```
/* vimrc
set number
set sw=4
set ts=4
set softtabstop=4
set background=dark
map <C-n> :tabnext<Return>
map <C-p> :tabprev<Return>
map <Space> :nohl<Return>
set expandtab
set noincsearch
filetype indent on
set autowrite
au FileType cpp set makeprg=g++\ -O2\ -g\ -o\ %<\ %\ -Wall\ -Wextra */

#include <algorithm>
#include <cassert>
#include <cmath>
#include <complex>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <deque>
#include <iostream>
#include <list>
#include <map>
#include <queue>
#include <set>
#include <sstream>
#include <stack>
#include <string>
#include <vector>
using namespace std;

typedef long long LL;
typedef long double LD;
typedef vector<int> VI;
typedef pair<int,int> PII;

#define REP(i,n) for(int i=0;i<(n);++i)
#define SIZE(c) ((int)((c).size()))
#define FOR(i,a,b) for (int i=(a); i<(b); ++i)
#define FOREACH(i,x) for (__typeof((x).begin()) i=(x).begin(); i!=(x).end(); ++i)
#define FORD(i,a,b) for (int i=((int)(a))-1; i>=(b); --i)
#define ALL(u) (u).begin(),(u).end()

#define pb push_back
#define mp make_pair
#define st first
#define nd second

/*
Days of week
January 1, 1600: Saturday   January 1, 1900: Monday     June 13, 2042: Friday
January 1, 2008: Tuesday    April 1, 2008: Tuesday      April 9, 2008: Wednesday
December 31, 1999: Friday   January 1, 3000: Wednesday
*/
/* Warnsdorff's heuristic for knight's tour. At each step choose a square which
   has the least number of valid moves that the knight can make from there. */

// Fenwick Tree
int a[MAXN];
// value[n] += x
void add(int n, int x) { for (; n < MAXN; n |= n + 1) a[n] += x; }
// Returns value[0] + value[1] + ... + value[n]
int sum(int n) { int s=0; while (n>=0) { s+=a[n]; n=(n&(n+1))-1; } return s; }

/*
Clearing the lowest 1 bit: x & (x - 1), all trailing 1's: x & (x + 1)
Setting the lowest 0 bit: x | (x + 1)
Enumerating subsets of a bitmask m: x=0; do { ...; x=(x+1+~m)&m; } while (x!=0);
__builtin_ctz/__builtin_clz returns the number of trailing/leading zero bits.
__builtin_popcount(unsigned x) counts 1-bits (slower than table lookups).
For 64-bit unsigned integer type, use the sumx 'll', i.e. __builtin_popcountll.
*/
/*
Flow-shop scheduling (Johnson's problem). Schedule N jobs on 2 machines to
minimize completition time. i-th job takes a_i and b_i time to execute on 1st
and 2nd machine, respectively. Each job must be first executed on the fitrst
machine, then on second. Both machines execute all jobs in the same order.
Solution: sort jobs by key a_i < b_i ? a_i : (INF-b_i)
*/
/*
Euler's theorem. For any planar graph, V - E + F = 1 + C, where V is the number
of graph's vertices, E is the number of edges, F is the number of faces in
graph's planar drawing, and C is the number of connected components.
Corollary: V - E + F = 2 for a 3D polyhedron.
*/
/*
Matrix-tree theorem. Let matrix T = [t_ij], where t_ij is the number of
multiedges between i and j, for i != j, and t_ii = -deg_i.
Number of spanning trees of a graph is equal to the determinant of a matrix
obtained by deleting any k-th row and k-th column from T.
*/
/*
Vertex covers and independent sets.
Let M, C, I be a max matching, a min vertex cover, and a max independent set.
Then |M| <= |C| = N - |I|, with equality for bipartite graphs. Complement
of an MVC is always a MIS, and vice versa. Given a bipartite graph with
partitions (A;B), build a network: connect source to A, and B to sink with
edges of capacities, equal to the corresponding nodes' weights, or 1 in the
unweighted case. Set capacities of the original graph's edges to the
infinity. Let (S; T) be a minimum s-t cut. Then a maximum(-weighted) independent
set is I = (A * S) + (B * T), and a minimum(-weighted) vertex cover is
C = (A * T) + (B * S).
*/

// Jesli G nie ma w. izolowanych: max skojarzenie + min pokrycie krawedziowe = |V|

// W dwudzielnym |min pokrycie wierzcholkowe| = |max skojarzenie|

/*
Pick's theorem. I = A - B/2 + 1, where A is the area of a lattice polygon,
I is number of lattice points inside it, and B is number of lattice points
on the boundary. Number of lattice points minus one on a line segment
from (0; 0) and (x; y) is gcd(x; y).
*/
/*
Kryterium Eulera (kiedy a jest reszta stopnia b modulo n?): jesli gcd(a,n)=1,
to (istnieje x: x^b=a(mod n)) <=> a^(phi(n)/gcd(phi(n),b))=1(mod n)

Duze liczby pierwsze:
    27281,    27397,        32237,    32251
    40507,    40591,        60331,    60353
    130079,   130367,       438443,   438499
    2023369,  2023453,      6384211,  6384709
    11796569, 11796767,     28892183, 28892267
    66926647, 66927067                              */

/*
#!/bin/bash
# Valgrind usage:
g++ -g a.cpp -o a
valgrind --tool=cachegrind ./a < t0.in
```

```
# You should now have a file named cachegrind.out.PID where PID is a number.
cg_annotate --auto=yes cachegrind.out.PID | less */

// Maksymalny leksykograficznie sufiks (algorytm Duvala)
// Bartosz Walczak
int duval(const char *s, int n) {
    int pb=0, pl=1, rb=1, rl=0;
    while (rb+rl<n) {
        if (s[rb+rl]>s[pb+rl]) { pb=rb++; pl=1; rl=0; }
        else if (s[rb+rl]<s[pb+rl]) { rb+=rl+1; pl=rb-pb; rl=0; }
        else if (++rl==pl) { rb+=pl; rl=0; }
    }
    return pb;
}
```

```
// SILNIE SPOJNE SKLADOWE (algrotym Tarjana) + 2SAT
// Adam Polak

const int N = 100*1000;
const int NIL = (-1);

int n;              // INPUT
vector<int> g[N];   // INPUT

int t, in[N], low[N];
stack<int> s;
bool stacked[N];

int scc[N], scc_n;  // OUTPUT (SCC)
bool value[N];      // OUTPUT (2SAT)

void tarjan(int u) {
    low[u] = in[u] = t++;
    s.push(u);
    stacked[u] = true;
    FOREACH(v, g[u]) {
        if (in[*v]==NIL) {
            tarjan(*v);
            low[u] = min(low[u], low[*v]);
        } else if (stacked[*v]) low[u] = min(low[u], in[*v]);
    }
    if (low[u]==in[u]) {
        for(;;) {
            int v = s.top(); s.pop();
            stacked[v] = false;
            scc[v] = scc_n;
            if (v==u) break;
        }
        scc_n++;
    }
}

void tarjan_scc() {
    REP(i,n) { in[i] = low[i] = NIL; stacked[i] = false; }
    scc_n = t = 0;
    REP(i,n) if(in[i]==NIL) tarjan(i);
}

// 2SAT usage:
// 1) n = 2*variables
// 2) REP(i,n) g[i].clear();
// 3) add_constr(...) //np. add_constr(zm1, 1, zm2, 0) = ((NOT zm1) OR zm2)
// 4) solve_2sat();
void add_constr(int a, bool neg_a, int b, bool neg_b) {
    g[2*a+neg_a].push_back(2*b+1-neg_b);
    g[2*b+neg_b].push_back(2*a+1-neg_a);
}

bool solve_2sat() {
    tarjan_scc();
    int v[scc_n], c[scc_n];
    REP(i,(n/2)) if (scc[2*i]==scc[2*i+1]) return false;
    REP(i,n) v[scc[i]] = i;
    REP(i,scc_n) c[i] = NIL;
    REP(i,scc_n) if (c[i]==NIL) {
        c[i] = 1;
        c[scc[v[i]^1]] = 0;
    }
    REP(i,(n/2)) value[i] = c[scc[2*i+1]];
    return true;
}
```

```
// Dwuspojne skladowe itd. - mosty traktowane jak dwuspojne!
// Linie oznaczone [D], [M], [A], [L] potrzebne tylko, jesli szukamy
//     numeracji [D]wuspojnych, [M]ostow, p.[A]rtykulacji, funkcji [L]ow
// Maciek Wawro

const int MAXN = 1000005;

struct Edge{ // W momencie rozpoczecia algorytmu musi byc bcc = -1 i bridge = 0
    Edge* rev;
    int dest;
    int bcc;        //OUT: Numer komponentu
    bool bridge;    //OUT: Czy most                          /*M*/
    Edge(int v) : dest(v), bcc(-1) {
        bridge = false;                                      /*M*/
    };
};

int N;                  //IN: Liczba wierzcholkow
list<Edge>  adj[MAXN];  //IN: Listy sasiedztwa
int  visit[MAXN];
bool  artp[MAXN];  //OUT: True, jesli dany wierzcholek jest p.art. /*A*/
int      bcc_num;  //OUT: Liczba komponentow                 /*D*/
int    low[MAXN];                                            /*L*/

stack<Edge*> _stack;                                         /*D*/
int        _dfsTime;
int bccDFS(int v, bool root = false) {
    int lo = visit[v] = ++_dfsTime;
    FOREACH(it, adj[v]) {
        if(it -> bcc != -1) continue;
        _stack.push(&*it);                                   /*D*/
        it->rev->bcc = -2;
        if(!visit[it->dest]) {
            int ulo = bccDFS(it->dest);
            lo = min(ulo, lo);
            it->bridge = it->rev->bridge = (ulo > visit[v]);/*M*/
            if(ulo >= visit[v]) {                            /*AD*/
                artp[v] = !root; root = false;               /*A*/
                Edge* edge;                                  /*D*/
                do {
                    edge = _stack.top();                     /*D*/
                    _stack.pop();                            /*D*/
                    edge->bcc = edge->rev->bcc = bcc_num;   /*D*/
                } while(edge != &*it);                       /*D*/
                ++bcc_num;                                   /*D*/
            }                                                /*AD*/
        } else lo = min(lo, visit[it->dest]);
    }
    low[v] = lo;                                             /*L*/
    return lo;
}

void computeBCC(){
    fill(artp, artp+N, false);                               /*A*/
    fill(visit, visit+N, false);
    _dfsTime = 1;
    bcc_num = 0;                                             /*D*/
    REP(i,N) if(!visit[i]) bccDFS(i, true);
}
```

```
// LCA i LAQ online (algorytm z drabinami)
// Adam Polak

const int N = 100000;

int n;                  // INPUT
vector<int> graph[N];   // INPUT

int in[N], out[N], p[N], size[N], l_ind[N], depth[N], dfstime;
vector<int> ladder[N]; // Clear these vectors after each testcase!

void dfs(int u) {
    in[u] = dfstime++; size[u] = 1; int best_s = 0; l_ind[u] = u;
    FOREACH(v, graph[u]) if (*v != p[u]) {
        p[*v] = u; depth[*v] = depth[u] + 1;
        dfs(*v);
        if (size[*v] > best_s) { best_s = size[*v]; l_ind[u] = l_ind[*v]; }
        size[u] += size[*v];
    }
    ladder[l_ind[u]].push_back(u);
    out[u] = dfstime++;
}

void init(int root) { dfstime = 0; p[root] = -1; depth[root] = 0; dfs(root); }

inline bool is_anc(int a, int b) { return (in[a] <= in[b] && out[b] <= out[a]); }

int LCA(int a, int b) {  // Lowest common ancestor
    int k = l_ind[a];
    while(!is_anc(*(ladder[k].end()-1), b)) k = l_ind[p[*(ladder[k].end()-1)]];
    int l = 0, r = ladder[k].size() - 1;
    while(l < r) {
        int mid = (l + r) / 2;
        if (is_anc(ladder[k][mid], b) && is_anc(ladder[k][mid], a))
            r = mid;
        else
            l = mid+1;
    }
    return ladder[k][l];
}

int LAQ(int a, int x) {  // Level ancestor query
    if (depth[a] < x) return -1;
    int d = depth[a] - x;
    int k = l_ind[a];
    while(depth[*(ladder[k].end()-1)] > d) k = l_ind[p[*(ladder[k].end()-1)]];
    return *(ladder[k].end() - 1 - d + depth[*(ladder[k].end()-1)]);
}
```

```
// LCA (algorytm Tarjana)
// Adam Polak

const int N = 1000000;
int n;                          // INPUT
vector<int> g[N];               // INPUT: graph
vector<pair<int,int*> > q[N];   // INPUT: queries
int p[N], anc[N];
bool col[N];

int set_find(int a) { return p[a]==a?a:p[a]=set_find(p[a]); }

void dfs(int u, int f) {
    FOREACH(v, g[u]) if (*v!=f) {
        dfs(*v, u);
        anc[p[set_find(u)]=set_find(*v)] = u;
    }
    col[u] = true;
    FOREACH(i,q[u])
        if (col[i->first]) *(i->second) = anc[set_find(i->first)];
}

void lca(int root) {
    REP(i,n) { p[i]=anc[i]=i; col[i]=0; }
    dfs(root, -1);
}

// CYKL EULERA w grafie eulerowskim O(V+E)

const int MAXN = 1100000, MAXM = 11000000; // max liczba wierzcholkow i krawedzi

struct edge {
    int v;
    int back_idx; //TYLKO DLA NIESKIEROWANYCH

    bool vis;     //TYLKO DLA NIESKIEROWANYCH
    edge(int vi):v(vi){
        vis = false; //TYLKO DLA NIESKIEROWANYCH
    }
};

int n;                // IN: liczba wierzcholkow
vector<edge> adj[MAXN]; // IN: lista sasiedztwa (dla skierowanych: do przodu)
int cc;               // OUT: dlugosc cyklu Eulera
edge *cycle[MAXM];    // OUT: kolejne krawedzie na cyklu Eulera
int cur[MAXN];

stack<edge*>e_stack_;

// aby znalezc sciezke Eulera: dodaj krawedz miedzy
// wierzcholkami o nieparzystym stopniu, znajdz cykl i
// potem usun ta krawedz
void search(int v) {
    while(true){
        if(cur[v] == adj[v].size()) {
            if (!e_stack_.empty()) {
                cycle[cc++] = e_stack_.top();
                e_stack_.pop();
            }
        } else {
            edge *e = &adj[v][cur[v]++];
            if (!e->vis) {                      //TYLKO DLA NIESKIEROWANYCH
                adj[e->v][e->back_idx].vis = true;  //TYLKO DLA NIESKIEROWANYCH
                e_stack_.push(e);               //TO ZOSTAJE ZAWSZE
            }                                   //TYLKO DLA NIESKIEROWANYCH
        }
        if (e_stack_.empty()) break;
        v = e_stack_.top()->v;
    }
}

void compute_cycle() {
    REP(v,n) cur[v] = 0;
    cc = 0;
    REP(v,n) search(v);
}

void add_edge(int a, int b){
    adj[a].push_back(edge(b));
    adj[b].push_back(edge(a));                      //TYLKO DLA NIESKIEROWANYCH
    adj[a].back().back_idx = adj[b].size()-1;    //TYLKO DLA NIESKIEROWANYCH
    adj[b].back().back_idx = adj[a].size()-1;    //TYLKO DLA NIESKIEROWANYCH
}
```

```
// MAKSYMALNE SKOJARZENIE w dowolnym grafie O(V^3)
// Bartosz Walczak

const int MAXN = 100; // maksymalna liczba wierzcholkow

int n;                  // IN: liczba wierzcholkow
bool edge[MAXN][MAXN]; // IN: macierz sasiedztwa (mozna zmienic na liste)
int mate[MAXN];        // OUT: wierzcholek skojarzony (-1 oznacza brak)
int label[MAXN], base[MAXN], prev1[MAXN], prev2[MAXN];
bool mark[MAXN];

bool prepare(int v) {
    for (;;) {
        mark[v] = !mark[v];
        if (mate[v]==-1) return mark[v];
        v = base[prev2[mate[v]]];
    }
}

int shrink(int v, int b1, int b2, queue<int> &Q) {
    while (mark[v]) {
        prev1[v] = b1; prev2[v] = b2;
        mark[mate[v]] = true;
        Q.push(mate[v]);
        v = base[prev2[mate[v]]];
    }
    return v;
}

bool make_blos(int i, int j, int bi, int bj, queue<int> &Q) {
    if (label[i]!=1 || i==j) return false;
    if (prepare(i), prepare(j)) return true;
    int b = (shrink(i, bi, bj, Q), shrink(j, bj, bi, Q));
    REP(v,n) if (mark[base[v]]) base[v] = b;
    return false;
}

void rematch(int i, int j) {
    int next = mate[i];
    mate[i] = j;
    if (next==-1) return;
    mate[next] = -1;
    rematch(prev2[next], prev1[next]);
    rematch(prev1[next], prev2[next]);
}

bool augment() {
    queue<int> Q;
    REP(i,n) {
        label[i] = mate[i]==-1;
        if (mate[i]==-1) Q.push(i);
        mark[i] = false;
        base[i] = i;
    }
    while (!Q.empty()) {
        int cur = Q.front(); Q.pop();
        REP(i,n) /*tu zmienic*/ if (edge[cur][i] && i!=mate[cur]) {
            if (!label[i]) {
                label[i] = -1;
                label[mate[i]] = 1;
                Q.push(mate[i]);
                prev1[i] = i; prev2[i] = cur;
            }
            else if (make_blos(base[i], base[cur], i, cur, Q)) {
                rematch(i, cur); rematch(cur, i);
                return true;
            }
        }
    }
    return false;
}

int compute_gcm() { // zwraca licznosc maksymalnego skojarzenia
    fill_n(mate, n, -1);
    int res = 0;
    while (augment()) ++res;
    return res;
}

// MAKSYMALNE SKOJARZENIE w grafie dwudzielnym ("Turbomatching")
// Adam Polak

const int N = 10000;
int n1,n2;          // INPUT
vector<int> g[N];   // INPUT
int m1[N], m2[N];   // OUTPUT
bool c[N];

bool dfs(int u) {
    if (u<0) return true;
    if (c[u]) return false; else c[u]=true;
    FOREACH(v, g[u])
        if (dfs(m2[*v])) { m1[u] = *v; m2[*v] = u; return true; }
    return false;
}

int matching() {
    REP(i,n1) m1[i]=-1;
    REP(i,n2) m2[i]=-1;
    bool changed;
    do {
        changed = 0;
        REP(i,n1) c[i]=false;
        REP(i,n1) if (m1[i]<0) changed |= dfs(i);
    } while(changed);
    int siz = 0;
    REP(i,n1) siz += (m1[i]!=-1);
    return siz;
}
```

```
// MAKSYMALNY PRZEPLYW O(V^3) (algorytm Push-Relabel)
// Adam Polak

const int N = 100*1000;

struct Edge {
    int v,cap,flow;
    int back_ind;
    Edge *back;
    Edge(int vi, int ci):v(vi),cap(ci){}
};

/* Usage:
   1) n=...; s=...; t=...;
   2) REP(i,n) g[i].clear();
   3) add_edge(...);
   4) compute_flow();
*/

int n,s,t;
int e[N],h[N];
vector<Edge> g[N];
vector<Edge>::iterator cur[N];

void bfs(int start, int start_h) {
    queue<int> q;
    h[start] = start_h;
    for(q.push(start);!q.empty();q.pop()) {
        int u = q.front();
        FOREACH(i,g[u])
            if (i->back->flow < i->back->cap && h[i->v]>h[u]+1) {
                    h[i->v] = h[u] + 1;
                    q.push(i->v);
            }
    }
}

int compute_flow() {
    queue<int> q;
    REP(i,n) {
        FOREACH(j,g[i]) {
            j->flow = 0;
            j->back = &g[j->v][j->back_ind];
        }
        cur[i] = g[i].begin();
        h[i] = e[i] = 0;
    }
    FOREACH(i,g[s]) {
        i->flow = i->cap;
        i->back->flow = -i->flow;
        if (e[i->v]==0 && i->v!=t) q.push(i->v);
        e[i->v] += i->flow;
    }
    h[s] = n;
    int relabel_counter = 0;
    for(;!q.empty();q.pop()) {
        int u = q.front();
        while (e[u]>0) {
            if (cur[u]==g[u].end()) { // relabel
                relabel_counter++;
                h[u] = 2*n+1;
                FOREACH(i,g[u]) if(i->flow < i->cap) h[u]=min(h[u],1+h[i->v]);
                cur[u] = g[u].begin();
                continue;
            }
            if (cur[u]->flow < cur[u]->cap && h[u]==h[cur[u]->v]+1) { // push
                int d = min(e[u], cur[u]->cap - cur[u]->flow);
                cur[u]->flow += d;
                cur[u]->back->flow -= d;
                e[u] -= d;
                e[cur[u]->v] += d;
                if (e[cur[u]->v]==d && cur[u]->v!=t && cur[u]->v!=s) q.push(cur[u]->v);
            } else cur[u]++;
        }
        if (relabel_counter >= n) {
            REP(i,n) h[i]=2*n+1;
            bfs(t,0);
            bfs(s,n);
            relabel_counter = 0;
        }
    }
    return e[t];
}

void add_edge(int a, int b, int c, int c_back=0) {
    assert(a != b);  // NIE wrzucac petelek!
    g[a].push_back(Edge(b,c));
    g[b].push_back(Edge(a,c_back));
    g[a].back().back_ind = g[b].size()-1;
    g[b].back().back_ind = g[a].size()-1;
}

// MAKSYMALNY PRZEPLYW (algorytm Edmondsa-Karpa)
// Adam Polak

const int N = 1000;
int n, cap[N][N];   // INPUT
int flow[N][N];     // OUTPUT

int edmonds(int s, int t) {
    int b,e,q[n],p[n],d,FLOW=0;
    REP(i,n) REP(j,n) flow[i][j]=0;
    for(;;) {
        REP(i,n) p[i]=-1;
        for(q[b=e=0]=s;b<=e;b++)
            REP(v,n)
                if (flow[q[b]][v] < cap[q[b]][v] && p[v]<0)
                    p[q[++e]=v] = q[b];
        if (p[t]<0) break;
        d = cap[p[t]][t] - flow[p[t]][t];
        for(int i=t;i!=s;i=p[i]) d=min(d,cap[p[i]][i]-flow[p[i]][i]);
        for(int i=t;i!=s;i=p[i]) {
            flow[p[i]][i] += d;
            flow[i][p[i]] -= d;
        }
        FLOW += d;
    }
    return FLOW;
}
```

```
// MAX FLOW MIN COST O(V^2F)
// Adam Polak

// Usage: vide push-relabel

typedef int capacity_t;
typedef int cost_t;

const int N = 10000;
// use INFINITY from cmath if cost_t is a double
const cost_t INF = 1000*1000*1000;

struct edge {
    int from, v;
    capacity_t cap, flow;
    cost_t cost, dist;
    int revIndex;
    bool residual() { return flow<cap; }
    edge(int _f, int _v, capacity_t _cap, cost_t _cost):
        from(_f),v(_v),cap(_cap),flow(0),cost(_cost),dist(_cost){}
};

int n,s,t;
cost_t d[N];
vector<edge>::iterator p[N];
vector<edge> g[N];

bool queued[N];
void bellmanFord() {
    REP(i,n) { d[i]=INF; queued[i]=0; }
    queue<int> q;
    d[s]=0; q.push(s); queued[s]=0;
    while(!q.empty()) {
        int u=q.front(); q.pop(); queued[u]=0;
        FOREACH(i, g[u])
            if (i->residual() && d[i->v] > d[u]+i->dist) {
                d[i->v] = d[u]+i->dist;
                p[i->v] = i;
                if (!queued[i->v]) { q.push(i->v); queued[i->v]=1; }
            }
    }
}

void dijkstra() {
    REP(i,n) { d[i]=INF; }
    priority_queue< pair<cost_t,int>, vector<pair<cost_t,int> >,
        greater<pair<cost_t,int> > > q;
    d[s]=0; q.push(make_pair(0,s));
    while(!q.empty()) {
        int u = q.top().second;
        cost_t dist = q.top().first;
        q.pop();
        if (dist!=d[u]) continue;
        FOREACH(i, g[u])
            if (i->residual() && d[i->v] > d[u]+i->dist) {
                d[i->v] = d[u]+i->dist;
                p[i->v] = i;
                q.push(make_pair(d[i->v],i->v));
            }
    }
}

void usePotentials() {
    REP(u,n) FOREACH(i, g[u])
        i->dist = i->dist + d[i->from] - d[i->v];
}

// Mozna (za/od)komentowac linie z gwiazdkami - w praktyce bywa szybciej
capacity_t FLOW;
cost_t COST;
void fordFulkerson() {
    FLOW = 0;
    COST = 0;
    bellmanFord();       // *
    usePotentials();     // *
    for(;;) {
        // bellmanFord();   // * dijkstra sux ;)
        dijkstra();      // *
        usePotentials(); // *
        if (d[t]==INF) break;
        cost_t cost = 0;
        for(int u=t;u!=s;u=p[u]->from) {
            p[u]->flow++;
            g[p[u]->v][p[u]->revIndex].flow--;
            cost += p[u]->cost;
        }
        FLOW ++; COST += cost;
    }
}

void addEdge(int a, int b, capacity_t f, cost_t c) {
    assert(a != b);  // NIE wrzucac petelek!
    g[a].push_back(edge(a,b,f,c));
    g[b].push_back(edge(b,a,0,-c));
    g[a].back().revIndex = g[b].size()-1;
    g[b].back().revIndex = g[a].size()-1;
}
```

```
// SKOJARZENIE O MINIMALNEJ WADZE w grafie dwudzielnym O(V^3)
// Bartosz Walczak

const int MAXN = 100; // maksymalne liczby wierzcholkow po obu stronach
const int INF = 1e9;
int n1, n2;             // IN: liczby wierzcholkow po obu stronach
int weight[MAXN][MAXN]; // IN: macierz wag krawedzi (INF oznacza brak)
int mate1[MAXN], mate2[MAXN]; // OUT: wierzcholki skojarzone (-1 oznacza brak)
int old_dist1[MAXN], dist1[MAXN], dist2[MAXN], prev2[MAXN];

int compute_bwm() { // zwraca wage skojarzenia
   REP(a,n1) {
      mate1[a] = -1;
      old_dist1[a] = 0;
   }
   REP(b,n2) {
      mate2[b] = -1;
      dist2[b] = INF;
      REP(a,n1) if (weight[a][b]<dist2[b]) {
         dist2[b] = weight[a][b];
         prev2[b] = a;
      }
   }
   int res = 0;
   for (;;) {
/* Jezeli szukamy skojarzenia licznosci k o minimalnej wadze, te petle wykonac
   dokladnie k razy                                                          */
      int cur, min_dist = INF;
      REP(b,n2) if (mate2[b]==-1 && dist2[b]<min_dist) {
         cur = b;
         min_dist = dist2[b];
      }
/* Jezeli szukamy skojarzenia dowolnej licznosci o minimalnej wadze, poprawic
   ponizszy warunek na: min_dist>=0                                          */
      if (min_dist==INF) break;
      res += min_dist;
      while (cur!=-1) {
         int next = mate1[prev2[cur]];
         mate2[cur] = prev2[cur];
         mate1[prev2[cur]] = cur;
         cur = next;
      }
      REP(a,n1) dist1[a] = mate1[a]==-1 ? 0 : INF;
      REP(b,n2) dist2[b] = INF;
      for (;;) {
         min_dist = INF;
         REP(a,n1) if (dist1[a]!=INF && dist1[a]-old_dist1[a]<min_dist) {
            cur = a;
            min_dist = dist1[a]-old_dist1[a];
         }
         if (min_dist==INF) break;
         REP(b,n2) if (b!=mate1[cur] && weight[cur][b]!=INF &&
          dist1[cur]+weight[cur][b]<dist2[b]) {
            dist2[b] = dist1[cur]+weight[cur][b];
            prev2[b] = cur;
            if (mate2[b]!=-1) dist1[mate2[b]] = dist2[b]-weight[mate2[b]][b];
         }
         old_dist1[cur] = dist1[cur];
         dist1[cur] = INF;
      }
   }
   return res;
}

// ALGORYTM WEGIERSKI O(n^3) [najdrozsze skojarzenie doskonale]
// Adam Polak

const int N = 500;
const int INF = 1e9;

int n;                  // INPUT
int w[N][N];            // INPUT
int m1[N], m2[N];       // OUTPUT
int l1[N], l2[N], p[N], slack[N];
bool blue[N];

int hungarian() {
    REP(i,n) { m1[i] = m2[i] = p[i] = -1; l1[i] = l2[i] = -INF; }
    REP(i,n) REP(j,n) {
        l1[i]=max(l1[i],w[i][j]);
        l2[j]=max(l2[j],w[i][j]);
    };
    REP(k,n) {
        REP(i,n) { slack[i] = INF; p[i] = -1; }
        REP(i,n) if (blue[i] = (m1[i]==-1))
            REP(j,n) slack[j] = min(slack[j], l1[i]+l2[j]-w[i][j]);
        int u;
        for(;;) {
            u = min_element(slack,slack+n)-slack;
            if (slack[u]) {
                int d = slack[u];
                REP(i,n) {
                    if (blue[i]) l1[i] -= d;
                    if (slack[i] == INF) l2[i] += d; else slack[i] -= d;
                }
            } else {
                REP(i,n) if (blue[i]&&w[i][u]==l1[i]+l2[u]) { p[u]=i; break; }
                slack[u]=INF;
                if (m2[u] == -1) break;
                blue[m2[u]]=true;
                REP(i,n) if (slack[i]!=INF)
                    slack[i]=min(slack[i], l1[m2[u]]+l2[i] - w[m2[u]][i]);
            }
        }
        while(u != -1) { m2[u] = p[u]; swap(u, m1[p[u]]); }
    }
    /* Jezeli wynik moze przekroczyc 10^9 zmienic result i typ zwracany na LL */
    int result = 0;
    REP(i,n) result += w[i][m1[i]];
    return result;
}
```

```
/* MINIMALNE SKIEROWANE DRZEWO ROZPINAJACE O(E log(V))
   Zalozenia: 1. do korzenia nie wchodzi zadna krawedz
              2. istnieje sciezka z korzenia do kazdego innego wierzcholka    */

// Bartosz Walczak

const int MAXN = 100, MAXM = 100; // maksymalna liczba wierzcholkow i krawedzi

struct edge { // krawedz/element kolejki zlaczalnej
    int u, v; // IN: poczatek i koniec krawedzi
    int key;  // IN: waga krawedzi (zmienia sie!)

    edge *left, *right; // poczatkowo: 0, 0
    int len, add;       // poczatkowo: 1, 0
};

struct node1 { // element zbioru
    node1 *parent;
    int size, scc;
};
struct node2 { // j.w.
    node2 *parent; // poczatkowa wartosc: this
    int size;      // poczatkowa wartosc: 1
};

// Operacje na zbiorach rozlacznych
template<class T>
T *set_find(T *p) { // znajduje reprezentanta
    if (p->parent != p) p->parent = set_find(p->parent);
    return p->parent;
}
template<class T>
T *set_union(T *p1, T *p2) { // laczy zbiory
    if (p1->size < p2->size) swap(p1, p2);
    p2->parent = p1;
    p1->size += p2->size;
    return p1;
}

// Operacje na kolejkach zlaczalnych
void tree_push(edge *p) {
    p->key += p->add;
    if (p->left) p->left->add += p->add;
    if (p->right) p->right->add += p->add;
    p->add = 0;
}
edge *tree_union(edge *p1, edge *p2) { // laczy kolejki
    if (!p1) return p2;
    if (!p2) return p1;
    if (p2->key+p2->add < p1->key+p1->add) swap(p1, p2);
    tree_push(p1);
    p1->right = tree_union(p1->right, p2);
    if (!p1->left || p1->left->len < p1->right->len) swap(p1->left, p1->right);
    p1->len = p1->right ? p1->right->len+1 : 1;
    return p1;
}
edge *tree_extract(edge *p) { // usuwa z kolejki element najmniejszy
    tree_push(p);
    return tree_union(p->left, p->right);
}
void tree_add(edge *p, int x) { // dodaje x do wszystkich wartosci w kolejce
    if (p) p->add += x;
}

int n, m;         // IN: liczba wierzcholkow, liczba krawedzi
edge edges[MAXM]; // IN: tablica wszystkich krawedzi
node1 scc_set[MAXN];
node2 wcc_set[MAXN];
int upper[2*MAXN], lower[2*MAXN];
edge *adj[2*MAXN];
edge *res[2*MAXN]; // OUT: krawedz do rodzica w drzewie (korzen ma NULL)

int compute_branching() { // zwraca wage drzewa
    FOR(i,0,n) {
        scc_set[i].parent = scc_set+i;
        scc_set[i].size = 1;
        scc_set[i].scc = i;
        wcc_set[i].parent = wcc_set+i;
        wcc_set[i].size = 1;
        upper[i] = lower[i] = -1;
        adj[i] = res[i] = 0;
    }
    FOR(j,0,m) {
        edges[j].left = edges[j].right = 0;
        edges[j].len = 1;
        edges[j].add = 0;
        adj[edges[j].v] = tree_union(adj[edges[j].v], edges+j);
    }
    int scc_c=n, value=0;
    FOR(i,0,n) {
        int c = set_find(scc_set+i)->scc;
        while (adj[c] && !res[c]) {
            edge *e = adj[c];
            adj[c] = tree_extract(adj[c]);
            node1 *s1 = set_find(scc_set+e->v), *s2 = set_find(scc_set+e->u);
            if (s1==s2) continue;
            res[c] = e;
            value += e->key;
            tree_add(adj[c], -e->key);
            node2 *w1 = set_find(wcc_set+e->v), *w2 = set_find(wcc_set+e->u);
            if (w1!=w2) { set_union(w1, w2); continue; }
            upper[c] = scc_c;
            do {
                e = res[s2->scc];
                upper[s2->scc] = scc_c;
                adj[c] = tree_union(adj[c], adj[s2->scc]);
                s1 = set_union(s1, s2);
                s2 = set_find(scc_set+e->u);
            } while (s1!=s2);
            s1->scc = scc_c;
            upper[scc_c] = lower[scc_c] = -1;
            adj[scc_c] = adj[c];
            res[scc_c] = 0;
            c = scc_c++;
        }
    }
    FORD(c,scc_c,n) {
        if (lower[c]==-1)
          for (int i=res[c]->v; i!=c; i=upper[i]) lower[upper[i]] = i;
        res[lower[c]] = res[c];
    }
    return value;
}
```

```
/* LEX-BFS I ZAWEZANIE PODZIALU. Rozpoznawanie grafow cieciwowych
   1. lex_bfs  - oblicza porzadek odwrotny Lex-BFS wierzcholkow
   2. test_peo - sprawdza czy obliczony porzadek to Perfect Elimination Order
                 grafu cieciwowego                                            */

// Bartosz Walczak

template<class Iter> inline Iter PREV/*NEXT*/(Iter i) { return --i/*++i*/; }

struct clas;
struct elem { // element klasy podzialu
    int v; list<clas>::iterator c; // numer elementu, iterator klasy
    elem(int vi, list<clas>::iterator ci):v(vi), c(ci) {}
};
struct clas { // klasa podzialu
    list<elem> L; int label; // lista elementow, pomocnicza etykieta
    clas(int li):label(li) {}
};

/* Zawezanie podzialu: umieszcza nowa klase elementu i przed stara
   Mozna zmienic tak, zeby umieszczal za stara                              */
void refine(list<clas> &C, list<elem>::iterator i, int lab) {
    list<clas>::iterator c=i->c;
    if (c->label!=lab) { C.insert(c/*NEXT(c)*/, clas(lab)); c->label=lab; }
    i->c = PREV(c)/*NEXT(c)*/;
    i->c->L.splice(i->c->L.end(), c->L, i);
    if (c->L.empty()) C.erase(c);
}

const int MAXN = 100; // maksymalna liczba wierzcholkow

int n;              // IN: liczba wierzcholkow
list<int> adj[MAXN]; // IN: lista sasiedztwa
int order[MAXN];     // OUT: porzadek Lex-BFS
int label[MAXN];     // OUT: pozycja wierzcholka w porzadku Lex-BFS
list<elem>::iterator iter[MAXN];

void lex_bfs() { // Lex-BFS
    fill_n(label, n, -1);
    list<clas> C; C.push_front(clas(-1));
    clas *c = &C.front();
    FOR(i,0,n) iter[i] = c->L.insert(c->L.end(), elem(i, C.begin()));
    FORD(cur,n,0) {
        c = &C.front();
        int v = c->L.front().v; c->L.pop_front();
        if (c->L.empty()) C.pop_front();
        order[cur]=v; label[v]=cur;
        FORE(i,adj[v]) if (label[*i]==-1) refine(C, iter[*i], cur);
    }
}

int par[MAXN]; // OUT: pierwszy prawy sasiad (-1 - brak)
int cnt[MAXN]; // OUT: liczba prawych sasiadow

bool test_peo() { // jesli porzadek jest PEO, zwraca true i oblicza par, cnt
    fill_n(par, n, -1);
    FOR(i,0,n) {
        int v=order[i]; par[v]=-1; cnt[v]=0;
        FORE(j,adj[v]) if (label[*j]>i) { par[*j]=v; ++cnt[v]; }
                       else if (par[*j]==-1) par[*j]=v;
        FORE(j,adj[v]) if (par[*j]==v && label[*j]<i)
          FORE(k,adj[*j]) if (label[*k]>i && par[*k]!=v) return false;
    }
    return true;
}
```

```
/* ALGORYTMY KOMBINATORYCZNE. Schemat generowania kolejnych obiektow:
   1. Inicjalizacja zmiennych i tablic pierwszym obiektem
   2. Generowanie nastepnych obiektow funkcja next_*, poki zwraca true         */

// Bartosz Walczak

const int MAXN = 100;

typedef int T; // typ, ktory miesci liczbe wszystkich obiektow

/* Generowanie i zliczanie podzbiorow k-elementowych zbioru {0,...,n-1} w
   porzadku leksykograficznym: od {0,...,k-1} do {n-k,...,n-1}               */

int n, k;
int elem[MAXN];  // elementy podzbioru w porzadku rosnacym
bool used[MAXN]; // czy element jest w podzbiorze? (nieuzywana w algorytmach)

bool next_subset() { // nastepny podzbior. Zwraca false, jesli wrocil do
    int cur=1;       // pierwszego. Czas zamortyzowany O(1)
    while (cur<=k && elem[k-cur]==n-cur) used[n-cur++]=false;
    if (cur>k) {
        FOR(i,0,k) { elem[i]=i; used[i]=true; }
        return false;
    }
    int pos=elem[k-cur]; used[pos++]=false;
    FOR(i,0,cur) { elem[k-cur+i]=pos+i; used[pos+i]=true; }
    return true;
}

T newton[MAXN+1][MAXN+1]; // newton[n][k] - symbol Newtona n po k

void init_subset() { // wypelnia tablice newton
    newton[0][0]=1;
    FOR(j,1,n+1) newton[0][j]=0;
    FOR(i,1,n+1) {
        newton[i][0]=newton[i-1][0];
        FOR(j,1,n+1) newton[i][j]=newton[i-1][j-1]+newton[i-1][j];
    }
}
T count_subset() { // zwraca numer podzbioru. Czas O(k)
    T res=0; int first=0;
    FOR(i,0,k) {
        res+=newton[n-first][k-i]-newton[n-elem[i]][k-i];
        first=elem[i]+1;
    }
    return res;
}
void gen_subset(T no) { // generuje podzbior o podanym numerze. Czas O(n)
    fill_n(used, n, false);
    int first=0, pos=0;
    FOR(i,0,k) {
        while (no>=newton[n-first][k-i]-newton[n-pos-1][k-i]) ++pos;
        no-=newton[n-first][k-i]-newton[n-pos][k-i];
        elem[i]=pos; used[pos]=true;
        first=++pos;
    }
}

/* Generowanie i zliczanie podzialow liczby n na sume dodatnich skladnikow w
   porzadku antyleksykograficznym: od n do 1+1+...+1                         */

int n, k;                  // k - liczba roznych skladnikow, k=O(sqrt(n))
int elem[MAXN], cnt[MAXN]; // elem - skladnik, cnt - ile razy?

bool next_partition1() { // nastepny podzial. Zwraca false, jesli wrocil do
    int sum=0;           // pierwszego. Czas O(1)
    if (elem[k-1]==1) sum+=cnt[--k];
```

```
    if (!k) { elem[k]=sum; cnt[k++]=1; return false; }
    sum+=elem[k-1];
    if (--cnt[k-1]) { elem[k]=elem[k-1]-1; ++k; }
    else --elem[k-1];
    cnt[k-1]=sum/elem[k-1]; sum%=elem[k-1];
    if (sum) { elem[k]=sum; cnt[k++]=1; }
    return true;
}

T pcnt[MAXN+1][MAXN+1]; // pcnt[n][k] - liczba podzialow liczby n o najwiekszym
                        // skladniku <=k == liczba podzialow na <=k skladnikow
void init_partition1() { // wypelnia tablice pcnt
    FOR(j,0,n+1) pcnt[0][j]=1;
    FOR(i,1,n+1) {
        pcnt[i][0]=0;
        FOR(j,1,i+1) pcnt[i][j]=pcnt[i][j-1]+pcnt[i-j][j];
        FOR(j,i+1,n+1) pcnt[i][j]=pcnt[i][i];
    }
}
T count_partition1() { // zwraca numer podzialu. Czas O(k)
    T res=0; int sum=n, first=n;
    FOR(i,0,k) {
        res+=pcnt[sum][first]-pcnt[sum][elem[i]];
        sum-=elem[i]*cnt[i]; first=elem[i];
    }
    return res;
}
void gen_partition1(T no) { // generuje podzial o podanym numerze. Czas O(n)
    k=0; int sum=n, first=n, cur=n;
    while (sum) {
        while (no>=pcnt[sum][first]-pcnt[sum][cur-1]) --cur;
        no-=pcnt[sum][first]-pcnt[sum][cur];
        elem[k]=cur; cnt[k++]=1;
        sum-=first=cur;
        while (no<pcnt[sum][cur]-pcnt[sum][cur-1]) { ++cnt[k-1]; sum-=cur; }
    }
}

/* Generowanie i zliczanie podzialow zbioru {0,...,n-1} na niepuste podzbiory
   w porzadku leksykograficznym: od {0,...,n-1} do {0},...,{n-1}
   Uwaga: Klasy podzialu sa numerowane w porzadku leksykograficznym elementow.
   W szczegolnosci element 0 zawsze nalezy do klasy 0.                        */

int n, cnt;     // cnt - liczba klas podzialu
int size[MAXN]; // rozmiar klasy podzialu
int pos[MAXN];  // numer klasy elementu. Ta tablica wyznacza porzadek podzialow

bool next_partition2() { // nastepny podzial. Zwraca false, jesli wrocil do
    FORD(i,n,1) {        // pierwszego. Czas zamortyzowany O(1)
        if (pos[i]==cnt-1) {
            if (size[cnt-1]==1) { pos[i]=0; ++size[0]; --cnt; continue; }
            size[cnt++]=0;
        }
        --size[pos[i]]; ++pos[i]; ++size[pos[i]];
        return true;
    }
    return false;
}

T pcnt[MAXN+1][MAXN+1]; // pcnt[n][0] - liczba podzialow zbioru n-elementowego

void init_partition2() { // wypelnia tablice pcnt
    FOR(j,0,n+1) pcnt[0][j]=1;
    FOR(i,1,n+1) FOR(j,0,n+1) pcnt[i][j]=j*pcnt[i-1][j]+pcnt[i-1][j+1];
}
T count_partition2() { // zwraca numer podzialu. Czas O(n)
    T res=0; int wd=0;
    FOR(i,0,n) {
        res+=pos[i]*pcnt[n-i-1][wd];
        if (pos[i]==wd) ++wd;
    }
    return res;
}
void gen_partition2(T no) { // generuje podzial o podanym numerze. Czas O(n)
    cnt=0;
    FOR(i,0,n) {
        int p=min(cnt, no/pcnt[n-i-1][cnt]);
        no-=p*pcnt[n-i-1][cnt];
        if (p==cnt) size[cnt++]=0;
        pos[i]=p; ++size[p];
    }
}

// Arkadiusz Pawlik

/* Kod Gray'a: gray(0),...,gray(2^n-1) - permutacja liczb 0,...,2^n-1, w ktorej
   kazde dwie kolejne oraz ostatnia z pierwsza roznia sie tylko na 1 bicie    */
unsigned gray(unsigned n) { return n^n>>1; }
unsigned igray(unsigned n) { // funkcja odwrotna do gray
    n^=n>>1; n^=n>>2; n^=n>>4; n^=n>>8; n^=n>>16;
    return n;
}

// Odwrocenie kolejnosci bitow (operuje na n najmniej znaczacych bitach)
unsigned rev(unsigned v, unsigned n) {
    v = (v & 0xffff0000)>>16 | (v & 0x0000ffff)<<16;
    v = (v & 0xff00ff00)>>8  | (v & 0x00ff00ff)<<8;
    v = (v & 0xf0f0f0f0)>>4  | (v & 0x0f0f0f0f)<<4;
    v = (v & 0xcccccccc)>>2  | (v & 0x33333333)<<2;
    v = (v & 0xaaaaaaaa)>>1  | (v & 0x55555555)<<1;
    return v >> 32-n;
}

// Liczba niezerowych bitow
unsigned bitcount(unsigned v) {
    v = ((v & 0xaaaaaaaa)>>1)  + (v & 0x55555555);
    v = ((v & 0xcccccccc)>>2)  + (v & 0x33333333);
    v = ((v & 0xf0f0f0f0)>>4)  + (v & 0x0f0f0f0f);
    v = ((v & 0xff00ff00)>>8)  + (v & 0x00ff00ff);
    v = ((v & 0xffff0000)>>16) + (v & 0x0000ffff);
    return v;
}
```

```
// TEST RABINA MILLERA
// + MNOZENIE MODULO 64bit LICZB
// Maciek Wawro

typedef unsigned long long ULL;
const int _k = 16;
const ULL _mask = (1<<_k)-1;
//   Zalozenia: b, MOD < 2^(64-_k)
ULL mul(ULL a, ULL b, ULL MOD){
    ULL result = 0;
    while(a){
        ULL temp = (b*(a&_mask)) % MOD;
        result = (result+temp) % MOD;
        a >>= _k;
        b = (b<<_k) % MOD;
    }
    return result;
}

/* inline ULL mul(ULL a, ULL b, ULL MOD) { return (a*b) % MOD; } */

ULL pow(ULL a, ULL w, ULL MOD) {
    ULL res = 1;
    while(w){
        if (w&1) res = mul(res, a, MOD);
        a = mul(a, a, MOD);
        w >>= 1;
    }
    return res;
}

bool primeTest(ULL N, int a) {
    if(a > N-1) return true;
    ULL d = N-1;
    int s = 0;
    while(!(d&1)){
        d >>= 1;
        s++;
    }
    ULL x = pow(a, d, N);
    if((x==1)||(x==N-1)) return true;
    REP(i,s-1){
        x = mul(x, x, N);
        if(x == 1) return false;
        if(x == N-1) return true;
    }
    return false;
}

/*  Dla N<2^32 testujemy 2, 7, 61
 *  Dla N<2^48 testujemy pierwsze z [2,17]
 *  Dla N<2^64 testujemy 2, 325, 9375, 28178,
        450775, 9780504, 1795265022             */

bool isPrime(ULL N) {
    if(N<4) return N>1;
    bool prime = N%2;
    prime = prime && primeTest(N, 2);
    prime = prime && primeTest(N, 7);
    prime = prime && primeTest(N, 61);
    return prime;
}

/*  Test mozna przyspieszyc, sprawdzajac najpierw podzielnosc przez
 *  pierwsze kilkanascie liczb pierwszych. */
```

```
// CHINSKIE TWIERDZENIE O RESZTACH
// + ALGORYTM EUKLIDESA
// Adam Polak
// + FAKTORYZACJA RHO POLLARDA
// Robet Obryk

/*  Ponizsze kody operuja od poczatku do konca na liczbach nieujemnych. */

typedef unsigned long long T;

T GCD(T a, T b) {
    while(b) { a%=b; swap(a,b); }
    return a;
}

/* gcd = ax - by; x,y >= 0 */
T eGCD(T a, T b, T &x, T &y) {
    if (!b) { x=1; y=0; return a; }
    T d = eGCD(b,a%b,y,x);
    y = a-x*(a/b)-y;
    x = b-x;
    return d;
}

T inverse(T a, T p) { T x,y; eGCD(a,p,x,y); return x % p; }

/*  x = a mod m, x = b mod n,  0 <= x < LCM(n,m)
 *  Jesli n <= m, to w zadnym miejscu obliczenia nie wykraczaja poza
 *      max(LCM(n,m), 2n)                                              */
T CRT(T a, T m, T b, T n) {
    b = (b+n-(a%n))%n;
    T d = GCD(m,n);
    if (b%d) throw 0;
    return (((b/d)*inverse(m/d,n/d))%(n/d))*m+a;
}

T RHO(T n) {
    if (n%2 == 0) return 2;
    T x = 2, y = 2;
    do {
        x = (x*x+1)%n; // uzyj __uint128_t jesli n > 10^9
        y = (y*y+1)%n; // uzyj __uint128_t jesli n > 10^9
        y = (y*y+1)%n; // uzyj __uint128_t jesli n > 10^9
    } while (GCD(x+n-y, n) == 1);
    return GCD(x+n-y, n);
}

typedef unsigned long long ULL;
typedef pair<ULL,ULL> UINT128;
// __int128_t mul(ULL a, ULL b) { return a * __int128_t(b); }
UINT128 mul(ULL a, ULL b) {
  ULL M = (1ULL<<32);
  ULL a1 = a>>32, a0 = a%M, b1 = b>>32, b0 = b%M;
  ULL hi = a1 * b1, lo = a0 * b0;
  a0 *= b1, b0 *= a1;
  hi += (a0>>32) + (b0>>32);
  a0<<=32, b0 <<= 32;
  if (a0 + lo < lo) ++hi; lo += a0;
  if (b0 + lo < lo) ++hi; lo += b0;
  return UINT128(hi, lo);
}
```

```
// Dominatory w grafie skierowanym O(E log(V))
// Zalozenie: Kazdy wierzcholek jest osiagalny z korzenia
// Bartosz Walczak

const int MAXN = 100; // maksymalna liczba wierzcholkow

int n, root;           // IN: liczba wierzcholkow, korzen
bool edge[MAXN][MAXN]; // IN: macierz sasiedztwa (mozna zmienic na liste)
int dom[MAXN];         // OUT: bezposredni dominator (dom[root]==-1)
int semi[MAXN], vertex[MAXN], parent[MAXN], anc[MAXN], label[MAXN];
int head[MAXN], next[MAXN];
int cur_time;

void search(int w) {
    semi[w] = cur_time;
    vertex[cur_time] = w;
    ++cur_time;
    FOR(v,0,n) /*tu zmienic*/ if (edge[w][v] && semi[v]==-1)
      { parent[v] = w; search(v); }
}

void compress(int v) {
    if (anc[anc[v]]==-1) return;
    compress(anc[v]);
    if (semi[label[anc[v]]] < semi[label[v]]) label[v] = label[anc[v]];
    anc[v] = anc[anc[v]];
}

int eval(int v) {
    if (anc[v]==-1) return v;
    compress(v);
    return label[v];
}

void compute_dominators() {
    FOR(v,0,n) { semi[v]=anc[v]=head[v]=-1; label[v]=v; }
    cur_time = 0;
    search(root);
    FORD(i,n,1) {
        int w = vertex[i];
        FOR(v,0,n) /*tu zmienic*/ if (edge[v][w])
          semi[w] = min(semi[w], semi[eval(v)]);
        next[w] = head[vertex[semi[w]]];
        head[vertex[semi[w]]] = w;
        anc[w] = parent[w];
        while (head[parent[w]]!=-1) {
            int v = head[parent[w]];
            head[parent[w]] = next[v];
            int u = eval(v);
            dom[v] = semi[u]<semi[v] ? u : parent[w];
        }
    }
    FOR(i,1,n) {
        int w = vertex[i];
        if (dom[w]!=vertex[semi[w]]) dom[w] = dom[dom[w]];
    }
    dom[root] = -1;
}
```

```
/* SIMPLEX
   Minimalizuje CX przy ograniczeniach AX=B, X>=0
   Zalozenie: B>=0
   Uwaga: Ograniczenia w postaci nierownosci mozna zamienic na rownania
          przez wprowadzenie dodatkowych zmiennych
*/
// Bartosz Walczak

const int MAXM = 100, MAXN = 100; // maksymalna liczba ograniczen i zmiennych

typedef double K;
const K EPS = 1e-9;

int m, n;                 // IN: liczba ograniczen i liczba zmiennych
K A[MAXM+1][MAXN+MAXM]; // IN: A[m x n] Uwaga: A i B zmieniaja sie!
K B[MAXM+1];            // IN: B[m]     Dodatkowe pola tablic A i B nie maja
K C[MAXN];              // IN: C[n]     znaczenia na wejsciu, ale sa uzywane
K X[MAXN];              // OUT: X[n]    w algorytmie
int base[MAXM];

bool optimize(int last) {
    for (;;) {
        int curj=last;
        FOR(j,0,last) if (A[m][j]<=-EPS) { curj=j; break; }
        if (curj==last) return true; // znaleziono rozwiazanie
        int curi;
        K tmp=INFINITY;
        FOR(i,0,m) {
            if (base[i]>=last && fabs(A[i][curj])>=EPS)
              { curi=i; tmp=0.0; break; }
            else if (A[i][curj]>=EPS) {
                K r=B[i]/A[i][curj];
                if (r<tmp) { curi=i; tmp=r; }
            }
        }
        if (tmp==INFINITY) return false; // rozwiazanie nieograniczone
        base[curi]=curj;
        tmp=A[curi][curj];
        FOR(j,0,last) A[curi][j]/=tmp;
        B[curi]/=tmp;
        FOR(i,0,m+1) if (i!=curi) {
            tmp=A[i][curj];
            FOR(j,0,last) A[i][j]-=A[curi][j]*tmp;
            B[i]-=B[curi]*tmp;
        }
    }
}

K simplex() { // zwraca optymalna wartosc
    FOR(i,0,m+1) { fill_n(A[i]+n, m, 0.0); A[i][n+i]=1.0; base[i]=n+i; }
    fill_n(A[m], n+m, 0.0);
    FOR(i,0,m) FOR(j,0,n) A[m][j]-=A[i][j];
    optimize(m+n); // powinno zawsze zwrocic true
    FOR(i,0,m) if (base[i]>=n && B[i]>=EPS) return INFINITY; // brak rozwiazan
    copy(C, C+n, A[m]);
    FOR(i,0,m) if (base[i]<n)
      { K tmp=A[m][base[i]]; FOR(j,0,n) A[m][j]-=A[i][j]*tmp; }
    if (!optimize(n)) return -INFINITY; // rozwiazanie nieograniczone
    fill_n(X, n, 0.0);
    K res=0.0;
    FOR(i,0,m) if (base[i]<n) { X[base[i]]=B[i]; res+=C[base[i]]*B[i]; }
    return res;
}
```

```
/* ARYTMETYKA DUZYCH LICZB I FFT
   Funkcje operuja na przedzialach [b,e) w tablicy intow, gdzie b to najmniej
   znaczaca cyfa. Wynik zapisywany w przedziale (r,R), gdzie r to zwykle
   ostatni argument funkcji, R to wynik funkcji.
*/

// Arkadiusz Pawlik, Bartosz Walczak

const int BASE = 1000000000; // podstawa systemu pozycyjnego: FFT -> 100000
const int MAXSIZE = 100000;  // maksymalny rozmiar liczby

// Usuwanie wiodacych zer
inline int *strip(int *b, int *e) {
    while (e>b && !e[-1]) --e;
    return e;
}

int *input(int *r) { // UWAGA: funkcje i/o zaleza od bazy!
    static char buf[MAXSIZE*10];
    scanf("%s", buf);
    int len = strlen(buf), *r0=r;
    while (len) {
        int st = max(0, len-9/*liczba cyfr*/);
        int a=0;
        FOR(i,st,len) a = a*10+(buf[i]-'0');
        *r++ = a;
        len = st;
    }
    return strip(r0, r); // potrzebne, zeby dobrze wczytywalo zero
}
void output(int *b, int *e) {
    if (b==e) { printf("0"); return; }
    printf("%d", *--e);
    while (e-->b) printf("%09d", *e); // dostosowac liczbe cyfr
}

// Dodawanie malej liczby s>=0. Moze byc r=b1
inline int *add(int *b1, int *e1, int s, int *r) {
    while (b1!=e1) {
        s += *b1++;
        int t = (s>=BASE); *r++ = s-(BASE&-t); s=t;
    }
    if (s) *r++ = s;
    return r;
}

// Dodawanie. Moze byc r=b1 lub r=b2
inline int *add(int *b1, int *e1, int *b2, int *e2, int *r) {
    int s=0;
    while (b1!=e1 && b2!=e2) {
        s += *b1++ + *b2++;
        int t = (s>=BASE); *r++ = s-(BASE&-t); s=t;
    }
    if (b1!=e1) return add(b1, e1, s, r);
    else return add(b2, e2, s, r);
}

// Odejmowanie malej liczby s>=0. Zalozenie: a>=s. Moze byc r=b1
inline int *sub(int *b1, int *e1, int s, int *r) {
    int *r0=r; s=1-s;
    while (b1!=e1) {
        s += *b1++ + (BASE-1);
        int t = (s>=BASE); *r++ = s-(BASE&-t); s=t;
    }
    return strip(r0, r);
}

/* Odejmowanie. Zalozenie: n1>=n2. Moze byc r=b1 lub r=b2
   Uwaga: n2 nie moze miec wiodacych zer                                */
inline int *sub(int *b1, int *e1, int *b2, int *e2, int *r) {
    int *r0=r, s=1;
    while (b2!=e2) {
        s += *b1++ + (BASE-1) - *b2++;
        int t = (s>=BASE); *r++ = s-(BASE&-t); s=t;
    }
    if (!s) {
        for (; !*b1; ++b1) *r++ = BASE-1;
        *r++ = *b1++ - 1;
    }
    return strip(r0, copy(b1, e1, r));
}

/* Porownanie. Uwaga: n1 i n2 nie moga miec wiodacych zer
   Wynik: <0 jesli n1<n2, >0 jesli n1>n2, 0 jesli n1==n2               */
inline int cmp(int *b1, int *e1, int *b2, int *e2) {
    if (e1-b1!=e2-b2) return (e1-b1)-(e2-b2);
    while (b1!=e1) if (*--e1 != *--e2) return *e1-*e2;
    return 0;
}

// Mnozenie przez mala liczbe O(n). Moze byc r=b1
inline int *mul(int *b1, int *e1, int v, int *r) {
    int *r0=r, s=0;
    while (b1!=e1) {
        long long tmp = s + 1LL*v**b1++;
        *r++ = (int)(tmp%BASE); s = (int)(tmp/BASE);
    }
    if (s) *r++ = s;
    return strip(r0, r);
}

// Mnozenie O(n^2), O(n) dzielen
const long long MAXS = 4000000000000000000LL; /* 4*10^18  Niepotrzebna, jesli
       MAXSIZE*BASE^2 miesci sie w long longu, wtedy mozna usunac t i kod (*) */
inline int *mul(int *b1, int *e1, int *b2, int *e2, int *r) {
    if (b1==e1 || b2==e2) return r;
    long long s=0, t=0; // t,s przechowuje wartosci do MAXSIZE*BASE^2
    for (int j=0; j<(e1-b1)+(e2-b2)-1; ++j) {
        for (int *i1 = min(b1+j, e1-1), *i2 = b2+j-(i1-b1); i1>=b1 && i2<e2;) {
            s += 1LL**i1--**i2++;
            if (s>=MAXS) { s -= MAXS; t += MAXS/BASE; } // <- (*)
        }
        *r++ = (int)(s%BASE); s/=BASE;
        s+=t; t=0; // <- (*)
    }
    while (s) { *r++ = (int)(s%BASE); s/=BASE; }
    return r;
}

// Dzielenie przez mala liczbe. W q umieszczana jest reszta
inline int *div(int *b1, int *e1, int d, int &q, int *r) {
    int *pos = e1; q=0;
    while (--pos >= b1) {
        long long tmp = 1LL*q*BASE + *pos;
        r[pos-b1] = (int)(tmp/d); q = (int)(tmp%d);
    }
    return strip(r, r+(e1-b1));
}

// Dzielenie. Uwaga: w [b1, e1) umieszczana jest reszta
int *div(int *b1, int *&e1, int *b2, int *e2, int *r) {
    static int divbuf[MAXSIZE];
    if (e1-b1<e2-b2) return r;
    int *pos = e1-(e2-b2), *last = r+((e1-b1)-(e2-b2)+1);
```

```
    do {
        int q1=0, q2=BASE, *e=divbuf;
        do {
            int q = q1+q2>>1;
            e = mul(b2, e2, q, divbuf);
            if (cmp(divbuf, e, pos, strip(pos, e1)) <= 0) q1=q;
            else q2=q;
        } while (q2-q1>1);
        r[pos-b1] = q1;
        e1 = sub(pos, e1, divbuf, mul(b2, e2, q1, divbuf), pos);
    } while (--pos>=b1);
    return strip(r, last);
}

// FFT, obliczanie splotu i szybkie mnozenie
typedef double K;

const int MAXN = 2*MAXSIZE; /* potega dwojki, o jeden wyzsza od
                              zaokraglenia nA, nB w gore do najblizszej      */
const K EPS = 1e-8;

int nA, nB; // IN: liczba wspolczynnikow wielomianow A i B
int nC;     // OUT: liczba wspolczynnikow wielomianu C
int n;      // Uwaga: zmienna pomocnicza

K A[MAXN]; // IN: wielomian A, A[0] - wyraz wolny
K B[MAXN]; // IN: wielomian B, B[0] - wyraz wolny
K C[MAXN]; // OUT: wielomian C, C[0] - wyraz wolny

complex<K> e[MAXN];
complex<K> tab1[MAXN];
complex<K> tab2[MAXN];

// Obliczanie FFT
void FFT(complex<K> tab[]) {
    FOR(i,0,n) {
        int j=0;
        for (int k=1; k<n; k<<=1, j<<=1) if (k&i) j++;
        j>>=1;
        if (i<j) swap(tab[i], tab[j]);
    }
    int step=1, n_step=0;
    while ((1<<n_step)<n) n_step++;
    for (int step=1; step<n; step<<=1) {
        --n_step;
        for (int i=0; i<n; i+=2*step) FOR(j,0,step) {
            complex<K> u=tab[i+j], v=tab[i+j+step];
            tab[i+j] = u+v*e[j<<n_step];
            tab[i+j+step] = u-v*e[j<<n_step];
        }
    }
}

// Obliczanie splotu wielomianow
void convolution() {
    n=1;
    while (n<nA || n<nB) n<<=1;
    n<<=1;
    FOR(i,0,n) {
        tab1[i] = complex<K>(0.0, 0.0);
        tab2[i] = complex<K>(0.0, 0.0);
        e[i] = complex<K>(cos(2*M_PI*i/n), -sin(2*M_PI*i/n));
    }
    FOR(i,0,nA) tab1[i] = complex<K>(A[i], 0.0);
    FOR(i,0,nB) tab2[i] = complex<K>(B[i], 0.0);
    FFT(tab1); FFT(tab2);
    K s = 1.0/n;
    FOR(i,0,n) {
        tab1[i] *= tab2[i]*s;
        e[i] = complex<K>(cos(2*M_PI*i/n), sin(2*M_PI*i/n));
    }
    FFT(tab1);
    FOR(i,0,n) C[i] = real(tab1[i]);
    FORD(i,n,0) if (abs(C[i])>EPS) { nC=i+1; break; }
}

// Mnozenie FFT O(n log(n))
int *fastmul(int *b1, int *e1, int *b2, int *e2, int *r) {
    nA=nB=0;
    for (int *p=b1; p!=e1; ++p) A[nA++] = (K)(*p);
    for (int *p=b2; p!=e2; ++p) B[nB++] = (K)(*p);
    convolution();
    long long s=0;
    FOR(i,0,nC) {
        s += (long long)(C[i]+0.5);
        r[i] = (int)(s%BASE); s/=BASE;
    }
    while (s) { r[nC++] = s%BASE; s/=BASE; }
    return strip(r, r+nC);
}

// Konwersje reprezentacji pozycyjnych O(n^2)
void mul5(char *a) {
    static char b1[1024]; // Uwaga: wpisac maksymalna dlugosc liczby
    char *a1=a, *b=b1;
    int d=0;
    while (*a) { d = d+(*a++-'0')*5; *b++ = d%10+'0'; d/=10; }
    while (d) { *b++ = d%10+'0'; d/=10; }
    *b=0;
    strcpy(a1, b1);
}
void d2b(char *a, char *b) { // dziesietne na binarne (w stringu)
    char *b1=b;
    reverse(a, a+strlen(a));
    while (*a) { mul5(a); *b++ = (*a++=='5')+'0'; }
    *b=0;
    reverse(b1, b);
}
void b2d(char *a, char *b) { // binarne na dziesietne (w stringu)
    *b=0;
    while (*a) {
        char *p=b; int d = *a=='1';
        while (*p) { d += (*p-'0')*2; *p++ = d%10+'0'; d/=10; }
        while (d) { *p++ = d%10+'0'; d/=10; }
        *p=0; ++a;
    }
    reverse(b, b+strlen(b));
}
```

```
//DRZEWO SUFIKSOWE O(n)
//TABLICA SUFIKSOWA O(n)
//Maciek Wawro

const int K = 3;  // Wielkosc alfabetu ([0..K-1])

struct Node{
  Node *next[K], *s;
  int leaf,L,R; // [L,R] odpowiedni zakres w wejsciowym slowie
                // [-1,-1] dla roota
                // leaf -nr sufiksu, -1 dla wezlow wewnetrznych
  Node(int le, int l,int r):leaf(le),L(l),R(r){
    REP(i,K)next[i] = NULL;
  }
  ~Node(){
    REP(i,K)if(next[i])
      delete next[i];
  }
  int len(){return R-L+1;}
};

//IN: N - dlugosc slowa
//IN: S - slowo; S[N-1]>S[i] dla i = 0,...,N-2 !
//OUT: Korzen drzewa sufiksowego
Node* suffixTree(char* S, int N){
  Node* root = new Node(-1,-1,-1);
  root->s = root;
  Node* cur = root, *next, *temp, *added;
  int dep = 0, suf = 0, split;
  REP(i,N){
    added = NULL;
    while(suf<=i){
      while((next = cur->next[S[suf+dep]]) && next->len() + dep <= i-suf){
        cur = next; dep += next->len();
      }
      if(!next){
        cur->next[S[i]] = temp = new Node(suf,i,N);
        if(added) added->s = cur; added = NULL;
      }else if(S[split = next->L + i-suf-dep] != S[i]){
        cur->next[S[suf+dep]] = temp = new Node(-1, next->L, split - 1);
        next->L = split;
        temp->next[S[split]] = next;
        temp->next[S[i]] = new Node(suf, i, N);
        if(added) added->s = temp; added = temp;
      }else{
        if(added) added->s = cur;
        break;
      }
      if(cur != root){cur = cur->s; --dep;}
      suf++;
    }
  }
  return root;
}

const int MAXN = 1<<21;
int _count[1<<21];
void countSort(int* in, int* out, const int* key, int N, int M){
  fill_n(_count, M, 0);
  REP(i,N)++_count[key[in[i]]];
  FOR(i,1,M)_count[i] += _count[i-1];
  FORD(i,N,0)out[--_count[key[in[i]]]] = in[i];
}

int temp[MAXN], s0[MAXN], s12[MAXN],_rank[MAXN], recOut[MAXN];
const int* _s;
inline bool cmp(int u, int v){
  while(true){
    if(_s[u] != _s[v]) return _s[u]<_s[v];
    if((u%3) && (v%3)) return _rank[u] < _rank[v];
    ++u;++v;
  }
}

/*IN: N - dlugosc
  IN: s - string
  IN: K - zakres alfabetu
  OUT: out - tablica sufiksowa
 !!ZALOZENIA: N>=2, 0<s[i]<K, s[N]=s[N+1]=s[N+2]=0!! */
void suffixArray(const int* s, int N, int* out, int K){
  int n0 = (N+2)/3, n1 = (N+1)/3, n12 = 0;
  REP(i,N)if(i%3)temp[n12++] = i;

  countSort(temp, s12, s+2, n12, K);
  countSort(s12, temp, s+1, n12, K);
  countSort(temp, s12, s,   n12, K);

  int recIn[n12+5], cnt = 2;
  REP(i,n12){
    if(i>0 && !equal(s+s12[i-1], s+s12[i-1]+3, s+s12[i]))++cnt;
    recIn[s12[i]%3==1?s12[i]/3:s12[i]/3+n1+1] = cnt;
  }

  if(cnt != n12+1){
    REP(i,3) recIn[n12+1+i] = 0;
    recIn[n1] = 1;
    suffixArray(recIn, n12+1, recOut, cnt+1);
    FOR(i,1,n12+1)s12[i-1] = recOut[i]<n1? 3*recOut[i]+1 : 3*(recOut[i]-n1)-1;
  }

  REP(i,n12)_rank[s12[i]] = i+1;
  _rank[N] = 0;

  REP(i,n0)s0[i] = 3*i;
  countSort(s0,temp,_rank+1,n0,n12+2);
  countSort(temp,s0,s,n0,K);

  _s = s;
  merge(s12, s12+n12, s0, s0+n0, out, cmp);
}

/*IN: sA - tablica sufiksowa
  IN: invSA - odwrotnosc tablicy sufiksowej
  IN: N - dlugosc
  IN: text - string; T[N]!=T[i] dla i<N!!
  OUT: lcp */
void computeLCP(const int* sA, const int* invSA, int N, int* text, int* lcp){
  int cur = 0;
  REP(i,N){
    int j = invSA[i];
    if(!j)continue;
    int k = sA[j-1];
    while(text[k+cur] == text[i+cur])cur++;
    lcp[j] = cur;
    cur = max(0,cur-1);
  }
}
```

```
// AUTOMAT SKONCZONY dla wielu wzorcow O(n SIGMA)
// Bartosz Walczak

const int SIGMA = 2;    // licznosc alfabetu
const int MAXBUF = 100; // maksymalna liczba wezlow + 1

inline int alpha(char c) { return c-'0'; } // numer znaku w alfabecie

// Algorytm zwraca nastepujaca strukture
struct node {
    node *prefix, *next[SIGMA]; // funkcja prefiksowa, funkcja przejsc
    const char *end;            // wzorzec, ktorego koncem jest dany stan
    node *accept;               // nastepnik na liscie stanow akceptujacych
/* Uwaga: Tablice next mozna zmienic na mape. Wtedy potrzebna jest funkcja:
    node *get_next(int a) {
        node *v = this;
        while (!v->next.count(a)) v = v->prefix;
        return v->next[a];
    }                                                                         */
    node *get_accept() { return end ? this : accept; }
    void evaluate(int a, queue<node*> &Q) { // mapa: zmienic funkcje tak, aby
        if (next[a]) {        // przyjmowala iterator, i wpisac tylko tresc ifa
            next[a]->prefix = prefix->next[a]; // mapa: prefix->get_next(i->FI)
            next[a]->accept = next[a]->prefix->get_accept();
            Q.push(next[a]);
        }
        else next[a] = prefix->next[a]; // mapa: pominac
    }
};

node buf[MAXBUF]; // mapa: po zakonczeniu wyczyscic wszystkie nexty
int bufc;         // przed konstrukcja automatu wyzerowac!

node *get_node() { // pobiera nowy wezel z bufora
  FOR(a,0,SIGMA) buf[bufc].next[a] = 0; // mapa: pominac
  buf[bufc].end = 0;
  buf[bufc].accept = 0;
  return buf+bufc++;
}

// Dodawanie wzorca do struktury automatu
void add_str(node *root, const char *str) { // na poczatku root = get_node()
    for (int i=0; str[i]; ++i) {
        int a = alpha(str[i]);
        if (!root->next[a]) root->next[a] = get_node(); // mapa: zmienic warunek
        root = root->next[a];                         // na !root->next.count(a)
    }
    root->end = str;
}

// Obliczenie funkcji prefiksowej i przejsc po dodaniu wszystkich wzorcow
void evaluate(node *root) {
    root->prefix = get_node();
    FOR(a,0,SIGMA) root->prefix->next[a] = root;
    queue<node*> Q; Q.push(root);
    while (!Q.empty()) {
        node *cur = Q.front(); Q.pop();
        FOR(a,0,SIGMA) cur->evaluate(a, Q); // mapa: FORE(i,cur->next)
    }                                       //         cur->evaluate(i, Q);
}

// Generowanie listy wzorcow akceptowanych w stanie v
for (node *u=v; u; u=u->accept) if (u->end) { /*znaleziono wzorzec u->end*/ }
```

```
// ALGORYTMY TEKSTOWE
// Adam Polak

// Funkcja prefikso-sufiksowa
void pref_suf(const char *w, int n, int *p) {
    p[0] = 0;
    for(int i=1;i<n;i++) {
        p[i] = p[i-1];
        while(p[i] > 0 && w[i] != w[p[i]]) p[i] = p[p[i]-1];
        if (w[i] == w[p[i]]) p[i]++;
    }
}

// Funkcja prefikso-prefiksowa
void pref_pref(const char *w, int n, int *p) {
    int g = 0;
    p[0] = 0;
    for(int i=1;i<n;i++) {
        p[i] = max(min(p[i-g], p[g]+g-i), 0);
        while(i+p[i] < n && w[p[i]] == w[i+p[i]]) p[i]++;
        if (p[i]+i > p[g]+g) g=i;
    }
}

// Promienie palindromiczne (algorytm Manachera)
// sh = 0 dla palindromow nieparzystych, 1 dla parzystych
int manacher(const char *w, int n, int *p, int sh) {
    int g = 0;
    p[0] = 1-sh;
    for(int i=1;i<n;i++) {
        if (2*g-i>=0) p[i] = max(min(p[2*g-i], p[g]+g-i),0);
        else p[i] = 0;
        while(i-p[i]-sh >= 0 && i+p[i] < n && w[i+p[i]] == w[i-p[i]-sh])
            p[i]++;
        if (p[i]+i > p[g]+g) g=i;
    }
}

// Slownik podslow bazowych (KMR)
int dbf[16][1<<16]; // OUTPUT
void kmr(const char *w, int n) {
    REP(i,n) dbf[0][i] = w[i];
    pair<pair<int,int>,int> h[n];
    int k, l;
    for(l=k=1; l<n; l<<=1, k++) {
        REP(i,n) h[i]=make_pair(
                make_pair(dbf[k-1][i], (i+l<n)?dbf[k-1][i+l]:0),i);
        sort(h,h+n);
        int count = 1;
        REP(i,n) {
            if (i>0 && h[i].first!=h[i-1].first) count++;
            dbf[k][h[i].second] = count;
        }
    }
}

// Rownowaznosc cykliczna
#define cyc(x) (x)<n?(x):(x-n)
bool cyc_equiv(char *w, char *u, int n) {
    int i=0,j=0,k;
    while(i<n && j<n) {
        for(k=0;k<n;k++) if (w[cyc(i+k)]!=u[cyc(j+k)]) break;
        if (k==n) return true;
        if (w[cyc(i+k)] < u[cyc(j+k)]) i+=k+1; else j+=k+1;
    }
    return false;
}
```

```
// GEOMETRIA - podstawowe struktury i operatory

// Bartosz Walczak

typedef double K;
const K EPS = 1e-9;

struct xy { // punkt w 2D
    K x, y;
    xy(K xi, K yi):x(xi), y(yi) {}
    xy() {}
    K norm() const { return x*x+y*y; } // kwadrat(!) normy euklidesowej
};

inline xy operator+(const xy&a, const xy&b) { return xy(a.x+b.x, a.y+b.y); }
inline xy operator-(const xy&a, const xy&b) { return xy(a.x-b.x, a.y-b.y); }
inline xy operator*(const xy&a, K f) { return xy(a.x*f, a.y*f); }
inline xy operator/(const xy&a, K f) { return xy(a.x/f, a.y/f); }
inline xy cross(const xy&a) { return xy(-a.y, a.x); } // obrot o 90 stopni
inline K operator*(const xy&a, const xy&b) { return a.x*b.x+a.y*b.y; }
inline K det(const xy&a, const xy&b) { return a.x*b.y-b.x*a.y; }
// mowi czy jak bylismy w X, jestesmy w Y i bedziemy w Z to skrecamy w lewo(right-prawo)
inline bool left(const xy& X, const xy& Y, const xy& Z) { return det(Y-X, Z-Y) > EPS; }
inline bool right(const xy& X, const xy& Y, const xy& Z) { return det(Y-X, Z-Y) < -EPS;}

struct xyz { // punkt w 3D
    K x, y, z;
    xyz(K xi, K yi, K zi):x(xi), y(yi), z(zi) {}
    xyz() {}
    K norm() const { return x*x+y*y+z*z; } // kwadrat(!) normy euklidesowej
};

xyz normal; // UWAGA! ustaw ten wektor!

inline xyz operator+(const xyz&a, const xyz&b)
  { return xyz(a.x+b.x, a.y+b.y, a.z+b.z); }
inline xyz operator-(const xyz&a, const xyz&b)
  { return xyz(a.x-b.x, a.y-b.y, a.z-b.z); }
inline xyz operator*(const xyz&a, K f) { return xyz(a.x*f, a.y*f, a.z*f); }
inline xyz operator/(const xyz&a, K f) { return xyz(a.x/f, a.y/f, a.z/f); }
// Iloczyn wektorowy. Uwaga: odwrotnie argumenty: cross(a,b)=bxa
inline xyz cross(const xyz&a, const xyz&b=normal)
  { return xyz(b.y*a.z-a.y*b.z, b.z*a.x-a.z*b.x, b.x*a.y-a.x*b.y); }
inline K operator*(const xyz&a, const xyz&b)
  { return a.x*b.x+a.y*b.y+a.z*b.z; }
inline K det(const xyz&a, const xyz&b, const xyz&c=normal)
  { return cross(a,b)*c; }

/* GEOMETRIA 2D. Dziala rowniez na dowolnej plaszczyznie w 3D. Wtedy normal
   jest unormowanym (tzn. |normal|=1) wektorem normalnym do plaszczyzny.
   Funkcje oznaczone (*) wymagaja przystosowania do 3D.                      */

// Bartosz Walczak

typedef xy P; // w wersji na plaszczyznie w 3D zmienic na:  typedef xyz P;

// Kat skierowany pomiedzy dwoma wektorami. Zalozenie: a,b!=0
K angle(const P&a, const P&b) { return atan2(det(a,b), a*b); }
// Obrot wektora o kat skierowany
P rot(const P&a, K phi) { return a*cos(phi)+cross(a)*sin(phi); }

/* Wzajemna orientacja 3 wektorow
   >0 - counterclockwise, <0 - clockwise, ==0 - 2 wektory sie pokrywaja       */
int orient(const P&a, const P&b, const P&c) {
    K d1=det(a,b), d2=det(b,c), d3=det(c,a);
    return (d1>=EPS)-(d1<=-EPS)+(d2>=EPS)-(d2<=-EPS)+(d3>=EPS)-(d3<=-EPS);
}

struct line { // prosta {v: n*v=c} (n - wektor normalny)
    P n; K c;
    line(const P&ni, K ci):n(ni), c(ci) {}
    line() {}
};

// Czy punkt lezy na prostej?
bool on_line(const P&a, const line &p) { return fabs(p.n*a-p.c)<EPS; }
// Prosta przechodzaca przez 2 punkty ccw. Zalozenie: a!=b
line span(const P&a, const P&b) { return line(cross(b-a), det(b,a)); }
// Symetralna odcinka. Zalozenie: a!=b
line median(const P&a, const P&b) { return line(b-a, (b-a)*(b+a)*0.5); }
// Przeciecie 2 prostych
P intersection(const line &p, const line &q) {
    K d=det(p.n,q.n);
    if (fabs(d)<EPS) throw "rownolegle";
    return cross(p.n*q.c-q.n*p.c)/d;
}
// Prosta rownolegla przechodzaca przez punkt
line parallel(const P&a, const line &p) { return line(p.n, p.n*a); }
// Prosta prostopadla przechodzaca przez punkt
line perp(const P&a, const line &p) { return line(cross(p.n), det(p.n,a)); }
// Odleglosc punktu od prostej
K dist(const P&a, const line &p) { return fabs(p.n*a-p.c)/sqrt(p.n.norm()); }

// PRZECINANIE ODCINKOW

// Bartosz Walczak

struct segment { P a, b; }; // odcinek domkniety

// Punkt p lezy na prostej zawierajacej s. Czy lezy na odcinku s?
bool on_segment(const P&p, const segment &s) // (*)
  { return min(s.a.x, s.b.x)<p.x+EPS && p.x<max(s.a.x, s.b.x)+EPS &&
           min(s.a.y, s.b.y)<p.y+EPS && p.y<max(s.a.y, s.b.y)+EPS; }
// Czy dwa odcinki maja wspolny punkt?
bool intersect(const segment &s1, const segment &s2) {
    K d1 = det(s2.b-s2.a, s1.a-s2.a), d2 = det(s2.b-s2.a, s1.b-s2.a),
      d3 = det(s1.b-s1.a, s2.a-s1.a), d4 = det(s1.b-s1.a, s2.b-s1.a);
    return (d1>=EPS && d2<=-EPS || d1<=-EPS && d2>=EPS) &&
           (d3>=EPS && d4<=-EPS || d3<=-EPS && d4>=EPS) ||
           fabs(d1)<EPS && on_segment(s1.a, s2) ||
           fabs(d2)<EPS && on_segment(s1.b, s2) ||
           fabs(d3)<EPS && on_segment(s2.a, s1) ||
           fabs(d4)<EPS && on_segment(s2.b, s1);
}

// WYPUKLA OTOCZKA 2D O(n log(n))

// Bartosz Walczak

const int MAXN = 100; // maksymalna liczba punktow

int n;          // IN: liczba punktow (zmienia sie przy wywolaniu remove)
P pts[MAXN];    // IN: tablica punktow (zmienia sie!)
int hc;         // OUT: liczba punktow na wypuklej otoczce
P *hull[MAXN]; // OUT: wskazniki na kolejne punkty na wypuklej otoczce

inline bool operator==(const xy&a, const xy&b) // (*) potrzebne do remove
  { return a.x==b.x && a.y==b.y; }
inline bool operator<(const xy&a, const xy&b) // (*)
  { return a.y<b.y || a.y==b.y && a.x<b.x; }

inline bool compare(const P&a, const P&b) {
    K d=det(a-*pts, b-a);
    return d>=EPS || fabs(d)<EPS && (a-*pts)*(b-a)>=EPS;
```

```
}

void compute_hull() {
    if (!n) { hc=0; return; } // pominac, jesli n>0
    swap(*min_element(pts, pts+n), *pts);
    n = remove(pts+1, pts+n, *pts)-pts; // pominac, jesli punkty sa rozne
    sort(pts+1, pts+n, compare);
    hull[0]=pts; hc=1;
    FOR(i,1,n) {
        while (hc>=2 && det(*hull[hc-1]-*hull[hc-2], pts[i]-*hull[hc-1])<EPS)
          --hc;
        hull[hc++]=pts+i;
    }
}

/* NAJMNIEJSZE KOLO zawierajace wszystkie punkty O(n)
   OUT: C - srodek kola, R - kwadrat(!) promienia
   Uwaga! Przed wywolaniem warto zrobic random_shuffle                        */

// Arkadiusz Pawlik

void minidisc2(const P *begin, const P *end, P &C,
               K &R, const P &p1, const P &p2) {
    R = (p1-p2).norm()*0.25; C = (p1+p2)*0.5;
    FOR(i,0,end-begin) if ((C-begin[i]).norm() > R) {
        line U = median(p2, begin[i]);
        line V = median(p1, begin[i]);
        C = intersection(U, V); // Uwaga na wyjatki!
        R = max((C-p1).norm(), max((C-p2).norm(), (C-begin[i]).norm()));
    }
}
void minidisc1(const P *begin, const P *end, P &C, K &R, const P &p1) {
    R = (p1-begin[0]).norm()*0.25; C = (p1+begin[0])*0.5;
    FOR(i,1,end-begin) if ((C-begin[i]).norm() > R)
      minidisc2(begin, begin+i, C, R, p1, begin[i]);
}
void minidisc(const P *begin, const P *end, P &C, K &R) {
    if (end-begin==0) { C=xy(0,0); R=0; }
    else if (end-begin==1) { C=*begin, R=0; }
    else {
        R = (begin[0]-begin[1]).norm()*0.25;
        C = (begin[0]+begin[1])*0.5;
        FOR(i,2,end-begin) if ((C-begin[i]).norm() > R)
          minidisc1(begin, begin+i, C, R, begin[i]);
    }
}

// GEOMETRIA OKREGOW W 2D

// Bartosz Walczak

struct circle { // okrag w 2D
    P c; K r; // srodek, promien
    circle(const P&ci, K ri=0):c(ci), r(ri) {}
    circle() {}
    K length() const { return 2*M_PI*r; } // dlugosc
    K area() const { return M_PI*r*r; } // pole kola
};

// Czy punkt lezy na okregu?
bool on_circle(const P&a, const circle &c)
  { return fabs((a-c.c).norm()-c.r*c.r)<EPS; }
// Czy kolo/punkt lezy wewnatrz lub na brzegu kola?
bool operator<(const circle&a, const circle&b)
  { return b.r+EPS>a.r && (a.c-b.c).norm()<(b.r-a.r)*(b.r-a.r)+EPS; }
// Srodek okragu opisanego na trojkacie
circle circumcircle(P a, P b, P c) {
    if ((a-b).norm() > (c-b).norm()) swap(a, c);
    if ((b-c).norm() > (a-c).norm()) swap(a, b);
    if (fabs(det(b-a, c-b))<EPS) throw "zdegenerowany";
    P v=intersection(median(a, b), median(b, c));
    return circle(v, sqrt((a-v).norm()));
}
// Przeciecie okregu i prostej. Zwraca liczbe punktow
int intersection(const circle &c, const line &p, P I[]/*OUT*/) {
    K d=p.n.norm(), a=(p.n*c.c-p.c)/d;
    P u=c.c-p.n*a; a*=a; K r=c.r*c.r/d;
    if (a>=r+EPS) return 0;
    if (a>r-EPS) { I[0]=u; return 1; }
    K h=sqrt(r-a);
    I[0]=u+cross(p.n)*h; I[1]=u-cross(p.n)*h; return 2;
}
// Przeciecie dwoch okregow. Zwraca liczbe punktow. Zalozenie: c1.c!=c2.c
int intersection(const circle &c1, const circle &c2, P I[]/*OUT*/) {
    K d=(c2.c-c1.c).norm(), r1=c1.r*c1.r/d, r2=c2.r*c2.r/d;
    P u=c1.c*((r2-r1+1)*0.5)+c2.c*((r1-r2+1)*0.5);
    if (r1>r2) swap(r1,r2);
    K a=(r1-r2+1)*0.5; a*=a;
    if (a>=r1+EPS) return 0;
    if (a>r1-EPS) { I[0]=u; return 1; }
    P v=cross(c2.c-c1.c); K h=sqrt(r1-a);
    I[0]=u+v*h; I[1]=u-v*h; return 2;
}

// GEOMETRIA 3D

// Bartosz Walczak

// Kat pomiedzy dwoma wektorami. Zawsze >=0. Zalozenie: a,b!=0
K angle3(const xyz &a, const xyz &b)
  { return atan2(sqrt(cross(b,a).norm()), a*b); }

struct plane { // plaszczyzna {v: n*v=c} (n - wektor normalny)
    xyz n; K c;
    plane(const xyz &ni, K ci):n(ni), c(ci) {}
    plane() {}
};

// Czy punkt lezy na plaszczyznie?
bool on_plane(const xyz &a, const plane &p) { return fabs(p.n*a-p.c)<EPS; }
// Plaszczyzna rozpieta przez 3 punkty ccw. Zalozenie: a,b,c niezalezne
plane span3(const xyz &a, const xyz &b, const xyz &c)
  { xyz n=cross(c-a,b-a); return plane(n, n*a); }
// Plaszczyzna symetralna odcinka. Zalozenie: a!=b
plane median3(const xyz &a, const xyz &b)
  { return plane(b-a, (b-a)*(b+a)*0.5); }
// Plaszczyzna rownolegla przechodzaca przez punkt
plane parallel3(const xyz &a, const plane &p) { return plane(p.n, p.n*a); }
// Odleglosc punktu od plaszczyzny
K dist3(const xyz &a, const plane &p)
  { return fabs(p.n*a-p.c)/sqrt(p.n.norm()); }

struct line3 { // prosta {v: cross(v,u)=w} (u - wektor kierunku)
    xyz u, w;
    // UWAGA! konstruktor dwuargumentowy nie tworzy prostej przechodzacej
    // przez 2 punkty, w tym celu nalezy uzyc span3!
    line3(const xyz &ui, const xyz &wi):u(ui), w(wi) {}
    line3() {}
};

// Czy punkt lezy na prostej?
bool on_line3(const xyz &a, const line3 &p)
  { return (cross(a,p.u)-p.w).norm()<EPS; }
// Prosta rozpieta przez 2 punkty. Zalozenie: a!=b
```

```
line3 span3(const xyz &a, const xyz &b)
  { return line3(b-a, cross(a,b-a)); }
// Plaszczyzna rozpieta przez prosta i punkt ccw. Zalozenie: cross(a,p.u)!=p.w
plane span3(const line3 &p, const xyz &a)
  { return plane(cross(a,p.u)-p.w, p.w*a); }
// Prosta przeciecia dwoch plaszczyzn
line3 intersection3(const plane &p, const plane &q) {
    xyz u=cross(q.n,p.n);
    if (u.norm()<EPS) throw "rownolegle";
    return line3(u, q.n*p.c-p.n*q.c);
}
// Punkt przeciecia plaszczyzny i prostej
xyz intersection3(const plane &p, const line3 &q) {
    K d=q.u*p.n;
    if (fabs(d)<EPS) throw "rownolegle";
    return (q.u*p.c+cross(p.n,q.w))/d;
}
// Prosta prostopadla do plaszczyzny przechodzaca przez punkt
line3 perp3(const xyz &a, const plane &p) { return line3(p.n, cross(a,p.n)); }
// Plaszczyzna prostopadla do prostej przechodzaca przez punkt
plane perp3(const xyz &a, const line3 &p) { return plane(p.u, p.u*a); }
// Odleglosc punktu od prostej
K dist3(const xyz &a, const line3 &p)
  { return sqrt((cross(a,p.u)-p.w).norm())/sqrt(p.u.norm()); }
// Odleglosc 2 prostych od siebie. Zalozenie: cross(q.u,p.u)!=0 (niestabilne
K dist3(const line3 &p, const line3 &q)    // przy bliskim 0)
  { return fabs(p.u*q.w+q.u*p.w)/sqrt(cross(q.u,p.u).norm()); }

// GEOMETRIA SFER W 3D

// Bartosz Walczak

struct sphere {
    xyz c; K r; // srodek, promien
    sphere(const xyz &ci, K ri=0):c(ci), r(ri) {}
    sphere() {}
    K area() const { return 4*M_PI*r*r; } // pole powierzchni
    K volume() const { return 4*M_PI*r*r*r/3; } // objetosc kuli
};

// Czy punkt lezy na sferze?
bool on_sphere(const xyz &a, const sphere &s)
  { return fabs((a-s.c).norm()-s.r*s.r)<EPS; }
// Czy sfera/punkt lezy wewnatrz lub na brzegu kuli?
bool in_sphere(const sphere &a, const sphere &b)
  { return b.r+EPS>a.r && (a.c-b.c).norm()<(b.r-a.r)*(b.r-a.r)+EPS; }
// Przeciecie sfery i prostej. Zwraca liczbe punktow przeciecia
int intersection3(const sphere &s, const line3 &p, xyz I[]/*OUT*/) {
    K d=p.u.norm(), a=(cross(s.c,p.u)-p.w).norm()/(d*d), r=s.r*s.r/d;
    if (a>=r+EPS) return 0;
    xyz u=(p.u*(p.u*s.c)+cross(p.u,p.w))/d;
    if (a>r-EPS) { I[0]=u; return 1; }
    K h=sqrt(r-a);
    I[0]=u+p.u*h; I[1]=u-p.u*h; return 2;
}
/* Przeciecie sfery i plaszczyzny. Zwraca true, jesli sie przecinaja. Wtedy u,r
   sa odp. srodkiem i promieniem okregu przeciecia. Zalozenie: s1.c!=s2.c     */
bool intersection3(const sphere &s, const plane &p, xyz &u, K &r) {
    K d=p.n.norm(), a=(p.n*s.c-p.c)/d;
    u=s.c-p.n*a; a*=a; K r1=s.r*s.r/d;
    if (a>=r1+EPS) return false;
    r=a>r1-EPS ? 0 : sqrt(r1-a)*sqrt(d); return true;
}
/* Przeciecie dwoch sfer. Zwraca true, jesli sie przecinaja. Wtedy u,r sa
   odp. srodkiem i promieniem okregu przeciecia. Zalozenie: s1.c!=s2.c        */
bool intersection3(const sphere &s1, const sphere &s2, xyz &u, K &r) {
    K d=(s2.c-s1.c).norm(), r1=s1.r*s1.r/d, r2=s2.r*s2.r/d;
    u=s1.c*((r2-r1+1)*0.5)+s2.c*((r1-r2+1)*0.5);
    if (r1>r2) swap(r1,r2);
    K a=(r1-r2+1)*0.5; a*=a;
    if (a>=r1+EPS) return false;
    r=a>r1-EPS ? 0 : sqrt(r1-a)*sqrt(d); return true;
}

// GEOMETRIA NA SFERZE

// Bartosz Walczak

// Odleglosc dwoch punktow na sferze
K distS(const xyz &a, const xyz &b)
  { return atan2(sqrt(cross(b,a).norm()), a*b); }

struct circleS { // okrag na sferze
    xyz c; K r; // srodek, promien katowy
    circleS(const xyz &ci, K ri):c(ci), r(ri) {}
    circleS() {}
    K area() const { return 2*M_PI*(1-cos(r)); } // pole kola
};

// Okrag rozpiety przez 3 punkty. Zalozenie: punkty sa parami rozne
circleS spanS(xyz a, xyz b, xyz c) {
    int tmp=1;
    if ((a-b).norm() > (c-b).norm()) { swap(a, c); tmp=-tmp; }
    if ((b-c).norm() > (a-c).norm()) { swap(a, b); tmp=-tmp; }
    xyz v=cross(c-b,b-a); v=v*(tmp/sqrt(v.norm()));
    return circleS(v, distS(a,v));
}
// Przeciecie 2 okregow na sferze. Zalozenie: cross(c2.c,c1.c)!=0
int intersectionS(const circleS &c1, const circleS &c2, xyz I[]/*OUT*/) {
    xyz n=cross(c2.c,c1.c), w=c2.c*cos(c1.r)-c1.c*cos(c2.r);
    K d=n.norm(), a=w.norm()/d;
    if (a>=1+EPS) return 0;
    xyz u=cross(n,w)/d;
    if (a>1-EPS) { I[0]=u; return 1; }
    K h=sqrt(1-a)/sqrt(d);
    I[0]=u+n*h; I[1]=u-n*h; return 2;
}

// Porzadek katowy na punktach gdzie "root" to punkt wokol ktorego krecimy. Zaczynamy
// od punktow lezacych bezposrednio na prawo od "root", potem idziemy przeciwnie do
// ruchu wskazowek zegara, najpierw krotsze wektory. Grzegorz Guspiel

// const K EPS = 0; // jezeli K jest typem calkowitoliczbowym

struct AngleCmp {
    xy root;
    AngleCmp(const xy& root_ = xy(0, 0)): root(root_) {}
    void makeSure(const xy& a) const {
        assert(a.x > EPS || a.x < -EPS || a.y > EPS || a.y < -EPS);
    }
    int hp(const xy& a) const {
        return a.y < -EPS || (a.y <= EPS && a.x < -EPS);
    }
    bool operator()(xy a, xy b) const {
        a = a - root; makeSure(a);
        b = b - root; makeSure(b);
        int cmpHP = hp(a) - hp(b);
        if (cmpHP) return cmpHP < 0;
        K d = det(a, b);
        if (d > EPS || d < -EPS) return d > 0;
        return a.norm() < b.norm() - EPS;
    }
};
```

```cpp
// Przesuniecie cykliczne poprzedniego porzadku gdzie "root" to poczatek a "first" to
// koniec wektora, ktory jest elementem najmniejszym porzadku.

struct ShiftedAngleCmp {
    AngleCmp cmp;
    xy first;
    ShiftedAngleCmp(const xy& root_, const xy& first_):
        cmp(AngleCmp(root_)), first(first_) {}
    bool operator()(const xy& a, const xy& b) const {
        int cmpFirst = int(cmp(a, first)) - cmp(b, first);
        if (cmpFirst) return cmpFirst < 0;
        return cmp(a, b);
    }
};
```

```cpp
// WYPUKLA OTOCZKA 2D ONLINE O(log n) / query
// + SPRAWDZANIE NAJDALEJ WYSUNIETEGO PUNKTU O(log^2 |H|)
// Maciek Wawro

typedef int T; const T EPS = 0;
//typedef double T; const T EPS = 1e-9;
typedef pair<T,T> Point;

inline bool ccw(Point a, Point b, Point c){
    return (a.st - b.st) * (b.nd-c.nd) - (a.nd-b.nd) * (b.st-c.st) > EPS;
};

template <typename Set, typename It>
bool checkAndRemove(Set& hull, It it) {
    It next = it, prev = it;
    if (it == hull.begin() || (++next) == hull.end()) return false;
    --prev;
    if (ccw(*prev,*it,*next)) return false;
    hull.erase(it);
    return true;
}

template <typename Set>
void insert(Set& hull, Point a) {
    typedef typename Set::iterator It;
    It it = hull.insert(a).first, prev, next;
    if (checkAndRemove(hull, it)) return;
    while (it != hull.begin() && checkAndRemove(hull, --(prev=it)));
    while (++(next=it) != hull.end() && checkAndRemove(hull, next));
}

set<Point, greater<Point> > upperHull; // Gorna (+ prawa) otoczka od prawej do lewej
set<Point> lowerHull; // Dolna (+lewa) otoczka od lewej do prawej
void insert(Point a) {
    insert(upperHull, a);
    insert(lowerHull, a);
}

// Zeby uzyc maximizeDot, zastapic deklaracje upperHull, lowerHull przez:

inline T dot(Point a, Point b) {
    return a.first * b.first + a.second * b.second;
}

bool bitonic = false;

struct UComparator { bool operator()(Point a, Point b); };
set<Point, UComparator> upperHull;

bool UComparator :: operator()(Point a, Point b) {
    if (!bitonic) return a > b;
    bitonic = false;
    set<Point>::iterator it = ++upperHull.find(a);
    bitonic = true;
    return it != upperHull.end() && dot(*it, b) >= dot(a, b);
}

struct LComparator { bool operator()(Point a, Point b); };
set<Point, LComparator > lowerHull;

bool LComparator :: operator()(Point a, Point b) {
    if (!bitonic) return a < b;
    bitonic = false;
    set<Point>::iterator it = ++lowerHull.find(a);
    bitonic = true;
    return it != lowerHull.end() && dot(*it, b) >= dot(a, b);
}
```

```
// Dziala w czasie O(log^2 |hull|)
Point maximizeDot(Point v) {
    bitonic = true;
    Point result = v.second >= 0 ? *upperHull.lower_bound(v)
                                 : *lowerHull.lower_bound(v);
    bitonic = false;
    return result;
}

// WYPUKLA OTOCZKA 2D
// Maciek Wawro

typedef int T; const T EPS = 0;
//typedef double T; const T EPS = 1e-9;
typedef pair<T,T> Point;
//class Point:public pair<T,T>{public: int id;};

const int MAXN = 100000;

int N;              //IN: Liczba punktow
Point points[MAXN]; //IN: Punkty - psute! (sort)
int H;              //OUT:Wielkosc otoczki
Point hull[MAXN+5]; //OUT:Kolejne punkty w kolejnosci CCW

inline bool ccw(Point& a, Point& b, Point& c){
  T A[] = {a.st-b.st,a.nd-b.nd,b.st-c.st,b.nd-c.nd};
  T det = A[0]*A[3] - A[1]*A[2];  //Dla T=int -> uwzglednic rozmiar (LL)!
  T cro = A[0]*A[2] + A[1]*A[3];  //Dla T=int -> uwzglednic rozmiar (LL)!
  return (det > EPS) || ((det>=-EPS) && (cro < -EPS));
}; //Zamienic na det>=-EPS dla "slabej" wypluczki (uwaga na wszystkie wspolliniowe!)

inline void addPoint(int i){
  while((H > 1) && (!ccw(hull[H-2], hull[H-1], points[i])))H--;
  hull[H++] = points[i];
}
void computeHull(){
  int d = 0;
  sort(points, points+N);
  //N = unique(points, points+N)-points;// Potrzebne TYLKO jesli N>1 i wszystkie rowne
                                      //     (mozna obsluzyc inaczej)
  H = 0;
  REP(i,N)addPoint(i);
  if(N<2)return;
  FORD(i,N-1,0)addPoint(i);
  H--;
}
```

```
/* WYPUKLA OTOCZKA 3D w dwoch wersjach: O(n^2) i O(n log(n)) randomizowana
   O(n^2) jest szybsza przy rownomiernie rozlozonych punktach
   Uwaga: Konstruuje wypukla otoczke tylko jezeli punkty nie sa koplanarne    */

// Bartosz Walczak

const int MAXN = 100; // maksymalna liczba punktow

struct tri;

struct edge { // krawedz
    tri *t; int i; // wskaznik na druga sciane, indeks krawedzi na tej scianie
    edge(tri *ti, int ii):t(ti), i(ii) {}
    edge() {}
};

struct tri { // trojkatna sciana
    xyz *v[3], normal; // wierzcholki, normalna skierowana na zewnatrz
    edge e[3];         // krawedzie numerowane przeciwlegle do wierzcholkow
/* Wersja O(n log(n)):
    list<pair<int, list<tri*>::iterator> > L;                              */
    bool mark;
    tri **self;
    void compute_normal() { normal=cross(*v[1]-*v[0], *v[2]-*v[0]); }
    bool visible(const xyz &p) const { return (p-*v[0])*normal>=EPS; }
};

inline edge *rev(edge *e) { return e->t->e+e->i; }
inline edge *next(edge *e) { return e->t->e+(e->i+1)%3; }

int n;             // IN: liczba punktow
xyz pts[MAXN];     // IN: punkty
int tc;            // OUT: liczba scian
tri *tris[3*MAXN]; // OUT: wskazniki na kolejne sciany
tri buf[3*MAXN];   // bufor zawierajacy sciany (niekoniecznie po kolei!)

tri *get_tri() {
    tris[tc]->mark=false;
    tris[tc]->self = tris+tc;
    return tris[tc++];
}
void put_tri(tri *t) {
    tris[--tc]->self = t->self;
    swap(tris[tc], *t->self);
}

/* Wersja O(n log(n)):
list<tri*> vis[MAXN];
int mark[MAXN];

void add_point(tri *t, int i) {
    if (t->visible(pts[i])) t->L.PB(MP(i, vis[i].insert(vis[i].end(), t)));
}                                                                          */

const int tri_adj[4][3] = { 1, 2, 3, 0, 3, 2, 0, 1, 3, 0, 2, 1 };
const int tri_rev[4][3] = { 0, 0, 0, 0, 2, 1, 1, 2, 1, 2, 2, 1 };

int compute_3dhull() { // zwraca wymiar
    int dim=0;
    FOR(i,1,n) if ((pts[i]-pts[0]).norm()>=EPS)
      { swap(pts[i],pts[1]); ++dim; break; }
    if (dim==0) return 0;
    FOR(i,2,n) if (cross(pts[1]-pts[0], pts[i]-pts[0]).norm()>=EPS)
      { swap(pts[i],pts[2]); ++dim; break; }
    if (dim==1) return 1;
    FOR(i,3,n) if (fabs(det(pts[1]-pts[0], pts[2]-pts[0], pts[i]-pts[0]))>=EPS)
      { swap(pts[i],pts[3]); ++dim; break; }
```

```
    if (dim==2) return 2;
    if (det(pts[1]-pts[0], pts[2]-pts[0], pts[3]-pts[0])<0) swap(pts[2],pts[3]);
    FOR(i,0,3*n) tris[i] = buf+i;
    tc=0;
    FOR(i,0,4) {
        tri *t=get_tri();
        FOR(j,0,3) {
            t->v[j]=pts+tri_adj[i][j];
            t->e[j]=edge(buf+tri_adj[i][j], tri_rev[i][j]);
        }
        t->compute_normal();
/* Wersja O(n log(n)):
        FOR(j,4,n) add_point(t, j);                                              */
    }
/* Wersja O(n log(n)):
    fill_n(mark, n, 0);
    int id=1;                                                                    */
    FOR(i,4,n) {
        edge *first, *cur;
// Wersja O(n^2):
        FOR(j,0,tc) tris[j]->mark=tris[j]->visible(pts[i]);
        FOR(j,0,tc) if (tris[j]->mark) FOR(k,0,3) if (!tris[j]->e[k].t->mark)
          { first=cur=tris[j]->e+k; goto label; }
/* Wersja O(n log(n)):
        FORE(j,vis[i]) (*j)->mark=true;
        FORE(j,vis[i]) FOR(k,0,3) if (!(*j)->e[k].t->mark)
          { first=cur=(*j)->e+k; goto label; }                                   */
        continue;
label:  int ti=tc;
        do {
            tri *t1=cur->t, *t2=get_tri(); int i1=cur->i;
            t2->v[0]=pts+i; t2->v[1]=t1->v[(i1+2)%3]; t2->v[2]=t1->v[(i1+1)%3];
            t2->compute_normal();
            t2->e[0]=*cur;
            cur=rev(cur);
/* Wersja O(n log(n)):
            FORE(j,t1->L) { add_point(t2, j->FI); mark[j->FI]=id; }
            FORE(j,cur->t->L) if (mark[j->FI]!=id) add_point(t2, j->FI);
            ++id;                                                                */
            do cur=next(cur); while (cur->t->mark);
            t1->e[i1]=edge(t2, 0);
        } while (cur!=first);
        tri *last=tris[tc-1];
        for (; ti<tc; ++ti) {
            last->e[1]=edge(tris[ti], 2);
            tris[ti]->e[2]=edge(last, 1);
            last=tris[ti];
        }
// Wersja O(n^2):
        FOR(j,0,tc) if (tris[j]->mark) put_tri(tris[j--]);
/* Wersja O(n log(n)):
        FORE(j,vis[i]) put_tri(*j);
        FORD(j,ti,tc) {
            FORE(k,tris[j]->L) vis[k->FI].erase(k->SE);
            tris[j]->L.clear();
        }                                                                        */
    }
    return 3;
}
```

```
typedef LL node_size_t; //suma wszystkich node_size() powinna sie miescic!

template<typename T>
struct SplayTree {
#define _subtree_size(x) ((x)?(x)->subtree : 0)
    struct Node{
        T val;

        Node* left_, *right_, *p_; //tych (i dalszych) pol nie dotykac bezposrednio
        Node(const T& v):
                val(v), left_(NULL), right_(NULL), p_(NULL), everted_(false){
            resize();
        }
        void set_left(Node *x){
            push();     /* EVERT */
            left_ = x; if(x) x->p_ = this; resize(); }
        void set_right(Node *x){
            push();     /* EVERT */
            right_ = x; if(x) x->p_ = this; resize();}

        //template<> node_size_t SplayTree<KTH_TEST>::Node::node_size(){ ... }
        node_size_t node_size(){return 1;}
        node_size_t subtree;

        void resize(){
            subtree = _subtree_size(left_) + _subtree_size(right_) + node_size();
        }

        bool everted_;                                 /* EVERT */
        void evert() { everted_ = !everted_; }         /* EVERT */
        void push() {                                  /* EVERT */
            if (everted_) {                            /* EVERT */
                if (left_) left_->evert();             /* EVERT */
                if (right_) right_->evert();           /* EVERT */
                std::swap(left_, right_);              /* EVERT */
                everted_ = false;                      /* EVERT */
            }                                          /* EVERT */
        }                                              /* EVERT */

        void rotate() {
            Node *parent = p_;
            this->p_ = parent->p_;
            if(parent->p_) {
                if (parent==parent->p_->left_) parent->p_->set_left(this);
                else parent->p_->set_right(this);
            }

            if (this==parent->left_) {
                parent->set_left(this->right_);
                set_right(parent);
            } else {
                parent->set_right(this->left_);
                set_left(parent);
            }
        }

        void dump_inorder_(vector<T>&out){                 /* DUMP */
            push();                                        /* DUMP, EVERT */
            if(left_) left_->dump_inorder_(out);           /* DUMP */
            out.push_back(val);                            /* DUMP */
            if(right_) right_->dump_inorder_(out);         /* DUMP */
        }                                                  /* DUMP */

        void clear_(){                                         /* CLEAR */
            if(left_) {left_->clear_(); delete left_;}         /* CLEAR */
            if(right_) {right_->clear_(); delete right_;}      /* CLEAR */
        }                                                      /* CLEAR */
```

```
    } *root;

    void dump_inorder(vector<T>&out){           /* DUMP */
        if(root) root->dump_inorder_(out);      /* DUMP */
    }                                           /* DUMP */

    SplayTree():root(NULL){}
    //UWAGA - piszac deepcopy pamietaj o pushowaniu
    SplayTree(const SplayTree<T>& rh); //XXX - konstruktor bez definicji!
    void operator = (const SplayTree<T>& rh); //XXX - j.w.
    ~SplayTree(){ clear(); }    /* CLEAR */
    void swap(SplayTree<T>& rh){ std::swap(root, rh.root); } /* SWAP */

    void clear(){           /* CLEAR */
        if(!root) return;   /* CLEAR */
        root->clear_();     /* CLEAR */
        delete root;        /* CLEAR */
        root = NULL;        /* CLEAR */
    }

    Node* splay(Node *v) { //uwaga, zmienia korzen drzewa
        while(v->p_) {
            if (v->p_->p_) v->p_->p_->push();       /* EVERT */
            v->p_->push();                          /* EVERT */
            v->push();                              /* EVERT */
            if (v->p_->p_ && ((v==v->p_->left_) == (v->p_==v->p_->p_->left_)))
                v->p_->rotate();
            v->rotate();
        }
        return root = v;
    }

    /* MULTISET */

    Node* lower_bound(const T &x){ //zmienia korzen, NIEKONIECZNIE na wynik!
        Node *v = root, *res = NULL, *prev = NULL;
        while(v){
            prev = v;
            if(v->val < x) v = v->right_;
            else {
                res = v;
                v = v->left_;
            }
        }
        if(prev) splay(prev); //XXX czy na pewno tak chcemy ?
        return res;
    }

    Node* insert(const T& x){
        Node *v = new Node(x);
        while(root){
            if(x < root->val){
                if(root->left_) root = root->left_;
                else { root->set_left(v); break; }
            } else {
                if(root->right_) root = root->right_;
                else { root->set_right(v); break; }
            }
        }
        return splay(v);
    };

    /* PATH */

    // Pierwszy wierzcholek taki ze suma rozmiarow wierzcholkow w porzadku
    // inorder do tego wierzcholka wlacznie jest > k. Jesli rozmiary sa = 1,
    // to jest to k-ty wierzcholek w porzadku inorder. (Liczac od 0)
    Node *splay_kth(node_size_t k){
        Node *v = root;
        if (_subtree_size(v) <= k) return NULL;
        for(;;) {
            if(!v) return NULL;
            v->push();      /* EVERT */
            if (_subtree_size(v->left_) <= k &&
                    _subtree_size(v->left_) + v->node_size()>k){
                return splay(v);
            }
            if (_subtree_size(v->left_) <= k) {
                k -= (v->node_size()+_subtree_size(v->left_));
                v = v->right_;
            } else v = v->left_;
        }
    }

    void reverse(){ if(root) root->evert(); }

    void append_(Node* nw){
        if(!nw) return;
        Node *v = root;
        while(v){
            v->push();  /* EVERT */
            if(v->right_) v = v->right_;
            else break;
        }

        if(v) v->set_right(nw);
        splay(nw);
    }

    void append(const T& val){ append_(new Node(val)); }

    void insert_at(node_size_t k, const T& val){ //val bedzie na ktej pozycji
        // (od zera). UWAGA - Jesli k jest duze, to val bedzie appendowane.
        // Jesli rozmiary wezlow nie sa jednostkowe, to val zostanie wsadzone
        // na najdalsza pozycje taka, ze suma elementow przed nia jest <= k
        if(!root){
            append(val);
            return;
        }
        SplayTree st = split_from(k);
        append(val);
        extend(st);
    }

    SplayTree split_from(node_size_t k){ //odrywa podsciezke o ind. [k, k+1, ...]
        SplayTree res;
        if(!splay_kth(k)) return res;

        Node *left_ = root->left_;
        root->set_left(NULL);
        root->resize();
        if(left_) left_->p_ = NULL;

        res.root = root;
        root = left_;

        return res;
    }

    void extend(SplayTree<T> &rh){ //wchlania sciezke rh (dokleja na koniec)
        assert(this != &rh);
        if(!rh.root) return;
        if(!root) root = rh.root;
        else append_(rh.root);
```

```
        rh.root = NULL;
    }
};

// Para najblizszych punktow O(n lg n)
// Adam Polak

const int MAXN = 100000;

typedef long long LL;
typedef pair<int, int> Point;
#define X first
#define Y second

bool cmpY(Point p, Point q) { return p.Y < q.Y || (p.Y == q.Y && p.X < q.X); }

inline LL dist2(Point p, Point q) { LL x = p.X-q.X, y = p.Y-q.Y; return x*x + y*y; }

int n;                  // INPUT, n >= 2
Point points[MAXN];     // INPUT
LL best_dist2;          // OUTPUT
Point best_p, best_q;   // OUTPUT

Point buf[MAXN];

void closest_pair(Point *points=points, int n=n, bool root=true) {
    if (n < 2) return;
    if (root) { sort(points, points+n, cmpY); best_dist2 = 1 + 8e18; }
    int mid = n/2; int midY = points[mid].Y;
    closest_pair(points, mid, false);
    closest_pair(points+mid, n-mid, false);
    double low = midY - sqrtl(best_dist2);
    double high = midY + sqrtl(best_dist2);
    inplace_merge(points, points+mid, points+n);
    int k = 0;
    for(int i=0; i<n; i++) if (points[i].Y > low && points[i].Y < high) {
        for(int j=max(0, k-6); j<k; j++)
            if (dist2(points[i], buf[j]) < best_dist2) {
                best_dist2 = dist2(points[i], buf[j]);
                best_p = points[i]; best_q = buf[j];
            }
        buf[k++] = points[i];
    }
}
```

```
// ZNAJDOWANIE PARY PRZECINAJACYCH SIE ODCINKOW
// NIE UWZGLEDNIA WSPOLNYCH KONCOW
// Robert Obryk

bool on_segment(const P&p, const segment &s) // (*)
  { return min(s.a.x, s.b.x)<p.x-EPS && p.x<max(s.a.x, s.b.x)-EPS &&
           min(s.a.y, s.b.y)<p.y-EPS && p.y<max(s.a.y, s.b.y)-EPS; }
// Czy dwa odcinki maja wspolny punkt?
bool intersect(const segment &s1, const segment &s2) {
    K d1 = det(s2.b-s2.a, s1.a-s2.a), d2 = det(s2.b-s2.a, s1.b-s2.a),
      d3 = det(s1.b-s1.a, s2.a-s1.a), d4 = det(s1.b-s1.a, s2.b-s1.a);
    return (d1>EPS && d2<-EPS || d1<-EPS && d2>EPS) &&
           (d3>EPS && d4<-EPS || d3<-EPS && d4>EPS) ||
           fabs(d1)<=EPS && on_segment(s1.a, s2) ||
           fabs(d2)<=EPS && on_segment(s1.b, s2) ||
           fabs(d3)<=EPS && on_segment(s2.a, s1) ||
           fabs(d4)<=EPS && on_segment(s2.b, s1);
}

struct event {
    xy pkt;
    int idx;
    bool begin;
    event(xy _pkt, int _idx, bool _begin) : pkt(_pkt), idx(_idx), begin(_begin) {}
};

bool operator<(const event&a, const event&b) {
    if (a.pkt.x != b.pkt.x)
        return a.pkt.x < b.pkt.x;
    if (a.pkt.y != b.pkt.y)
        return a.pkt.y < b.pkt.y;
    return a.begin && !b.begin;
}

struct idx_segment {
    segment s;
    int idx;
    idx_segment(segment _s, int _idx) : s(_s), idx(_idx) {}
};

bool operator<(const idx_segment& a, const idx_segment& b)
{
    if (intersect(a.s, b.s) && a.idx != b.idx)
        throw make_pair(a.idx, b.idx);
    if (a.idx == b.idx)
        return false;
    K d1 = det(b.s.b-b.s.a, a.s.a-b.s.a);
    K d2 = det(b.s.b-b.s.a, a.s.b-b.s.a);
    if (d1 <= 0 && d2 <= 0)
        return true;
    if (d1 >= 0 && d2 >= 0)
        return false;
    return !(b < a);
}

bool do_cross(vector&lt;segment&gt;& segments, pair<int, int>& which)
{
    vector<event> events;
    for(int i=0;i<segments.size();i++) {
        segment &seg = segments[i];
        if (seg.b.x < seg.a.x || (seg.b.x == seg.a.x && seg.b.y < seg.a.y))
            swap(seg.a, seg.b);
        events.push_back(event(seg.a, i, true));
        events.push_back(event(seg.b, i, false));
    }
    sort(events.begin(), events.end());
    try {
```

```
        set<idx_segment> s;
        for(vector<event>::iterator eit = events.begin(); eit != events.end(); ++eit) {
            if (!eit->begin) {
                set<idx_segment>::iterator it, jt, kt;
                it = s.find(idx_segment(segments[eit->idx], eit->idx));
                if (it != s.begin()) {
                    jt = it; jt--;
                    kt = it; kt++;
                    if (kt != s.end())
                        (void)(*kt < *jt);
                }
                s.erase(idx_segment(segments[eit->idx], eit->idx));
            } else
                s.insert(idx_segment(segments[eit->idx], eit->idx));
        }
    } catch (pair<int,int>& c) {
        which = c;
        return true;
    }
    return false;
}

// Eliminacja Gaussa O(nm^2)
// Linie oznaczone [Z2], [Zp], [Zn], [R] sa specyficzne dla
// poszczegolnych cial/pierscieni.
// Adam Polak
// Jezeli jestesmy w ciele liczb rzeczywistych, przepisujemy:
// [R] oraz [R-nieosobl] jezeli wiemy ze uklad ma jednoznaczne rozwiazanie,
// [R] oraz [R-osobl] jezeli tego nie wiemy.
const int N = 100;
const int M = 100;

typedef unsigned long long ULL; // [Z2]
const double EPS = 1e-9;        // [R-osobl]

// INPUT, jest psuty! (psuty, nie pusty!)
int A[N][M], B[N];              // [Zp], [Zn]
ULL A[N][(M+63)/64]; bool B[N]; // [Z2]
double A[N][M], B[N];           // [R]

int MOD;                        // [Zp], [Zn]

// OUTPUT
int X[M];       // [Zp], [Zn]
bool X[M];      // [Z2]
double X[M];    // [R]

/* Rozwiazuje rownanie AX = B
   Zwraca wymiar przestrzeni rozwiazan (-1 - brak rozwiazan) */
int gauss(int n, int m) {
    int dim=0, P[m]; REP(i,m) P[i]=i;
    REP(i,n) {
        int r=i,c=i;
        FOR(j,i,n) FOR(k,i,m) {
            if (A[j][k]!=0) { r=j; c=k; goto found; }            // [Zp], [Zn]
            if (fabs(A[j][k]) > EPS) { r=j; c=k; goto found; }         // [R-osobl]
            if (fabs(A[j][k]) > fabs(A[r][c])) { r=j; c=k; }           // [R-nieosobl]
            if (A[j][k/64]&(1ULL<<(k&63))) { r=j; c=k; goto found; }   // [Z2]
        }
        break;                   // [Zp], [Zn], [Z2], [R-osobl]
        found:                   // [Zp], [Zn], [Z2], [R-osobl]
        dim = i+1;
        if (r != i) {
            REP(j,m)                        // [Zp], [Zn], [R]
            REP(j,(m+63)/64)                // [Z2]
                swap(A[i][j], A[r][j]);
            swap(B[i], B[r]);
        }
        if (c != i) {
            REP(j,n) {
                swap(A[j][i], A[j][c]);                   // [Zp], [Zn], [R]
                if ((((A[j][i/64]&(1ULL<<(i&63)))>>(i&63))) !=      // [Z2]
                        ((A[j][c/64]&(1ULL<<(c&63)))>>(c&63))) {    // [Z2]
                    A[j][i/64] ^= (1ULL<<(i&63));                   // [Z2]
                    A[j][c/64] ^= (1ULL<<(c&63));                   // [Z2]
                }                                                   // [Z2]
            }
            swap(P[i], P[c]);
        }
        FOR(j,i+1,n) {
            if (A[j][i/64]&(1ULL<<(i&63))) {           // [Z2]
                REP(k,(m+63)/64) A[j][k] ^= A[i][k];    // [Z2]
                if (B[i]) B[j] ^= 1   ;                 // [Z2]
            }                                           // [Z2]
            int d = (A[j][i] * inverse(A[i][i],MOD)) % MOD;     // [Zp]
            double d = A[j][i] / A[i][i];                       // [R]
            FOR(k,i,m) A[j][k] = (A[j][k]-d*A[i][k]) /*%MOD*/;  // [Zp], [R]
            B[j] = (B[j]-d*B[i])                     /*%MOD*/;  // [Zp], [R]
            while(A[j][i] != 0) {                               // [Zn]
                int d = A[j][i] / A[i][i];                      // [Zn]
                FOR(k,i,m) {                                    // [Zn]
                    A[j][k] = (A[j][k]-d*A[i][k]) % MOD;        // [Zn]
                    swap(A[j][k], A[i][k]);                     // [Zn]
                }                                               // [Zn]
                B[j] = (B[j]-d*B[i]) % MOD;                     // [Zn]
                swap(B[i], B[j]);                               // [Zn]
            }                                                   // [Zn]
        }
    }
    FOR(i,dim,n) if (B[i]!=0) return -1;    // [Z2], [Zp], [Zn]
    FOR(i,dim,n) if (fabs(B[i]) > EPS) return -1;       // [R-osobl]
    FOR(i,dim,m) X[i] = 0;
    FORD(i,dim,0) {
        FOR(j,i+1,m) {
            B[i] = (B[i]-A[i][j]*X[j]) /*%MOD*/;    // [Zp], [Zn], [R]
            B[i] ^= (X[j] && (A[i][j/64]&(1ULL<<(j&63))));      // [Z2]
        }
        X[i] = B[i];                                            // [Z2]
        X[i] = (inverse(A[i][i], MOD) * B[i]) % MOD;            // [Zp]
        int D = GCD(A[i][i], MOD);                              // [Zn]
        if (B[i] % D != 0) return -1;                           // [Zn]
        X[i] = (inverse(A[i][i]/D, MOD/D) * (B[i]/D)) % MOD;    // [Zn]
        X[i] = B[i] / A[i][i];                                  // [R]
    }
    REP(i,m) REP(j,m) if (P[j]==i) {
        swap(P[j], P[i]);
        swap(X[j], X[i]);
        break;
    }
    return m-dim;
}
```

Całki:

$\int \sqrt{a^2-x^2}\,dx = \frac{x}{2}\sqrt{a^2-x^2}+\frac{a^2}{2}\arcsin\frac{x}{|a|} (|x| \leqslant |a|)$

$\int \sqrt{x^2+a^2}\,dx = \frac{x}{2}\sqrt{x^2+a^2}+\frac{a^2}{2}\ln\left(x+\sqrt{x^2+a^2}\right) = \frac{x}{2}\sqrt{x^2+a^2}+\frac{a^2}{2}\operatorname{arsinh}\frac{x}{|a|}$

$\int \frac{dx}{\sqrt{a^2-x^2}} = \arcsin\frac{x}{|a|} \qquad (|x|<|a|)$

$\int \frac{x\,dx}{\sqrt{a^2-x^2}} = -\sqrt{a^2-x^2} \qquad (|x|<|a|)$

$\int \frac{dx}{\sqrt{x^2+a^2}} = \ln\left(x+\sqrt{x^2+a^2}\right)$

Wzory trygonometryczne:

$\sin(x\pm y) = \sin x\cdot\cos y \pm \cos x\cdot\sin y$ $\qquad$ $\cos(x\pm y) = \cos x\cdot\cos y \mp \sin x\cdot\sin y$

$\operatorname{tg}(x\pm y) = \frac{\operatorname{tg}x \pm \operatorname{tg}y}{1\mp\operatorname{tg}x\cdot\operatorname{tg}y}$ $\qquad$ $\operatorname{ctg}(x\pm y) = \frac{\operatorname{ctg}x\cdot\operatorname{ctg}y\mp 1}{\operatorname{ctg}y\pm\operatorname{ctg}x}$

$\left|\sin\frac{1}{2}x\right| = \sqrt{\frac{1-\cos x}{2}}$ $\qquad$ $\left|\cos\frac{1}{2}x\right| = \sqrt{\frac{1+\cos x}{2}}$

$\left|\operatorname{tg}\frac{1}{2}x\right| = \sqrt{\frac{1-\cos x}{1+\cos x}}$ $\qquad$ $\left|\operatorname{ctg}\frac{1}{2}x\right| = \sqrt{\frac{1+\cos x}{1-\cos x}}$

$\operatorname{tg}\frac{1}{2}x = \frac{1-\cos x}{\sin x} = \frac{\sin x}{1+\cos x}$ $\qquad$ $\operatorname{ctg}\frac{1}{2}x = \frac{1+\cos x}{\sin x} = \frac{\sin x}{1-\cos x}$

Długość krzywej $f(x)$: $s=\int_a^b\sqrt{1+[f'(x)]^2}\,dx$

Długość krzywej $X(t),Y(t)$: $s=\int_a^b\sqrt{[X'(t)]^2+[Y'(t)]^2}\,dt$

Symbol Legendre'a ($p\in P$, $p>2$): $\left(\frac{a}{p}\right)\equiv a^{(p-1)/2}$

$\left(\frac{ab}{p}\right)=\left(\frac{a}{p}\right)\left(\frac{b}{p}\right)$ $\qquad$ $\left(\frac{2}{p}\right)=(-1)^{\frac{p^2-1}{8}}$

$\left(\frac{-1}{p}\right)=(-1)^{\frac{p-1}{2}}$ $\qquad$ $\left(\frac{q}{p}\right)\left(\frac{p}{q}\right)=(-1)^{\frac{(p-1)(q-1)}{4}}$

Symbol Newtona:

$\sum_{k=0}^n\binom{n}{k}^2=\binom{2n}{n}$ $\qquad$ $\sum_{k=1}^n k\binom{n}{k}^2=n2^{n-1}$ $\qquad$ $\sum_{k=0}^n\binom{r}{k}\binom{s}{n-k}=\binom{r+s}{m+n}$

Liczby Stirlinga I rodzaju (liczba permutacji n elementów o k cyklach):

$\left[{n+1 \atop k}\right]=n\left[{n \atop k}\right]+\left[{n \atop k-1}\right]$ $\qquad$ $\sum_{p=k}^n\left[{n \atop p}\right]\binom{p}{k}=\left[{n+1 \atop k+1}\right]$

Liczby Stirlinga II rodzaju (liczba podziałów zbioru n-elementowego na k klas)

$\left\{{n \atop k}\right\}=\frac{1}{k!}\sum_{j=0}^k(-1)^{k-j}\binom{k}{j}j^n$ $\qquad$ $\left\{{n+1 \atop k}\right\}=k\left\{{n \atop k}\right\}+\left\{{n \atop k-1}\right\}$

$\left\{{n \atop k}\right\}\equiv\binom{z}{w} \pmod 2,\quad z=n-\left\lceil\frac{k+1}{2}\right\rceil,\ w=\left\lfloor\frac{k-1}{2}\right\rfloor$

Jeśli każde 2 elementy zbioru muszą być odległe o co najmniej d:

$S^d(n,k)=S(n-d+1,k-d+1), n\geqslant k\geqslant d$

Liczby Bella (liczba podziałów zbioru n - elementowego):

$B_{n+1}=\sum_{k=0}^n\binom{n}{k}B_k$ $\qquad$ $B_{p^m+n}\equiv mB_n+B_{n+1} \pmod p$

Lemat Burnside'a: $|X/G|=\frac{1}{|G|}\sum_{g\in G}|X^g|$ gdzie G jest grupą działającą na X,

$Gx\stackrel{def}{=}\{g(x):g\in G\}$ (orbita elementu $x\in X$), X/G jest zbiorem wszystkich orbit,

$X^g\stackrel{def}{=}\{x\in X:g(x)=x\}$ (zbiór punktów stałych dla elementu $g\in G$).

Inne:

$\operatorname{perm}(A)=(-1)^n\sum_{S\subseteq\{1,\dots,n\}}(-1)^{|S|}\prod_{i=1}^n\sum_{j\in S}a_{ij}$

Jeśli $f(n)=\sum_{d|n}g(d)$, to $g(n)=\sum_{d|n}\mu(d)f\left(\frac{n}{d}\right)$

gdzie $\mu(1)=1$, $\mu(p^2\cdot a)=0$, $\mu(p_1\cdot p_2\cdot\ldots\cdot p_k)=(-1)^k$ dla p, p_i pierwszych.

Liczba Catalana (nawiasowania): $C_n=\frac{1}{n+1}\binom{2n}{n}$

Liczba nieporządków (permutacji bez p. stałych): $!n=n!\sum_{i=0}^n\frac{(-1)^i}{i!}$

Pole trójkąta na sferze: $R^2(A+B+C-\pi)$, gdzie A,B,C - kąty na sferze.

Twierdzenie tangensów: $\frac{a-b}{a+b}=\frac{\tan[\frac{1}{2}(\alpha-\beta)]}{\tan[\frac{1}{2}(\alpha+\beta)]}$.

Objętość bryły obrotowej: $\pi\int_a^b f(x)^2dx$

Powierzchnia bryły obrotowej: $2\pi\int_a^b|f(x)|\sqrt{1+(f'(x))^2}dx$

Parametryzowana (obrót wokół x): $2\pi\int_a^b|y(t)|\sqrt{(\frac{dx}{dt})^2+(\frac{dy}{dt})^2}dt$

Obrót 3D wokół osi (u_x,u_y,u_z) o kąt θ (cs = cos, sn = sin):

$$\begin{bmatrix} cs\theta+u_x^2(1-cs\theta) & u_xu_y(1-cs\theta)-u_zsn\theta & u_xu_z(1-cs\theta)+u_ysn\theta \\ u_yu_x(1-cs\theta)+u_zsn\theta & cs\theta+u_y^2(1-cs\theta) & u_yu_z(1-cs\theta)-u_xsn\theta \\ u_zu_x(1-cs\theta)-u_ysn\theta & u_zu_y(1-cs\theta)+u_xsn\theta) & cs\theta+u_z^2(1-cs\theta) \end{bmatrix}$$