

California Institute of Technology

May 7, 2016

Contents

1	1 Templates	2
1.1	Start	2
1.2	Template - C++	2
1.3	Template - Java	2
2	2 Data Structures	2
2.1	2
2.2	BIT	3
2.3	KD Tree	4
2.4	LCA	6
2.5	Lazy Segment Tree	7
2.6	Segment Tree	8
2.7	Splay Tree	8
2.8	Union Find	10
3	3 Graph	10
3.1	2-SAT	10
3.2	Dense Dijkstra	11
3.3	Dijkstra	11
3.4	Eulerian Path	12
3.5	Heavy Light	13
3.6	Poset Width	13
3.7	SCC	14
3.8	Topological Sort	14
4	4 Combinatorial Optimization	15
4.1	Bipartite Graph	15
4.2	Max Flow - Dinic	16
4.3	Max Flow - Push-Relabel	17
4.4	Min Cost Matching	18

4.5	Min Cost Max Flow	20
4.6	Min Cut	22
5	5 Geometry	22
5.1	Convex Hull	22
5.2	Delaunay	23
5.3	Geometry - Java	24
5.4	Geometry 3D - Java	26
5.5	Geometry	26
6	6 Numerics	30
6.1	Euclid	30
6.2	FFT	31
6.3	Gauss-Jordan	32
6.4	Matrix	33
6.5	Reduced Row Echelon Form	34
6.6	Simplex	35
7	7 String	36
7.1	Aho-Corasick	36
7.2	KMP	37
7.3	Rabin-Karp	37
7.4	Suffix Array	38
7.5	Suffix Arrays	39
8	8 Misc	40
8.1	Dates	40
8.2	IO	41
8.3	Longest Increasing Subsequence	41
8.4	Primes	42
8.5	Regular Expressions - Java	42

1 1 Templates

1.1 Start

```
// .vimrc
syn on
set mouse=a sw=4 ts=4 ai si nu wrap
nnoremap ; :
map H : '<,'>!. /hash.sh<CR>

// Hash Utility
#!/bin/bash
input=$(cat /dev/stdin)
hash=$(cpp -fpreprocessed -P <(echo "$input")|sed ':a;N;$!ba;s/\t\n//g'|md5sum)
echo "// start: ${hash}"
echo "$input"
echo "// end"

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

1.2 Template - C++

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

typedef long long ll;
typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
typedef vector<vi> vvi;
typedef vector<ll> vll;
typedef vector<vll> vvll;

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

```
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

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

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

```

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

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

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

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

    if(p == q)
        return p;

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

```

```

    return A[p][0];
}

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

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

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

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

    // precompute L
    DFS(root, 0);

    return 0;
}

```

2.2 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);
    }
};

```

2.3 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 kdnnode
{
    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

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

    kdnnode() : leaf(false), first(0), second(0) {}
    ~kdnnode() { 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 kdnnode(); first->construct(vl);
            second = new kdnnode(); second->construct(vr);
        }
    }
};

// simple kd-tree class to hold the tree and handle queries
struct kdtree
{
    kdnnode *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 kdnnode();
        root->construct(v);
    }
    ~kdtree() { delete root; }

    // recursive search method returns squared distance to nearest point
    ntype search(kdnnode *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;
    }

    return 0;
}

//
// -----

```

2.4 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];
}
};

```

2.5 Lazy Segment Tree

```

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

    T zero;
    vector<T> value;

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

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

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

```

```

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

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

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

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

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

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

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

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

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

```

```

        rebuild(i), rebuild(j);
    }

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

        T res_left = zero, res_right = zero;
        for(; i <= j; i /= 2, j /= 2){
            if((i&1) == 1) res_left = res_left + value[i++];
            if((j&1) == 0) res_right = value[j--] + res_right;
        }
        return res_left + res_right;
    }
};

```

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 Splay Tree

```

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

```

```

        {
            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);
}
}

```

2.8 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:  $O(|E| \log |V|)$ 

#include <queue>

```

```

#include <stdio.h>

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

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

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

```

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

```

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

```

```

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

```

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).

#include <iostream>
#include <vector>

using namespace std;
typedef long long LL;

struct Edge {
    int from, to, cap, flow, index;
    Edge(int from, int to, int cap, int flow, int index) :
        from(from), to(to), cap(cap), flow(flow), index(index) {}
    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 Max Flow - Push-Relabel

```

// Adjacency list implementation of FIFO push relabel maximum flow
// with the gap relabeling heuristic. This implementation is
// significantly faster than straight Ford-Fulkerson. It solves
// random problems with 10000 vertices and 1000000 edges in a few
// seconds, though it is possible to construct test cases that
// achieve the worst-case.
//
// Running time:
//  $O(|V|^3)$ 

```

```

//
// INPUT:
//   - graph, constructed using AddEdge()
//   - source
//   - sink
//
// OUTPUT:
//   - maximum flow value
//   - To obtain the actual flow values, look at all edges with
//     capacity > 0 (zero capacity edges are residual edges).

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

using namespace std;

typedef long long LL;

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

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

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

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

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

```

```

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

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

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

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

LL GetMaxFlow(int s, int t) {
    count[0] = N-1;
    count[N] = 1;
    dist[s] = N;

```

```

    active[s] = active[t] = true;
    for (int i = 0; i < G[s].size(); i++) {
        excess[s] += G[s][i].cap;
        Push(G[s][i]);
    }

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

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

```

4.4 Min Cost Matching

```

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

#include <algorithm>
#include <cstdio>
#include <cmath>
#include <vector>

using namespace std;

```

```

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

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

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

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

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

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

        // find an unmatched left node
        int s = 0;

```

```

        while (Lmate[s] != -1) s++;

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

        int j = 0;
        while (true) {

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

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

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

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

        // augment along path
        while (dad[j] >= 0) {

```

```

    const int d = dad[j];
    Rmate[j] = Rmate[d];
    Lmate[Rmate[j]] = j;
    j = d;
}
Rmate[j] = s;
Lmate[s] = j;

mated++;
}

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

return value;
}

```

4.5 Min Cost Max Flow

```

// Generic flow 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:  $O(\min(|V|^2 * \text{totflow}, |V|^3 * \text{totcost}))$ 
//
// 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
//
//          excess -- a vector such that the total flow from i == excess[i]
//
//
// OUTPUT: cost of the resulting flow; the matrix flow will contain
//          the actual flow values (all nonnegative).
//          The vector excess will contain node excesses that could not be
//          eliminated. Remember to check it.
//
// To use this, create a MinCostCirc object, and call it like this:
//

```

```

// MinCostCirc circ(N);
// circ.cap = <whatever>; circ.cost = <whatever>;
// circ.excess[foo] = bar;
// circ.flow[i][j] = something; (if you want)
// int finalcost = circ.solve();
//
// If you want min-cost max-flow, leave excess blank and call
// min_cost_max_flow.
// Andy says to use caution in min-cost max-flow mode if you have negative
// costs.

```

```

typedef vector<int64_t> VI64;
typedef vector<int> VI;
typedef vector<VI64> VVI64;

```

```
const int64_t INF = 1LL<<60;
```

```

struct MinCostCirc {
    int N;
    VVI64 cap, flow, cost;
    VI dad, found, src, add;
    VI64 pi, dist, excess;

```

```

    MinCostCirc(int N) : N(N), cap(N, VI64(N)), flow(cap), cost(cap),
                        dad(N), found(N), src(N), add(N),
                        pi(N), dist(N+1), excess(N) {}

```

```

void search() {
    fill(found.begin(), found.end(), false);
    fill(dist.begin(), dist.end(), INF);

```

```

    int here = N;
    for (int i = 0; i < N; i++)
        if (excess[i] > 0) {
            src[i] = i;
            dist[i] = 0;
            here = i;
        }

```

```

while (here != N) {
    int best = N;
    found[here] = 1;
    for (int k = 0; k < N; k++) {
        if (found[k]) continue;
        int64_t x = dist[here] + pi[here] - pi[k];

```

```

    if (flow[k][here]) {
        int64_t val = x - cost[k][here];
        assert(val >= dist[here]);
        if (dist[k] > val) {
            dist[k] = val;
            dad[k] = here;
            add[k] = 0;
            src[k] = src[here];
        }
    }
    if (flow[here][k] < cap[here][k]) {
        int64_t val = x + cost[here][k];
        assert(val >= dist[here]);
        if (dist[k] > val) {
            dist[k] = val;
            dad[k] = here;
            add[k] = 1;
            src[k] = src[here];
        }
    }

    if (dist[k] < dist[best]) best = k;
}
here = best;

for (int k = 0; k < N; k++)
    if (found[k])
        pi[k] = min(pi[k] + dist[k], INF);
}

int64_t solve() {
    int64_t totcost = 0;
    int source, sink;
    for(int i = 0; i < N; i++)
        for(int j = 0; j < N; j++)
            if (cost[i][j] < 0)
            {
                flow[i][j] += cap[i][j];
                totcost += cost[i][j] * cap[i][j];
                excess[i] -= cap[i][j];
                excess[j] += cap[i][j];
            }

    bool again = true;

```

```

while (again) {
    search();
    int64_t amt = INF;
    fill(found.begin(), found.end(), false);
    again = false;
    for(int sink = 0; sink < N; sink++)
    {
        if (excess[sink] >= 0 || dist[sink] == INF || found[src[sink]]++)
            continue;
        again = true;
        int source = src[sink];

        for (int x = sink; x != source; x = dad[x])
            amt = min(amt, flow[x][dad[x]] ? flow[x][dad[x]] :
                cap[dad[x]][x] - flow[dad[x]][x]);
        amt = min(amt, min(excess[source], -excess[sink]));
        for (int x = sink; x != source; x = dad[x]) {
            if (add[x]) {
                flow[dad[x]][x] += amt;
                totcost += amt * cost[dad[x]][x];
            } else {
                flow[x][dad[x]] -= amt;
                totcost -= amt * cost[x][dad[x]];
            }
            excess[x] += amt;
            excess[dad[x]] -= amt;
        }
        assert(amt != 0);
        break; // Comment out at your peril if you need speed.
    }
}

return totcost;
}

// returns (flow, cost)
pair<int,int> min_cost_max_flow(int source, int sink) {
    excess[source] = INF;
    excess[sink] = -INF;
    pair<int, int> ret;
    ret.second = solve();
    ret.first = INF - excess[source];
    return ret;
}
};

```

4.6 Min Cut

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

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

using namespace std;

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

const int INF = 1000000000;

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

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

#include <cstdio>
#include <cassert>
#include <vector>
#include <algorithm>
#include <cmath>

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
}

```

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

```

```

#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, 0.9};
    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 - Java

```

// 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.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,
// double 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)).
}
}

```

5.4 Geometry 3D - Java

```

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));
}
}

```

5.5 Geometry

// C++ routines for computational geometry.

```

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

using namespace std;

```

```

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

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

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

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

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

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

// compute distance from c to segment between a and b

```

```

double DistancePointSegment(PT a, PT b, PT c) {
    return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}

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

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

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

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

// compute intersection of line passing through a and b
// with line passing through c and d, assuming that unique
// intersection exists; for segment intersection, check if
// segments intersect first
PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {
    b=b-a; d=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(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].x*p[j].y - p[j].x*p[i].y);
}
return c / scale;
}

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

int main() {

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

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

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

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

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

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

    // expected: 1 0 1
    cerr << LinesParallel(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "

```

```

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

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

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

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

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

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

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

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

    // expected: (1,6)
    //      (5,4) (4,5)
    //      blank line
    //      (4,5) (5,4)

```

```
//          blank line
//          (4,5) (5,4)
vector<PT> u = CircleLineIntersection(P(0,6), P(2,6), P(1,1), 5);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
u = CircleLineIntersection(P(0,9), P(9,0), P(1,1), 5);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
u = CircleCircleIntersection(P(1,1), P(10,10), 5, 5);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
u = CircleCircleIntersection(P(1,1), P(8,8), 5, 5);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
u = CircleCircleIntersection(P(1,1), P(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(P(1,1), P(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[] = { P(0,0), P(5,0), P(1,1), P(0,5) };
vector<PT> p(pa, pa+4);
PT c = ComputeCentroid(p);
cerr << "Area: " << ComputeArea(p) << endl;
cerr << "Centroid: " << c << endl;

return 0;
}
```

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.
```

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

```
using namespace std;
```

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

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

// computes gcd(a,b)
int gcd(int a, int b) {
    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[] []

```

```

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

```

```
using namespace std;
```

```
const double EPS = 1e-10;
```

```

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

```

```

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

```



```

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

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

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

return det;
}

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

```

```

}

double det = GaussJordan(a, b);

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

// expected: -0.233333 0.166667 0.133333 0.066667
//           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;
}
}

```

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 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[] []

const double EPSILON = 1e-7;

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

// 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;
}

```

6.6 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).

```

```

#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) {
    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 Rabin-Karp

```

namespace rabin_karp {
    const int NPRIME = 3;
    const int P1 = 1e9+7, P2 = 1e9+9, P3 = 1e9+21;
    const int primes[3] = { P1, P2, P3 };

    struct strong_mod {
        int vals[NPRIME];

        strong_mod(ll value = 0) {
            for (int i = 0; i < NPRIME; i++) {
                vals[i] = value % primes[i];
                if (vals[i] < 0) vals[i] += primes[i];
            }
        }

        #define combine(h, op) \
            strong_mod res; \
            for (int i = 0; i < NPRIME; i++) \
                res.vals[i] = (vals[i] op h.vals[i]) % primes[i]; \
            return res;

        bool operator==(const strong_mod &h) { return !memcmp(this, &h,
            sizeof(strong_mod)); }
        strong_mod operator*(const strong_mod &h) { combine(h, * 111 *); }
        strong_mod operator+(const strong_mod &h) { combine(h, +); }

        strong_mod operator*(ll m) { return (*this) * strong_mod(m); }
        strong_mod operator+(ll a) { return (*this) + strong_mod(a); }
    };

    const int MAXL = 2e5 + 200;
    const int BASE = 3001;

    vector<strong_mod> powers;

    void init() {
        powers.resize(MAXL);
        powers[0] = strong_mod(1);
        for (int i = 1; i < MAXL; i++)
            powers[i] = powers[i-1] * BASE;
    }

    void compute(vector<strong_mod> &hashes, string str) {
        hashes.resize(str.size() + 1);
        for (int i = 1; i <= str.size(); i++)

```

```

            hashes[i] = hashes[i-1] * BASE + int(str[i-1]);
        }

        strong_mod read(vector<strong_mod> &hashes, int i, int l) {
            strong_mod sub = hashes[i] * powers[l];
            return hashes[i+1] + (sub * -1);
        }
    };

```

7.4 Suffix Array

```

struct suffix_array{
    const int INF = 0x7fffffff;

    int L, D;
    string str;
    vector<vector<int>> > suff;
    vector<int> rank_of, at_rank;

    pair<pair<int, int>, int> __make_rep(int l, int i, int p){
        if (!l) return {{str[i], INF}, i};
        return {{suff[l-1][i], (i+p<L) ? suff[l-1][i+p] : INF}, i};
    }

    suffix_array(string _str){
        str = _str;
        L = str.size();
        D = 33 - __builtin_clz(L-1);
        suff.resize(D);

        vector<pair<pair<int, int>, int> > keys(L);
        for (int l=0; l<D; l++){
            for (int i=0; i<L; i++)
                keys[i] = __make_rep(l, i, 1<<(l-1));
            sort(keys.begin(), keys.end());

            suff[l].resize(L);
            for (int i=0, r=0; i<L; i++){
                if (i>0 && keys[i].first != keys[i-1].first) r++;
                suff[l][keys[i].second] = r;
            }
        }
    }

```

```

    rank_of.resize(L);
    at_rank.resize(L);
    for(int i=0; i<L; i++){
        rank_of[i] = suff.back()[i];
        at_rank[rank_of[i]] = i;
    }
}

// compare the string at [i, i+l1) to the string at [j, j+l2)
int comp(int i, int l1, int j, int l2){
    int c1 = min(l1, l2);
    for(int l=0; l<D; l++){
        if((c1>>l)&1){
            if(suff[l][i] != suff[l][j])
                return suff[l][i] < suff[l][j] ? -1 : 1;
            i += 1<<l, j += 1<<l;
        }
    }

    return (l1!=l2) ? (l1<l2) ? -1 : 1 : 0;
}

// find the interval of suffix ranks corresponding to instances of
// the substring at [i, i+len)
pair<int, int> find_range(int i, int len){
    int left = rank_of[i];
    for(int lo=0, hi=left; lo<=hi; ){
        int mi = (lo + hi)/2;
        if(comp(i, len, at_rank[mi], len) == 0){
            left = mi;
            hi = mi-1;
        }
        else lo = mi+1;
    }

    int right = rank_of[i];
    for(int lo=right, hi=L-1; lo<=hi; ){
        int mi = (lo + hi)/2;
        if(comp(i, len, at_rank[mi], len) == 0){
            right = mi;
            lo = mi+1;
        }
        else hi = mi-1;
    }

    return make_pair(left, right);
}

```

```

    }
};

```

7.5 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.

```

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

```

```
using namespace std;
```

```

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

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

```

```

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

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

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 Dates

// Routines for performing computations on dates. In these routines,

```

// months are expressed as integers from 1 to 12, days are expressed
// as integers from 1 to 31, and years are expressed as 4-digit
// integers.

#include <iostream>
#include <string>

using namespace std;

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

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

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

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

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

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

```



```

string day = intToDay (jd);

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

```

8.2 IO

```

#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;
}

```

8.3 Longest Increasing Subsequence

```

// Given a list of numbers of length n, this routine extracts a
// longest increasing subsequence.
//
// Running time: O(n log n)
//
// INPUT: a vector of integers
// OUTPUT: a vector containing the longest increasing subsequence

```

```

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

```

```
using namespace std;
```

```

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

```

```
#define STRICTLY_INCREASNG
```

```

VI LongestIncreasingSubsequence(VI v) {
    VPII best;
    VI dad(v.size(), -1);

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

```

```

VI ret;
for (int i = best.back().second; i >= 0; i = dad[i])

```

```

    ret.push_back(v[i]);
    reverse(ret.begin(), ret.end());
    return ret;
}

```

8.4 Primes

```

// O(sqrt(x)) Exhaustive Primality Test
#include <cmath>
#define EPS 1e-7
typedef long long LL;
bool IsPrimeSlow (LL x)
{
    if(x<=1) return false;
    if(x<=3) return true;
    if (!(x%2) || !(x%3)) return false;
    LL s=(LL)(sqrt((double)(x))+EPS);
    for(LL i=5;i<=s;i+=6)
    {
        if (!(x%i) || !(x%(i+2))) return false;
    }
    return true;
}

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

// Other primes:
//      The largest prime smaller than 10 is 7.
//      The largest prime smaller than 100 is 97.
//      The largest prime smaller than 1000 is 997.

```

```

//      The largest prime smaller than 10000 is 9973.
//      The largest prime smaller than 100000 is 99991.
//      The largest prime smaller than 1000000 is 999983.
//      The largest prime smaller than 10000000 is 9999991.
//      The largest prime smaller than 100000000 is 99999989.
//      The largest prime smaller than 1000000000 is 999999937.
//      The largest prime smaller than 10000000000 is 9999999967.
//      The largest prime smaller than 100000000000 is 99999999977.
//      The largest prime smaller than 1000000000000 is 99999999989.
//      The largest prime smaller than 10000000000000 is 999999999971.
//      The largest prime smaller than 100000000000000 is 9999999999973.
//      The largest prime smaller than 1000000000000000 is 9999999999989.
//      The largest prime smaller than 10000000000000000 is
999999999999937.
//      The largest prime smaller than 100000000000000000 is
999999999999997.
//      The largest prime smaller than 1000000000000000000 is
999999999999989.

```

8.5 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!");
}
}
}
}

```