3DConvexHull

```
const db eps = 1e-9;

template<class Type>
struct pt{
    Type x, y, z;
    pt() {}
    pt<Type> (Type x, Type y, Type z): x(x), y(y), z(z) {}
    pt<Type> operator- (const pt<Type> &nxt) const { return pt<Type>(x - nxt.x, y -
nxt.y, z - nxt.z); }
    Type operator* (const pt<Type> &nxt) const { return x * nxt.x + y * nxt.y + z *
nxt.z; }
    pt<Type> operator% (const pt<Type> &nxt) const { return pt<Type>(y * nxt.z - z *
nxt.y, z * nxt.x - x * nxt.z, x * nxt.y - y * nxt.x); }
    operator pt<db>() const { return pt<db>(x, y, z); }
    db len() const {
        return sqrt(x * x + y * y + z * z);
    }
    pt<db> resize(db new_len){
        if (len() < eps) return pt<db>(0, 0, 0);
        db cur_len = len();
        new_len /= cur_len;
        return pt<db>(x * new_len, y * new_len, z * new_len);
    }
};

template<class Type>
struct plane{
    pt<Type> a, b, c;
    plane() {}
    plane(pt<Type>& a, pt<Type>& b, pt<Type>& c): a(a), b(b), c(c) {}
};

template<int SZ>
struct matrix{
    int a[SZ][SZ];

    ll det(){
        if (SZ == 3){
            return (ll)a[0][0] * (ll)a[1][1] * a[2][2] +
                   (ll)a[1][0] * (ll)a[2][1] * a[0][2] +
                   (ll)a[2][0] * (ll)a[0][1] * a[1][2] -
                   (ll)a[2][0] * (ll)a[1][1] * a[0][2] -
                   (ll)a[0][0] * (ll)a[2][1] * a[1][2] -
                   (ll)a[1][0] * (ll)a[0][1] * a[2][2];
        }

        vector<int> t(SZ);
        for (int i = 0; i < SZ; i++) t[i] = i;
        ll ans = 0;
        do {
            ll now = 1;
            for (int i = 0; i < SZ; i++) now *= a[i][t[i]];
            for (int i = 0; i < SZ; i++) for (int j = 0; j < i; j++) if (t[i] < t[j])
now *= -1;
            ans += now;
        } while(next_permutation(t.begin(), t.end()));
        return ans;
    }
};

ll calcDirectedVolume(pt<ll>& a, pt<ll>& b, pt<ll>& c, pt<ll>& d){
    pt<ll> w[3] = {b - a, c - a, d - a};
    matrix<3> m;
    for (int i = 0; i < 3; i++){
```

```
        m.a[i][0] = w[i].x;
        m.a[i][1] = w[i].y;
        m.a[i][2] = w[i].z;
    }
    return m.det();
}

vector<plane<ll>> slowBuild3DConvexHull(vector<pt<ll>> &a){
    vector<plane<ll>> ans;
    for (int i = 0; i < a.size(); i++) for (int j = i + 1; j < a.size(); j++) for
(int k = j + 1; k < a.size(); k++){
        int w[3] = {0, 0, 0};
        for (int s = 0; s < a.size(); s++){
            ll val = calcDirectedVolume(a[i], a[j], a[k], a[s]);
            if (val > 0) val = 2;
            else if (val == 0) val = 1;
            else val = 0;
            w[val]++;
        }
        if ((w[0] > 0) + (int)(w[2] > 0) <= 1) ans.push_back(plane<ll>(a[i], a[j],
a[k]));
    }
    return ans;
}

bool wasEdge[1001][1001];

vector<plane<ll>> build3DConvexHull(vector<pt<ll>> &a){
    vector<tuple<int, int, int>> pl;

    for (int i = 0; i < 4; i++) for (int j = i + 1; j < 4; j++) for (int k = j + 1; k
< 4; k++){
        int last = (0 ^ 1 ^ 2 ^ 3) ^ (i ^ j ^ k);
        if (calcDirectedVolume(a[i], a[j], a[k], a[last]) > 0){
            pl.push_back(make_tuple(i, k, j));
        } else {
            pl.push_back(make_tuple(i, j, k));
        }
    }

    for (int i = 4; i < a.size(); i++){
        vector<int> rem;
        for (int j = (int)pl.size() - 1; j >= 0; j--){
            int w[3] = { get<0>(pl[j]), get<1>(pl[j]), get<2>(pl[j]) };
            if (calcDirectedVolume(a[w[0]], a[w[1]], a[w[2]], a[i]) > 0){
                rem.push_back(j);
                wasEdge[w[0]][w[1]] = 1;
                wasEdge[w[1]][w[2]] = 1;
                wasEdge[w[2]][w[0]] = 1;
            }
        }
        if (rem.size() == 0) continue;
        for (int v : rem){
            int w[3] = { get<0>(pl[v]), get<1>(pl[v]), get<2>(pl[v]) };
            for (int j = 0; j < 3; j++){
                int k = j == 2 ? 0 : (j + 1);
                if (wasEdge[w[j]][w[k]] + (int)wasEdge[w[k]][w[j]] == 1){
                    pl.push_back(make_tuple(i, w[j], w[k]));
                }
                wasEdge[w[j]][w[k]] = 0;
                wasEdge[w[k]][w[j]] = 0;
            }
            swap(pl[v], pl.back());
            pl.pop_back();
        }
    }

    vector<plane<ll>> ans;
```

```
    for (const auto &c : pl) ans.push_back(plane<ll>(a[get<0>(c)], a[get<1>(c)],
a[get<2>(c)]));
    return ans;
}
```

AndConvolution

```
const int K = 1<<17;

// u can set modular arithmetic here
void ANDConvolution(vector<int>& v){
    for (int step=K; step > 1; step /= 2){
        for (int start=0; start < K; start += step){
            for (int w=0; w < step/2; w++){
                v[start+w] += v[start + w + step / 2];
            }
        }
    }
}

void inverseANDConvolution(vector<int>& v){
    for (int step=K; step > 1; step /= 2){
        for (int start=0; start < K; start += step){
            for (int w=0; w < step/2; w++){
                v[start+w] -= v[start + w + step / 2];
            }
        }
    }
}

/* Usage Example
    ANDConvolution(f);
    ANDConvolution(g);
    for (int i = 0; i < K; i++) f[i] *= g[i];
    inverseANDConvolution(f);
    f is ur answer
*/
```

Cartesian

```
#include <bits/stdc++.h>
#define merge merg
#define ll long long

using namespace std;

//a cartesian tree is represented as just an index of the root in the global array
//0 is a fictitious vertex here, don`t forget!

struct Vertex{int l; int r; int pr; int sz; int value;};
const int INF = 1e9;
const int N = 1e5+11; //possible number of vertex here
Vertex decart[N];
int ptr=0;

int create_vertex(int value){
        decart[ptr++] = {0, 0, rand()%1000000000, 1, value};
        return ptr-1;
}

void update(int vertex){
    int L = decart[vertex].l, R = decart[vertex].r;
    decart[vertex].sz = 1+decart[L].sz+decart[R].sz;
}
```

```
pair<int, int> split(int father, int number){ //it lefts number elements within the
left node and the remainings in the right one.
    if (father <= 0) return make_pair(0, 0);
    int L = decart[father].l, R = decart[father].r;
    int l = 1+decart[L].sz;
    if (l <= number){
        pair<int, int> p = split(R, number - l);
        decart[father].r = p.first;
        p.first = father;
        update(father);
        return p;
    }
    else{
            pair<int, int> p = split(L, number);
            decart[father].l = p.second;
            p.second = father;
            update(father);
            return p;
        }
}

int merge(int first, int second){ //merges two cartesians having roots first and
second
    if (first <= 0) return second;
    if (second <= 0) return first;
    if (decart[first].pr >= decart[second].pr){
        int v = merge(decart[first].r, second);
        decart[first].r = v;
        update(first);
        return first;
    }
    else{
            int v = merge(first, decart[second].l);
            decart[second].l = v;
            update(second);
            return second;
        }
}

//DON`T FORGET THAT 0 is a fictitious vertex HERE.

void init(){
        decart[ptr++] = {-1, -1, rand()%1000000000, 0, -1}; //put a fictitious vertex
with your parameters here
}

int main()
{
        init();
}
```

DinicWithScaling

```
#define pb push_back

struct Dinic{
    struct edge{
        int to, flow, cap;
    };

    const static int N = 555; //count of vertices

    vector<edge> e;
    vector<int> g[N + 7];
    int dp[N + 7];
```

```
    int ptr[N + 7];

    void clear(){
        for (int i = 0; i < N + 7; i++) g[i].clear();
        e.clear();
    }

    void addEdge(int a, int b, int cap){
        g[a].pb(e.size());
        e.pb({b, 0, cap});
        g[b].pb(e.size());
        e.pb({a, 0, 0});
    }

    int minFlow, start, finish;

    bool bfs(){
        for (int i = 0; i < N; i++) dp[i] = -1;
        dp[start] = 0;
        vector<int> st;
        int uk = 0;
        st.pb(start);
        while(uk < st.size()){
            int v = st[uk++];
            for (int to : g[v]){
                auto ed = e[to];
                if (ed.cap - ed.flow >= minFlow && dp[ed.to] == -1){
                    dp[ed.to] = dp[v] + 1;
                    st.pb(ed.to);
                }
            }
        }
        return dp[finish] != -1;
    }

    int dfs(int v, int flow){
        if (v == finish) return flow;
        for (; ptr[v] < g[v].size(); ptr[v]++){
            int to = g[v][ptr[v]];
            edge ed = e[to];
            if (ed.cap - ed.flow >= minFlow && dp[ed.to] == dp[v] + 1){
                int add = dfs(ed.to, min(flow, ed.cap - ed.flow));
                if (add){
                    e[to].flow += add;
                    e[to ^ 1].flow -= add;
                    return add;
                }
            }
        }
        return 0;
    }

    int dinic(int start, int finish){
        Dinic::start = start;
        Dinic::finish = finish;
        int flow = 0;
        for (minFlow = (1 << 30); minFlow; minFlow >>= 1){
            while(bfs()){
                for (int i = 0; i < N; i++) ptr[i] = 0;
                while(int now = dfs(start, (int)2e9 + 7)) flow += now;
            }
        }
        return flow;
    }
} dinic;
```

DiophantineInequality

```
// returns min k : L <= (a * k) mod M <= R, or -1 if there`re no solution
// L <= R
// O(logM)
int solveDiophantineInequality(int a, int M, int L, int R) {
    a %= M;
    if (a==0){
        if (L==0) return 0;
        return -1;
    }
    int solution = (L + a - 1) / a;
    if (a * (long long)solution <= R) {
        return solution;
    }

    if (2 * a > M) {
        return solveDiophantineInequality(M - a, M, M - R, M - L);
    }

    if (M % a == 0) {
        return -1;
    }

    solution = solveDiophantineInequality(a - M % a, a, L % a, R % a);
    if (solution == -1) {
        return -1;
    }
    solution = (solution * (long long)M + L + a - 1) / a;
    return solution;
}
```

DominatorTree

```
struct DominatorTree{
    struct DSU{
        struct Vert{
            int p;
            pair<int, int> val;
        };

        vector<Vert> t;
        vector<int> ord;

        DSU(vector<int> &ord): ord(ord) { t.resize(ord.size()); for (int i = 0; i <
ord.size(); i++) t[i].p = i; }

        int get(int v){
                if (t[v].p == v) return v;
                int new_p = get(t[v].p);
                if (ord[t[v].val.first] > ord[t[t[v].p].val.first]) t[v].val =
t[t[v].p].val;
                t[v].p = new_p;
                return t[v].p;
        }

        void merge(int a, int b){
            a = get(a); b = get(b);
            if (a != b){
                t[b].p = a;
            }
        }

        int setVal(int v, pair<int, int> val){
            t[v].val = val;
        }

        pair<int, int> getVal(int v){
```

```
            get(v);
            return t[v].val;
        }
    };

    vector<vector<int> > g, gr, lg;
    vector<int> idom, sdom, was, tin;

    int timer;
    void dfs(int v){
        tin[v] = timer++;
        was[v] = 1;
        for (int to : g[v]) if (!was[to]) dfs(to);
    }

    vector<vector<int> > req;

    DominatorTree(int n, vector<pair<int, int> > &edges, int root){
        g.resize(n); gr.resize(n); lg.resize(n);
        idom.resize(n, -1); sdom.resize(n);
        was.resize(n, 0), tin.resize(n);
        req.resize(n);
        for (auto &&e : edges){
            g[e.first].push_back(e.second);
            gr[e.second].push_back(e.first);
        }
        timer = 0; dfs(root);
        vector<int> ord;
        for (int i = 0; i < n; i++) ord.push_back(i);
        sort(ord.begin(), ord.end(), [this](int w1, int w2){ return tin[w1] >
tin[w2]; });
        DSU dsu(tin);
        for (int v : ord){
            sdom[v] = v;
            for (int to : gr[v]){
                if (v == to) continue;
                int val = tin[to] < tin[v] ? to : dsu.getVal(to).first;
                if (tin[val] < tin[sdom[v]]) sdom[v] = val;
            }

            req[sdom[v]].push_back(v);
            for (auto &&r : req[v]){
                auto val = dsu.getVal(r);
                if (tin[val.first] < tin[sdom[r]]){
                    lg[val.second].push_back(r);
                } else {
                    idom[r] = sdom[r];
                }
            }

            dsu.setVal(v, make_pair(sdom[v], v));
            for (int to : g[v]){
                if (tin[to] > tin[v] && dsu.t[to].p == to){
                    dsu.merge(v, to);
                }
            }
        }

        for (int i = 0; i < n; i++) was[i] = 0;

        for (int i = 0; i < n; i++) if (!was[i] && idom[i] != -1){
            vector<int> st;
            st.push_back(i);
            was[i] = 1;
            while(st.size()){
                int v = st.back(); st.pop_back();
                idom[v] = idom[i];
                for (int to : lg[v]) if (!was[to]) was[to] = 1, st.push_back(to);
```

```
            }
        }
    }
};
```

FastTwoChinese

```
#include <bits/stdc++.h>

#define ll long long
#define db long double
#define x first
#define y second
#define mp make_pair
#define pb push_back
#define all(a) a.begin(), a.end()
#define ipair pair<int, int>

using namespace std;
const ll LINF = 1e15;

namespace twoc { //addEdge(u, v, cost)
    //init(n) - before the running
    //solve(root) - flow
    //the result is ll (probably?)
    struct Heap {
        static Heap *null;
        ll x, xadd;
        int ver, h;
        int ei;
        Heap *l, *r;

        Heap(ll xx, int vv) : x(xx), xadd(0), ver(vv), h(1), l(null), r(null) {}

        Heap(const char *) : x(0), xadd(0), ver(0), h(0), l(this), r(this) {}

        void add(ll a) {
            x += a;
            xadd += a;
        }

        void push() {
            if (l != null)l->add(xadd);
            if (r != null)r->add(xadd);
            xadd = 0;
        }
    };

    Heap *Heap::null = new Heap("wqeqw");

    Heap *merge(Heap *l, Heap *r) {
        if (l == Heap::null)return r;
        if (r == Heap::null)return l;
        l->push();
        r->push();
        if (l->x > r->x)
            swap(l, r);
        l->r = merge(l->r, r);
        if (l->l->h < l->r->h)
            swap(l->l, l->r);
        l->h = l->r->h + 1;

        return l;
    }

    Heap *pop(Heap *h) {
```

```
        h->push();
        return merge(h->l, h->r);
    }

    const int N = 666666;

    struct DSU {
        int p[N];

        void init(int nn) { iota(p, p + nn, 0); }

        int get(int x) { return p[x] == x ? x : p[x] = get(p[x]); }

        void merge(int x, int y) { p[get(y)] = get(x); }
    } dsu;

    Heap *eb[N];
    int n;
    struct Edge {
        int x, y;
        ll c;
    };
    vector<Edge> edges;
    int answer[N];

    void init(int nn) {
        n = nn;
        dsu.init(n);
        fill(eb, eb + n, Heap::null);
        edges.clear();
    }

    void addEdge(int x, int y, ll c) {
        Heap *h = new Heap(c, x);
        h->ei = edges.size();
        edges.push_back({x, y, c});
        eb[y] = merge(eb[y], h);
    }

    ll solve(int root = 0) {
        ll ans = 0;
        static int done[N], pv[N];
        memset(done, 0, sizeof(int) * n);
        done[root] = 1;
        int tt = 1;
        int cnum = 0;
        static vector<ipair > eout[N];
        for (int i = 0; i < n; ++i)eout[i].clear();
        for (int i = 0; i < n; ++i) {
            int v = dsu.get(i);
            if (done[v])
                continue;
            ++tt;
            while (true) {
                done[v] = tt;
                int nv = -1;
                while (eb[v] != Heap::null) {
                    nv = dsu.get(eb[v]->ver);
                    if (nv == v) {
                        eb[v] = pop(eb[v]);
                        continue;
                    }
                    break;
                }
                if (nv == -1)
                    return LINF;
                ans += eb[v]->x;
                eb[v]->add(-eb[v]->x);
```

```
                int ei = eb[v]->ei;
                eout[edges[ei].x].push_back({++cnum, ei});
                if (!done[nv]) {
                    pv[v] = nv;
                    v = nv;
                    continue;
                }
                if (done[nv] != tt)
                    break;
                int v1 = nv;
                while (v1 != v) {
                    eb[v] = merge(eb[v], eb[v1]);
                    dsu.merge(v, v1);
                    v1 = dsu.get(pv[v1]);
                }
            }
        }
        memset(answer, -1, sizeof(int) * n);
        answer[root] = 0;
        set<ipair > es(all(eout[root]));
        while (!es.empty()) {

            auto it = es.begin();

            int ei = it->second;

            es.erase(it);

            int nv = edges[ei].y;

            if (answer[nv] != -1)

                continue;

            answer[nv] = ei;

            return ans;

        }
    }
}
```

FFT

```
#define db long double

class cn{
public:
        db x, y;
        cn(){}
        cn(db xx, db yy): x(xx), y(yy) {}
        cn(db xx): x(xx), y(0) {}
        db real() { return x; }
        void operator /= (double f) { x /= f; y /= f; }
};

cn operator + (cn a, cn b) { return cn(a.x + b.x, a.y + b.y); }
cn operator - (cn a, cn b) { return cn(a.x - b.x, a.y - b.y); }
cn operator * (cn a, cn b) { return cn(a.x * b.x - a.y * b.y, a.x * b.y + a.y * b.x);
}

class FFT{
public:
        constexpr const static db pi = acos(-1.0);
        const static int MAX_SIZE = 1 << 21;
```

```
        //#define cn complex<db>

        int n;
        cn a[MAX_SIZE * 2 + 7], b[MAX_SIZE * 2 + 7];

        int getReverse(int a, int k){
                int ans = 0;
                for (int i = 0; i < k; i++) if ((a >> i) & 1) ans ^= (1 << (k - i -
1));
                return ans;
        }

        void fft(cn *a, int type){
                int k = -1;
                for (int i = 0; i < 25; i++) if ((n >> i) & 1){
                        k = i;
                        break;
                }
                for (int i = 0; i < n; i++){
                        int j = getReverse(i, k);
                        if (i < j) swap(a[i], a[j]);
                }
                for (int len = 2; len <= n; len *= 2){
                        cn w(cos(2 * pi / (db)len), sin(2 * pi / (db)len) * type);
                        for (int i = 0; i < n; i += len){
                                cn g = cn(1, 0);
                                for (int j = 0; j < len / 2; j++){
                                        cn x = a[i + j];
                                        cn y = a[i + j + len / 2] * g;
                                        a[i + j] = x + y;
                                        a[i + j + len / 2] = x - y;
                                        g = g * w;
                                }
                        }
                }
                if (type == -1) for (int i = 0; i < n; i++) a[i] /= n;
        }

        vector<int> mult(vector<int> &w1, vector<int> &w2){
                n = 1;
                while(n < w1.size() + w2.size()) n *= 2;
                for (int i = 0; i < w1.size(); i++) a[i] = w1[i];
                for (int i = 0; i < w2.size(); i++) b[i] = w2[i];
                for (int i = w1.size(); i < n; i++) a[i] = 0;
                for (int i = w2.size(); i < n; i++) b[i] = 0;
                fft(a, 1);
                fft(b, 1);
                for (int i = 0; i < n; i++) a[i] = a[i] * b[i];
                fft(a, -1);
                vector<int> ans(n);
                for (int i = 0; i < n; i++) ans[i] = floor((db)a[i].real()
                 + 0.5);
                while(ans.size() && ans.back() == 0) ans.pop_back();
                return ans;
        }
};
```

FlowCirculation

```
#define pb push_back

struct Dinic{
    struct edge{
        int to, flow, cap;
    };
```

```
    const static int N = 555; //count of vertices

    vector<edge> e;
    vector<int> g[N + 7];
    int dp[N + 7];
    int ptr[N + 7];

    void clear(){
        for (int i = 0; i < N + 7; i++) g[i].clear();
        e.clear();
    }

    void addEdge(int a, int b, int cap){
                g[a].pb(e.size());
                e.pb({b, 0, cap});
        g[b].pb(e.size());
        e.pb({a, 0, 0});
    }

    void addCircular(int a, int b, int l, int r) {
        addEdge(S, b, l); //S - source
        addEdge(a, T, l); //T - sink
        addEdge(a, b, r - l);
    }

    int minFlow, start, finish;

    bool bfs(){
        for (int i = 0; i < N; i++) dp[i] = -1;
        dp[start] = 0;
        vector<int> st;
        int uk = 0;
        st.pb(start);
        while(uk < st.size()){
            int v = st[uk++];
            for (int to : g[v]){
                auto ed = e[to];
                if (ed.cap - ed.flow >= minFlow && dp[ed.to] == -1){
                    dp[ed.to] = dp[v] + 1;
                    st.pb(ed.to);
                }
            }
        }
        return dp[finish] != -1;
    }

    int dfs(int v, int flow){
        if (v == finish) return flow;
        for (; ptr[v] < g[v].size(); ptr[v]++){
            int to = g[v][ptr[v]];
            edge ed = e[to];
            if (ed.cap - ed.flow >= minFlow && dp[ed.to] == dp[v] + 1){
                int add = dfs(ed.to, min(flow, ed.cap - ed.flow));
                if (add){
                    e[to].flow += add;
                    e[to ^ 1].flow -= add;
                    return add;
                }
            }
        }
        return 0;
    }

    int dinic(int start, int finish){
        Dinic::start = start;
        Dinic::finish = finish;
        int flow = 0;
        for (minFlow = (1 << 30); minFlow; minFlow >>= 1){
```

```
            while(bfs()){
                for (int i = 0; i < N; i++) ptr[i] = 0;
                while(int now = dfs(start, (int)2e9 + 7)) flow += now;
            }
        }
        return flow;
    }
} dinic;
```

FlowNetwork_Malhotra_Goldberg

```
#include <iostream>
#include <stdexcept>
#include <cassert>
#include <limits.h>
#include <optional>
#include <type_traits>
#include <vector>
#include <queue>


//Flow Network - addEdge(from, to, cap)
//Malhotra/Goldberg(network), getNetwork()

namespace NFlow{

template<typename TFlow>
class TNetwork {
private:
    struct TEdge_;

public:
    typedef unsigned int TVertex;
    typedef unsigned int TVertexNumber;
    typedef unsigned int TEdgeNum;

    class TEdgeIterator {
    friend class TNetwork;

    public:
        TFlow getFlow() const {
            return getEdge_().flow;
        }

        TFlow getCapacity() const {
            return getEdge_().capacity;
        }

        TFlow getResudialCapacity() const {
            return getCapacity() - getFlow();
        }

        TVertex getFinish() const {
            return getEdge_().finish;
        }

        void pushFlow(TFlow flow_value) {
            const auto edge_num = network_->graph_[vertex_][edge_num_];
            auto& edges_ = network_->edges_;
            if (edges_[edge_num].flow + flow_value > edges_[edge_num].capacity) {
                throw std::logic_error("Edge's flow is bigger than capacity");
            }
            edges_[edge_num].flow     += flow_value;
            edges_[edge_num ^ 1].flow -= flow_value;
        }
```

```
        TEdgeIterator& operator++() {
            if (edge_num_ < network_->graph_[vertex_].size()) {
                ++edge_num_;
            }
            return *this;
        }

        bool isEnd() const {
            return edge_num_ == network_->graph_[vertex_].size();
        }

    private:
        typedef unsigned int TEdgeNum_;

        TNetwork* network_;
        TVertex   vertex_;
        TEdgeNum_ edge_num_;

        TEdgeIterator(TNetwork* network, TVertex vertex) :
            network_(network),
            vertex_(vertex),
            edge_num_(0)
        {}

        const TEdge_& getEdge_() const {
            if (isEnd()) {
                throw std::out_of_range("Iterator out of range");
            }
            const auto edge_num = network_->graph_[vertex_][edge_num_];
            return network_->edges_[edge_num];
        }
    };

    TNetwork(TVertexNumber vertex_number, TVertex source, TVertex sink) :
        vertex_number_(vertex_number),
        source_(source),
        sink_(sink)
    {
        if (source >= vertex_number || sink   >= vertex_number) {
            throw std::out_of_range("Source or sink index is too large");
        }
        if (source == sink) {
            throw std::logic_error("Source and sink are the same");
        }
        graph_.resize(vertex_number_);
    }

    void addEdge(TVertex start, TVertex finish, TFlow capacity) {
        // add forward edge
        graph_[start].push_back(edges_.size());
        edges_.emplace_back(finish, /* flow = */ 0, capacity);
        // add backward edge
        graph_[finish].push_back(edges_.size());
        edges_.emplace_back(start,  /* flow = */ 0, /* capacity = */ 0);
    }

    TEdgeIterator getEdgeIterator(TVertex vertex) {
        return TEdgeIterator(this, vertex);
    }

    TVertexNumber getVertexNumber() const {
        return vertex_number_;
    }

    TVertex getSource() const {
        return source_;
    }
```

```
    TVertex getSink() const {
        return sink_;
    }

    TFlow getFlowValue() const {
        TFlow flow = 0;

        for (auto edge_num : graph_[source_]) {
            const auto& edge = edges_[edge_num];
            flow += edge.flow;
        }

        return flow;
    }

private:
    struct TEdge_ {
        TVertex finish;
        TFlow   flow;
        TFlow   capacity;

        TEdge_(TVertex finish, TFlow flow, TFlow capacity) :
            finish(finish),
            flow(flow),
            capacity(capacity)
        {}
    };

    std::vector< std::vector<TEdgeNum> > graph_;
    std::vector<TEdge_> edges_;
    TVertex vertex_number_;
    TVertex source_;
    TVertex sink_;
};

} // end of namespace NFlow


namespace NMalhotra {

template<typename TFlow>
class TMalhotra {
public:
    typedef NFlow::TNetwork<TFlow> TNetwork;

    TMalhotra(const TNetwork& network) :
        network_(network)
    {
        const auto vertex_number = network.getVertexNumber();
        incoming_potential_.resize(vertex_number);
        outcoming_potential_.resize(vertex_number);
        is_available_.resize(vertex_number);
        graph_.resize(vertex_number);
        reversed_graph_.resize(vertex_number);

        findMaxFlow_();
    }

    const TNetwork& getNetwork() const {
        return network_;
    }

private:
    typedef typename TNetwork::TVertex       TVertex_;
    typedef typename TNetwork::TVertexNumber TVertexNumber_;
    typedef typename TNetwork::TEdgeNum      TEdgeNum_;
    typedef typename TNetwork::TEdgeIterator TEdgeIterator_;
    typedef unsigned int                TDist_;
```

```
    typedef std::make_unsigned_t<TFlow> TPotential_;

    struct Edge_ {
        TVertex_       finish;
        TEdgeIterator_ network_edge;
        Edge_(TVertex_ finish, TEdgeIterator_ network_edge) :
            finish(finish),
            network_edge(network_edge)
        {}
    };

    TNetwork network_;
    std::vector<TPotential_>          incoming_potential_;
    std::vector<TPotential_>          outcoming_potential_;
    std::vector<bool>                 is_available_;
    std::vector< std::vector<Edge_> > graph_;
    std::vector< std::vector<Edge_> > reversed_graph_;

    TPotential_ getPotential_(TVertex_ vertex) {
        if (vertex == network_.getSource()) {
            return outcoming_potential_[vertex];
        }
        if (vertex == network_.getSink()) {
            return incoming_potential_[vertex];
        }
        return std::min(incoming_potential_[vertex], outcoming_potential_[vertex]);
    }

    TVertex_ getMinPotentialVertex_() {
        TVertex_ min_potential_vertex = network_.getSource();
        for (TVertexNumber_ vertex = 0; vertex < network_.getVertexNumber();
++vertex) {
            if (is_available_[vertex] && getPotential_(vertex) <
getPotential_(min_potential_vertex)) {
                min_potential_vertex = vertex;
            }
        }
        return min_potential_vertex;
    }

    void removeZeroPotentialVertex_(TVertex_ vertex) {
        is_available_[vertex] = false;
        for (const auto edge : graph_[vertex]) {
            incoming_potential_[edge.finish] -=
edge.network_edge.getResudialCapacity();
        }
        for (const auto edge : reversed_graph_[vertex]) {
            outcoming_potential_[edge.finish] -=
edge.network_edge.getResudialCapacity();
        }
    }

    void findMaxFlow_() {
        while(build_graph_()) {
            removeIncorrectEdges_();
            calcPotential_();
            const auto source = network_.getSource();
            const auto sink   = network_.getSink();
            while(std::min(getPotential_(source), getPotential_(sink)) > 0) {
                const auto min_potential_vertex = getMinPotentialVertex_();
                if (getPotential_(min_potential_vertex) == 0) {
                    removeZeroPotentialVertex_(min_potential_vertex);
                } else {
                    pushFlow_(min_potential_vertex);
                }
            }
        }
    }
```

```
    bool build_graph_() {
        const auto INF           = std::numeric_limits<TDist_>::max();
        const auto vertex_number = network_.getVertexNumber();
        const auto source        = network_.getSource();
        const auto sink          = network_.getSink();
        for (TVertexNumber_ vertex = 0; vertex < vertex_number; ++vertex) {
            is_available_[vertex] = false;
            graph_[vertex].clear();
            reversed_graph_[vertex].clear();
        }
        std::vector<TDist_> dist(vertex_number, INF);
        dist[source] = 0;

        std::queue<TVertex_> queue;
        queue.push(source);

        while(!queue.empty()) {
            const auto cur_vertex = queue.front();
            queue.pop();
            for (auto it = network_.getEdgeIterator(cur_vertex); !it.isEnd(); ++it) {
                const auto cur_finish = it.getFinish();
                if (it.getResudialCapacity() > 0){
                    if (dist[cur_finish] == INF) {
                        dist[cur_finish] = dist[cur_vertex] + 1;
                        queue.push(cur_finish);
                    }

                    if (dist[cur_finish] == dist[cur_vertex] + 1) {
                        graph_[cur_vertex].emplace_back(cur_finish, it);
                        reversed_graph_[cur_finish].emplace_back(cur_vertex, it);
                    }
                }
            }
        }

        if (dist[sink] == INF) {
            return false;
        }

        dist.assign(vertex_number, INF);
        dist[sink] = 0;
        queue.push(sink);

        while(!queue.empty()) {
            const auto cur_vertex = queue.front();
            queue.pop();
            for (const auto& edge : reversed_graph_[cur_vertex]) {
                if (dist[edge.finish] == INF) {
                    dist[edge.finish] = dist[cur_vertex] + 1;
                    queue.push(edge.finish);
                }
            }
        }

        for (TVertexNumber_ vertex = 0; vertex < vertex_number; ++vertex) {
            is_available_[vertex] = dist[vertex] != INF;
        }

        return true;
    }

    void removeIncorrectEdges_() {
        for (TVertexNumber_ vertex = 0; vertex < network_.getVertexNumber();
++vertex) {
            if (!is_available_[vertex]) {
                graph_[vertex].clear();
                reversed_graph_[vertex].clear();
```

```
            } else {
                TEdgeNum_ edge_num = 0;
                auto& graph = graph_[vertex];
                while(edge_num < graph.size()) {
                    if (!is_available_[graph[edge_num].finish]) {
                        std::swap(graph[edge_num], graph.back());
                        graph.pop_back();
                    } else {
                        ++edge_num;
                    }
                }

                auto& reversed_graph = reversed_graph_[vertex];
                while(edge_num < reversed_graph.size()) {
                    if (!is_available_[reversed_graph[edge_num].finish]) {
                        std::swap(reversed_graph[edge_num], reversed_graph.back());
                        reversed_graph.pop_back();
                    } else {
                        ++edge_num;
                    }
                }
            }
        }
    }

    void calcPotential_() {
        for (TVertexNumber_ vertex = 0; vertex < network_.getVertexNumber();
++vertex) {
            incoming_potential_[vertex]  = 0;
            outcoming_potential_[vertex] = 0;
            for (const auto& edge : reversed_graph_[vertex]) {
                incoming_potential_[vertex] +=
edge.network_edge.getResudialCapacity();
            }
            for (const auto& edge : graph_[vertex]) {
                outcoming_potential_[vertex] +=
edge.network_edge.getResudialCapacity();
            }
        }
    }

    void pushFlow_(TVertex_ min_potential_vertex) {
        TFlow flow_value = getPotential_(min_potential_vertex);
        std::queue< std::pair<TVertex_, TFlow> > queue;
        queue.push({min_potential_vertex, flow_value});

        while(!queue.empty()) {
            auto [cur_vertex, flow] = queue.front();
            queue.pop();
            if (cur_vertex == network_.getSink()) {
                continue;
            }
            auto& graph = graph_[cur_vertex];
            while(flow) {
                auto& cur_edge = graph.back();
                if (!is_available_[cur_edge.finish] ||
cur_edge.network_edge.getResudialCapacity() == 0) {
                    graph.pop_back();
                } else {
                    TFlow cur_flow = std::min(flow,
cur_edge.network_edge.getResudialCapacity());
                    cur_edge.network_edge.pushFlow(cur_flow);
                    outcoming_potential_[cur_vertex]     -= cur_flow;
                    incoming_potential_[cur_edge.finish] -= cur_flow;
                    flow                                 -= cur_flow;
                    queue.push({cur_edge.finish, cur_flow});
                }
            }
```

```
        }

        queue.push({min_potential_vertex, flow_value});

        while(!queue.empty()) {
            auto [cur_vertex, flow] = queue.front();
            queue.pop();
            if (cur_vertex == network_.getSource()) {
                continue;
            }
            auto& graph = reversed_graph_[cur_vertex];
            while(flow) {
                auto& cur_edge = graph.back();
                if (!is_available_[cur_edge.finish] ||
cur_edge.network_edge.getResudialCapacity() == 0) {
                    graph.pop_back();
                } else {
                    TFlow cur_flow = std::min(flow,
cur_edge.network_edge.getResudialCapacity());
                    cur_edge.network_edge.pushFlow(cur_flow);
                    incoming_potential_[cur_vertex]       -= cur_flow;
                    outcoming_potential_[cur_edge.finish] -= cur_flow;
                    flow                                  -= cur_flow;
                    queue.push({cur_edge.finish, cur_flow});
                }
            }
        }
    }
};

} // end of namespace NMalhotra


namespace NGoldberg {

template<typename TFlow>
class TGoldberg {
public:
    typedef NFlow::TNetwork<TFlow> TNetwork;

    TGoldberg(const TNetwork& network) :
        network_(network)
    {
        const auto vertex_number = network.getVertexNumber();
        height_.resize(vertex_number);
        potential_.resize(vertex_number);
        for (TVertexNumber_ vertex = 0; vertex < vertex_number; ++vertex) {
            edge_iterator_.push_back(network_.getEdgeIterator(vertex));
        }
        findMaxFlow_();
    }

    const TNetwork& getNetwork() const {
        return network_;
    }

private:
    typedef typename TNetwork::TVertex       TVertex_;
    typedef typename TNetwork::TVertexNumber TVertexNumber_;
    typedef typename TNetwork::TEdgeIterator TEdgeIterator_;
    typedef unsigned int THeight_;
    typedef std::make_unsigned_t<TFlow> TPotential_;

    TNetwork network_;
    std::vector<THeight_>       height_;
    std::vector<TPotential_>    potential_;
    std::vector<TEdgeIterator_> edge_iterator_;
    std::queue<TVertex_>        overflowed_vertexes_;
```

```
    void pushFlow(TVertex_ vertex, TEdgeIterator_ edge) {
        const TFlow flow = std::min(potential_[vertex],
(TPotential_)edge.getResudialCapacity());
        const auto source = network_.getSource();
        const auto sink   = network_.getSink();
        if (vertex != source && vertex != sink) {
            potential_[vertex] -= flow;
        }
        const auto finish = edge.getFinish();
        if (finish != source && finish != sink) {
            potential_[finish] += flow;
        }
        edge.pushFlow(flow);
    }

    void relabel(TVertex_ vertex) {
        THeight_ new_height = std::numeric_limits<THeight_>::max();
        for (auto it = network_.getEdgeIterator(vertex); !it.isEnd(); ++it) {
            if (it.getResudialCapacity() > 0) {
                new_height = std::min(new_height, height_[it.getFinish()] + 1);
            }
        }
        height_[vertex] = new_height;
    }

    void discharge(TVertex_ vertex) {
        auto& edge = edge_iterator_[vertex];
        while(potential_[vertex] > 0) {
            if (edge.isEnd()) {
                relabel(vertex);
                edge = network_.getEdgeIterator(vertex);
            } else {
                const TVertex_ finish = edge.getFinish();
                if (edge.getResudialCapacity() > 0 && height_[vertex] ==
height_[finish] + 1) {
                    const bool was_overflowed = potential_[finish] > 0;
                    pushFlow(vertex, edge);
                    if (!was_overflowed && potential_[finish] > 0) {
                        overflowed_vertexes_.push(finish);
                    }
                } else {
                    ++edge;
                }
            }
        }
    }

    void findMaxFlow_() {
        for (TVertexNumber_ vertex = 0; vertex < network_.getVertexNumber();
++vertex) {
            potential_[vertex] = 0;
            height_[vertex]    = 0;
        }

        const auto source = network_.getSource();
        height_[source] = network_.getVertexNumber();

        for (auto it = network_.getEdgeIterator(source); !it.isEnd(); ++it) {
            const auto cur_finish = it.getFinish();
            const auto sink       = network_.getSink();
            const auto flow       = it.getResudialCapacity();
            it.pushFlow(flow);
            if (cur_finish != sink) {
                potential_[cur_finish] += flow;
            }

            if (potential_[cur_finish] > 0) {
```

```
                overflowed_vertexes_.push(cur_finish);
            }
        }

        while(!overflowed_vertexes_.empty()) {
            const auto cur_vertex = overflowed_vertexes_.front();
            overflowed_vertexes_.pop();
            discharge(cur_vertex);
        }
    }
};

} // end of namespace NGoldberg


int main(){
    int n;
    std::cin >> n;
    std::vector<int> cost(n);
    for (int i = 0; i < n; i++) {
        std::cin >> cost[i];
    }

    const int INF = std::numeric_limits<int>::max();
    const unsigned int source = n;
    const unsigned int sink = n + 1;
    NFlow::TNetwork<int> network(n + 2, source, sink);

    for (int vertex = 0; vertex < n; vertex++) {
        int cnt;
        std::cin >> cnt;
        while(cnt--) {
            int parent;
            std::cin >> parent;
            parent--;
            network.addEdge(vertex, parent, INF);
        }
    }

    int result = 0;

    for (int vertex = 0; vertex < n; vertex++) {
        if (cost[vertex] > 0) {
            result += cost[vertex];
            network.addEdge(source, vertex, cost[vertex]);
        } else {
            network.addEdge(vertex, sink, -cost[vertex]);
        }
    }

    NMalhotra::TMalhotra malhotra(network);
    const auto malhotra_result_network = malhotra.getNetwork();

    NGoldberg::TGoldberg goldberg(network);
    const auto goldberg_result_network = goldberg.getNetwork();

    assert(malhotra_result_network.getFlowValue() ==
goldberg_result_network.getFlowValue());

    std::cout << result - malhotra_result_network.getFlowValue();
}
```

formulas&ideas.txt

```
Формула Кэли для кол-ва деревьев:
```

```
n^(n-2)

================================================================================
=========================

Количество деревьев с путем длины k:

(k+1) * n^(n-k-2) * (n-2)! / (n-k-2)!

================================================================================
=========================

Количество лесов из n вершин и k деревьев:

sum by i from 0 to k:

(-1/2)^i * (k+i) * i! * C(k, i) * C(n-k, i) * n^(n-k-i-1)

================================================================================
=========================

Решаем задачи на посчитать по всем объектам char(object)^k:
<=>
количество способов выбрать объект + последовательность длины k, членами которой
являются члены char
а теперь решаем для последовательности

(к примеру, посчитаь size^k для всех связных подграфов дерева, k <= 10
вместо size^k выбираем подграф + последовательность из k его вершин
теперь простая DP под поддеревьям)

================================================================================
=========================

Задача по модулю p - возможно стоит дискретно логарифмировать (т.е если x = pr_root ^
y, то заменим x на y)

тогда произведение переходит в сумму (a*b = pr_root ^ x * pr_root ^ y =
(pr_root)^(x+y))

================================================================================
=========================
```

GaussModulo

```
struct GaussModulo {
    int mult(int a, int b){
        return a * (ll)b % mod;
    }

    int pow(int val, int deg){
        if (deg == 0) return 1;
        if (deg & 1) {
            return mult(val, pow(val, deg - 1));
        } else {
            int cur_val = pow(val, deg >> 1);
            return mult(cur_val, cur_val);
        }
    }

    int get_rev(int val) {
        return pow(val, mod - 2);
    }

    enum GaussSolution {
        ZERO, ONE, MANY
```

```
    };

    int n;
    GaussSolution solutions_cnt;
    vector<int> solutions;

    GaussModulo(vector< vector<int> > &eqs) {
        n = (int)eqs.back().size() - 1;
        solutions.resize(n);

        int cur_eq = 0;
        for (int v = 0; v < n; v++) {
            int correct_eq_num = -1;
            for (int eq_num = cur_eq; eq_num < eqs.size(); eq_num++) {
                if (eqs[eq_num][v] != 0) {
                    correct_eq_num = eq_num;
                    break;
                }
            }

            if (correct_eq_num == -1) continue;

            swap(eqs[cur_eq], eqs[correct_eq_num]);

            int rev_val = get_rev(eqs[cur_eq][v]);
            for (int i = v; i < eqs[cur_eq].size(); i++) {
                eqs[cur_eq][i] = mult(eqs[cur_eq][i], rev_val);
            }

            for (int eq_num = cur_eq + 1; eq_num < eqs.size(); eq_num++) {
                int cur_val = eqs[eq_num][v];
                for (int i = v; i < eqs[eq_num].size(); i++) {
                    eqs[eq_num][i] -= mult(eqs[cur_eq][i], cur_val);
                    if (eqs[eq_num][i] < 0) eqs[eq_num][i] += mod;
                }
            }

            cur_eq++;
        }

        if (cur_eq < n) {
            solutions_cnt = MANY;
            return;
        }

        for (int i = cur_eq; i < eqs.size(); i++) {
            if (eqs[i].back() != 0) {
                solutions_cnt = ZERO;
                return;
            }
        }

        for (int v = n - 1; v >= 0; v--) {
            for (int eq_num = v - 1; eq_num >= 0; eq_num--) {
                eqs[eq_num].back() -= mult(eqs[eq_num][v], eqs[v].back());
                if (eqs[eq_num].back() < 0) eqs[eq_num].back() += mod;
                eqs[eq_num][v] = 0;
            }
        }

        solutions_cnt = ONE;

        for (int v = 0; v < n; v++) solutions[v] = eqs[v].back();
    }
};
```

GomoryHuTree

```
struct Dinic{
    struct edge{
        int to, flow, cap;
    };

    static const int N = 3003;

    vector<edge> e;
    vector<int> g[N];
    int ptr[N], dp[N];

    void clear(int n){
        e.clear();
        for (int i = 0; i < n; i++) g[i].clear();
    }

    void addEdge(int a, int b, int cap){
        g[a].pb(e.size());
        e.pb({b, 0, cap});
        g[b].pb(e.size());
        e.pb({a, 0, 0});
    }

    int minFlow, start, finish;

    bool bfs(int n){
        for (int i = 0; i < n; i++) dp[i] = -1;
        dp[start] = 0;
        vector<int> st;
        int uk = 0;
        st.pb(start);
        while(uk < st.size()){
            int v = st[uk++];
            for (int to : g[v]){
                auto ed = e[to];
                if (ed.cap - ed.flow >= minFlow && dp[ed.to] == -1){
                    dp[ed.to] = dp[v] + 1;
                    st.pb(ed.to);
                }
            }
        }
        return dp[finish] != -1;
    }

    int dfs(int v, int flow){
        if (v == finish) return flow;
        for (; ptr[v] < g[v].size(); ptr[v]++){
            int to = g[v][ptr[v]];
            edge ed = e[to];
            if (ed.cap - ed.flow >= minFlow && dp[ed.to] == dp[v] + 1){
                int add = dfs(ed.to, min(flow, ed.cap - ed.flow));
                if (add){
                    e[to].flow += add;
                    e[to ^ 1].flow -= add;
                    return add;
                }
            }
        }
        return 0;
    }

    int dinic(int start, int finish, int n){
        Dinic::start = start;
        Dinic::finish = finish;
        int flow = 0;
        for (minFlow = 1; minFlow; minFlow >>= 1){
            while(bfs(n)){
                for (int i = 0; i < n; i++) ptr[i] = 0;
```

```
                while(int now = dfs(start, (int)2e9 + 7)) flow += now;
            }
        }
        return flow;
    }
} dinic;

// Работает за n - 1 min-cut
// Передавать связный граф
// Номера вершин 0..n-1
struct GomoryHuTree{
    // еще в Динице поставить ll, если нужно
    using w_type = int;

    static const int N = 3003;

    struct Edge{
        int a, b;
        w_type w;
        Edge() = default;
        Edge(int a, int b, w_type w): a(a), b(b), w(w) {}
    };

    int color[N];
    bool was[N];
    vector< pair<int, w_type> > g[N];

    void clear(int n){
        for (int i = 0; i < n; i++) g[i].clear();
    }

    vector<Edge> build(int n, const vector<Edge>& edges){
        for (auto&& edge : edges) g[edge.a].pb({edge.b, edge.w});

        vector< vector<int> > nodes;
        vector<Edge> tree_edges;

        nodes.emplace_back(vector<int>(n));
        for (int i = 0; i < n; i++) nodes.back()[i] = i;

        while(1){
            int v = -1;
            for (int i = 0; i < nodes.size(); i++) if (nodes[i].size() > 1){
                    v = i;
                    break;
                }
            if (v == -1) break;

            split(n, edges, nodes, v, tree_edges);

            /*cout << nodes.size() << ' ' << tree_edges.size() << endl;
            for (auto& c : nodes){
                cout << "node: ";
                for (int v : c) cout << v << ' ';
                cout << endl;
            }
            for (auto&& c : tree_edges){
                cout << "edge: " << c.a << ' ' << c.b << ' ' << c.w << endl;
            }
            cout << endl;*/
        }

        vector<Edge> ans(n - 1);

        for (int i = 0; i < tree_edges.size(); i++){
            ans[i] = {nodes[tree_edges[i].a][0], nodes[tree_edges[i].b][0],
tree_edges[i].w};
        }
```

```
        return ans;
    }

    vector<int> g_comp[N];
    vector<int> comps;

    void dfs(int v, int p){
        comps.pb(v);
        for (int to : g_comp[v]) if (to != p) dfs(to, v);
    }

    void split(int n, const vector<Edge>& edges, vector< vector<int> >& nodes, int
node_num, vector<Edge>& tree_edges){
        auto& node = nodes[node_num];

        memset(was, 0, sizeof(bool) * n);

        int cc = 0;

        for (int v : node) was[v] = 1, color[v] = cc++;

        for (int i = 0; i < nodes.size(); i++) g_comp[i].clear();
        for (auto&& edge : tree_edges) g_comp[edge.a].pb(edge.b),
g_comp[edge.b].pb(edge.a);

        for (int to : g_comp[node_num]){
            comps.clear();
            dfs(to, node_num);
            for (int comp : comps) for (int v : nodes[comp]) color[v] = cc;
            cc++;
        }

        dinic.clear(cc);

        for (auto&& edge : edges) if (color[edge.a] != color[edge.b]){
                // можно в одно ребро сумму засунуть
                dinic.addEdge(color[edge.a], color[edge.b], edge.w);
                dinic.addEdge(color[edge.b], color[edge.a], edge.w);
            }

        w_type cut_size = dinic.dinic(color[node[0]], color[node[1]], cc);

        vector<int> left_node, right_node, other_left_nodes;

        memset(was, 0, sizeof(bool) * n);

        vector<int> st; st.pb(color[node[0]]); was[color[node[0]]] = 1,
left_node.pb(node[0]);
        while(st.size()){
            int now = st.back(); st.pop_back();
            for (int edge_num : dinic.g[now]) if (dinic.e[edge_num].flow !=
dinic.e[edge_num].cap){
                    int to = dinic.e[edge_num].to;
                    if (!was[to]){
                        st.pb(to);
                        was[to] = 1;
                        if (to < node.size()){
                            left_node.pb(node[to]);
                        } else {
                            other_left_nodes.pb(to);
                        }
                    }
                }
        }

        memset(was, 0, sizeof(bool) * n);
        for (int v : left_node) was[v] = 1;
```

```
        for (int v : node) if (!was[v]) right_node.pb(v);

        nodes[node_num] = std::move(left_node);
        nodes.emplace_back(std::move(right_node));

        memset(was, 0, sizeof(bool) * n);
        for (int v : other_left_nodes) was[v] = 1;

        for (auto& edge : tree_edges) if (edge.a == node_num || edge.b == node_num){
            if (edge.a != node_num) swap(edge.a, edge.b);

            if (!was[color[nodes[edge.b][0]]]){
                edge = {(int)nodes.size() - 1, edge.b, edge.w};
            }
        }

        tree_edges.emplace_back(node_num, (int)nodes.size() - 1, cut_size);
    }
};
```

LevelAncestor

```
const int N = 150007, LG = 20; //set it here
// init from list of tree edges
// get(x, y) returns y-th ancestor of x by O(1)

struct LA{
    int dv[LG][N];
    int n, m, u;
    int szlad = 0;
    vector<int> ladders[N], data[N];
    int what_ladder[N], what_number[N], logs[2*N], lengths[N];
    int fathers[N], d[N];

    void first_dfs(int vertex){
        int l = 1;
        for (int i=0; i < (int) data[vertex].size(); i++){
            int to = data[vertex][i];
            first_dfs(to);
            l = max(l, lengths[to] + 1);
        }
        lengths[vertex] = l;
    }

    void up(int vertex, int ost){
        if (vertex == 0 || ost == 0){
            ladders[szlad-1].push_back(vertex);
            return;
        }
        up(fathers[vertex], ost - 1);
        ladders[szlad-1].push_back(vertex);
    }

    void binup_dfs(int vertex, int last){
        if (last != -1){
            dv[0][vertex] = last;
            int nv = last;
            int now_level = 1;
            while (dv[now_level-1][nv] != -1){
                dv[now_level][vertex] = dv[now_level-1][nv];
                nv = dv[now_level-1][nv];
                now_level++;
            }
        }
        for (int i=0; i < (int) data[vertex].size(); i++){
            binup_dfs(data[vertex][i], vertex);
```

```
        }
    }

    void dfs(int vertex, int ladder, int depth){
        d[vertex] = depth;
        if (szlad == ladder){
            szlad++;
            up(vertex, lengths[vertex]);
            what_ladder[vertex] = szlad - 1;
            what_number[vertex] = ladders[szlad - 1].size() - 1;
        }
        else{
            ladders[ladder].push_back(vertex);
            what_ladder[vertex] = ladder;
            what_number[vertex] = ladders[ladder].size() - 1;
        }
        bool go = false;
        for (int i=0; i < (int) data[vertex].size(); i++){
            int to = data[vertex][i];
            if (go || lengths[to] + 1 != lengths[vertex]){
                dfs(to, szlad, depth + 1);
            }
            else{
                dfs(to, ladder, depth + 1);
                go = true;
            }
        }
    }

    int get(int vertex, int when){
        if (d[vertex] <= when) return 0;
        if (when == 0) return vertex;
        vertex = dv[logs[when]][vertex];
        when -= (1LL << logs[when]);
        return ladders[what_ladder[vertex]][what_number[vertex] - when];
    }

    void pre_dfs(int vertex, int last){
        if (last != -1) fathers[vertex] = last;

        int I = -1;

        for (int i=0; i < data[vertex].size(); ++i){
            int to = data[vertex][i];
            if (to==last){
                I=i;
                continue;
            }
            pre_dfs(to, vertex);
        }

        if (I!=-1){
            swap(data[vertex][I], data[vertex].back());
            data[vertex].pop_back();
        }
    }

    void init(vector<pair<int, int> > edges) {
        for (int i=0; i < edges.size(); ++i) {
            data[edges[i].first].push_back(edges[i].second);
            data[edges[i].second].push_back(edges[i].first);
        }

        pre_dfs(0, -1);
        first_dfs(0);
        int start = 0;
        for (int i=0; i < LG; i++){
            for (int j=0; j < N; j++){
```

```
                dv[i][j] = -1;
            }
        }
        for (int i=2; i <= 2*N; i*=2){
            for (int j=i/2; j < i; j++){
                logs[j] = start;
            }
            start++;
        }
        dfs(0, 0, 0);
        binup_dfs(0, -1);

    }

};
```

Manaker

```
// для четных палиндромов:
// 1) запустить со строкой вида a#b#a#c#a#b#a
// 2) взять значения в позициях решеток
vector<int> manaker(const string &s) {
    vector<int> man(s.size(), 0);
    int l = 0, r = 0;
    int n = s.size();
    for (int i = 1; i < n; i++) {
        if (i <= r) {
            man[i] = min(r - i, man[l + r - i]);
        }
        while (i + man[i] + 1 < n && i - man[i] - 1 >= 0
               && s[i + man[i] + 1] == s[i - man[i] - 1]) {
            man[i]++;
        }
        if (i + man[i] > r) {
            l = i - man[i];
            r = i + man[i];
        }
    }
    return man;
}
```

MinCostMaxFlow

```
struct MinCostMaxFlow {
    struct Edge{
        int to, cap;
        int flow;
        int cost;
    };

    static const int MAX_V = 222;
    static const int MAX_E = 4444;
    static const int INF = 1e9 + 7;

    int sz = 0;
    Edge e[MAX_E];
    vector<int> g[MAX_V];
    int fb[MAX_V];
    int was[MAX_V];
    pair<int, int> prev[MAX_V];

    void addEdge(int v, int to, int cap, int cost){
        g[v].push_back(sz);
        e[sz++] = { to, cap, 0, cost };
```

```
        g[to].push_back(sz);
        e[sz++] = { v, 0, 0, -cost };
    }

    ll find(int start, int finish, int required_flow) {
        ll ans = 0;

        while (required_flow) {
            for (int i = 0; i < MAX_V; i++) fb[i] = INF, prev[i] = { -1, -1 }, was[i]
= 0;
            fb[start] = 0;
            vector<int> st;
            int uk = 0;
            st.push_back(start);
            while (uk < st.size()) {
                int v = st[uk++];
                was[v] = 0;
                for (int to : g[v]) {
                    auto ed = e[to];
                    if (ed.flow < ed.cap && fb[ed.to] > fb[v] + ed.cost) {
                        prev[ed.to] = { v, to };
                        fb[ed.to] = fb[v] + ed.cost;
                        if (!was[ed.to]) {
                            st.push_back(ed.to);
                            was[ed.to] = 1;
                        }
                    }
                }
            }

            if (fb[finish] == INF) {
                return -1;
            }

            int max_flow = required_flow;
            int v = finish;
            while (1) {
                auto now = prev[v];
                if (now.x == -1) break;
                max_flow = min(max_flow, e[now.y].cap - e[now.y].flow);
                v = now.x;
            }
            ans += fb[finish] * (ll)max_flow;

            v = finish;
            while (1) {
                auto now = prev[v];
                if (now.x == -1) break;
                e[now.y].flow     += max_flow;
                e[now.y ^ 1].flow -= max_flow;
                v = now.x;
            }
            required_flow -= max_flow;
        }

        return ans;
    }
} min_cost_max_flow;
```

MinCostMaxFlowPotenc

```
struct MinCostMaxFlow {
    struct Edge{
        int to, cap;
        int flow;
        int cost;
```

```
    };

    static const int MAX_V = 603;
    static const int MAX_E = 2 * 333 * 333;
    static const int INF = 1e9 + 7;
    static const int MAX_COST = 1e9 + 7; // change to ll if it is exceeded in FB

    int sz = 0;
    Edge e[MAX_E];
    vector<int> g[MAX_V];
    int dp[MAX_V];
    pair<int, int> prev[MAX_V];
    int phi[MAX_V];

    void addEdge(int v, int to, int cap, int cost){
        g[v].push_back(sz);
        e[sz++] = { to, cap, 0, cost };
        g[to].push_back(sz);
        e[sz++] = { v, 0, 0, -cost };
    }

    void calcPhi(int start) {
        // FB for calculating phi, add vertex q and q->v for all v with cost 0
        for (int i = 0; i < MAX_V; ++i) phi[i] = MAX_COST;
        phi[start] = 0;
        for (int k = 0; k < MAX_V; k++) {
            for (int v = 0; v < MAX_V; v++) {
                for (int to : g[v]) {
                    Edge &ed = e[to];
                    if (ed.cap == ed.flow) continue;
                    phi[ed.to] = min(phi[ed.to], phi[v] + ed.cost);
                }
            }
        }
    }

    ll find(int start, int finish, int required_flow) {
        calcPhi(start);

        ll ans = 0;

        while (required_flow) {
            for (int i = 0; i < MAX_V; i++) dp[i] = INF, prev[i] = { -1, -1 };
            dp[start] = 0;

            set< pair<int, int> > se;
            se.insert({ 0, start });

            while (!se.empty()) {
                auto [dist, v] = *se.begin(); se.erase(se.begin());
                for (int to : g[v]) {
                    auto ed = e[to];
                    if (ed.flow < ed.cap && dp[ed.to] > dp[v] + ed.cost - phi[ed.to]
+ phi[v]) {
                        prev[ed.to] = { v, to };
                        se.erase({ dp[ed.to], ed.to });
                        dp[ed.to] = dp[v] + ed.cost - phi[ed.to] + phi[v];
                        se.insert({ dp[ed.to], ed.to });
                    }
                }
            }

            if (dp[finish] == INF) {
                return -1;
            }

            int max_flow = required_flow;
            int v = finish;
```

```
            while (1) {
                auto now = prev[v];
                if (now.x == -1) break;
                max_flow = min(max_flow, e[now.y].cap - e[now.y].flow);
                v = now.x;
            }
            ans += (dp[finish] + phi[finish]) * (ll)max_flow;

            v = finish;
            while (1) {
                auto now = prev[v];
                if (now.x == -1) break;
                e[now.y].flow     += max_flow;
                e[now.y ^ 1].flow -= max_flow;
                v = now.x;
            }
            required_flow -= max_flow;

            // recalc phi
            int min_phi = 0;
            for (int i = 0; i < MAX_V; ++i) {
                if (dp[i] == INF) {
                    min_phi = min(min_phi, phi[i]);
                } else {
                    phi[i] += dp[i];
                }
            }
            for (int i = 0; i < MAX_V; ++i) {
                if (dp[i] == INF) {
                    phi[i] -= min_phi;
                }
            }
            //
        }

        return ans;
    }
} min_cost_max_flow;
```

Minkowski

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

struct MinkowskiSum{

        struct Pt
        {
                ll x, y;
        };

        ll vector_multiple(Pt &a, Pt &b){
                return a.x * b.y - a.y * b.x;
        }

        Pt sum(Pt &a, Pt &b){
                return {a.x+b.x, a.y+b.y};
        }

        // точки отдавать в порядке сортировки против часовой стрелки

        vector<Pt> minkowski_sum(vector<Pt> &a, vector<Pt> &b){ //возможно не
работает для min(n, m) <= 2
                int n = a.size(), m = b.size();
                a.push_back(a[0]), a.push_back(a[1]);
```

```
                b.push_back(b[0]), b.push_back(b[1]);
                int i = 0, j = 0;
                vector<Pt> res;
                while (i < n || j < m){
                        res.push_back(sum(a[i], b[j]));
                        Pt first_vector = {a[i+1].x-a[i].x, a[i+1].y - a[i].y};
                        Pt second_vector = {b[j+1].x-b[j].x, b[j+1].y - b[j].y};
                        ll vp = vector_multiple(first_vector, second_vector);
                        if (vp > 0 || j==m){
                                ++i;
                        }
                        else if (vp < 0 || i==n){
                                ++j;
                        }
                        else{
                                ++i, ++j;
                        }
                }
                return res;
        }

};
```

NTT

```
class NTT{
public:
        #define db long double
        #define ll long long
        const static int mod = 998244353;
        const static int root = 646; // 646^(2^20) == 1 (998244353)
        const static int rev_root = 208611436;
        const static int MAX_SIZE = 1 << 21;

        void add(int &a, int b){
                a += b;
                if (a < 0) a += mod;
                if (a >= mod) a -= mod;
        }

        int sum(int a, int b){
                add(a, b);
                return a;
        }

        int mult(int a, int b){
                return a * (ll)b % mod;
        }

        int bp(int a, int k){
                if (k == 0) return 1;
                if (k & 1){
                        return mult(a, bp(a, k - 1));
                } else {
                        int q = bp(a, k >> 1);
                        return mult(q, q);
                }
        }

        int rev(int a){
                return bp(a, mod - 2);
        }

        int n;
        int a[MAX_SIZE * 2 + 7], b[MAX_SIZE * 2 + 7];
```

```
        int getReverse(int a, int k){
                int ans = 0;
                for (int i = 0; i < k; i++) if ((a >> i) & 1) ans ^= (1 << (k - i -
1));
                return ans;
        }

        void ntt(int *a, int type){
                int k = -1;
                for (int i = 0; i < 25; i++) if ((n >> i) & 1){
                        k = i;
                        break;
                }
                for (int i = 0; i < n; i++){
                        int j = getReverse(i, k);
                        if (i < j) swap(a[i], a[j]);
                }
                for (int len = 2; len <= n; len *= 2){
                        int w = bp(root, (1 << 20) / len);
                        if (type == -1) w = bp(rev_root, (1 << 20) / len);
                        for (int i = 0; i < n; i += len){
                                int g = 1;
                                for (int j = 0; j < len / 2; j++){
                                        int x = a[i + j];
                                        int y = mult(a[i + j + len / 2], g);
                                        a[i + j] = sum(x, y);
                                        a[i + j + len / 2] = sum(x, mod - y);
                                        g = mult(g, w);
                                }
                        }
                }
                if (type == -1){
                        int rev_n = rev(n);
                        for (int i = 0; i < n; i++) a[i] = mult(a[i], rev_n);
                }
        }

        vector<int> mult(vector<int> &w1, vector<int> &w2){
                n = 1;
                while(n < w1.size() + w2.size()) n *= 2;
                for (int i = 0; i < w1.size(); i++){
                        a[i] = w1[i];
                        a[i] %= mod;
                        if (a[i] < 0) a[i] += mod;
                }
                for (int i = 0; i < w2.size(); i++){
                        b[i] = w2[i];
                        b[i] %= mod;
                        if (b[i] < 0) b[i] += mod;
                }
                for (int i = w1.size(); i < n; i++) a[i] = 0;
                for (int i = w2.size(); i < n; i++) b[i] = 0;
                ntt(a, 1);
                ntt(b, 1);
                for (int i = 0; i < n; i++) a[i] = mult(a[i], b[i]);
                ntt(a, -1);
                vector<int> ans(n);
                for (int i = 0; i < n; i++) ans[i] = a[i];
                while(ans.size() && ans.back() == 0) ans.pop_back();
                return ans;
        }
};
```

OrConvolution

```
const int K = 1<<17;
```

```
// u can set modular arithmetic here
void ORConvolution(vector<int>& v){
    for (int step=K; step > 1; step /= 2){
        for (int start=0; start < K; start += step){
            for (int w=0; w < step/2; w++){
                v[start+step/2+w] += v[start + w];
            }
        }
    }
}

void inverseORConvolution(vector<int>& v){
    for (int step=K; step > 1; step /= 2){
        for (int start=0; start < K; start += step){
            for (int w=0; w < step/2; w++){
                v[start+step/2+w] -= v[start + w];
            }
        }
    }
}

/* Usage Example
    ORConvolution(f);
    ORConvolution(g);
    for (int i = 0; i < K; i++) f[i] *= g[i];
    inverseORConvolution(f);
    f is ur answer
*/
```

PalindromeTree

```
struct PalindromeTree {
    static const int SZ = 5e5;
    static const int SIGMA = 26;

    vector<int> s;
    int to[SZ][SIGMA];
    int suf[SZ];
    int len[SZ];
    int last;
    int sz;

    // 0, 1 - roots
    PalindromeTree() {
        s.push_back(-1);
        for (int i = 0; i < SZ; ++i) for (int j = 0; j < SIGMA; ++j) to[i][j] = -1;
        sz = 2; last = 1; len[0] = -1; suf[1] = 0; suf[0] = -1;
    }

    void clear() {
        s.clear();
        s.push_back(-1);
        for (int i = 0; i < sz; ++i) {
            for (int j = 0; j < SIGMA; ++j) {
                to[i][j] = -1;
            }
        }
        sz = 2; last = 1; len[0] = -1; suf[1] = 0; suf[0] = -1;
    }

    void add(int c) {
        s.push_back(c);
        while (c != s[(int)s.size() - len[last] - 2]){
            last = suf[last];
        }
```

```
        if (to[last][c] == -1){
            int v = sz++;
            to[last][c] = v;
            len[v] = len[last] + 2;
            do {
                last = suf[last];
            } while(last != -1 && s[(int)s.size() - len[last] - 2] != c);
            if (last == -1){
                suf[v] = 1;
            } else {
                suf[v] = to[last][c];
            }
            last = v;
        } else {
            last = to[last][c];
        }
    }
} PT;
```

PrimitiveRoot

```
#include <bits/stdc++.h>
#define ll long long
#define ull unsigned long long
#define db long double

using namespace std;

struct PrimitiveRoot{

        int mod, root; //modulo must be prime
        //call initialization and answer will be in 'root'

        int mult(int x, int y){
                return ((ll) x * (ll) y) % (ll) mod;
        }

        int pw(int x, int y){
                if (y==0) return 1;
                if (y==1) return x%mod;
                if (y%2) return mult(x, pw(x, y-1));
                int R = pw(x, y/2);
                return mult(R, R);
        }

        vector<int> get_primes(int v){
            vector<int> ans;
            int uk = 2;
            while(uk * uk <= v){
                int was = 0;
                while(v % uk == 0){
                    v /= uk;
                    was = 1;
                }
                if (was) ans.push_back(uk);
                uk++;
            }
            if (v > 1) ans.push_back(v);
            return ans;
        }

        PrimitiveRoot(int given_mod){
                mod = given_mod;
            int phi = mod - 1;
            auto now = get_primes(phi);
```

```
            for (int v = 1; ; v++){
                bool ok = 1;

                for (int p : now) if (pw(v, phi / p) == 1){
                    ok = 0;
                    break;
                }

                if (ok){
                        root = v;
                        return;
                }
            }
        }
};
```

SmallestCircleProblem

```
namespace SCP{ //Smallest Circle Problem
    //it is supposed to work O(n) averagely
    struct pt{
        db x, y;
        pt() {}
        pt(db x, db y): x(x), y(y) {}
        pt operator- (const pt &nxt) const { return pt(x - nxt.x, y - nxt.y); }
        db len(){
            return sqrt(x * x + y * y);
        }
    };

    struct line{
        db a, b, c;
    };

    db getSquare(db r){
        return M_PI * r * r;
    }

    pt getMedian(pt &a, pt &b){
        return pt((a.x + b.x) / 2, (a.y + b.y) / 2);
    }

    pair<pt, db> SCP(pt &a, pt &b){
        return make_pair(getMedian(a, b), (a - b).len() / 2);
    }

    pt intersectLines(line &l1, line &l2){
        if (abs(l1.a * l2.b - l2.a * l1.b) < eps) throw 42;
        db x = (l2.c * l1.b - l1.c * l2.b) / (l1.a * l2.b - l2.a * l1.b);
        db y = (l2.c * l1.a - l1.c * l2.a) / (l1.b * l2.a - l2.b * l1.a);
        return pt(x, y);
    }

    pair<pt, db> SCP(pt &a, pt &b, pt &c){
        pt o1 = getMedian(a, b);
        pt o2 = getMedian(b, c);
        line l1, l2;
        l1.a = (b - a).x; l1.b = (b - a).y; l1.c = -(l1.a * o1.x + l1.b * o1.y);
        l2.a = (b - c).x; l2.b = (b - c).y; l2.c = -(l2.a * o2.x + l2.b * o2.y);
        try {
            pt o = intersectLines(l1, l2);
            return make_pair(o, (o - a).len());
        } catch(...) {
            throw;
        }
    }
```

```
    bool inCircle(pt &a, pt &O, db r){
        return (O - a).len() <= r + eps;
    }

    pair<pt, db> recSolve(vector<pt> &a, vector<pt> &b){
        assert(b.size() <= 3);
        if (b.size() == 3){
            auto [O, r] = SCP(b[0], b[1], b[2]);
            bool ok = 1;
            for (auto p : a) if (!inCircle(p, O, r)){
                ok = 0;
                break;
            }
            if (ok) return make_pair(O, r);
            else return make_pair(O, -2);
        } else {
            if (a.size() == 0){
                if (b.size() == 0) return make_pair(pt(0, 0), 0);
                if (b.size() == 1) return make_pair(b[0], 0);
                if (b.size() == 2) return SCP(b[0], b[1]);
            } else {
                pt p = a.back(); a.pop_back();
                auto [O, r] = recSolve(a, b);
                a.push_back(p);
                if (inCircle(p, O, r)) return make_pair(O, r);
                a.pop_back(), b.push_back(p);
                auto res = recSolve(a, b);
                a.push_back(p), b.pop_back();
                return res;
            }
        }
    }

    db solve(vector<pt> &a){
        if (a.size() == 1) return 0;
        random_shuffle(a.begin(), a.end());
        vector<pt> b;
        db ans = recSolve(a, b).second;
        return getSquare(ans);
    }
}
```

SuffixArray

```
struct SuffixArray {
    static const int SZ = 3e5;

    int c[SZ];
    int cnt[SZ];
    int p[SZ];
    int pn[SZ];
    int cn[SZ];

    vector<int> buildSA(const vector<int>& s) {
        int n = s.size();
        int alpha = (*max_element(s.begin(), s.end())) + 1;
        memset(cnt, 0, alpha * sizeof(int));
        for (int c : s) ++cnt[c];
        for (int i = 1; i < alpha; ++i) cnt[i] += cnt[i - 1];
        for (int i = 0; i < n; ++i) p[--cnt[s[i]]] = i;
        c[p[0]] = 0;
        int cs = 1;
        for (int i = 1; i < n; ++i) {
            if (s[p[i]] != s[p[i - 1]]) ++cs;
            c[p[i]] = cs - 1;
```

```
        }

        for (int h = 0; (1 << h) < n; ++h) {
            for (int i = 0; i < n; ++i) {
                pn[i] = p[i] - (1 << h);
                if (pn[i] < 0) pn[i] += n;
            }
            memset(cnt, 0, cs * sizeof(int));
            for (int i = 0; i < n; ++i) ++cnt[c[pn[i]]];
            for (int i = 1; i < cs; ++i) cnt[i] += cnt[i - 1];
            for (int i = n - 1; i >= 0; --i) p[--cnt[c[pn[i]]]] = pn[i];
            cn[p[0]] = 0;
            cs = 1;
            for (int i = 1; i < n; ++i) {
                int mid1 = (p[i] + (1 << h)) % n, mid2 = (p[i-1] + (1 << h)) % n;
                if (c[p[i]] != c[p[i-1]] || c[mid1] != c[mid2]) ++cs;
                cn[p[i]] = cs - 1;
            }
            memcpy (c, cn, n * sizeof(int));
        }

        vector<int> result(p, p + n);
        return result;
    }

    // suf = sa from func above
    vector<int> buildLCP(const vector<int>& s, const vector<int>& suf) const {
        int n = s.size();
        vector<int> rsuf(n);
        vector<int> lcp(n);
        for (ll i = 0; i < n; i++) {
            rsuf[suf[i]] = i;
        }

        int k = 0;
        for (int i = 0; i < n; ++i) {
            if (k > 0) --k;
            if (rsuf[i] == n - 1) {
                lcp[n - 1] = -1;
                k = 0;
                continue;
            } else {
                int j = suf[rsuf[i] + 1];
                while (max(i + k, j + k) < n && s[i + k] == s[j + k]) ++k;
                lcp[rsuf[i]] = k;
            }
        }

        return lcp;
    }
} SA;
```

SuffixAutomata

```
struct Automata{
    static const int K = 1000000; //choose K as twice string length + const
    int counter;
    int go[K][26];
    int last;
    int suf[K], len[K];
    Automata(){
        for (int i=0; i < K; i++){
            suf[i] = -1;
            len[i] = -1;
            for (int j=0; j < 26; j++){
                go[i][j] = -1;
```

```
            }
        }
        len[0] = -1;
        last = 0;
        counter = 1;
    }
    void add(int number){
        int newlast = counter; len[newlast] = len[last] + 1; int p = last; counter++;
        while (p!=-1 && go[p][number] == -1){
            go[p][number] = newlast;
            p = suf[p];
        }
        if (p == -1){
            suf[newlast] = 0;
        }
        else{
            int q = go[p][number];
            if (len[q] == len[p] + 1){
                suf[newlast] = q;
            }
            else{
                int r = counter; counter ++;
                for (int i=0;i<26;i++){
                    go[r][i] = go[q][i];
                }
                suf[r] = suf[q];
                suf[q] = r;
                suf[newlast] = r;
                len[r] = len[p] + 1;
                while (p!=-1 && go[p][number] == q){
                    go[p][number] = r;
                    p = suf[p];
                }
            }
        }
        last = newlast;
    }
    void add_total(string &s){
        for (int i=0; i < s.size(); i++){
            add(s[i] - 'a');
        }
    }
};
```

SumLine

```
// sum(i=0..n-1) (a+b*i) div m
ll solve(ll n, ll a, ll b, ll m) {
    if (b == 0) return n * (a / m);
    if (a >= m) return n * (a / m) + solve(n, a % m, b, m);
    if (b >= m) return n * (n - 1) / 2 * (b / m) + solve(n, a, b % m, m);
    return solve((a + b * n) / m, (a + b * n) % m, m, b);
}
```

Tandems

```
#include <bits/stdc++.h>
#define ctr CompressedTandemRepeats
#define ll long long
#define ull unsigned long long
#define db long double

using namespace std;
```

```
struct CompressedTandemRepeats{int l; int r; int x;};
//we represent all tandem repeats as triples (l, r, x)
//what means that all substrings beginning in [l, ..., r] and having size x are
tandem repeats
//it can be proved that triples number is O(n)
//the algorithm works in O(n) space and O(n*logn) time

//just call get function to get all triples

struct TandemRepeats{
        int n;
        int how_reverse, how_add;
        vector<ctr> res; //answer will be here
        vector<pair<int, int> > current_pair;

        vector<int> z_function(string &s){
                int n = s.size();
                vector<int> z (n);
                for (int i=1, l=0, r=0; i<n; ++i) {
                        if (i <= r)
                                z[i] = min (r-i+1, z[i-l]);
                        while (i+z[i] < n && s[z[i]] == s[i+z[i]])
                                ++z[i];
                        if (i+z[i]-1 > r)
                                l = i,  r = i+z[i]-1;
                }
                return z;
        }

        void add_to_list(int index, int len, int k1, int k2){
                int L = len-k2, R = k1;
                if (L>R) return;
                swap(L, R);
                L = index-L, R = index-R;
                if (how_reverse > 0){
                        L += 2*len-1, R += 2*len-1;
                        L = (how_reverse-1-L), R = (how_reverse-1-R);
                        swap(L, R);
                }
                if (current_pair[2*len].second != -1 && current_pair[2*len].second+1
== L+how_add){
                        current_pair[2*len].second = R+how_add;
                }
                else{
                        if (current_pair[2*len].second != -1)
res.emplace_back(current_pair[2*len].first, current_pair[2*len].second, 2*len);
                        current_pair[2*len] = {L+how_add, R+how_add};
                }
        }

        void main_part(string &u, string &v, bool if_forget){
                string u_rev = u;
                reverse(u_rev.begin(), u_rev.end());
                vector<int> ZU = z_function(u_rev);
                string spec = v+'#'+u;
                vector<int> ZUV = z_function(spec);
                for (int i=0; i < u.size(); ++i){
                        int len = (u.size()-i);
                        if (len > v.size()) continue;
                        int k1 = 0;
                        if (i > 0) k1 = ZU[u.size()-i];
                        k1 = min(k1, len-1);
                        int k2 = ZUV[v.size()+1+u.size()-len];
                        if (if_forget) k2 = min(k2, len-1);
                        add_to_list(i, len, k1, k2);
                }
        }
```

```
        void MainLorenz(string &s, int add){
                if (s.size() == 1) return;
                string u, v;
                for (int i=0; i < s.size(); ++i){
                        if (2*i < s.size()) u += s[i];
                        else v += s[i];
                }

                string Q = v;
                int R = u.size();
                MainLorenz(u, add);

                how_reverse = -1, how_add=add;
                main_part(u, v, false);
                reverse(u.begin(), u.end()), reverse(v.begin(), v.end());
                how_reverse = s.size();
                main_part(v, u, true);

                MainLorenz(Q, add+R);
        }

        vector<ctr> get(string &s){
                n = s.size();
                current_pair.assign(n+1, {-1, -1});
                MainLorenz(s, 0);
                for (int i=0;i<=n;++i) if (current_pair[i].second!=-1){
                        res.emplace_back(current_pair[i].first,
current_pair[i].second, i);
                }
                return res;
        }
};
```

WeightedMatroids

```
#include <bits/stdc++.h>
#define vo vector<Object>
// Матроид над множеством X - такое множество I подмножеств X, что
// 1) пустое множество лежит в I
// 2) Если A лежит в I и B лежит в A, то B лежит в I
// 3) Если A, B лежат в I и |A| > |B|, найдется непустое x принадлежащее A/B, что x U
B принадлежит I
// Алгоритм пересечения имеет ответ answer на данный момент и other - все, что не
входит в ответ
// Затем он проводит ребра из y в z, где y лежит в answer, z лежит в other и
(answer/y) U z лежит в I1
// и проводит ребра из z в y, где y лежит в answer, z лежит в other и (answer/y) U z
лежит в I2
// X1 - множество z из other, таких, что answer U z лежит в I1, аналогично X2
// запускаем dfs из x1 в x2, находим кратчайший путь. Если пути нет, ответ найден
// иначе на этом кр.пути вершины из other переносим в answer и наоборот
//во взвешенном случае ставим веса -w[i] в вершины из other и w[i] из answer. Затем
ищем
//кратчайший путь по {len, size}
using namespace std;
//в Object любые поля
struct Object{int index; int npc; int u; int v;};
struct WeightedMatroids{
        static const int INF = 1e9;
        vo all_objects;
        vector<vector<int> > data;
        int required_size;
        int res;
        vector<int> w;
        WeightedMatroids(vo o, vector<int> W, int K){
                w = W, required_size = K, res = 0;
```

```
                all_objects = o;
        }

        // из answer убираем i, из other к answer добавляем j
        // проверяем свойство матройда (например, связность)
        // i, j могут быть -1

        bool valid2(vo &answer, vo &other, int i, int j){

        }
        bool valid1(vo &answer, vo &other, int i, int j){

        }
        pair<vo, vo> solve(vo answer, vo other){
            int N = answer.size() + other.size();
            data.assign(N, {});
            vector<bool> x1, x2;
            x1.assign(N, false);
            x2.assign(N, false);
            int S = answer.size();
            for (int i=0; i < answer.size(); i++){
                for (int j=0; j < other.size(); j++){
                        if (valid1(answer, other, i, j)) data[i].push_back(S+j);
                        if (valid2(answer, other, i, j)) data[S+j].push_back(i);
                }
            }
            for (int i=0; i < other.size(); i++){
                if (valid1(answer, other, -1, i)) x1[S+i] = true;
                if (valid2(answer, other, -1, i)) x2[S+i] = true;
            }
            //for (int i=0; i < other.size(); i++) cout << x1[i] << " " << x2[i] <<
endl;
            vector<pair<int, int> > path;
            vector<int> last;
            path.assign(N, {INF, -1}), last.assign(N, -1);
            for (int i=0; i < N; i++) if (x1[i]) path[i] = {-w[other[i-S].index], 1};
            for (int i=0; i < N; i++){
                for (int j=0; j < N; j++){
                        for (int k=0; k < data[j].size(); k++){
                                int to = data[j][k];
                                pair<int, int> R = {path[j].first, path[j].second+1};
                                if (to < S) R.first += w[answer[to].index];
                                else R.first -= w[other[to-S].index];
                                if (R < path[to]){
                                        path[to] = R, last[to] = j;
                                }
                        }
                }
            }
            pair<int, int> best = {INF, -1};
            int where = -1;
            for (int i=0; i < N; i++) if (x2[i]) if (path[i] < best){
                best = path[i];
                where = i;
            }
            if (where == -1) return {answer, other};
            res -= best.first;
            vo na, nold;
            set<int> sused;
            int now = 1;
            while (true){
                sused.insert(where);
                if (now==1) na.push_back(other[where - answer.size()]);
                else nold.push_back(answer[where]);
                if (last[where] == -1) break;
                where = last[where];
                now = 1-now;
            }
```

```
            for (int i=0; i < answer.size(); i++) if (!sused.count(i))
na.push_back(answer[i]);
            for (int i=0; i < other.size(); i++) if (!sused.count(i+answer.size()))
nold.push_back(other[i]);
            return {na, nold};
        }

        int get_w(){
            vo ans = {}, other = all_objects;
            for (int i=0; i < required_size; i++){
                pair<vo, vo> res = solve(ans, other);
                if (res.first.size() == ans.size()) return -INF;
                ans = res.first, other = res.second;
            }
            return res;
        }
};
```

XorConvolution

```
const int K = 1<<17;

// u can set modular arithmetic here
void hadamard(vector<int>& v){
    for (int step=K; step > 1; step /= 2){
        for (int start=0; start < K; start += step){
            for (int w=0; w < step/2; w++){
                int F = v[start+w] + v[start+step/2+w];
                int S = v[start+w] - v[start+step/2+w];
                v[start + w] = F;
                v[start+step/2+w] = S;
            }
        }
    }
}

/* Usage Example
    vector<int> f((1<<K)), g((1<<K));
    hadamard(f);
    hadamard(g);
    for (int i=0; i < K; i++) f[i] *= g[i];
    hadamard(f);
    for (int i=0; i < K; i++) f[i] /= K;
    // f is ur answer
*/
```

DynamicConvexHullTrick

```
#define ALL(c) (c).begin(),(c).end()
#define IN(x,c) (find(c.begin(),c.end(),x) != (c).end())
#define REP(i,n) for (int i=0;i<(int)(n);i++)
#define FOR(i,a,b) for (int i=(a);i<=(b);i++)
#define INIT(a,v) memset(a,v,sizeof(a))
#define SORT_UNIQUE(c) (sort(c.begin(),c.end()),
c.resize(distance(c.begin(),unique(c.begin(),c.end()))))
template<class A, class B> A cvt(B x) { stringstream ss; ss<<x; A y; ss>>y; return y;
}

typedef pair<int,int> PII;
typedef long long int64;

#define N 100000

int n;
```

```
int64 h[N],w[N];

int64 sqr(int64 x) { return x*x; }

struct line {
        char type;
        double x;
        int64 k, n;
};

bool operator<(line l1, line l2) {
        if (l1.type+l2.type>0) return l1.x<l2.x;
        else return l1.k>l2.k;
}

set<line> env;
typedef set<line>::iterator sit;

bool hasPrev(sit it) { return it!=env.begin(); }
bool hasNext(sit it) { return it!=env.end() && next(it)!=env.end(); }

double intersect(sit it1, sit it2) {
        return (double)(it1->n-it2->n)/(it2->k-it1->k);
}

void calcX(sit it) {
        if (hasPrev(it)) {
                line l = *it;
                l.x = intersect(prev(it), it);
                env.insert(env.erase(it), l);
        }
}

bool irrelevant(sit it) {
        if (hasNext(it) && next(it)->n <= it->n) return true; // x=0 cutoff //useless
        return hasPrev(it) && hasNext(it) && intersect(prev(it),next(it)) <=
intersect(prev(it),it);
}

void add(int64 k, int64 a) {
        sit it;
        // handle collinear line
        it=env.lower_bound({0,0,k,a});
        if (it!=env.end() && it->k==k) {
                if (it->n <= a) return;
                else env.erase(it);
        }
        // erase irrelevant lines
        it=env.insert({0,0,k,a}).first;
        if (irrelevant(it)) { env.erase(it); return; }
        while (hasPrev(it) && irrelevant(prev(it))) env.erase(prev(it));
        while (hasNext(it) && irrelevant(next(it))) env.erase(next(it));
        // recalc left intersection points
        if (hasNext(it)) calcX(next(it));
        calcX(it);
}

int64 query(int64 x) {
        auto it = env.upper_bound((line){1,(double)x,0,0});
        it--;
        return it->n+x*it->k;
}

int64 g[N];

int64 solve() {
        int64 a=0;
        REP (i,n) a+=w[i];
```

```
        g[0]=-w[0];
        FOR (i,1,n-1) {
                add(-2*h[i-1],g[i-1]+sqr(h[i-1]));
                int64 opt=query(h[i]);
                g[i]=sqr(h[i])-w[i]+opt;
        }
        return a+g[n-1];
}
```

FASTIO

```
const int MAX_MEM = 1e8;
int mpos = 0;
char mem[MAX_MEM];
void * operator new ( size_t n ) {
    char *res = mem + mpos;
    mpos += n;
    return (void *)res;
}
void operator delete ( void * ) { }

/** Interface */

inline int readChar();
template <class T = int> inline T readInt();
template <class T> inline void writeInt( T x, char end = 0 );
inline void writeChar( int x );
inline void writeWord( const char *s );

/** Read */

static const int buf_size = 4096;

inline int getChar() {
    static char buf[buf_size];
    static int len = 0, pos = 0;
    if (pos == len)
        pos = 0, len = fread(buf, 1, buf_size, stdin);
    if (pos == len)
        return -1;
    return buf[pos++];
}

inline int readChar() {
    int c = getChar();
    while (c <= 32)
        c = getChar();
    return c;
}

template <class T>
inline T readInt() {
    int s = 1, c = readChar();
    T x = 0;
    if (c == '-')
        s = -1, c = getChar();
    while ('0' <= c && c <= '9')
        x = x * 10 + c - '0', c = getChar();
    return s == 1 ? x : -x;
}

/** Write */

static int write_pos = 0;
static char write_buf[buf_size];

```

```
46. inline void writeChar( int x ) {
47.     if (write_pos == buf_size)
48.         fwrite(write_buf, 1, buf_size, stdout), write_pos = 0;
49.     write_buf[write_pos++] = x;
50. }
51.
52. template <class T>
53. inline void writeInt( T x, char end ) {
54.     if (x < 0)
55.         writeChar('-'), x = -x;
56.
57.     char s[24];
58.     int n = 0;
59.     while (x || !n)
60.         s[n++] = '0' + x % 10, x /= 10;
61.     while (n--)
62.         writeChar(s[n]);
63.     if (end)
64.         writeChar(end);
65. }
66.
67. inline void writeWord( const char *s ) {
68.     while (*s)
69.         writeChar(*s++);
70. }
71.
72. struct Flusher {
73.     ~Flusher() {
74.         if (write_pos)
75.             fwrite(write_buf, 1, write_pos, stdout), write_pos = 0;
76.     }
77. } flusher;
78.
79. /** Example */
```

HalfplaneIntersection

```
#define ld double
struct point{
        ld x, y;
        point() {}
        point(ld x1, ld y1) { x = x1, y = y1; }
        ld operator% (point nxt) const { return x * nxt.y - y * nxt.x; }
        ld operator* (point nxt) const { return x * nxt.x + y * nxt.y; }
        point operator- (point nxt) const { return point(x - nxt.x, y - nxt.y); }
        point operator+ (point nxt) const { return point(x + nxt.x, y + nxt.y); }
};
struct line{
        ld a, b, c;
        point s, t;
        line() {}
        line(point s1, point t1){
                s = s1, t = t1;
                a = t.y - s.y;
                b = s.x - t.x;
                c = (t.x - s.x) * s.y - s.x * (t.y - s.y);
                if ((t - s) % point(a, b) < 0){
                        a = -a, b = -b, c = -c;
                }
        }
};
const ld BOX = 1e18;
const ld pi = acos(-1.0);
bool equal(point s, point t){
        return (s % t) == 0 && (s * t) > 0;
}
```

```
bool cmp(line s, line t){
        if (equal(s.t - s.s, t.t - t.s)){
                if (abs(s.s.x) == BOX) return 0;
                if (abs(t.s.x) == BOX) return 1;
                return (s.t - s.s) % (t.s - s.s) < 0;
        }
        ld val1 = atan2(s.b, s.a);
        ld val2 = atan2(t.b, t.a);
        if (val1 < 0) val1 += pi * 2;
        if (val2 < 0) val2 += pi * 2;
        return val1 < val2;
}

point crossLineLine(line s, line t){
        ld x = (t.c * s.b - s.c * t.b) / (s.a * t.b - s.b * t.a);
        ld y = (t.c * s.a - s.c * t.a) / (s.b * t.a - t.b * s.a);
        return point(x, y);
}
void halfplanesIntersection(vector<line> a){
        //==========BOX=================
        a.pub(line(point(-BOX, -BOX), point(BOX, -BOX)));
        a.pub(line(point(-BOX, BOX), point(-BOX, -BOX)));
        a.pub(line(point(BOX, -BOX), point(BOX, BOX)));
        a.pub(line(point(BOX, BOX), point(-BOX, BOX)));
        //==============================
        sort(all(a), cmp);
        vector<line> q;
        for (int i = 0; i < a.size(); i++){
                if (i == 0 || !equal(a[i].t - a[i].s, a[i - 1].t - a[i - 1].s))
q.pub(a[i]);
        }
        //for (auto c : q){
        //      cout << "Line " << fixed << c.a << ' ' << c.b << ' ' << c.c << endl;
        //}
        vector<int> st;
        for (int it = 0; it < 2; it++){
                for (int i = 0; i < q.size(); i++){
                        while(st.size() > 1){
                                int j = st.back(), k = st[(int)st.size() - 2];
                                if (((q[i].t - q[i].s) % (q[j].t - q[j].s)) == 0)
break;
                                auto pt = crossLineLine(q[i], q[j]);
                                if ((q[k].t - q[k].s) % (pt - q[k].s) > 0) break;
                                st.pop_back();
                        }
                        st.pub(i);
                }
        }
    vector<int> was((int)a.size(), -1);
    bool ok = 0;
    for (int i = 0; i < st.size(); i++){
        int uk = st[i];
        if (was[uk] == -1){
                was[uk] = i;
        } else {
                st = vector<int>(st.begin() + was[uk], st.begin() + i);
                ok = 1;
                break;
        }
    }
    if (!ok){
                cout << "Impossible", exit(0);
    }
    point ans = point(0, 0);
    for (int i = 0; i < st.size(); i++){
        line l1 = q[st[i]], l2 = q[st[(i + 1) % (int)st.size()]];
        ans = ans + crossLineLine(l1, l2);
    }
```

```
    ans.x /= (ld)st.size();
    ans.y /= (ld)st.size();
    for (int i = 0; i < a.size(); i++){
        line l = a[i];
        if ((l.t - l.s) % (ans - l.s) <= 0) cout << "Impossible", exit(0);
    }
    cout << "Possible\n";
    cout.precision(10);
    cout << fixed << ans.x << ' ' << ans.y;
}
```

Hungarian

```
int n, ai;
int matrix[300][300];
vector<int> column_p, string_p, where, minv, strv, where_string;
vector<bool> see;
int INF = 1e15;
int32_t main()
{
    ios_base::sync_with_stdio(false);
    cin >> n;
    for (int i=0; i < n; i++){
        column_p.push_back(0);
        string_p.push_back(0);
        where.push_back(-1);
        where_string.push_back(-1);
        minv.push_back(-1);
        strv.push_back(-1);
        see.push_back(true);
        for (int j=0; j < n; j++){
            cin >> ai;
            matrix[i][j] = ai;
        }
    }
    for (int it=0; it < n; it++){
        vector<int> strings, columns;
        int now_string = it;
        fill(see.begin(), see.end(), true);
        fill(minv.begin(), minv.end(), INF);
        while (true){
            int minimum = INF;
            int mincol = -1;
            strings.push_back(now_string);
            for (int i=0; i < see.size(); i++){
                if (see[i]){
                    if (minv[i] > matrix[now_string][i] - string_p[now_string] -
column_p[i]){
                        minv[i] = matrix[now_string][i] - string_p[now_string] -
column_p[i];
                        strv[i] = now_string;
                    }
                    if (minv[i] < minimum){
                        minimum = minv[i];
                        mincol = i;
                    }
                }
            }
            for (int i=0; i < strings.size(); i++){
                string_p[strings[i]] += minimum;
            }
            for (int i=0; i < columns.size(); i++){
                column_p[columns[i]] -= minimum;
            }
            for (int i=0; i < n; i++){
                minv[i] -= minimum;
```

```
            }
            if (where[mincol] == -1){
                int nc = mincol;
                int str = strv[mincol];
                while (where_string[str] != -1){
                    int col = where_string[str];
                    where[nc] = str;
                    where_string[str] = nc;
                    str = strv[col];
                    nc = col;
                }
                where_string[str] = nc;
                where[nc] = str;
                break;
            }
            else{
                now_string = where[mincol];
                columns.push_back(mincol);
                see[mincol] = false;
            }
        }
    }
    int cost = 0;
    for (int i=0; i < n; i++){
        cost += string_p[i] + column_p[i];
    }
    cout << cost << endl;
    for (int i=0; i < n; i++){
        cout << i + 1 << " " << where_string[i] + 1 << endl;
    }
    return 0;
}
```

Matroids

```
#include <bits/stdc++.h>
#define int long long
// Матроид над множеством X - такое множество I подмножеств X, что
// 1) пустое множество лежит в I
// 2) Если A лежит в I и B лежит в A, то B лежит в I
// 3) Если A, B лежат в I и |A| > |B|, найдется непустое x принадлежащее A/B, что x U
B принадлежит I
// Алгоритм пересечения имеет ответ answer на данный момент и other - все, что не
входит в ответ
// Затем он проводит ребра из y в z, где y лежит в answer, z лежит в other и
(answer/y) U z лежит в I1
// и проводит ребра из z в y, где y лежит в answer, z лежит в other и (answer/y) U z
лежит в I2
// X1 - множество z из other, таких, что answer U z лежит в I1, аналогично X2
// запускаем dfs из x1 в x2, находим кратчайший путь. Если пути нет, ответ найден
// иначе на этом кр.пути вершины из other переносим в answer и наоборот
using namespace std;
struct Heap{int index; int value;};
const int K = 62;
vector<vector<int> > data;
int n, m;
vector<Heap> solve(vector<Heap> answer, vector<Heap> other){
//    for (int i=0; i < answer.size(); i++) cout << answer[i].index << " " <<
answer[i].value << " / ";
//    cout << endl;
//    for (int i=0; i < other.size(); i++) cout << other[i].index << " " <<
other[i].value << " / ";
//    cout << endl;
    vector<pair<int, int> > hauss(K);
    fill(hauss.begin(), hauss.end(), make_pair(0, 0));
    for (int i=0; i < answer.size(); i++){
```

```
        int T = answer[i].value, e = (1LL<<i);
        for (int j=K-1; j >= 0; j--){
            int ba = T&(1LL<<j);
            if (ba==0) continue;
            if (hauss[j].first == 0){
                hauss[j] = {T, e};
                break;
            }
            else{
                T ^= hauss[j].first, e ^= hauss[j].second;
            }
        }
    }
    int N = answer.size() + other.size();
    data.assign(N, {});
    vector<bool> x1, x2;
    x1.assign(N, false);
    x2.assign(N, false);
    vector<int> last;
    last.assign(N, -1);
    for (int i=0; i < other.size(); i++){
        int T = other[i].value, e = 0;
        for (int j=K-1; j >= 0; j--){
            int ba = T&(1LL<<j);
            if (ba==0) continue;
            //if (answer.size()==5) cout << T << " " << hauss[j].first << endl;
            if (hauss[j].first == 0){
                continue;
            }
            else{
                T ^= hauss[j].first, e ^= hauss[j].second;
            }
        }
        //if (answer.size()==5) cout << T << " " << e << endl;
        for (int j=0; j < answer.size(); j++){
            if (T != 0){
                data[j].push_back(answer.size() + i);
                x1[answer.size()+i] = true;
                last[answer.size()+i] = answer.size()+i;
            }
            else{
                int ba = e & (1LL<<j);
                //if (answer.size()==5) cout << "!!" << e << endl;
                if (ba != 0){
                    data[j].push_back(answer.size() + i);
                    //if (answer.size()==5) cout << "!!" << i << endl;
                }
            }
        }
    }
    vector<bool> used;
    used.resize(n+m, false);
    for (int i=0; i < answer.size(); i++) used[answer[i].index] = true;
    for (int i=0; i < answer.size(); i++){
        for (int j=0; j < other.size(); j++){
            if (answer[i].index != other[j].index){
                if (!used[other[j].index]) data[answer.size() + j].push_back(i);
            }
            else data[answer.size() + j].push_back(i);
            if (!used[other[j].index]){
                x2[answer.size()+j] = true;
            }
        }
    }
    int shortest = -1;
    queue<int> vrt;
    for (int i=0; i < N; i++) if (x1[i]) vrt.push(i);
    while (vrt.size()){
```

```
        int V = vrt.front();
        vrt.pop();
        if (x2[V]){
            shortest = V;
            break;
        }
        for (int i=0; i < data[V].size();i++){
            int to = data[V][i];
            if (last[to] != -1) continue;
            last[to] = V;
            vrt.push(to);
        }
    }
    if (shortest == -1) return answer;
    vector<Heap> na, nold;
    set<int> sused;
    int now = 1;
    while (true){
        sused.insert(shortest);
        if (now==1) na.push_back(other[shortest - answer.size()]);
        else nold.push_back(answer[shortest]);
        if (last[shortest] == shortest) break;
        shortest = last[shortest];
        now = 1-now;
    }
    for (int i=0; i < answer.size(); i++) if (!sused.count(i))
na.push_back(answer[i]);
    for (int i=0; i < other.size(); i++) if (!sused.count(i+answer.size()))
nold.push_back(other[i]);
    return solve(na, nold);
}
main() {
    //freopen("input.txt", "r", stdin);
    vector<Heap> v;
    cin >> n;
    vector<Heap> answer = {};
    for (int i=0; i < n; i++){
        int t;
        cin >> t;
        if (i == 0) answer.push_back({i, t});
        else v.push_back({i, t});
    }
    cin >> m;
    for (int i=0; i < m; i++){
        int k;
        cin >> k;
        for (int j=0; j < k; j++){
            int t;
            cin >> t;
            if (answer.size() == 0) answer.push_back({i+n,t});
            else v.push_back({i + n, t});
        }
    }
    answer = solve(answer, v);
    if (answer.size() < n+m){
        cout << -1;
        return 0;
    }
    vector<int> res(m);
    for (int i=0; i < answer.size(); i++){
        if (answer[i].index >= n) res[answer[i].index - n] = answer[i].value;
    }
    for (int i=0; i < m; i++) cout << res[i] << endl;
}
```

Mincut

```
/*
Для наиболее простой и ясной реализации (с асимптотикой O(n^3)) было выбрано
представление графа в виде матрицы смежности. Ответ хранится в переменных \rm
best\_cost и \rm best\_cut (искомые стоимость минимального разреза и сами вершины,
содержащиеся в нём).

Для каждой вершины в массиве \rm exist хранится, существует ли она, или она была
объединена с какой-то другой вершиной. В списке {\rm v}[i] для каждой сжатой вершины
i хранятся номера исходных вершин, которые были сжаты в эту вершину i.

Алгоритм состоит из n-1 фазы (цикл по переменной \rm ph). На каждой фазе сначала все
вершины находятся вне множества A, для чего массив \rm in\_a заполняется нулями, и
связности w всех вершин нулевые. На каждой из n-{\rm ph} итерации находится вершина
\rm sel с наибольшей величиной w. Если это итерация последняя, то ответ, если надо,
обновляется, а предпоследняя \rm prev и последняя \rm sel выбранные вершины
объединяются в одну. Если итерация не последняя, то \rm sel добавляется в множество
A, после чего пересчитываются веса всех остальных вершин.

Следует заметить, что алгоритм в ходе своей работы "портит" граф \rm g, поэтому, если
он ещё понадобится позже, надо сохранять его копию перед вызовом функции.
*/


const int MAXN = 500;
int n, g[MAXN][MAXN];
int best_cost = 1000000000;
vector<int> best_cut;

void mincut() {
        vector<int> v[MAXN];
        for (int i=0; i<n; ++i)
                v[i].assign (1, i);
        int w[MAXN];
        bool exist[MAXN], in_a[MAXN];
        memset (exist, true, sizeof exist);
        for (int ph=0; ph<n-1; ++ph) {
                memset (in_a, false, sizeof in_a);
                memset (w, 0, sizeof w);
                for (int it=0, prev; it<n-ph; ++it) {
                        int sel = -1;
                        for (int i=0; i<n; ++i)
                                if (exist[i] && !in_a[i] && (sel == -1 || w[i] >
w[sel]))
                                        sel = i;
                        if (it == n-ph-1) {
                                if (w[sel] < best_cost)
                                        best_cost = w[sel],  best_cut = v[sel];
                                v[prev].insert (v[prev].end(), v[sel].begin(),
v[sel].end());
                                for (int i=0; i<n; ++i)
                                        g[prev][i] = g[i][prev] += g[sel][i];
                                exist[sel] = false;
                        }
                        else {
                                in_a[sel] = true;
                                for (int i=0; i<n; ++i)
                                        w[i] += g[sel][i];
                                prev = sel;
                        }
                }
        }
}
```

SuffixTree

```
char s[N + 1];
```

```
map<char, int> t[VN];
int l[VN], r[VN], p[VN];
int n = 0, suf[VN], vn = 2, v = 1, pos = 0;

set<int> lens[VN];
fraction ans(1, 1);

int dfs(int v, int len) {
    if (t[v].empty()) {
        lens[v].insert(len);
        return N;
    }
    int md = N;
    for (auto [c, u] : t[v]) {
        md = min(md, dfs(u, len + min(n, r[u]) - l[u]));
        if (lens[v].size() < lens[u].size()) {
            lens[v].swap(lens[u]);
        }
        for (int x : lens[u]) {
            auto it = lens[v].lower_bound(x);
            if (it != lens[v].end()) {
                md = min(md, *it - x);
            }
            if (it != lens[v].begin()) {
                --it;
                md = min(md, x - *it);
            }
            lens[v].insert(x);
        }
    }
    if (md != N) {
        ans = max(ans, fraction(len + md, md));
    }
    return md;
}

int main() {
    string w;
    cin >> w;
    w += '$';
    for (char c = 0; c < 127; c++) {
        t[0][c] = 1;
    }
    l[1] = -1;

    for (n = 0; n < int(w.size()); n++) {
                char c = s[n] = w[n];
                auto new_leaf = [&]( int v ) {
                        p[vn] = v, l[vn] = n, r[vn] = N, t[v][c] = vn++;
                };
                go:;
                if (r[v] <= pos) {
                        if (!t[v].count(c)) {
                                new_leaf(v), v = suf[v], pos = r[v];
                                goto go;
                        }
                        v = t[v][c], pos = l[v] + 1;
                } else if (c == s[pos]) {
                        pos++;
                } else {
                        int x = vn++;
                        l[x] = l[v], r[x] = pos, l[v] = pos;
                        p[x] = p[v], p[v] = x;
                        t[p[x]][s[l[x]]] = x, t[x][s[pos]] = v;
                        new_leaf(x);
                        v = suf[p[x]], pos = l[x];
                        while (pos < r[x])
                                v = t[v][s[pos]], pos += r[v] - l[v];
```

```
                        suf[x] = (pos == r[x] ? v : vn);
                        pos = r[v] - (pos - r[x]);
                        goto go;
                    }
            }

        for (auto [c, v] : t[1]) {
            if (c != '$') {
                dfs(v, min(n, r[v]) - l[v]);
            }
        }

}
```