    if (x == y->ch[0])
                        rotate(y, 1), rotate(x, 1);
                    else
                        rotate(x, 0), rotate(x, 1);
                }
                else
                {

```

5.6 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 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] - (i<<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;
}

```

5.7 Lazy segment tree(Java)

```

public class SegmentTreeRangeUpdate {
    public long[] leaf;
    public long[] update;
    public int origSize;
    public SegmentTreeRangeUpdate(int[] list) {
        origSize = list.length;
        leaf = new long[4*list.length];
        update = new long[4*list.length];
        build(1, 0, list.length-1, list);
    }
    public void build(int curr, int begin, int end, int[] list)
    {
        if (begin == end)
            leaf[curr] = list[begin];
        else
        {
            int mid = (begin+end)/2;
            build(2 * curr, begin, mid, list);
            build(2 * curr + 1, mid+1, end, list);
            leaf[curr] = leaf[2*curr] + leaf[2*curr+1];
        }
    }
    public void update(int begin, int end, int val) {
        update(1, 0, origSize-1, begin, end, val);
    }
    public void update(int curr, int tBegin, int tEnd, int begin,
        int end, int val)
    {
        if (tBegin >= begin && tEnd <= end)
            update[curr] += val;
        else
        {
            leaf[curr] += (Math.min(end, tEnd) - Math.max(
                begin, tBegin) + 1) * val;
            int mid = (tBegin+tEnd)/2;
            if (mid >= begin && tBegin <= end)
                update(2*curr, tBegin, mid, begin, end
                    , val);
            if (tEnd >= begin && mid+1 <= end)
                update(2*curr+1, mid+1, tEnd, begin,
                    end, val);
        }
    }
    public long query(int begin, int end) {
        return query(1, 0, origSize-1, begin, end);
    }
    public long query(int curr, int tBegin, int tEnd, int begin,
        int end)
    {

```

```

if (tBegin >= begin && tEnd <= end) {
    if (update[curr] != 0) {
        leaf[curr] += (tEnd - tBegin + 1) * update
        [curr];
        if (2*curr < update.length) {
            update[2*curr] += update[curr]
        ];
            update[2*curr+1] += update[
            curr];
        }
        update[curr] = 0;
    }
    return leaf[curr];
}
else {
    leaf[curr] += (tEnd - tBegin + 1) * update[curr];
    if (2*curr < update.length) {
        update[2*curr] += update[curr];
        update[2*curr+1] += update[curr];
    }
    update[curr] = 0;
    int mid = (tBegin + tEnd) / 2;
    long ret = 0;
    if (mid >= begin && tBegin <= end)
        ret += query(2*curr, tBegin, mid,
            begin, end);
    if (tEnd >= begin && mid+1 <= end)
        ret += query(2*curr+1, mid+1, tEnd,
            begin, end);
    return ret;
}
}
}

```

6 Miscellaneous

6.1 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] = it == best.begin() ? -1 : prev(it)->second;
            *it = item;
        }
    }

    VI ret;
    for (int i = best.back().second; i >= 0; i = dad[i])
        ret.push_back(v[i]);
    reverse(ret.begin(), ret.end());
    return ret;
}
```

6.2 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, respectively.
*/
```

```
#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 = "AABACAADAABAABA", b = "AABA";
        KMP(a, b); // expected matches at: 0, 9, 12
        return 0;
    }
}
```

6.3 Constraint satisfaction problems

```
// Constraint satisfaction problems

#include <cstdlib>
#include <iostream>
#include <vector>
#include <set>
using namespace std;

#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];
        VVI& 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
    }
}

```

6.4 Fast exponentiation

```

/*
Uses powers of two to exponentiate numbers and matrices. Calculates
nk in O(log(k)) time when n is a number. If A is an n x n matrix,
calculates Ak in O(nlog(k)) time.
*/
#include <iostream>
#include <vector>
using namespace std;

```



```
}

```

6.5 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<VI>::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++)

```

```
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:
       = 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;
    }
}

```

```

    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 Formatting, STL

7.1 C++ input/output

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

using namespace std;

int main()
{
    // Output 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;
}
```

7.2 STL next permutation

```
// Example for using stringstream and next_permutation

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

using namespace std;

int main(void) {
    vector<int> v;

    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    v.push_back(4);

    // Expected output: 1 2 3 4
    // 1 2 4 3
    // ...
    // 4 3 2 1

    do {
        ostringstream oss;
        oss << v[0] << " " << v[1] << " " << v[2] << " " << v[3];

        // for input from a string s,
        // istream iss(s);
        // iss >> variable;

        cout << oss.str() << endl;
    } while (next_permutation(v.begin(), v.end()));

    v.clear();

    v.push_back(1);
```

```
v.push_back(2);
v.push_back(1);
v.push_back(3);

// To use unique, first sort numbers. Then call
// unique to place all the unique elements at the beginning
// of the vector, and then use erase to remove the duplicate
// elements.

sort(v.begin(), v.end());
v.erase(unique(v.begin(), v.end()), v.end());

// Expected output: 1 2 3
for (size_t i = 0; i < v.size(); i++)
    cout << v[i] << " ";
cout << endl;
```

7.3 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;
    intToDay (jd, m, d, y);
    string day = intToDay (jd);
```

```

System.out.println(fmt.format(12345.6789)); // produces
12345.68
System.out.println(fmt.format(12345.0)); // produces 12345.00
System.out.println(fmt.format(0.0)); // produces 0.00

// round to precisely 2 digits, force leading zeros
fmt = new DecimalFormat("00000000.00");
System.out.println(fmt.format(12345.6789)); // produces
000012345.68
System.out.println(fmt.format(12345.0)); // produces
000012345.00
System.out.println(fmt.format(0.0)); // produces 0000000000.00

// force leading '+'
fmt = new DecimalFormat("+0;-0");
System.out.println(fmt.format(12345.6789)); // produces +12346
System.out.println(fmt.format(-12345.6789)); // produces
-12346
System.out.println(fmt.format(0)); // produces +0

// force leading positive/negative, pad to 2
fmt = new DecimalFormat("positive 00;negative 0");
System.out.println(fmt.format(1)); // produces "positive 01"
System.out.println(fmt.format(-1)); // produces "negative 01"

// quote special chars (#)
fmt = new DecimalFormat("text with '#' followed by #");
System.out.println(fmt.format(12.34)); // produces "text with
# followed by 12"

// always show "."
fmt = new DecimalFormat("#.#");
fmt.setDecimalSeparatorAlwaysShown(true);
System.out.println(fmt.format(12.34)); // produces "12.3"
System.out.println(fmt.format(12)); // produces "12."
System.out.println(fmt.format(0.34)); // produces "0.3"

// different grouping distances:
fmt = new DecimalFormat("###,###,###"); // produces
"1,2345,6789.123"

// scientific:
fmt = new DecimalFormat("0.000E00");
System.out.println(fmt.format(123456789.123)); // produces
"1.235E08"
System.out.println(fmt.format(-0.000234)); // produces "-2.34E
-04"

// using variable number of digits:
fmt = new DecimalFormat("0");
System.out.println(fmt.format(123.123)); // produces "123"
fmt.setMinimumFractionDigits(8);
System.out.println(fmt.format(123.123)); // produces
"123.123000000"
fmt.setMaximumFractionDigits(0);
System.out.println(fmt.format(123.123)); // produces "123"

// note: to pad with spaces, you need to do it yourself:
// String out = fmt.format(...);
// while (out.length() < targetlength) out = " "+out;
}
}

```

7.6 Regular expressions

```

// Code which demonstrates the use of Java's regular expression
libraries.
// This is a solution for

```

```

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

```

7.4 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, 1);
            }
        }
    }
}

```

7.5 Decimal output formatting (Java)

```

// examples for printing floating point numbers
import java.util.*;
import java.io.*;
import java.text.DecimalFormat;

public class DecFormat {
    public static void main(String[] args) {
        DecimalFormat fmt;

        // round to at most 2 digits, leave of digits if not needed
        fmt = new DecimalFormat("#.##");
        System.out.println(fmt.format(12345.6789)); // produces
12345.68
        System.out.println(fmt.format(12345.0)); // produces 12345
        System.out.println(fmt.format(0.0)); // produces 0
        System.out.println(fmt.format(0.01)); // produces .1

        // round to precisely 2 digits
        fmt = new DecimalFormat("#.00");
        System.out.println(fmt.format(12345.6789)); // produces
12345.68
        System.out.println(fmt.format(12345.0)); // produces 12345.00
        System.out.println(fmt.format(0.0)); // produces .00

        // round to precisely 2 digits, force leading zero
        fmt = new DecimalFormat("0.00");
    }
}

```

```

// Loglan: a logical language
// http://acm.uva.es/p/v1/i34.html
//
// In this problem, we are given a regular language, whose rules can
// be
// inferred directly from the code. For each sentence in the input,
// we must
// determine whether the sentence matches the regular expression or
// not. The
// code consists of (1) building the regular expression (which is
// fairly
// complex) and (2) using the regex to match sentences.
import java.util.*;
import java.util.regex.*;
public class Loglan {
    public static String BuildRegex () {
        String space = " ";
        String A = "[a-z]";
        String C = "[a-zaeiou]";
        String MOD = "(g" + A + ")";
        String BA = "(b" + A + ")";
        String DA = "(d" + A + ")";
        String LA = "(l" + A + ")";
        String NAM = "[a-z]*" + C + " ";
        String PREDA = "(" + C + C + A + C + A + " | " + C + A + C + C + C +
            A + ")";
        String predstring = "(" + PREDA + "(" + space + PREDA + ")*)";
        String predname = "(" + LA + space + predstring + " | " + NAM +
            " | " + DA + space +
            predstring + ")*";
        String predclain = "(" + predname + space + BA + space + preds
            + " | " + DA + space +
            preds + ")*";
        String verbpred = "(" + MOD + space + predstring + ")*";
        String statement = "(" + predname + space + verbpred + space +
            predname + " | " +
            predname + space + verbpred + ")*";
        String sentence = "(" + statement + " | " + predclain + ")*";
        return " " + sentence + "$";
    }
    public static void main (String args[]){
        String regex = BuildRegex();
        Pattern pattern = Pattern.compile (regex);
        Scanner s = new Scanner(System.in);
        while (true) {
            // In this problem, each sentence consists of multiple
            // lines, where the last
            // line is terminated by a period. The code below reads
            // lines until
            // encountering a line whose final character is a ' '.
            // Note the use of
            // s.length() to get length of string
            // s.charAt() to extract characters from a Java string
            // s.trim() to remove whitespace from the beginning and
            // end of Java string
            // Other useful String manipulation methods include

```

```

// s.compareTo(t) < 0 if s < t, lexicographically
// s.indexOf("apple") returns index of first occurrence
// of "apple" in s
// s.lastIndexOf("apple") returns index of last
// occurrence of "apple" in s
// s.replace(c,d) replaces occurrences of character c
// with d
// s.startsWith("apple") returns (s.indexOf("apple") ==
// 0)
// s.toLowerCase() / s.toUpperCase() returns a new
// lower/uppercased string
// Integer.parseInt(s) converts s to an integer (32-bit)
// Long.parseLong(s) converts s to a long (64-bit)
// Double.parseDouble(s) converts s to a double
String sentence = "";
while (true) {
    sentence = (sentence + " " + s.nextLine()).trim();
    if (sentence.equals("#")) return;
    if (sentence.charAt(sentence.length()-1) == '.') break;
}
// now, we remove the period, and match the regular
// expression
String removed_period = sentence.substring(0, sentence.
length()-1).trim();
if (pattern.matcher (removed_period).find()){
    System.out.println ("Good");
} else {
    System.out.println ("Bad!");
}
}
}
}
}

```

8 Classical Problems

8.1 Illumination (Tarjan 2SAT)

```
/*
tarjan + 2sat
You inherited a haunted house. Its floor plan is an n-by-n square grid
with l lamps in fixed
locations and no interior walls. Each lamp can either illuminate its
row or its column, but not both
simultaneously. The illumination of each lamp extends by r squares in
both directions, so a lamp
unobstructed by an exterior wall of the house can illuminate as many
as 2r + 1 squares.

n, r and l (1 < n, r, l < 1,000).
ri and ci (1 < ri, ci < n), indicating that there is a lamp in row ri
and column ci.

```

YES if it is possible to illuminate all lamps as stated above
*/

```
#include <cstdio>
#include <vector>
#include <utility>
#include <algorithm>
#include <string>
#include <iostream>
#include <stack>

using namespace std;
typedef pair<int, int> pii;

const int MAXN = 10005;
vector<int> g[MAXN];
int d[MAXN], low[MAXN], scc[MAXN];
bool stacked[MAXN];
int ticks, current_scc;
stack<int> s;

void tarjan(int u) {
    d[u] = low[u] = ticks++;
    s.push(u);
    stacked[u] = true;
    const vector<int> &out = g[u];
    for (int k = 0, m = (int)out.size(); k < m; k++) {
        const int &v = out[k];
        if (d[v] == -1) {
            tarjan(v);
            low[u] = min(low[u], low[v]);
        } else if (stacked[v]) {
            low[u] = min(low[u], low[v]);
        }
    }
    if (d[u] == low[u]) {
        int v;
        do {
            v = s.top();
            s.pop();
            stacked[v] = false;
            scc[v] = current_scc;
        } while (u != v);
        current_scc++;
    }

    int n, r, l;
    vector<pii> lamps;

```

```
int main() {
    cin >> n >> r >> l;
    for (int i = 1; i <= l; i++) {
        int x, y;
        cin >> x >> y;
        lamps.push_back(make_pair(x, y));
    }

    memset(d, -1, sizeof(d));
    for (int i = 0; i < l; i++) {
        int mr = lamps[i].first;
        for (int j = i + 1; j < l; j++) {
            if (mr == lamps[j].first && abs(lamps[i].second - lamps[j].second) <= 2 * r) {
                g[i].push_back(j + 1);
                g[j].push_back(i + 1);
            }
        }
    }

    for (int i = 0; i < l; i++) {
        int mc = lamps[i].second;
        for (int j = i + 1; j < l; j++) {
            if (mc == lamps[j].second && abs(lamps[i].first - lamps[j].first) <= 2 * r) {
                g[i + 1].push_back(j);
                g[j + 1].push_back(i);
            }
        }
    }

    for (int i = 0; i < l * 2; i++) {
        if (d[i] == -1) tarjan(i);
    }

    for (int i = 0; i < l; i++) {
        if (scc[i] == scc[i + 1]) {
            cout << "0" << endl;
            return 0;
        }
    }

    cout << "1" << endl;
    return 0;
}

```

8.2 BuggyRobot (Dijkstra + state hash)

// go from S to G with minimal amount of edition to instruction L, R, U, and D

```
#include <string>
#include <vector>
#include <queue>
#include <iostream>
using namespace std;
int moves[4];
const int INF = 1000000000;
int main() {
    int N, M;
    cin >> N >> M;
    string lin;
    for (int i = 0; i < M + 2; i++)
        lin.push_back('#');
    for (int i = 0; i < N; i++) {
        string inp;

```

```

    cout << finished << endl ;
}

```

8.3 Paint (dp + binary search backtrack)

```

//https://open.kattis.com/problems/paint
// the smallest number of slats that go unpainted with an optimal
// selection of painters
// dp with binary search backtrack
#include <bits/stdc++.h>

```

```

using namespace std;

```

```

typedef long long ll;
typedef pair<ll, ll> pll;

```

```

#define pb push_back
#define mp make_pair

```

```

ll n, k;

```

```

ll find_idx(vector<pll> &slabs, ll maxlen) {
    ll low = 0, high = k-1, mid;
    if (slabs[0].second > maxlen) return -1;
    while (low < high) {
        if (high == low+1) {
            if (slabs[high].second <= maxlen) return high;
            else return low;
        }
    }
    mid = (low + high) / 2;
    if (slabs[mid].second > maxlen) high = mid;
    else low = mid;
    return mid;
}

```

```

int main() {
    cin >> n >> k;
    vector<pll> slabs;
    for (ll i=0; i<k; i++) {
        ll a, b; cin >> a >> b;
        slabs.pb(mp(a, b));
    }
    sort(slabs.begin(), slabs.end(), [](pll &p1, pll &p2){return (p1.
        second < p2.second) || (p1.second == p2.second && p1.first <
        p2.first)});
    ll dp[k], cnt[k];
    dp[0] = slabs[0].second - slabs[0].first + 1;
    cnt[0] = 1;
    for (ll i=1; i<k; i++) {
        ll val1 = dp[i-1];
        ll idx = find_idx(slabs, slabs[i].first-1);
        ll val2 = idx < 0 ? 0 : dp[idx];
        val2 += slabs[i].second - slabs[i].first + 1;
        if (val1 > val2) {
            dp[i] = val1;
            cnt[i] = cnt[i-1];
        } else if (val1 < val2) {
            dp[i] = val2;
            cnt[i] = idx < 0 ? 1 : cnt[idx] + 1;
        } else {
            dp[i] = val1;
            cnt[i] = max(cnt[i-1], (idx < 0 ? 1 : cnt[idx] + 1));
        }
    }
    cout << n - dp[k-1] << endl;
}

```

```

    cin >> inp ;
    lin.append(inp) ;
    lin.push_back('#') ;
}
for (int i=0; i<M+2; i++)
    lin.push_back('#') ;
int st = 0 ;
int end = 0 ;
for (int i=0; i<lin.size(); i++)
    if (lin[i] == 'S')
        st = i ;
    else if (lin[i] == 'G')
        end = i ;
string cmd ;
cin >> cmd ;
moveind['L'] = -1 ;
moveind['R'] = 1 ;
moveind['U'] = -M-1 ;
moveind['D'] = M+1 ;
moves[0] = -1 ;
moves[1] = 1 ;
moves[2] = -M-1 ;
moves[3] = M+1 ;
int atmod = lin.size() ;
vector<int> cost(lin.size()*(cmd.size()+1), INF) ;
vector<int> thislev ;
thislev.push_back(st) ;
cost[st] = 0 ;
int finished = -1 ;
for (int d=0; finished < 0; d++) {
    // first, all the additional nodes we can reach for free
    for (int i=0; i<thislev.size(); i++) {
        int at = thislev[i] ;
        int sqat = at % atmod ;
        int cmdat = at / atmod ;
        if (cmdat < cmd.size()) {
            int nsq = sqat + moveind[cmd[cmdat]] ;
            if (nsq == end)
                finished = d ;
            if (lin[nsq] == '#')
                nsq = sqat ;
            int st2 = nsq + (cmdat + 1) * atmod ;
            if (cost[st2] > d) {
                cost[st2] = d ;
                thislev.push_back(st2) ;
            }
        }
    }
    // now advance by additional moves; these cost
    vector<int> nextlev ;
    for (int i=0; finished < 0 && i<thislev.size(); i++) {
        int at = thislev[i] ;
        int sqat = at % atmod ;
        for (int mv=0; mv<4; mv++) {
            int nsq = sqat + moves[mv] ;
            if (nsq == end)
                finished = d + 1 ;
            int nat = at + moves[mv] ;
            if (lin[nsq] != '#' && cost[nat] > d + 1) {
                cost[nat] = d + 1 ;
                nextlev.push_back(nat) ;
            }
        }
    }
    swap(thislev, nextlev) ;
    nextlev.clear() ;
    if (thislev.size() == 0)
        break ;
}
}

```

```
}
}
```

8.4 Rainbow (dfs + mark)

```
// https://open.kattis.com/problems/rainbowroads
/*
You are given a tree with n nodes (stations), conveniently numbered
from 1 to n.
Each edge in this tree has one of n colors.
A path in this tree is called a rainbow if all adjacent edges in the
path have different colors.
Also, a node is called good if every simple path with that node as one
of its endpoints is a rainbow path.
(A simple path is a path that does not repeat any vertex or edge.)
Find all the good nodes in the given tree.
*/
#include <iostream>
#include <map>
#include <vector>
#include <cstdlib>
using namespace std;

map<int, vector<int>> > adj[50005];
int last[50005];
bool locked[50005];

void mark(int src, int dst) {
    if (locked[dst]) return;
    if (last[dst] == 0) {
        last[dst] = src;
        for (auto entry : adj[dst]) {
            for (auto next : entry.second) {
                if (next != src) mark(dst, next);
            }
        }
    } else if (last[dst] != src) {
        mark(dst, last[dst]);
    }
}

int main() {
    int n;
    scanf("%d", &n);
    for (int i = 1; i < n; i++) {
        int a, b, c;
        scanf("%d %d %d", &a, &b, &c);
        adj[a][c].push_back(b);
        adj[b][c].push_back(a);
    }
    for (int i = 1; i <= n; i++) {
        if (locked[i]) continue;
        for (auto entry : adj[i]) {
            if (entry.second.size() > 1) {
                for (auto next : entry.second) mark(i, next);
            }
        }
    }
    int count = 0;
    for (int i = 1; i <= n; i++) {
        if (last[i] == 0) count++;
    }
    cout << count << endl;
    for (int i = 1; i <= n; i++) {
        if (last[i] == 0) cout << i << endl;
    }
}
```

8.5 Security badge (dfs + memorization)

```
//https://open.kattis.com/problems/securitybadge
// N, L, and B, denoting the number of rooms, of locks, and of badge
// numbers
// S and D noting the starting and destination rooms that we are
// interested in
// a b x y indicating that a lock permits passage from room a to room
// b (but not from b to a) for badges numbered from x to y, inclusive

#include <iostream>
#include <vector>
#include <set>
using namespace std;

struct door {
    int dst, lo, hi;
    door(int dst, int lo, int hi) : dst(dst), lo(lo), hi(hi) {}
};

typedef vector<door> VD;
VD adj[1024];
int s, t;
bool visited[1024];

void dfs(int cur, int id) {
    if (visited[cur]) return;
    visited[cur] = true;
    for (auto edge : adj[cur]) {
        if (id >= edge.lo && id <= edge.hi) dfs(edge.dst, id);
    }
}

bool accessible(int id) {
    for (int i = 0; i < 1024; i++) visited[i] = false;
    dfs(s, id);
    return visited[t];
}

int main() {
    int n, m, k;
    cin >> n >> m >> k;
    cin >> s >> t;
    set<int> boundaries;
    for (int i = 1; i <= m; i++) {
        int src, dst, lo, hi;
        cin >> src >> dst >> lo >> hi;
        adj[src].push_back(door(dst, lo, hi));
        boundaries.insert(lo);
        boundaries.insert(hi + 1);
    }
    int last = 0;
    bool lastGood = false;
    int ans = 0;
    for (auto id : boundaries) {
        if (lastGood) ans += id - last;
        lastGood = accessible(id);
        last = id;
    }
    cout << ans << endl;
}
```


8.6 Radio (string hashing)

```
// find smallest possible substring S'+S'+S' = S cabcabca abc
#include <iostream>
#include <string>
#include <vector>

using namespace std;
typedef uint64_t ull;

ull prime = (ull) (1e9+7);

ull hashstr(string s) {
    ull hash = 0;
    ull expo = 1;
    for (int k=0; k<s.length(); k++) {
        hash = (hash + (ull)(s[k] - 'a'+1) * expo) % prime;
        expo = (expo * 32) % prime;
    }
    return hash;
}

int main() {
    int n; cin >> n;
    string s; cin >> s;

    vector<ull> lhash(n), rhash(n), expos(n);
    ull hash = 0;
    ull expo = 1;
    for (int k=0; k<n; k++) {
        hash = (hash + (ull)(s[k] - 'a'+1) * expo) % prime;
        lhash[k] = hash;
        expos[k] = expo;
        expo = (expo * 32) % prime;
    }
    hash = 0;
    for (int k=n-1; k>=0; k--) {
        hash = (hash * 32 + (ull)(s[k] - 'a'+1)) % prime;
        rhash[k] = hash;
    }
    int l;
    for (l=1; l<n; l++) if (rhash[l] == lhash[n-l-1]) break;
    cout << l << endl;
}
```

8.7 Hanoi (recursive)

```
//https://open.kattis.com/problems/thathanoi
// num of step to complete hanoi state otherwise output no
#include <iostream>
#include <cmath>
using namespace std;
const int MAX = 50;
int loc[MAX+1];

bool count(int start, int dest, int work, int disk, long long moves,
           int long long& ans)
{
    if (disk == 0)
        return true;
    else if (loc[disk] == dest) {
        if (!count(work, dest, start, disk-1, moves/2, ans))
            return false;
        ans += moves;
    }
}
```

```
return true;
}
else if (loc[disk] == start) {
    if (!count(start, work, dest, disk-1, moves/2, ans))
        return false;
    return true;
}
else
    return false;
}

int main()
{
    int n=0;
    long long moves = 1;
    bool valid = true;
    for(int i=0; i<3; i++) {
        int m;
        cin >> m;
        n += m;
        int prev = MAX+1;
        for(int j=0; j<m; j++) {
            int disk;
            cin >> disk;
            loc[disk] = i;
            moves *=2;
            if (disk > prev)
                valid = false;
            prev = disk;
        }
        long long ans = 0;
        if (!valid || !count(0, 2, 1, n, moves/2, ans)) // moves = n
            cout << "No" << endl;
        else
            cout << moves-1-ans << endl;
    }
}
```

8.8 basesum (number theory)

```
/**
 * given n, a, and b, find the smallest m>n such that the sum of the
 * digits of m in
 * base a is the same as the sum of digits of m in base b.
 */
inline ll ceil_div(ll a, ll b) { return b ? ((a/b) + ((a%b) != 0)) :
    INF; }
inline ll add(ll a, ll b) { return (a >= INF - b) ? INF : (a + b); }
inline ll mul(ll a, ll b) { return b ? (a >= ceil_div(INF,b) ? INF : a
    *b) : 0; }

ll iters;

void convert(ll N, int BASE, int ans[MAXD]) {
    memset(ans, 0, sizeof(ans[0])*MAXD); iters += MAXD;
    int i = 0;
    while(N) ans[i++] = (N%BASE), N /= BASE, ++iters;
    assert(i<MAXD);
}

int tmp3[MAXD];
string convert_to_string(ll N, int BASE) {
    convert(N, BASE, tmp3);
    string ans;
    bool non_zero = false;
    FORB(i, MAXD-1, 0) {
```

```

++iters;
if (tmp3[i] != 0) non_zero = true;
if (non_zero) ans += (tmp3[i] < 10 ? '0' + tmp3[i] : 'A' + tmp3[i]
    - 10);
}

if (ans.empty()) return "0";
return ans;
}

// find the next number > N with sum K (in given BASE)
int tmp[MAXD];
ll next(ll N, int BASE, int K) {
    convert(N, BASE, tmp);

    int sum = 0;
    FOR(i, MAXD) sum += tmp[i], ++iters;
    bool found = false;
    FOR(i, MAXD) {
        ++iters;
        if (tmp[i] < BASE-1 && sum + 1 <= K && sum - tmp[i] + (BASE-1)*(i
            +1) >= K) {
            // we can (and should) bump up here
            K -= sum + 1;
            tmp[i]++;
            FOR(j, MAXD) {
                ++iters;
                assert(K>=0);
                if (!K) break;
                assert(j <= i);
                assert(tmp[j] <= BASE-1);
                int add = min(K, BASE-1-tmp[j]);
                tmp[j] += add;
                K -= add;
            }
            found = true;
            break;
        } else {
            // continue step
            sum -= tmp[i];
            tmp[i] = 0;
        }
    }

    if (!found) return INF;

    ll ans = 0;
    FORB(i, MAXD-1, 0) {
        ++iters;
        ans = mul(ans, BASE);
        ans = add(ans, tmp[i]);
    }
    return ans;
}

int tmp2[MAXD];
ll digit_sum(ll N, int BASE) {
    convert(N, BASE, tmp2);
    int ans = 0;
    FOR(i, MAXD) ans += tmp2[i];
    return ans;
}

ll points[2*MAXS];
ll nxtA[MAXS];
ll nxtB[MAXS];

int main() {
    ll N; int A, B;
    cin >> N >> A >> B;
    N++;
    iters = 0;

    while(true) {
        ll x = digit_sum(N, A);
        ll y = digit_sum(N, B);
        if (x==y) break;
        ++iters;
        // find all points where the ranges change
        FOR(v, MAXS) {
            if (nxtA[v] > N) continue;
            nxtA[v] = next(N, A, v);
            assert(nxtA[v] > N);
        }
        FOR(v, MAXS) {
            if (nxtB[v] > N) continue;
            nxtB[v] = next(N, B, v);
            assert(nxtB[v] > N);
        }
        int K = 0;
        FOR(v, MAXS) {
            if (nxtA[v] >= INF) continue;
            points[K++] = nxtA[v];
        }
        FOR(v, MAXS) {
            if (nxtB[v] >= INF) continue;
            points[K++] = nxtB[v];
        }
        sort(points, points+K);
        K = unique(points, points+K) - points;
        // find the next point where the ranges overlap
        ll low_x = x, high_x = x;
        ll low_y = y, high_y = y;
        ll p;
        FOR(z, K) {
            p = points[z];
            ll x = digit_sum(p, A);
            low_x = min(low_x, x);
            high_x = max(high_x, x);
            ll y = digit_sum(p, B);
            low_y = min(low_y, y);
            high_y = max(high_y, y);
            if (high_x >= low_y && low_x <= high_y) {
                // they now overlap
                N = p;
                break;
            }
        }
        cout << N << /* " " << convert_to_string(N, A) << " " <<
            convert_to_string(N, B) <<*/ endl;
    }
}

```

8.9 Vin Diagram (flood)

/*

```

7 7
AXXX..
X...X..
X.XXXX
X.X.X.X
XXXXX.X
...X...X
...XXXXB

A B A intersect B
5 5 I

*/
#include <iostream>
#include <vector>
using namespace std;

const int MAXS = 1000;

vector<vector<int>>> DIR = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};

char graph[MAXS+2][MAXS+2];

inline bool isIntersection(int r, int c){
    return (graph[r][c] == 'X'
        && graph[r+1][c] == 'X'
        && graph[r-1][c] == 'X'
        && graph[r][c+1] == 'X'
        && graph[r][c-1] == 'X');
}

int flood(int r, int c, char mark, int nrows, int ncols){
    if (graph[r][c] != '.') return 0;
    graph[r][c] = mark;
    int count = 1;
    if (r > 0)
        count += flood(r-1, c, mark, nrows, ncols);
    if (r < nrows-1)
        count += flood(r+1, c, mark, nrows, ncols);
    if (c > 0)
        count += flood(r, c-1, mark, nrows, ncols);
    if (c < ncols-1)
        count += flood(r, c+1, mark, nrows, ncols);
    return count;
}

void findstart(int&r, int&c, char mark, int n, int m){
    int x, y;
    for(int i=1; i<=n; i++) for(int j=1; j<=m; j++) {
        if (graph[i][j] != mark) continue;
        for (auto &txy: DIR) {
            x = txy[0]; y = txy[1];
            if (graph[i+x][j+y] == '.' && graph[i-x][j-y] == '*') {
                r = i+x;
                c = j+y;
                return;
            }
        }
    }

    int main(){
        int n, m;
        cin >> n >> m;

        int ax, ay, bx, by;
        int edges = 0;

        // extra side helper
        for (int j=0; j<m+2; j++)
            graph[0][j] = graph[n+1][j] = '.';
        for (int i=1; i<n; i++)

```

```

graph[i][0] = graph[i][m+1] = '.';
for (int i=1; i<=n; i++) for (int j=1; j<=m; j++) {
    cin >> graph[i][j];
    if (graph[i][j] == 'A'){ // mark 'A' coordinate
        ax = i; ay = j;
    } else if (graph[i][j] == 'B'){ // mark 'B' coordinate
        bx = i; by = j;
    }
    if (graph[i][j] != '.') edges++;
}

// set intersections
for (int i=1; i<=n; i++) for (int j=1; j<=m; j++)
    if (isIntersection(i, j)) graph[i][j] = 'I';

// fill edge connected to 'A'
int r=ax, c=ay;
bool done = false;
int x, y;
while(!done) {
    graph[r][c] = 'A';
    done = true;
    for (auto &txy: DIR) {
        x = txy[0]; y = txy[1];
        if (graph[r+x][c+y] == 'X' || graph[r+x][c+y] == 'I') {
            if (graph[r+x][c+y] == 'I') {
                r = r + 2*x;
                c = c + 2*y;
            } else {
                r = r + x;
                c = c + y;
            }
            done = false;
            break;
        }
    }
}

// set remaining edge to 'B'
for (int i=1; i<=n; i++) for (int j=1; j<=m; j++)
    if (graph[i][j] == 'X') graph[i][j] = 'B';

// fill exterior '*' from (0,0)
int exterior = flood(0, 0, '*', n+2, m+2);
exterior -= 2*(n+m+2);

// fill A area
findstart(r, c, 'A', n, m);
int anum = flood(r, c, 'a', n+2, m+2);

// fill B area
findstart(r, c, 'B', n, m);
int bnum = flood(r, c, 'b', n+2, m+2);
int abnum = n*m - edges - exterior - anum - bnum;

cout << anum << ' ' << bnum << ' ' << abnum << endl;
return 0;
}

```

8.10 substring (suffix)

```

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

using namespace std;

```

```

struct suffix
{
    int index;
    int rank[2];

    bool operator<(const suffix &other) const
    {
        if (rank[0] < other.rank[0])
            return true;
        if (rank[0] > other.rank[0])
            return false;
        return rank[1] < other.rank[1];
    }
};

void buildSuffixArray(string &txt, int n, vector<int> &sufarray)
{
    std::vector<suffix> suffixes;
    suffixes.resize(n);

    for (int i = 0; i < n; i++)
    {
        suffixes[i].index = i;
        suffixes[i].rank[0] = txt[i] - 'a';
        suffixes[i].rank[1] = ((i+1) < n)? (txt[i+1] - 'a') : -1;
    }

    std::sort(suffixes.begin(), suffixes.end());
    std::vector<int> ind;
    ind.resize(n);

    for (int k = 4; k < 2*n; k = k*2)
    {
        int rank = 0;
        int prev_rank = suffixes[0].rank[0];
        suffixes[0].rank[0] = rank;
        ind[suffixes[0].index] = 0;

        for (int i = 1; i < n; i++)
        {
            if (suffixes[i].rank[0] == prev_rank &&
                suffixes[i].rank[1] == suffixes[i-1].rank[1])
            {
                prev_rank = suffixes[i].rank[0];
                suffixes[i].rank[0] = rank;
            }
            else
            {
                prev_rank = suffixes[i].rank[0];
                suffixes[i].rank[0] = ++rank;
            }
            ind[suffixes[i].index] = i;
        }

        for (int i = 0; i < n; i++)
        {
            int nextindex = suffixes[i].index + k/2;
            suffixes[i].rank[1] = (nextindex < n)?
                suffixes[ind[nextindex]].rank[0] : -1;

            std::sort(suffixes.begin(), suffixes.end());
            sufarray.clear();
            for (int i = 0; i < n; i++)
                sufarray.push_back(suffixes[i].index);
        }

        void kasai(string &txt, vector<int> &suffixArr, vector<int> &result)
        {
            int n = suffixArr.size();

```

```

        result.resize(n, 0);
        vector<int> invSuff(n, 0);
        for (int i=0; i < n; i++)
            invSuff[suffixArr[i]] = i;

        int k = 0;
        for (int i=0; i<n; i++)
        {
            if (invSuff[i] == n-1)
            {
                k = 0;
                continue;
            }

            int j = suffixArr[invSuff[i]+1];
            while (i+k<n && j+k<n && txt[i+k]==txt[j+k])
                k++;

            result[invSuff[i]] = k;
            if (k>0)
                k--;
        }

        int main()
        {
            std::string s;
            std::cin >> s;
            std::vector<int> sufarray;
            buildSuffixArray(s, s.length(), sufarray);
            kasai(s, sufarray, lcp);
            int maxlen = -1;
            int maxstr = -1;
            for (int i = 0; i < lcp.size(); i++)
            {
                if (lcp[i] > maxlen)
                {
                    maxlen = lcp[i];
                    maxstr = i;
                }
            }
            for (int j = 0; j < maxlen; j++)
            {
                std::cout << s[sufarray[maxstr] + j];
            }
            std::cout << std::endl;
        }
    }
}

```

8.11 average manhattan (comptational geometry)

```

/*
Within a zone:  Z = (llr+rw
Cx(i) = x0 + (1+2r)/(3(1+r))w
*/
#include <iostream>
#include <algorithm>
#include <iomanip>
#include <cstdlib>
#include <vector>
#include <math.h>
#include <tuple>
using namespace std ;

```

```

    int n = suffixArr.size();

```

```

using ll = long long ;
using t3 = tuple<double, double, double> ;
int main() {
    cout << setprecision(15) ;
    int N{0} ;
    cin >> N ;
    vector<ll> xs(N), ys(N) ;
    for (int i=0; i<N; i++)
        cin >> xs[i] >> ys[i] ;
    double r {} ;
    for (int outer=0; outer<2; outer++) {
        int hix0 = min_element(xs.begin(), xs.end()) - xs.begin() ;
        int lox0 { hix0 } ;
        auto ht=[&](ll x, int p0, int p1) -> double {
            if (x == xs[p0])
                return ys[p0] ;
            if (x == xs[p1])
                return ys[p1] ;
            return ys[p0]+(x-xs[p0])*(ys[p1]-ys[p0])/(double)(xs[p1]-xs[
p0]) ;
        } ;
        vector<t3> zones ;
        while (1) {
            int hix1 { (hix0 + 1) % N } ;
            int lox1 { (lox0 + N - 1) % N } ;
            ll x0 = max(xs[hix0], xs[lox0]) ;
            if (x0 == xs[hix1]) {
                hix0 = hix1 ;
                continue ;
            }
            if (x0 == xs[lox1]) {
                lox0 = lox1 ;
                continue ;
            }
            ll x1 { min(xs[hix1], xs[lox1]) } ;
            if (x1 < x0)
                break ;
            if (x1 == x0)
                throw "Failed while building zones" ;
            double lft { ht(x0, hix0, hix1) - ht(x0, lox0, lox1) } ;
            double rgt { ht(x1, hix0, hix1) - ht(x1, lox0, lox1) } ;
            zones.push_back(make_tuple(lft, rgt, (double)(x1-x0)) ) ;
            // cout << "Adding tuple xs " << x0 << " " << x1 << " heights " <<
            lft << " " << rgt << endl ;
            if (x1 == xs[hix1])
                hix0 = hix1 ;
            if (x1 == xs[lox1])
                lox0 = lox1 ;
        }
        double s {} ;
        double cxa {} ;
        double sa {} ;
        double a2 {} ;
        double x0 {} ;
        for (int i=0; i<zones.size(); i++) {
            t3 &z = zones[i] ;
            double lft { get<0>(z) }, rgt { get<1>(z) }, w { get<2>(z) } ;
            s += (lft*lft+3*lft*rgt+rgt*rgt)*w*w*w/15 ;
            double ta { (lft+rgt)*w/2 } ;
            double cx { x0 + (lft+2*rgt)/(3*(lft+rgt))*w } ;
            if (i)
                s += 2 * (cx - cxa / sa) * sa * ta ;
            a2 += ta*(2*sa+ta) ;
            sa += ta ;
            cxa += cx * ta ;
            x0 += w ;
        }
        r += s / a2 ;
        swap(xs, ys) ;
    }
    reverse(xs.begin(), xs.end()) ;
    reverse(ys.begin(), ys.end()) ;
    cout << r << endl ;
}

```