Contents

1 1 Templates

1.1 Start

```
// .vimrc
syn on
set mouse=a sw=4 ts=4 ai si nu wrap
nnoremap ; :

// Terminal: comparing generated output to sample output
./my_program < sample.in | diff sample.out -
```

1.2 Template - C++

```cpp
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
static bool DBG = 1;

ll mod(ll a, ll b) { return ((a%b)+b)%b; }

int main() {
    ios_base::sync_with_stdio(0);
    cout << fixed << setprecision(15);
    int n;
    cin >> n;
    cout << n << endl;
    return 0;
}
```

1.3 Template - Java

```java
import java.util.*;
import java.math.*;
import java.io.*;

class modelo {
    static final double EPS = 1.e-10;
    static final boolean DBG = true;

    private static int cmp(double x, double y = 0, double tol = EPS) {
        return (x <= y + tol)? (x + tol < y)? -1 : 0 : 1;
    }

    public static void main(String[] argv) {
        Scanner s = new Scanner(System.in);
    }
}
```

2 2 Data Structures

2.1 BIT

```
template<typename T> struct BIT{
    int S;
    vector<T> v;

    BIT<T>(int _S){
        S = _S;
        v.resize(S+1);
    }
    void update(int i, T k){
        for(i++; i<=S; i+=i&-i)
            v[i] = v[i] + k;
    }
    T read(int i){
        T sum = 0;
        for(i++; i; i-=i&-i)
            sum = sum + v[i];
        return sum;
    }
    T read(int l, int r){
        return read(r) - read(l-1);
    }
};
```

2.2 KD Tree

```
// -----------------------------------------------------------------
// A straightforward, but probably sub-optimal KD-tree implmentation
// that's probably good enough for most things (current it's a
// 2D-tree)
//
//  - constructs from n points in O(n lg^2 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) {
            if (p.y < y0)    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) {
            if (p.y < y0)    return pdist2(point(x1, y0), p);
            else if (p.y > y1) 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 <<
            ")"
            << " is " << tree.nearest(q) << endl;
    }
}
```

2.3 LCA

```
struct lca {
    int L, N;
    vector<int> depth, size, link;

    lca(){}
    lca(const vvi &graph, int root = 0) {
        N = graph.size();
        for (L = 0; (1 << L) <= N; L++);
        depth.resize(N);
        size.resize(N);
        link.resize(L*N);
        init(root, root, graph);
    }
    void init(int loc, int par, const vvi &graph) {
        link[loc] = par;
        for (int l = 1; l < L; l++)
            link[l*N + loc] = link[(l-1)*N + link[(l-1)*N + loc]];
        for (int nbr : graph[loc]) {
            if (nbr == par) continue;
            depth[nbr] = depth[loc] + 1;
            init(nbr, loc, graph);
            size[loc] += size[nbr];
        }
        size[loc]++;
    }
    int above(int loc, int dist) {
        for (int l = 0; l < L; l++)
            if ((dist >> l)&1)
                loc = link[l*N + loc];
```

```
        return loc;
    }
    int find(int u, int v) {
        if (depth[u] > depth[v]) swap(u, v);
        v = above(v, depth[v] - depth[u]);
        if (u == v) return u;
        for (int l = L - 1; l >= 0; l--) {
            if (link[l*N + u] != link[l*N + v])
                u = link[l*N + u], v = link[l*N + v];
        }
        return link[u];
    }
};
```

2.4 Lazy Segment Tree

```
// Modular lazy segment tree
// It takes a type T for vertex values and a type U for update
// operations. Type T should have an operator '+' specifying how to
// combine vertices. Type U should have an operator '()' specifying how
// to apply updates to vertices and an operator '+' for combining two
// updates. Example below.
template<typename T, typename U> struct seg_tree_lazy {
    int S, H;
    T zero;
    vector<T> value;
    U noop;
    vector<bool> dirty;
    vector<U> prop;

    seg_tree_lazy<T, U>(int _S, T _zero = T(), U _noop = U()) {
        zero = _zero, noop = _noop;
        for (S = 1, H = 1; S < _S; ) S *= 2, H++;
        value.resize(2*S, zero);
        dirty.resize(2*S, false);
        prop.resize(2*S, noop);
    }
    void set_leaves(vector<T> &leaves) {
        copy(leaves.begin(), leaves.end(), value.begin() + S);
        for (int i = S - 1; i > 0; i--)
            value[i] = value[2 * i] + value[2 * i + 1];
    }
    void apply(int i, U &update) {
        value[i] = update(value[i]);
        if(i < S) {
            prop[i] = prop[i] + update;
            dirty[i] = true;
        }
    }
    void rebuild(int i) {
        for (int l = i/2; l; l /= 2) {
            T combined = value[2*l] + value[2*l+1];
            value[l] = prop[l](combined);
        }
    }
    void propagate(int i) {
        for (int h = H; h > 0; h--) {
            int l = i >> h;
            if (dirty[l]) {
                apply(2*l, prop[l]);
                apply(2*l+1, prop[l]);
                prop[l] = noop;
                dirty[l] = false;
            }
        }
    }
    void upd(int i, int j, U update) {
        i += S, j += S;
        propagate(i), propagate(j);
        for (int l = i, r = j; l <= r; l /= 2, r /= 2) {
            if((l&1) == 1) apply(l++, update);
            if((r&1) == 0) apply(r--, update);
        }
        rebuild(i), rebuild(j);
    }
    T query(int i, int j){
        i += S, j += S;
        propagate(i), propagate(j);
        T res_left = zero, res_right = zero;
        for(; i <= j; i /= 2, j /= 2){
            if((i&1) == 1) res_left = res_left + value[i++];
            if((j&1) == 0) res_right = value[j--] + res_right;
        }
        return res_left + res_right;
    }
};

// Example that supports following operations:
// 1: Add amount V to the values in range [L,R].
// 2: Reset the values in range [L,R] to value V.
// 3: Query for the sum of the values in range [L,R].

// Here's what T looks like:
struct node {
    int sum, width;
    node operator+(const node &n) {
        return { sum + n.sum, width + n.width }
    }
};
// Here's what U looks like:
struct update {
    bool type; // 0 for add, 1 for reset
    int value;
    node operator()(const node &n) {
        if(type) return { n.width * value, n.width };
        else return { n.sum + n.width * value, n.width };
    }
    update operator+(const update &u) {
        if(u.type) return u;
        return { type, value + u.value };
    }
};
int main() {
    int N = 100;
    node zero = {0,0};
    update noop = {false, 0};
    vector<node> leaves(N, {0,1});
```

```
    seg_tree_lazy<node, update> st(N, zero, noop);
    st.set_leaves(leaves);
}
```

2.5 Segment Tree 2D

```
#define Max 506
#define INF (1 << 30)
int P[Max][Max]; // container for 2D grid

/* 2D Segment Tree node */
struct Point {
    int x, y, mx;
    Point() {}
    Point(int x, int y, int mx) : x(x), y(y), mx(mx) {}

    bool operator < (const Point& other) const {
        return mx < other.mx;
    }
};

struct Segtree2d {
    Point T[2 * Max * Max];
    int n, m;
    // initialize and construct segment tree
    void init(int n, int m) {
        this -> n = n;
        this -> m = m;
        build(1, 1, 1, n, m);
    }
    // build a 2D segment tree from data [ (a1, b1), (a2, b2) ]
    // Time: O(n logn)
    Point build(int node, int a1, int b1, int a2, int b2) {
        // out of range
        if (a1 > a2 or b1 > b2)
            return def();

        // if it is only a single index, assign value to node
        if (a1 == a2 and b1 == b2)
            return T[node] = Point(a1, b1, P[a1][b1]);

        // split the tree into four segments
        T[node] = def();
        T[node] = maxNode(T[node], build(4 * node - 2, a1, b1, (a1 + a2) /
            2, (b1 + b2) / 2 ) );
        T[node] = maxNode(T[node], build(4 * node - 1, (a1 + a2) / 2 + 1,
            b1, a2, (b1 + b2) / 2 ));
        T[node] = maxNode(T[node], build(4 * node + 0, a1, (b1 + b2) / 2 +
            1, (a1 + a2) / 2, b2) );
        T[node] = maxNode(T[node], build(4 * node + 1, (a1 + a2) / 2 + 1,
            (b1 + b2) / 2 + 1, a2, b2) );
        return T[node];
    }
    // helper function for query(int, int, int, int);
    Point query(int node, int a1, int b1, int a2, int b2, int x1, int y1,
        int x2, int y2) {
        // if we out of range, return dummy
        if (x1 > a2 or y1 > b2 or x2 < a1 or y2 < b1 or a1 > a2 or b1 > b2)
            return def();
        // if it is within range, return the node
        if (x1 <= a1 and y1 <= b1 and a2 <= x2 and b2 <= y2)
            return T[node];
        // split into four segments
        Point mx = def();
        mx = maxNode(mx, query(4 * node - 2, a1, b1, (a1 + a2) / 2, (b1 +
            b2) / 2, x1, y1, x2, y2) );
        mx = maxNode(mx, query(4 * node - 1, (a1 + a2) / 2 + 1, b1, a2,
            (b1 + b2) / 2, x1, y1, x2, y2) );
        mx = maxNode(mx, query(4 * node + 0, a1, (b1 + b2) / 2 + 1, (a1 +
            a2) / 2, b2, x1, y1, x2, y2) );
        mx = maxNode(mx, query(4 * node + 1, (a1 + a2) / 2 + 1, (b1 + b2)
            / 2 + 1, a2, b2, x1, y1, x2, y2));
        // return the maximum value
        return mx;
    }
    // query from range [ (x1, y1), (x2, y2) ]
    // Time: O(logn)
    Point query(int x1, int y1, int x2, int y2) {
        return query(1, 1, 1, n, m, x1, y1, x2, y2);
    }

    // helper function for update(int, int, int);
    Point update(int node, int a1, int b1, int a2, int b2, int x, int y,
        int value) {
        if (a1 > a2 or b1 > b2)
            return def();
        if (x > a2 or y > b2 or x < a1 or y < b1)
            return T[node];
        if (x == a1 and y == b1 and x == a2 and y == b2)
            return T[node] = Point(x, y, value);
        Point mx = def();
        mx = maxNode(mx, update(4 * node - 2, a1, b1, (a1 + a2) / 2, (b1 +
            b2) / 2, x, y, value) );
        mx = maxNode(mx, update(4 * node - 1, (a1 + a2) / 2 + 1, b1, a2,
            (b1 + b2) / 2, x, y, value));
        mx = maxNode(mx, update(4 * node + 0, a1, (b1 + b2) / 2 + 1, (a1 +
            a2) / 2, b2, x, y, value));
        mx = maxNode(mx, update(4 * node + 1, (a1 + a2) / 2 + 1, (b1 + b2)
            / 2 + 1, a2, b2, x, y, value) );
        return T[node] = mx;
    }
    // update the value of (x, y) index to 'value'
    // Time: O(logn)
    Point update(int x, int y, int value) {
        return update(1, 1, 1, n, m, x, y, value);
    }
    // utility functions; these functions are virtual because they will
        be overridden in child class
    virtual Point maxNode(Point a, Point b) {
        return max(a, b);
    }
    // dummy node
    virtual Point def() {
        return Point(0, 0, -INF);
    }
};
```

```
/* 2D Segment Tree for range minimum query; a override of Segtree2d class
    */
struct Segtree2dMin : Segtree2d {
    // overload maxNode() function to return minimum value
    Point maxNode(Point a, Point b) { return min(a, b); }
    Point def() { return Point(0, 0, INF); }
};

// initialize class objects
Segtree2d Tmax;
Segtree2dMin Tmin;

/* Drier program */
int main(void) {
    int n, m;
    // input
    scanf("%d %d", &n, &m);
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= m; j++)
            scanf("%d", &P[i][j]);
    // initialize
    Tmax.init(n, m);
    Tmin.init(n, m);
    // query
    int x1, y1, x2, y2;
    scanf("%d %d %d %d", &x1, &y1, &x2, &y2);
    Tmax.query(x1, y1, x2, y2).mx;
    Tmin.query(x1, y1, x2, y2).mx;
    // update
    int x, y, v;
    scanf("%d %d %d", &x, &y, &v);
    Tmax.update(x, y, v);
    Tmin.update(x, y, v);
    return 0;
}
```

2.6 Segment Tree

```
template<typename T> struct seg_tree {
    int S;
    T zero;
    vector<T> value;

    seg_tree<T>(int _S, T _zero = T()) {
        S = _S, zero = _zero;
        value.resize(2*S+1, zero);
    }
    void set_leaves(vector<T> &leaves) {
        copy(leaves.begin(), leaves.end(), value.begin() + S);
        for (int i = S - 1; i > 0; i--)
            value[i] = value[2 * i] + value[2 * i + 1];
    }
    void upd(int i, T v) {
        i += S;
        value[i] = v;
        while(i>1){
            i/=2;
            value[i] = value[2*i] + value[2*i+1];
        }
    }
    T query(int i, int j) {
        T res_left = zero, res_right = zero;
        for(i += S, j += S; i <= j; i /= 2, j /= 2){
            if((i&1) == 1) res_left = res_left + value[i++];
            if((j&1) == 0) res_right = value[j--] + res_right;
        }
        return res_left + res_right;
    }
};
```

2.7 Union Find

```
// (struct) also keeps track of sizes
struct union_find {
    vector<int> P,S;

    union_find(int N) {
        P.resize(N), S.resize(N, 1);
        for(int i = 0; i < N; i++) P[i] = i;
    }
    int rep(int i) {return (P[i] == i) ? i : P[i] = rep(P[i]);}
    bool unio(int a, int b) {
        a = rep(a), b = rep(b);
        if(a == b) return false;
        P[b] = a;
        S[a] += S[b];
        return true;
    }
};
// (Shorter) union-find set: the vector/array contains the parent of each
    node
int find(vector <int>& C, int x){return (C[x]==x) ? x : C[x]=find(C,
    C[x]);} //C++
int find(int x){return (C[x]==x)?x:C[x]=find(C[x]);} //C
```

3 3 Graph

3.1 2-SAT

```
struct two_sat {
    int N;
    vector<vector<int>> impl;

    two_sat(int _N) {
        N = _N;
        impl.resize(2 * N);
    }
    void add_impl(int var1, bool neg1, int var2, bool neg2) {
        impl[2 * var1 + neg1].push_back(2 * var2 + neg2);
        impl[2 * var2 + !neg2].push_back(2 * var1 + !neg1);
    }
    void add_clause(int var1, bool neg1, int var2, bool neg2) {
        add_impl(var1, !neg1, var2, neg2);
    }
```

```
    void add_clause(int var1, bool neg1) {
        add_clause(var1, neg1, var1, neg1);
    }

    int V, L, C;
    stack<int> view;

    int dfs(int loc) {
        visit[loc] = V;
        label[loc] = L++;
        int low = label[loc];
        view.push(loc);
        in_view[loc] = true;
        for (int nbr : impl[loc]) {
            if(!visit[nbr]) low = min(low, dfs(nbr));
            else if(in_view[nbr]) low = min(low, label[nbr]);
        }
        if(low == label[loc]) {
            while (true) {
                int mem = view.top();
                comp[mem] = C;
                in_view[mem] = false;
                view.pop();
                if(mem == loc) break;
            }
            C++;
        }
        return low;
    }

    vector<int> visit, label, comp, in_view;

    void reset(vector<int> &v) {
        v.resize(2 * N);
        fill(v.begin(), v.end(), 0);
    }
    bool consistent() {
        V = 0, L = 0, C = 0;
        reset(visit), reset(label), reset(comp), reset(in_view);
        for (int i = 0; i < 2 * N; i++) {
            if(!visit[i]) {
                V++;
                dfs(i);
            }
        }
        for (int i = 0; i < N; i++)
            if(comp[2 * i] == comp[2 * i + 1]) return false;
        return true;
    }
};
```

3.2 Dense Dijkstra

```
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;
    }
}
```

3.3 Dijkstra

```
// Implementation of Dijkstra's algorithm using adjacency lists
// and priority queue for efficiency.
//
// Running time: O(|E| log |V|)
typedef pair<int,int> PII;
const int INF = 2000000000;

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

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

3.4 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 = 100;
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;
}
```

3.5 Heavy Light

```
template<typename T> struct heavy_light {
    lca links;
    seg_tree<T> st;
    vector<int> preorder, index, jump;

    heavy_light(const vvi &graph, int root) {
        links = lca(graph, 0);
        st = seg_tree<T>(graph.size());
        index.resize(graph.size()), jump.resize(graph.size());
        dfs(root, root, root, graph);
    }
    void dfs(int loc, int par, int lhv, const vvi &graph) {
        jump[loc] = lhv;
        index[loc] = preorder.size();
        preorder.push_back(loc);
        vector<int> ch = graph[loc];
        sort(ch.begin(), ch.end(), [&](int i, int j) {
                return links.size[i] > links.size[j]; });
        if (loc != par) ch.erase(ch.begin());
        for (int c = 0; c < ch.size(); c++)
            dfs(ch[c], loc, c ? ch[c] : lhv, graph);
    }
    void assign(int loc, T value) {
        st.upd(index[loc], value);
    }
    T __sum(int u, int r) {
        T res;
        while (u != r) {
            int go = max(index[r] + 1, index[jump[u]]);
            res = res + st.query(go, index[u]);
            u = links.link[preorder[go]];
        }
        return res;
    }
    T sum(int u, int v) {
        int r = links.find(u, v);
        return st.query(index[r], index[r]) + __sum(u, r) + __sum(v, r);
    }
};
```

3.6 Poset Width

```
// requires bipartite graph (4.1)
vector<int> width(vector<vector<int>> poset) {
    int N = poset.size();
    bipartite_graph g(N, N);
    for (int i = 0; i < N; i++) {
        for (int j : poset[i])
            g.edge(j, i);
    }
    g.matching();
    vector<bool> vis[2];
    vis[false].resize(2 * N, false);
    vis[true].resize(2 * N, false);
    for (int i = 0; i < N; i++) {
        if (g.match[i] != -1) continue;
        if (vis[false][i]) continue;
        queue<pair<bool, int>> bfs;
        bfs.push(make_pair(false, i));
        vis[false][i] = true;
        while (!bfs.empty()) {
            bool inm = bfs.front().first;
            int loc = bfs.front().second;
            bfs.pop();
            for (int nbr : g.adj[loc]) {
                if (vis[!inm][nbr]) continue;
                if ((g.match[loc] == nbr) ^ inm) continue;
                vis[!inm][nbr] = true;
                bfs.push(make_pair(!inm, nbr));
            }
        }
    }
```

```
    vector<bool> inz(2 * N, false);
    for (int i = 0; i < 2 * N; i++)
        inz[i] = vis[true][i] || vis[false][i];
    vector<bool> ink(N, false);
    for (int i = 0; i < N; i++)
        if (!inz[i])
            ink[i]= true;
    for (int i = N; i < 2 * N; i++)
        if (inz[i])
            ink[i - N] = true;
    vector<int> res;
    for (int i = 0; i < N; i++) {
        if (!ink[i])
            res.push_back(i);
    }
    return res;
}
```

3.7 SCC

```
struct edge{int e, nxt;};
int V, E;
edge e[MAXE], er[MAXE];
int sp[MAXV], spr[MAXV];
int group_cnt, group_num[MAXV];
bool v[MAXV];
int stk[MAXV];
void fill_forward(int x) {
    int i;
    v[x]=true;
    for(i=sp[x];i;i=e[i].nxt) if(!v[e[i].e]) fill_forward(e[i].e);
    stk[++stk[0]]=x;
}
void fill_backward(int x) {
    int i;
    v[x]=false;
    group_num[x]=group_cnt;
    for(i=spr[x];i;i=er[i].nxt) if(v[er[i].e]) fill_backward(er[i].e);
}
void add_edge(int v1, int v2) { //add edge v1->v2
    e [++E].e=v2; e [E].nxt=sp [v1]; sp [v1]=E;
    er[ E].e=v1; er[E].nxt=spr[v2]; spr[v2]=E;
}
void SCC() {
    int i;
    stk[0]=0;
    memset(v, false, sizeof(v));
    for(i=1;i<=V;i++) if(!v[i]) fill_forward(i);
    group_cnt=0;
    for(i=stk[0];i>=1;i--) if(v[stk[i]]){group_cnt++;
        fill_backward(stk[i]);}
}
```

3.8 Topological Sort

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

typedef double TYPE;
typedef vector<TYPE> VT;
typedef vector<VT> VVT;
typedef vector<int> VI;
typedef vector<VI> VVI;

bool TopologicalSort (const VVI &w, VI &order){
    int n = w.size();
    VI parents (n);
    queue<int> q;
    order.clear();
    for (int i = 0; i < n; i++){
        for (int j = 0; j < n; j++)
            if (w[j][i]) parents[i]++;
        if (parents[i] == 0) q.push (i);
    }
    while (q.size() > 0){
        int i = q.front();
        q.pop();
        order.push_back (i);
        for (int j = 0; j < n; j++) if (w[i][j]){
            parents[j]--;
            if (parents[j] == 0) q.push (j);
        }
    }
    return (order.size() == n);
}
```

4 4 Combinatorial Optimization

4.1 Bipartite Graph

```
struct bipartite_graph {
    int A, B;
    vector<vector<int>> adj;

    bipartite_graph(int _A, int _B) {
        A = _A, B = _B;
        adj.resize(A + B);
    }
    void edge(int i, int j) {
        adj[i].push_back(A+j);
        adj[A+j].push_back(i);
    }

    vector<int> visit, match;
```

```
    bool augment(int loc, int run) {
        if(visit[loc] == run) return false;
        visit[loc] = run;
        for (int nbr : adj[loc]) {
            if (match[nbr] == -1 || augment(match[nbr], run)) {
                match[loc] = nbr, match[nbr] = loc;
                return true;
            }
        }
        return false;
    }
    int matching() {
        visit = vector<int>(A+B, -1);
        match = vector<int>(A+B, -1);
        int ans = 0;
        for (int i = 0; i < A; i++)
            ans += augment(i, i);
        return ans;
    }
    vector<bool> vertex_cover() {
        vector<bool> res(A + B, false);
        queue<int> bfs;
        for (int i = 0; i < A; i++) {
            if (match[i] == -1) bfs.push(i);
            else res[i] = true;
        }
        while (!bfs.empty()) {
            int loc = bfs.front();
            bfs.pop();
            for (int nbr : adj[loc]) {
                if (res[nbr]) continue;
                res[nbr] = true;
                int loc2 = match[nbr];
                if (loc2 == -1) continue;
                res[loc2] = false;
                bfs.push(loc2);
            }
        }
        return res;
    }
};
```

4.2 Max Flow - Dinic

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

struct Edge {
    int from, to, cap, flow, index;
    Edge(int from, int to, int cap, int flow, int index) :
        from(from), to(to), cap(cap), flow(flow), index(index) {}
    ll rcap() { return cap - flow; }
};

struct Dinic {
    int N;
    vector<vector<Edge> > G;
    vector<vector<Edge *> > Lf;
    vector<int> layer;
    vector<int> Q;

    Dinic(int N) : N(N), G(N), Q(N) {}

    void AddEdge(int from, int to, int cap) {
        if (from == to) return;
        G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
        G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1));
    }
    ll BlockingFlow(int s, int t) {
        layer.clear(); layer.resize(N, -1);
        layer[s] = 0;
        Lf.clear(); Lf.resize(N);
        int head = 0, tail = 0;
        Q[tail++] = s;
        while (head < tail) {
            int x = Q[head++];
            for (int i = 0; i < G[x].size(); i++) {
                Edge &e = G[x][i]; if (e.rcap() <= 0) continue;
                if (layer[e.to] == -1) {
                    layer[e.to] = layer[e.from] + 1;
                    Q[tail++] = e.to;
                }
                if (layer[e.to] > layer[e.from]) {
                    Lf[e.from].push_back(&e);
                }
            }
        }
        if (layer[t] == -1) return 0;
        ll totflow = 0;
        vector<Edge *> P;
        while (!Lf[s].empty()) {
            int curr = P.empty() ? s : P.back()->to;
            if (curr == t) { // Augment
                ll amt = P.front()->rcap();
                for (int i = 0; i < P.size(); ++i) {
                    amt = min(amt, P[i]->rcap());
                }
                totflow += amt;
                for (int i = P.size() - 1; i >= 0; --i) {
                    P[i]->flow += amt;
                    G[P[i]->to][P[i]->index].flow -= amt;
                    if (P[i]->rcap() <= 0) {
                        Lf[P[i]->from].pop_back();
                        P.resize(i);
                    }
```

```
            }
        } else if (Lf[curr].empty()) { // Retreat
            P.pop_back();
            for (int i = 0; i < N; ++i)
                for (int j = 0; j < Lf[i].size(); ++j)
                    if (Lf[i][j]->to == curr)
                        Lf[i].erase(Lf[i].begin() + j);
        } else { // Advance
            P.push_back(Lf[curr].back());
        }
    }
    return totflow;
}
ll GetMaxFlow(int s, int t) {
    ll totflow = 0;
    while (ll flow = BlockingFlow(s, t))
        totflow += flow;
    return totflow;
}
};
```

4.3 Min Cost Matching

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

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;
}
```

4.4 Min Cost Max Flow

```
// Min cost max flow algorithm using an adjacency matrix. If you
// want just regular max flow, setting all edge costs to 1 gives
// running time O(|E|^2 |V|).
//
// Running time: O(min(|V|^2 * totflow, |V|^3 * totcost))
//
// INPUT: cap -- a matrix such that cap[i][j] is the capacity of
//        a directed edge from node i to node j
//   cost -- a matrix such that cost[i][j] is the (positive)
//       cost of sending one unit of flow along a
//       directed edge from node i to node j
//   source -- starting node
//   sink -- ending node
//
// OUTPUT: max flow and min cost; the matrix flow will contain
//    the actual flow values (note that unlike in the MaxFlow
//    code, you don't need to ignore negative flow values -- there
//    shouldn't be any)
typedef vector<ll> vll;
typedef vector<vll> vvll;
const ll INF = 1LL << 60;

struct MCMF {
    int N;
    vll found, dad, dist, pi;
    vvll cap, flow, cost;

    MCMF(int N) : N(N), cap(N, vll(N)), flow(cap), cost(cap),
    dad(N), found(N), pi(N), dist(N+1) {};

    void add_edge(int from, int to, ll ca, ll co) {
        cap[from][to] = ca; cost[from][to] = co; }
    bool search(int source, int sink) {
        fill(found.begin(), found.end(), 0);
        fill(dist.begin(), dist.end(), INF);
        dist[source] = 0;
        while(source != N) {
            int best = N;
            found[source] = 1;
            for(int k = 0; k < N; k++) {
                if(found[k]) continue;
                if(flow[k][source]) {
                    ll val = dist[source] + pi[source] - pi[k] -
                        cost[k][source];
                    if(dist[k] > val) {
                        dist[k] = val;
                        dad[k] = source;
                    }
                }
                if(flow[source][k] < cap[source][k]) {
                    ll val = dist[source] + pi[source] - pi[k] +
                        cost[source][k];
                    if(dist[k] > val) {
                        dist[k] = val;
                        dad[k] = source;
                    }
                }
                if(dist[k] < dist[best]) best = k;
            }
            source = best;
        }
        for(int k = 0; k < N; k++)
            pi[k] = min((ll)(pi[k] + dist[k]), INF);
        return found[sink];
    }
    pair<ll,ll> mcmf(int source, int sink) {
        ll totflow = 0, totcost = 0;
        while(search(source, sink)) {
            ll amt = INF;
            for(int x = sink; x != source; x = dad[x])
                amt = min(amt, (ll)(flow[x][dad[x]] != 0 ?
                        flow[x][dad[x]] : cap[dad[x]][x] -
                            flow[dad[x]][x]));
            for(int x = sink; x != source; x = dad[x]) {
                if(flow[x][dad[x]] != 0) {
                    flow[x][dad[x]] -= amt;
                    totcost -= amt * cost[x][dad[x]];
                } else {
                    flow[dad[x]][x] += amt;
                    totcost += amt * cost[dad[x]][x];
                }
            }
            totflow += amt;
        }
        return {totflow, totcost};
    }
};
```

4.5 Min Cut

```
// Adjacency matrix implementation of Stoer-Wagner min cut algorithm.
//
// Running time:
//   O(|V|^3)
//
// INPUT:
//   - graph, constructed using AddEdge()
//
```

```
// OUTPUT:
//   - (min cut value, nodes in half of min cut)
typedef vector<int> VI;
typedef vector<VI> VVI;
const int INF = 1000000000;

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

5 5 Geometry

5.1 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

#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
}
```

5.2 Delaunay

```
// Slow but simple Delaunay triangulation. Does not handle
// degenerate cases (from O'Rourke, Computational Geometry in C)
//
// Running time: O(n^4)
//
```

```
// INPUT: x[] = x-coordinates
//        y[] = y-coordinates
//
// OUTPUT:  triples = a vector containing m triples of indices
//          corresponding to triangle vertices
typedef double T;

struct triple {
    int i, j, k;
    triple() {}
    triple(int i, int j, int k) : i(i), j(j), k(k) {}
};
vector<triple> delaunayTriangulation(vector<T>& x, vector<T>& y) {
    int n = x.size();
    vector<T> z(n);
    vector<triple> ret;
    for (int i = 0; i < n; i++)
        z[i] = x[i] * x[i] + y[i] * y[i];
    for (int i = 0; i < n-2; i++) {
        for (int j = i+1; j < n; j++) {
            for (int k = i+1; k < n; k++) {
                if (j == k) continue;
                double xn = (y[j]-y[i])*(z[k]-z[i]) -
                    (y[k]-y[i])*(z[j]-z[i]);
                double yn = (x[k]-x[i])*(z[j]-z[i]) -
                    (x[j]-x[i])*(z[k]-z[i]);
                double zn = (x[j]-x[i])*(y[k]-y[i]) -
                    (x[k]-x[i])*(y[j]-y[i]);
                bool flag = zn < 0;
                for (int m = 0; flag && m < n; m++)
                    flag = flag && ((x[m]-x[i])*xn +
                            (y[m]-y[i])*yn +
                            (z[m]-z[i])*zn <= 0);
                if (flag) ret.push_back(triple(i, j, k));
            }
        }
    }
    return ret;
}
int main() {
    T xs[]={0, 0, 1, 0.9};
    T ys[]={0, 1, 0, 0.9};
    vector<T> x(&xs[0], &xs[4]), y(&ys[0], &ys[4]);
    vector<triple> tri = delaunayTriangulation(x, y);
    //expected: 0 1 3
    //          0 3 2
    int i;
    for(i = 0; i < tri.size(); i++)
        printf("%d %d %d\n", tri[i].i, tri[i].j, tri[i].k);
    return 0;
}
```

5.3 Geometry

```
// C++ routines for computational geometry.
double INF = 1e100;
double EPS = 1e-12;

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

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

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

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

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

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

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

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

bool LinesCollinear(PT a, PT b, PT c, PT d) {
    return LinesParallel(a, b, c, d)
```

```
        && fabs(cross(a-b, a-c)) < EPS
        && fabs(cross(c-d, c-a)) < EPS;
}

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

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

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

// determine if point is in a possibly non-convex polygon (by William
// Randolph Franklin); returns 1 for strictly interior points, 0 for
// strictly exterior points, and 0 or 1 for the remaining points.
// Note that it is possible to convert this into an *exact* test using
// integer arithmetic by taking care of the division appropriately
// (making sure to deal with signs properly) and then by writing exact
// tests for checking point on polygon boundary
bool PointInPolygon(const vector<PT> &p, PT q) {
    bool c = 0;
    for (int i = 0; i < p.size(); i++){
        int j = (i+1)%p.size();
        if ((p[i].y <= q.y && q.y < p[j].y ||
                p[j].y <= q.y && q.y < p[i].y) &&
            q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) /
                (p[j].y - p[i].y))
            c = !c;
    }
    return c;
}

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

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

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

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

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

PT ComputeCentroid(const vector<PT> &p) {
    PT c(0,0);
    double scale = 6.0 * ComputeSignedArea(p);
```

```
    for (int i = 0; i < p.size(); i++){
        int j = (i+1) % p.size();
        c = c + (p[i]+p[j])*(p[i].x*p[j].y - p[j].x*p[i].y);
    }
    return c / scale;
}

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

// Plane distance between parallel planes aX + bY + cZ + d1 = 0 and
// aX + bY + cZ + d2 = 0 is abs(d1 - d2) / 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);
}

int main() {
    // expected: (-5,2)
    cerr << RotateCCW90(PT(2,5)) << endl;
    // expected: (5,-2)
    cerr << RotateCW90(PT(2,5)) << endl;
    // expected: (-5,2)
    cerr << RotateCCW(PT(2,5),M_PI/2) << endl;
    // expected: (5,2)
    cerr << ProjectPointLine(PT(-5,-2), PT(10,4), PT(3,7)) << endl;
    // expected: (5,2) (7.5,3) (2.5,1)
    cerr << ProjectPointSegment(PT(-5,-2), PT(10,4), PT(3,7)) << " "
        << ProjectPointSegment(PT(7.5,3), PT(10,4), PT(3,7)) << " "
        << ProjectPointSegment(PT(-5,-2), PT(2.5,1), PT(3,7)) << endl;
    // expected: 6.78903
    cerr << DistancePointPlane(4,-4,3,2,-2,5,-8) << endl;
    // expected: 1 0 1
    cerr << LinesParallel(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
        << LinesParallel(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
        << LinesParallel(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;
    // expected: 0 0 1
    cerr << LinesCollinear(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
        << LinesCollinear(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
        << LinesCollinear(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;
    // expected: 1 1 1 0
    cerr << SegmentsIntersect(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) << " "
        << SegmentsIntersect(PT(0,0), PT(2,4), PT(4,3), PT(0,5)) << " "
        << SegmentsIntersect(PT(0,0), PT(2,4), PT(2,-1), PT(-2,1)) << " "
        << SegmentsIntersect(PT(0,0), PT(2,4), PT(5,5), PT(1,7)) << endl;
    // expected: (1,2)
    cerr << ComputeLineIntersection(PT(0,0), PT(2,4), PT(3,1), PT(-1,3))
         << endl;
    // expected: (1,1)
    cerr << ComputeCircleCenter(PT(-3,4), PT(6,1), PT(4,5)) << endl;
    vector<PT> v;
    v.push_back(PT(0,0));
    v.push_back(PT(5,0));
    v.push_back(PT(5,5));
    v.push_back(PT(0,5));
    // expected: 1 1 1 0 0
    cerr << PointInPolygon(v, PT(2,2)) << " "
        << PointInPolygon(v, PT(2,0)) << " "
        << PointInPolygon(v, PT(0,2)) << " "
        << PointInPolygon(v, PT(5,2)) << " "
        << PointInPolygon(v, PT(2,5)) << endl;
    // expected: 0 1 1 1 1
    cerr << PointOnPolygon(v, PT(2,2)) << " "
        << PointOnPolygon(v, PT(2,0)) << " "
        << PointOnPolygon(v, PT(0,2)) << " "
        << PointOnPolygon(v, PT(5,2)) << " "
        << PointOnPolygon(v, PT(2,5)) << endl;
    // expected: (1,6)
    //         (5,4) (4,5)
    //         blank line
    //         (4,5) (5,4)
    //         blank line
    //         (4,5) (5,4)
    vector<PT> u = CircleLineIntersection(PT(0,6), PT(2,6), PT(1,1), 5);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = CircleLineIntersection(PT(0,9), PT(9,0), PT(1,1), 5);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = CircleCircleIntersection(PT(1,1), PT(10,10), 5, 5);
```

```
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = CircleCircleIntersection(PT(1,1), PT(8,8), 5, 5);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 10, sqrt(2.0)/2.0);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 5, sqrt(2.0)/2.0);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    // area should be 5.0
    // centroid should be (1.1666666, 1.166666)
    PT pa[] = { PT(0,0), PT(5,0), PT(1,1), PT(0,5) };
    vector<PT> p(pa, pa+4);
    PT c = ComputeCentroid(p);
    cerr << "Area: " << ComputeArea(p) << endl;
    cerr << "Centroid: " << c << endl;
}
```

6 6 Numerics

6.1 Euclid

```
// 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.
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) {
    int tmp;
    while(b){a%=b; tmp=a; a=b; b=tmp;}
    return a;
}
// computes lcm(a,b)
int lcm(int a, int b) {
    return a/gcd(a,b)*b;
}
// returns d = 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 solutions;
    int d = extended_euclid(a, n, x, y);
    if (!(b%d)) {
        x = mod (x*(b/d), n);
        for (int i = 0; i < d; i++)
            solutions.push_back(mod(x + i*(n/d), n));
    }
    return solutions;
}
// computes b such that ab = 1 (mod n), returns -1 on failure
int mod_inverse(int a, int n) {
    int x, y;
    int d = extended_euclid(a, n, x, y);
    if (d > 1) return -1;
    return mod(x,n);
}
// Chinese remainder theorem (special case): find z such that
// z % x = a, z % y = b. Here, z is unique modulo M = lcm(x,y).
// Return (z,M). On failure, M = -1.
PII chinese_remainder_theorem(int x, int a, int y, int b) {
    int s, t;
    int d = extended_euclid(x, y, s, t);
    if (a%d != b%d) return make_pair(0, -1);
    return make_pair(mod(s*b*x+t*a*y,x*y)/d, x*y/d);
}
// Chinese remainder theorem: find z such that
// z % x[i] = a[i] for all i. Note that the solution is
// unique modulo M = lcm_i (x[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 &x, const VI &a) {
    PII ret = make_pair(a[0], x[0]);
    for (int i = 1; i < x.size(); i++) {
        ret = chinese_remainder_theorem(ret.second, ret.first, x[i], a[i]);
        if (ret.second == -1) break;
    }
    return ret;
}
// computes x and y such that ax + by = c; on failure, x = y =-1
void linear_diophantine(int a, int b, int c, int &x, int &y) {
    int d = gcd(a,b);
    if (c%d) {
        x = y = -1;
    } else {
        x = c/d * mod_inverse(a/d, b/d);
        y = (c-a*x)/b;
    }
}

int main() {
    // expected: 2
    cout << gcd(14, 30) << endl;
    // expected: 2 -2 1
    int x, y;
    int d = extended_euclid(14, 30, x, y);
    cout << d << " " << x << " " << y << endl;
    // expected: 95 45
    VI sols = modular_linear_equation_solver(14, 30, 100);
    for (int i = 0; i < (int) sols.size(); i++) cout << sols[i] << " ";
    cout << endl;
    // expected: 8
```

```
    cout << mod_inverse(8, 9) << endl;
    // expected: 23 56
    //        11 12
    int xs[] = {3, 5, 7, 4, 6};
    int as[] = {2, 3, 2, 3, 5};
    PII ret = chinese_remainder_theorem(VI (xs, xs+3), VI(as, as+3));
    cout << ret.first << " " << ret.second << endl;
    ret = chinese_remainder_theorem (VI(xs+3, xs+5), VI(as+3, as+5));
    cout << ret.first << " " << ret.second << endl;
    // expected: 5 -15
    linear_diophantine(7, 2, 5, x, y);
    cout << x << " " << y << endl;
}
```

6.2 FFT

```
namespace fft {
    struct cnum {
        double a, b;
        cnum operator+(const cnum &c) { return { a + c.a, b + c.b }; }
        cnum operator-(const cnum &c) { return { a - c.a, b - c.b }; }
        cnum operator*(const cnum &c) { return { a*c.a - b*c.b, a*c.b +
            b*c.a }; }
        cnum operator/(double d) { return { a / d, b / d }; }
    };

    const double PI = 2 * atan2(1, 0);
    int deg;
    vector<int> rev;

    void set_degree(int _deg) {
        assert(__builtin_popcount(_deg) == 1);
        deg = _deg;
        rev.resize(deg);
        for (int i = 1, j = 0; i < deg; i++) {
            int bit = deg / 2;
            for (; j >= bit; bit /= 2)
                j -= bit;
            j += bit;
            rev[i] = j;
        }
    }
    void transform(vector<cnum> &poly, bool invert) {
        if(deg != poly.size()) set_degree(poly.size());
        for (int i = 1; i < deg; i++)
            if(rev[i] > i)
                swap(poly[i], poly[rev[i]]);
        for (int len = 2; len <= deg; len *= 2) {
            double ang = 2 * PI / len * (invert ? -1 : 1);
            cnum base = { cos(ang), sin(ang) };
            for (int i = 0; i < deg; i += len) {
                cnum w = {1, 0};
                for (int j = 0; j < len / 2; j++) {
                    cnum u = poly[i+j];
                    cnum v = w * poly[i+j+len/2];
                    poly[i+j] = u + v;
                    poly[i+j+len/2] = u - v;
                    w = w * base;
                }
            }
        }
        if(invert) {
            for (int i = 0; i < deg; i++)
                poly[i] = poly[i] / double(deg);
        }
    }
};
```

6.3 Gauss-Jordan

```
// Gauss-Jordan elimination with full pivoting.
//
// Uses:
//   (1) solving systems of linear equations (AX=B)
//   (2) inverting matrices (AX=I)
//   (3) computing determinants of square matrices
//
// Running time: O(n^3)
//
// INPUT: a[][] = an nxn matrix
//        b[][] = an nxm matrix
//
// OUTPUT:  X    = an nxm matrix (stored in b[][])
//        A^{-1} = an nxn matrix (stored in a[][])
//        returns determinant of a[][]
typedef vector<int> VI;
typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;
const double EPS = 1e-10;

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

    for (int i = 0; i < n; i++) {
        int pj = -1, pk = -1;
        for (int j = 0; j < n; j++) if (!ipiv[j])
            for (int k = 0; k < n; k++) if (!ipiv[k])
                if (pj == -1 || fabs(a[j][k]) > fabs(a[pj][pk])) { pj = j;
                    pk = k; }
        if (fabs(a[pj][pk]) < EPS) { cerr << "Matrix is singular." <<
            endl; exit(0); }
        ipiv[pk]++;
        swap(a[pj], a[pk]);
        swap(b[pj], b[pk]);
        if (pj != pk) det *= -1;
        irow[i] = pj;
        icol[i] = pk;
        T c = 1.0 / a[pk][pk];
        det *= a[pk][pk];
        a[pk][pk] = 1.0;
        for (int p = 0; p < n; p++) a[pk][p] *= c;
        for (int p = 0; p < m; p++) b[pk][p] *= c;
```

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

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

    double det = GaussJordan(a, b);
    // expected: 60
    cout << "Determinant: " << det << endl;
    // expected: -0.233333 0.166667 0.133333 0.0666667
    //      0.166667 0.166667 0.333333 -0.333333
    //      0.233333 0.833333 -0.133333 -0.0666667
    //      0.05 -0.75 -0.1 0.2
    cout << "Inverse: " << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            cout << a[i][j] << ' ';
        cout << endl;
    }
    // expected: 1.63333 1.3
    //      -0.166667 0.5
    //      2.36667 1.7
    //      -1.85 -1.35
    cout << "Solution: " << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++)
            cout << b[i][j] << ' ';
        cout << endl;
    }
}
```

6.4 Matrix

```
template<typename T> struct matrix {
    int N;
    vector<T> dat;

    matrix<T> (int _N, T fill = T(0), T diag = T(0)) {
        N = _N;
        dat.resize(N * N, fill);
        for (int i = 0; i < N; i++)
            (*this)(i, i) = diag;
    }
    T& operator()(int i, int j) {
        return dat[N * i + j];
    }
    matrix<T> operator *(matrix<T> &b){
        matrix<T> r(N);
        for(int i=0; i<N; i++)
            for(int j=0; j<N; j++)
                for(int k=0; k<N; k++)
                    r(i, j) = r(i, j) + (*this)(i, k) * b(k, j);
        return r;
    }
    matrix<T> pow(ll expo){
        if(!expo) return matrix<T>(N, T(0), T(1));
        matrix<T> r = (*this * *this).pow(expo/2);
        return expo&1 ? r * *this : r;
    }
    friend ostream& operator<<(ostream &os, matrix<T> &m){
        os << "{";
        for(int i=0; i<m.N; i++){
            if(i) os << "},\n ";
            os << "{";
            for(int j=0; j<m.N; j++){
                if(j) os << ", ";
                os << setw(10) << m(i, j) << setw(0);
            }
        }
        return os << "}}";
    }
};

struct mll {
    const int MOD;
    ll val;
    mll(ll _val = 0) {
        val = _val % MOD;
        if (val < 0) val += MOD;
    }
    mll operator+(const mll &o) {
        return mll((val + o.val) % MOD);
    }
    mll operator*(const mll &o) {
        return mll((val * o.val) % MOD);
    }
    friend ostream& operator<<(ostream &os, mll &m) {
        return os << m.val;
    }
};
```

6.5 Primes

```
// O(sqrt(x)) Exhaustive Primality Test
#define EPS 1e-7
typedef long long LL;
bool IsPrimeSlow (LL x) {
    if(x<=1) return false;
```

```
    if(x<=3) return true;
    if (!(x%2) || !(x%3)) return false;
    LL s=(LL)(sqrt((double)(x))+EPS);
    for(LL i=5;i<=s;i+=6)
        if (!(x%i) || !(x%(i+2))) return false;
    return true;
}
// Primes less than 1000:
//    2    3    5    7   11   13   17   19   23   29   31   37
//   41   43   47   53   59   61   67   71   73   79   83   89
//   97  101  103  107  109  113  127  131  137  139  149  151
//  157  163  167  173  179  181  191  193  197  199  211  223
//  227  229  233  239  241  251  257  263  269  271  277  281
//  283  293  307  311  313  317  331  337  347  349  353  359
//  367  373  379  383  389  397  401  409  419  421  431  433
//  439  443  449  457  461  463  467  479  487  491  499  503
//  509  521  523  541  547  557  563  569  571  577  587  593
//  599  601  607  613  617  619  631  641  643  647  653  659
//  661  673  677  683  691  701  709  719  727  733  739  743
//  751  757  761  769  773  787  797  809  811  821  823  827
//  829  839  853  857  859  863  877  881  883  887  907  911
//  919  929  937  941  947  953  967  971  977  983  991  997
// Other primes:
//  The largest prime smaller than 10^x:
//  7 97 997 9973 99991 999983 9999991 99999989 999999937 9999999967
//  99999999977 999999999989 9999999999971 99999999999973 999999999999989
//  9999999999999937 99999999999999997 999999999999999989
```

6.6 Reduced Row Echelon Form

```
// Reduced row echelon form via Gauss-Jordan elimination
// with partial pivoting. This can be used for computing
// the rank of a matrix.
//
// Running time: O(n^3)
//
// INPUT: a[][] = an nxn matrix
//
// OUTPUT:  rref[][] = an nxm matrix (stored in a[][])
//          returns rank of a[][]
typedef vector<double> VD;
typedef vector<VD> VVD;
const double EPSILON = 1e-7;

// returns rank
int rref (VVD &a){
    int i,j,r,c;
    int n = a.size();
    int m = a[0].size();
    for (r=c=0;c<m;c++){
        j=r;
        for (i=r+1;i<n;i++) if (fabs(a[i][c])>fabs(a[j][c])) j = i;
        if (fabs(a[j][c])<EPSILON) continue;
        for (i=0;i<m;i++) swap(a[j][i],a[r][i]);
        double s = a[r][c];
        for (j=0;j<m;j++) a[r][j] /= s;
        for (i=0;i<n;i++) if (i != r){
            double t = a[i][c];
            for (j=0;j<m;j++) a[i][j] -= t*a[r][j];
        }
        r++;
    }
    return r;
}
```

6.7 Simplex

```
// Two-phase simplex algorithm for solving linear programs of the form
//
//     maximize     c^T x
//     subject to   Ax <= b
//                  x >= 0
//
// INPUT: A -- an m x n matrix
//        b -- an m-dimensional vector
//        c -- an n-dimensional vector
//        x -- a vector where the optimal solution will be stored
//
// OUTPUT: value of the optimal solution (infinity if unbounded
//         above, nan if infeasible)
//
// To use this code, create an LPSolver object with A, b, and c as
// arguments. Then, call Solve(x).
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) {
        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] / D[r][s];
        for (int j = 0; j < n + 2; j++) if (j != s) D[r][j] /= D[r][s];
        for (int i = 0; i < m + 2; i++) if (i != r) D[i][s] /= -D[r][s];
        D[r][s] = 1.0 / D[r][s];
        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;
}
```

7 7 String

7.1 Aho-Corasick

```
namespace aho_corasick {
    const int SIGMA = 2;
    const int TOTL = 1e7 + 100;

    struct node {
        int link[SIGMA];
        int suff, dict, patt;
        node() {
            suff = 0, dict = 0, patt = -1;
            memset(link, 0, sizeof(link));
        }
        // link[]: contains trie links + failure links
        // suff: link to longest proper suffix that exists in the trie
        // dict: link to longest suffix that exists in the dictionary
        // patt: index of this node's word in the dictionary
    };

    int tail = 1;
    vector<node> trie(TOTL);
    vector<string> patterns;

    void add_pattern(string &s) {
        int loc = 0;
        for (char c : s) {
            int &nloc = trie[loc].link[c-'a'];
            if (!nloc) nloc = tail++;
            loc = nloc;
        }
        trie[loc].dict = loc;
        trie[loc].patt = patterns.size();
        patterns.push_back(s);
    }
    void calc_links() {
        queue<int> bfs({0});
        while (!bfs.empty()) {
            int loc = bfs.front(); bfs.pop();
            int fail = trie[loc].suff;
            if (!trie[loc].dict)
                trie[loc].dict = trie[fail].dict;
            for (int c = 0; c < SIGMA; c++) {
                int &succ = trie[loc].link[c];
                if (succ) {
                    trie[succ].suff = loc ? trie[fail].link[c] : 0;
                    bfs.push(succ);
                } else succ = trie[fail].link[c];
            }
```

```
        }
    }
    void match(string &s, vector<bool> &matches) {
        int loc = 0;
        for (char c : s) {
            loc = trie[loc].link[c-'a'];
            for (int dm = trie[loc].dict; dm; dm =
                 trie[trie[dm].suff].dict) {
                if (matches[trie[dm].patt]) break;
                matches[trie[dm].patt] = true;
            }
        }
    }
}
```

7.2 KMP

```
template<typename T> struct kmp {
    int M;
    vector<T> needle;
    vector<int> succ;

    kmp(vector<T> _needle) {
        needle = _needle;
        M = needle.size();
        succ.resize(M + 1);
        succ[0] = -1, succ[1] = 0;
        int cur = 0;
        for (int i = 2; i <= M; ) {
            if (needle[i-1] == needle[cur]) succ[i++] = ++cur;
            else if (cur) cur = succ[cur];
            else succ[i++] = 0;
        }
    }
    vector<bool> find(vector<T> &haystack) {
        int N = haystack.size(), i = 0;
        vector<bool> res(N);
        for (int m = 0; m + i < N; ) {
            if (i < M && needle[i] == haystack[m + i]) {
                if (i == M - 1) res[m] = true;
                i++;
            } else if (succ[i] != -1) {
                m = m + i - succ[i];
                i = succ[i];
            } else {
                i = 0;
                m++;
            }
        }
        return res;
    }
};
```

7.3 Suffix Arrays

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

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

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

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

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

```
  for (int i = 0; i < v.size(); i++) cout << v[i] << " ";
  cout << endl;
  cout << suffix.LongestCommonPrefix(0, 2) << endl;
}
```

8 8 Misc

8.1 IO

```
int main() {
  // Ouput a specific number of digits past the decimal point,
  // in this case 5
  cout.setf(ios::fixed); cout << setprecision(5);
  cout << 100.0/7.0 << endl;
  cout.unsetf(ios::fixed);
  // Output the decimal point and trailing zeros
  cout.setf(ios::showpoint);
  cout << 100.0 << endl;
  cout.unsetf(ios::showpoint);
  // Output a '+' before positive values
  cout.setf(ios::showpos);
  cout << 100 << " " << -100 << endl;
  cout.unsetf(ios::showpos);
  // Output numerical values in hexadecimal
  cout << hex << 100 << " " << 1000 << " " << 10000 << dec << endl;
}
```

8.2 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
typedef vector<int> VI;
typedef pair<int,int> PII;
typedef vector<PII> VPII;

#define STRICTLY_INCREASNG
VI LongestIncreasingSubsequence(VI v) {
  VPII best;
  VI dad(v.size(), -1);
  for (int i = 0; i < v.size(); i++) {
#ifdef STRICTLY_INCREASNG
    PII item = make_pair(v[i], 0);
    VPII::iterator it = lower_bound(best.begin(), best.end(), item);
    item.second = i;
#else
    PII item = make_pair(v[i], i);
    VPII::iterator it = upper_bound(best.begin(), best.end(), item);
#endif
    if (it == best.end()) {
      dad[i] = (best.size() == 0 ? -1 : best.back().second);
      best.push_back(item);
    } else {
      dad[i] = dad[it->second];
      *it = item;
    }
  }
  VI ret;
  for (int i = best.back().second; i >= 0; i = dad[i])
    ret.push_back(v[i]);
  reverse(ret.begin(), ret.end());
  return ret;
}
```

8.3 Regular Expressions - Java

```
// Code which demonstrates the use of Java's regular expression libraries.
// This is a solution for
//
//   Loglan: a logical language
//   http://acm.uva.es/p/v1/134.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 = "([aeiou])";
    String C = "([a-z&&[^aeiou]])";
    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 + A +
        ")";

    String predstring = "(" + PREDA + "(" + space + PREDA + ")*)";
    String predname = "(" + LA + space + predstring + "|" + NAM + ")";
    String preds = "(" + predstring + "(" + space + A + space +
        predstring + ")*)";
    String predclaim = "(" + 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 + "|" + predclaim + ")";
```

```
    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!");
      }
    }
  }
}
```

8.4 Tokenizer

```
// char tokens
vector<string> &split(const string &s, char delim, vector<string> &elems){
    stringstream ss(s);
    string item;
    while (getline(ss, item, delim)) {
        elems.push_back(item);
    }
    return elems;
}
vector<string> split(const string &s, char delim) {
    vector<string> elems;
    split(s, delim, elems);
    return elems;
}
// string tokens
vector<string> split(string str, string token) {
    vector<string> result;
    int next;
    while((next = str.find(token)) != string::npos) {
        result.push_back(str.substr(0, next));
        str = str.substr(next + token.size());
    }
    return result;
}

int main() {
    string test = "Hello, this is a string that we might, like, like to
        parse! ";
    auto v_space = split(test, ' ');
    auto v_comma = split(test, ',');
    cout << "|"; for(auto s : v_space) cout << s << "|"; cout << endl;
    cout << "|"; for(auto s : v_comma) cout << s << "|"; cout << endl;
    // |Hello,|this|is|a|string|that|we|might,|like,|like|to|parse!|
    // |Hello| this is a string that we might| like| like to parse! |
    return 0;
}
```