# 1 DP

## 1.1 divide-and-conquer-optimization

```
int m, n;
vector<long long> dp_before(n), dp_cur(n);
long long C(int i, int j);
// compute dp_cur[l], ... dp_cur[r] (inclusive)
void compute(int l, int r, int optl, int optr){
    if (l > r)
        return;
    int mid = (l + r) >> 1;
    pair<long long, int> best = {LLONG_MAX, -1};
    for (int k = optl; k <= min(mid, optr); k++){
        best = min(best, {(k ? dp_before[k - 1] : 0) +
            C(k, mid), k});
    }
    dp_cur[mid] = best.first;
    int opt = best.second;
    compute(l, mid - 1, optl, opt);
    compute(mid + 1, r, opt, optr);
}
int solve(){
    for (int i = 0; i < n; i++)
        dp_before[i] = C(0, i);
    for (int i = 1; i < m; i++){
        compute(0, n - 1, 0, n - 1);
        dp_before = dp_cur;
    }
    return dp_before[n - 1];
}
```

## 1.2 knuth-optimization

```
int solve() {
    int N;
    ... // read N and input
    int dp[N][N], opt[N][N];
    auto C = [&](int i, int j) {
        ... // Implement cost function C.
    };
    for (int i = 0; i < N; i++) {
        opt[i][i] = i;
        ... // Initialize dp[i][i] according to the
            problem
    }
    for (int i = N-2; i >= 0; i--) {
        for (int j = i+1; j < N; j++) {
            int mn = INT_MAX;
            int cost = C(i, j);
            for (int k = opt[i][j-1]; k <= min(j-1,
                opt[i+1][j]); k++) {
                if (mn >= dp[i][k] + dp[k+1][j] + cost) {
                    opt[i][j] = k;
                    mn = dp[i][k] + dp[k+1][j] + cost;
                }
            }
            dp[i][j] = mn;
        }
    }
    cout << dp[0][N-1] << endl;
}
```

## 1.3 li-chao-tree

```
typedef long long ll;
class LiChaoTree{
    ll L,R;
    bool minimize;
    int lines;
    struct Node{
        complex<ll> line;
        Node *children[2];
        Node(complex<ll> ln= {0,1000000000000000000}){
            line=ln;
            children[0]=0;
            children[1]=0;
        }
    } *root;
    ll dot(complex<ll> a, complex<ll> b){
        return (conj(a) * b).real();
    }
    ll f(complex<ll> a, ll x){
        return dot(a, {x, 1});
    }
    void clear(Node* &node){
        if(node->children[0]){
            clear(node->children[0]);
        }
        if(node->children[1]){
            clear(node->children[1]);
        }
        delete node;
    }
    void add_line(complex<ll> nw, Node* &node, ll l, ll
        r){
        if(node==0){
            node=new Node(nw);
            return;
        }
        ll m = (l + r) / 2;
        bool lef = (f(nw, l) < f(node->line,
            l)&&minimize)||((!minimize)&&f(nw, l) >
            f(node->line, l));
        bool mid = (f(nw, m) < f(node->line,
            m)&&minimize)||((!minimize)&&f(nw, m) >
            f(node->line, m));
        if(mid){
            swap(node->line, nw);
        }
        if(r - l == 1){
            return;
        }
        else if(lef != mid){
            add_line(nw, node->children[0], l, m);
        }
        else{
            add_line(nw, node->children[1], m, r);
        }
    }
    ll get(ll x, Node* &node, ll l, ll r){
        ll m = (l + r) / 2;
        if(r - l == 1){
            return f(node->line, x);
        }
        else if(x < m){
            if(node->children[0]==0) return f(node->line,
                x);
            if(minimize) return min(f(node->line, x),
                get(x, node->children[0], l, m));
            else return max(f(node->line, x), get(x,
                node->children[0], l, m));
        }
        else{
            if(node->children[1]==0) return f(node->line,
                x);
            if(minimize) return min(f(node->line, x),
                get(x, node->children[1], m, r));
            else return max(f(node->line, x), get(x,
                node->children[1], m, r));
        }
    }
public:
    LiChaoTree(ll l=-1000000001,ll r=1000000001,bool
        mn=false){
        L=l;
        R=r;
        root=0;
        minimize=mn;
        lines=0;
    }
    void AddLine(pair<ll,ll> ln){
        add_line({ln.first,ln.second},root,L,R);
        lines++;
    }
    int number_of_lines(){
        return lines;
    }
    ll getOptimumValue(ll x){
        return get(x,root,L,R);
    }
    ~LiChaoTree(){
        if(root!=0) clear(root);
    }
};
```

## 1.4 zero-matrix

```
int zero_matrix(vector<vector<int>> a) {
    int n = a.size();
    int m = a[0].size();
    int ans = 0;
    vector<int> d(m, -1), d1(m), d2(m);
    stack<int> st;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            if (a[i][j] == 1)
                d[j] = i;
        }
        for (int j = 0; j < m; ++j) {
            while (!st.empty() && d[st.top()] <= d[j])
                st.pop();
            d1[j] = st.empty() ? -1 : st.top();
            st.push(j);
        }
        while (!st.empty())
            st.pop();
        for (int j = m - 1; j >= 0; --j) {
            while (!st.empty() && d[st.top()] <= d[j])
                st.pop();
            d2[j] = st.empty() ? m : st.top();
            st.push(j);
        }
        while (!st.empty())
            st.pop();
        for (int j = 0; j < m; ++j)
            ans = max(ans, (i - d[j]) * (d2[j] - d1[j] -
                1));
    }
    return ans;
}
```

# 2 DS

## 2.1 MO$_with_update$

```
const int N = 1e5 +5;
const int P = 2000; //block size = (2*n^2)^(1/3)
struct query{
    int t, l, r, k, i;
};
vector<query> q;
```

```
vector<array<int, 3>> upd;
vector<int> ans;
vector<int>a;
void add(int x);
void rem(int x);
int get_answer();
void mos_algorithm(){
    sort(q.begin(), q.end(), [](const query &a, const
        query &b){
        if (a.t / P != b.t / P)
            return a.t < b.t;
        if (a.l / P != b.l / P)
            return a.l < b.l;
        if ((a.l / P) & 1)
            return a.r < b.r;
        return a.r > b.r;
    });
    for (int i = upd.size() - 1; i >= 0; --i)
        a[upd[i][0]] = upd[i][1];
    int L = 0, R = -1, T = 0;
    auto apply = [&](int i, int fl){
        int p = upd[i][0];
        int x = upd[i][fl + 1];
        if (L <= p && p <= R){
            rem(a[p]);
            add(x);
        }
        a[p] = x;
    };
    ans.clear();
    ans.resize(q.size());
    for (auto qr : q){
        int t = qr.t, l = qr.l, r = qr.r, k = qr.k;
        while (T < t)
            apply(T++, 1);
        while (T > t)
            apply(--T, 0);
        while (R < r)
            add(a[++R]);
        while (L > l)
            add(a[--L]);
        while (R > r)
            rem(a[R--]);
        while (L < l)
            rem(a[L++]);
        ans[qr.i] = get_answer();
    }
}
void TEST_CASES(int cas){
    int n, m;
    cin>>n>>m;
    a.resize(n);
    for(int i=0;i<n;i++){
        cin>>a[i];
    }
    for(int i=0;i<m;i++){
        int tp;
        scanf("%d", &tp);
        if (tp == 1){
            int l, r, k;
            cin>>l>>r>>k;
            q.push_back({upd.size(), l - 1, r - 1, k,
                q.size()});
        }
        else{
            int p, x;
            cin>>p>>x;
            --p;
            upd.push_back({p, a[p], x});
            a[p] = x;
        }
    }
    mos_algorithm();
}
```

## 2.2 bipartite-disjoint-set-union

```
void make_set(int v) {
    parent[v] = make_pair(v, 0);
    rank[v] = 0;
    bipartite[v] = true;
}
pair<int, int> find_set(int v) {
    if (v != parent[v].first) {
        int parity = parent[v].second;
        parent[v] = find_set(parent[v].first);
        parent[v].second ^= parity;
    }
    return parent[v];
}
void add_edge(int a, int b) {
    pair<int, int> pa = find_set(a);
    a = pa.first;
    int x = pa.second;

    pair<int, int> pb = find_set(b);
    b = pb.first;
    int y = pb.second;

    if (a == b) {
        if (x == y)
            bipartite[a] = false;
    } else {
        if (rank[a] < rank[b])
            swap (a, b);
        parent[b] = make_pair(a, x^y^1);
        bipartite[a] &= bipartite[b];
        if (rank[a] == rank[b])
            ++rank[a];
    }
}
bool is_bipartite(int v) {
    return bipartite[find_set(v).first];
}
```

## 2.3 dsu-rollback

```
struct dsu_save {
    int v, rnkv, u, rnku;
    dsu_save() {}
    dsu_save(int _v, int _rnkv, int _u, int _rnku)
        : v(_v), rnkv(_rnkv), u(_u), rnku(_rnku) {}
};
struct dsu_with_rollbacks {
    vector<int> p, rnk;
    int comps;
    stack<dsu_save> op;
    dsu_with_rollbacks() {}
    dsu_with_rollbacks(int n) {
        p.resize(n);
        rnk.resize(n);
        for (int i = 0; i < n; i++) {
            p[i] = i;
            rnk[i] = 0;
        }
        comps = n;
    }
    int find_set(int v) {
        return (v == p[v]) ? v : find_set(p[v]);
    }
    bool unite(int v, int u) {
        v = find_set(v);
        u = find_set(u);
        if (v == u)
            return false;
        comps--;
        if (rnk[v] > rnk[u])
            swap(v, u);
        op.push(dsu_save(v, rnk[v], u, rnk[u]));
        p[v] = u;
        if (rnk[u] == rnk[v])
            rnk[u]++;
        return true;
    }
    void rollback() {
        if (op.empty())
            return;
        dsu_save x = op.top();
        op.pop();
        comps++;
        p[x.v] = x.v;
        rnk[x.v] = x.rnkv;
        p[x.u] = x.u;
        rnk[x.u] = x.rnku;
    }
};
struct query {
    int v, u;
    bool united;
    query(int _v, int _u) : v(_v), u(_u) {
    }
};
struct QueryTree {
    vector<vector<query>> t;
    dsu_with_rollbacks dsu;
    int T;
    QueryTree() {}
    QueryTree(int _T, int n) : T(_T) {
        dsu = dsu_with_rollbacks(n);
        t.resize(4 * T + 4);
    }
    void add_to_tree(int v, int l, int r, int ul, int
        ur, query& q) {
        if (ul > ur)
            return;
        if (l == ul && r == ur) {
            t[v].push_back(q);
            return;
        }
        int mid = (l + r) / 2;
        add_to_tree(2 * v, l, mid, ul, min(ur, mid), q);
        add_to_tree(2 * v + 1, mid + 1, r, max(ul, mid +
            1), ur, q);
    }
    void add_query(query q, int l, int r) {
        add_to_tree(1, 0, T - 1, l, r, q);
    }
    void dfs(int v, int l, int r, vector<int>& ans) {
        for (query& q : t[v]) {
            q.united = dsu.unite(q.v, q.u);
        }
        if (l == r)
            ans[l] = dsu.comps;
        else {
            int mid = (l + r) / 2;
            dfs(2 * v, l, mid, ans);
            dfs(2 * v + 1, mid + 1, r, ans);
        }
        for (query q : t[v]) {
            if (q.united)
                dsu.rollback();
```

```
        }
    }
    vector<int> solve() {
        vector<int> ans(T);
        dfs(1, 0, T - 1, ans);
        return ans;
    }
};
```

## 2.4 mo

```
struct Query {
    int l, r,k, idx;
    bool operator<(Query other) const
    {
        if(l/block_size!=other.l/block_size) return
            (l<other.l);
        return (l/block_size&1)? (r<other.r) :
            (r>other.r);
    }
};
vector<int> mo_s_algorithm(vector<Query> queries) {
    vector<int> answers(queries.size());
    sort(queries.begin(), queries.end());
    // TODO: initialize data structure
    int cur_l = 0;
    int cur_r = -1;
    // invariant: data structure will always reflect the
        range [cur_l, cur_r]
    for (Query q : queries) {
        while (cur_l > q.l) {
            cur_l--;
            add(cur_l);
        }
        while (cur_r < q.r) {
            cur_r++;
            add(cur_r);
        }
        while (cur_l < q.l) {
            remove(cur_l);
            cur_l++;
        }
        while (cur_r > q.r) {
            remove(cur_r);
            cur_r--;
        }
        answers[q.idx] = get_answer();
    }
    return answers;
}
```

## 2.5 treap

```
template <class T>
class treap{
    struct item{
        int prior, cnt;
        T key;
        item *l,*r;
        item(T v)
        {
            key=v;
            l=NULL;
            r=NULL;
            cnt=1;
            prior=rand();
        }
    } *root,*node;
    int cnt (item * it){
        return it ? it->cnt : 0;
    }
    void upd_cnt (item * it){
        if (it) it->cnt = cnt(it->l) + cnt(it->r) + 1;
    }
    void split (item * t, T key, item * & l, item * & r){
        if (!t)
            l = r = NULL;
        else if (key < t->key)
            split (t->l, key, l, t->l), r = t;
        else
            split (t->r, key, t->r, r), l = t;
        upd_cnt(t);
    }
    void insert (item * & t, item * it){
        if (!t)
            t = it;
        else if (it->prior > t->prior)
            split (t, it->key, it->l, it->r), t = it;
        else
            insert (it->key < t->key ? t->l : t->r, it);
        upd_cnt(t);
    }
    void merge (item * & t, item * l, item * r){
        if (!l || !r)
            t = l ? l : r;
        else if (l->prior > r->prior)
            merge (l->r, l->r, r), t = l;
        else
            merge (r->l, l, r->l), t = r;
        upd_cnt(t);
    }
    void erase (item * & t, T key){
        if (t->key == key)
            merge (t, t->l, t->r);
        else
            erase (key < t->key ? t->l : t->r, key);
        upd_cnt(t);
    }
    T elementAt(item * &t,int key){
        T ans;
        if(cnt(t->l)==key) ans=t->key;
        else if(cnt(t->l)>key) ans=elementAt(t->l,key);
        else ans=elementAt(t->r,key-1-cnt(t->l));
        upd_cnt(t);
        return ans;
    }
    item * unite (item * l, item * r){
        if (!l || !r) return l ? l : r;
        if (l->prior < r->prior) swap (l, r);
        item * lt, * rt;
        split (r, l->key, lt, rt);
        l->l = unite (l->l, lt);
        l->r = unite (l->r, rt);
        upd_cnt(l);
        upd_cnt(r);
        return l;
    }
    void heapify (item * t){
        if (!t) return;
        item * max = t;
        if (t->l != NULL && t->l->prior > max->prior)
            max = t->l;
        if (t->r != NULL && t->r->prior > max->prior)
            max = t->r;
        if (max != t)
        {
            swap (t->prior, max->prior);
            heapify (max);
        }
    }
    item * build (T * a, int n){
        if (n == 0) return NULL;
        int mid = n / 2;
        item * t = new item (a[mid], rand ());
        t->l = build (a, mid);
        t->r = build (a + mid + 1, n - mid - 1);
        heapify (t);
        return t;
    }
    void output (item * t,vector<T> &arr){
        if (!t) return;
        output (t->l,arr);
        arr.push_back(t->key);
        output (t->r,arr);
    }
public:
    treap(){
        root=NULL;
    }
    treap(T *a,int n){
        build(a,n);
    }
    void insert(T value){
        node=new item(value);
        insert(root,node);
    }
    void erase(T value){
        erase(root,value);
    }
    T elementAt(int position){
        return elementAt(root,position);
    }
    int size(){
        return cnt(root);
    }
    void output(vector<T> &arr){
        output(root,arr);
    }
    int range_query(T l,T r){ //(l,r]
        item *previous,*next,*current;
        split(root,l,previous,current);
        split(current,r,current,next);
        int ans=cnt(current);
        merge(root,previous,current);
        merge(root,root,next);
        previous=NULL;
        current=NULL;
        next=NULL;
        return ans;
    }
};
template <class T>
class implicit_treap{
    struct item{
        int prior, cnt;
        T value;
        bool rev;
        item *l,*r;
        item(T v){
            value=v;
            rev=false;
            l=NULL;
            r=NULL;
            cnt=1;
            prior=rand();
        }
    } *root,*node;
    int cnt (item * it){
        return it ? it->cnt : 0;
    }
    void upd_cnt (item * it){
        if (it)
            it->cnt = cnt(it->l) + cnt(it->r) + 1;
```

```
    }
    void push (item * it){
        if (it && it->rev){
            it->rev = false;
            swap (it->l, it->r);
            if (it->l) it->l->rev ^= true;
            if (it->r) it->r->rev ^= true;
        }
    }
    void merge (item * & t, item * l, item * r){
        push (l);
        push (r);
        if (!l || !r)
            t = l ? l : r;
        else if (l->prior > r->prior)
            merge (l->r, l->r, r), t = l;
        else
            merge (r->l, l, r->l), t = r;
        upd_cnt (t);
    }
    void split (item * t, item * & l, item * & r, int
     key, int add = 0){
        if (!t)
            return void( l = r = 0 );
        push (t);
        int cur_key = add + cnt(t->l);
        if (key <= cur_key)
            split (t->l, l, t->l, key, add), r = t;
        else
            split (t->r, t->r, r, key, add + 1 +
                cnt(t->l)), l = t;
        upd_cnt (t);
    }
    void insert(item * &t,item * element,int key){
        item *l,*r;
        split(t,l,r,key);
        merge(l,l,element);
        merge(t,l,r);
        l=NULL;
        r=NULL;
    }
    T elementAt(item * &t,int key){
        push(t);
        T ans;
        if(cnt(t->l)==key) ans=t->value;
        else if(cnt(t->l)>key) ans=elementAt(t->l,key);
        else ans=elementAt(t->r,key-1-cnt(t->l));
        return ans;
    }
    void erase (item * & t, int key){
        push(t);
        if(!t) return;
        if (key == cnt(t->l))
            merge (t, t->l, t->r);
        else if(key<cnt(t->l))
            erase(t->l,key);
        else
            erase(t->r,key-cnt(t->l)-1);
        upd_cnt(t);
    }
    void reverse (item * &t, int l, int r){
        item *t1, *t2, *t3;
        split (t, t1, t2, l);
        split (t2, t2, t3, r-l+1);
        t2->rev ^= true;
        merge (t, t1, t2);
        merge (t, t, t3);
    }
    void cyclic_shift(item * &t,int L,int R){
        if(L==R) return;
        item *l,*r,*m;
        split(t,t,l,L);
        split(l,l,m,R-L+1);
        split(l,l,r,R-L);
        merge(t,t,r);
        merge(t,t,l);
        merge(t,t,m);
        l=NULL;
        r=NULL;
        m=NULL;
    }
    void output (item * t,vector<T> &arr){
        if (!t) return;
        push (t);
        output (t->l,arr);
        arr.push_back(t->value);
        output (t->r,arr);
    }
public:
    implicit_treap(){
        root=NULL;
    }
    void insert(T value,int position){
        node=new item(value);
        insert(root,node,position);
    }
    void erase(int position){
        erase(root,position);
    }
    void reverse(int l,int r){
        reverse(root,l,r);
    }
    T elementAt(int position){
        return elementAt(root,position);
    }
    void cyclic_shift(int L,int R){
        cyclic_shift(root,L,R);
    }
    int size(){
        return cnt(root);
    }
    void output(vector<T> &arr){
        output(root,arr);
    }
};
```

# 3 Geo

## 3.1 basic-area-geometry

```
struct point2d {
    ftype x, y;
    point2d() {}
    point2d(ftype x, ftype y): x(x), y(y) {}
    point2d& operator+=(const point2d &t) {
        x += t.x;
        y += t.y;
        return *this;
    }
    point2d& operator-=(const point2d &t) {
        x -= t.x;
        y -= t.y;
        return *this;
    }
    point2d& operator*=(ftype t) {
        x *= t;
        y *= t;
        return *this;
    }
    point2d& operator/=(ftype t) {
        x /= t;
        y /= t;
        return *this;
    }
    point2d operator+(const point2d &t) const {
        return point2d(*this) += t;
    }
    point2d operator-(const point2d &t) const {
        return point2d(*this) -= t;
    }
    point2d operator*(ftype t) const {
        return point2d(*this) *= t;
    }
    point2d operator/(ftype t) const {
        return point2d(*this) /= t;
    }
};
point2d operator*(ftype a, point2d b) {
    return b * a;
}
struct point3d {
    ftype x, y, z;
    point3d() {}
    point3d(ftype x, ftype y, ftype z): x(x), y(y), z(z)
      {}
    point3d& operator+=(const point3d &t) {
        x += t.x;
        y += t.y;
        z += t.z;
        return *this;
    }
    point3d& operator-=(const point3d &t) {
        x -= t.x;
        y -= t.y;
        z -= t.z;
        return *this;
    }
    point3d& operator*=(ftype t) {
        x *= t;
        y *= t;
        z *= t;
        return *this;
    }
    point3d& operator/=(ftype t) {
        x /= t;
        y /= t;
        z /= t;
        return *this;
    }
    point3d operator+(const point3d &t) const {
        return point3d(*this) += t;
    }
    point3d operator-(const point3d &t) const {
        return point3d(*this) -= t;
    }
    point3d operator*(ftype t) const {
        return point3d(*this) *= t;
    }
    point3d operator/(ftype t) const {
        return point3d(*this) /= t;
    }
};
point3d operator*(ftype a, point3d b) {
    return b * a;
}
ftype dot(point2d a, point2d b) {
    return a.x * b.x + a.y * b.y;
}
ftype dot(point3d a, point3d b) {
    return a.x * b.x + a.y * b.y + a.z * b.z;
}
```

```
ftype norm(point2d a) {
    return dot(a, a);
}
double abs(point2d a) {
    return sqrt(norm(a));
}
double proj(point2d a, point2d b) {
    return dot(a, b) / abs(b);
}
double angle(point2d a, point2d b) {
    return acos(dot(a, b) / abs(a) / abs(b));
}
point3d cross(point3d a, point3d b) {
    return point3d(a.y * b.z - a.z * b.y,
                   a.z * b.x - a.x * b.z,
                   a.x * b.y - a.y * b.x);
}
ftype triple(point3d a, point3d b, point3d c) {
    return dot(a, cross(b, c));
}
ftype cross(point2d a, point2d b) {
    return a.x * b.y - a.y * b.x;
}
point2d intersect(point2d a1, point2d d1, point2d a2,
    point2d d2) {
    return a1 + cross(a2 - a1, d2) / cross(d1, d2) * d1;
}
point3d intersect(point3d a1, point3d n1, point3d a2,
    point3d n2, point3d a3, point3d n3) {
    point3d x(n1.x, n2.x, n3.x);
    point3d y(n1.y, n2.y, n3.y);
    point3d z(n1.z, n2.z, n3.z);
    point3d d(dot(a1, n1), dot(a2, n2), dot(a3, n3));
    return point3d(triple(d, y, z),
                   triple(x, d, z),
                   triple(x, y, d)) / triple(n1, n2, n3);
}
int signed_area_parallelogram(point2d p1, point2d p2,
    point2d p3) {
    return cross(p2 - p1, p3 - p2);
}

double triangle_area(point2d p1, point2d p2, point2d p3)
    {
    return abs(signed_area_parallelogram(p1, p2, p3)) /
        2.0;
}

bool clockwise(point2d p1, point2d p2, point2d p3) {
    return signed_area_parallelogram(p1, p2, p3) < 0;
}

bool counter_clockwise(point2d p1, point2d p2, point2d
    p3) {
    return signed_area_parallelogram(p1, p2, p3) > 0;
}
double area(const vector<point>& fig) {
    double res = 0;
    for (unsigned i = 0; i < fig.size(); i++) {
        point p = i ? fig[i - 1] : fig.back();
        point q = fig[i];
        res += (p.x - q.x) * (p.y + q.y);
    }
    return fabs(res) / 2;
}
//Pick: S = I + B/2 - 1
int count_lattices(Fraction k, Fraction b, long long n) {
    auto fk = k.floor();
    auto fb = b.floor();
    auto cnt = 0LL;
    if (k >= 1 || b >= 1) {
        cnt += (fk * (n - 1) + 2 * fb) * n / 2;
        k -= fk;
        b -= fb;
    }
    auto t = k * n + b;
    auto ft = t.floor();
    if (ft >= 1) {
        cnt += count_lattices(1 / k, (t - t.floor()) /
            k, t.floor());
    }
    return cnt;
}
```

## 3.2 delaunay-voronoi

```
typedef long long ll;
bool ge(const ll& a, const ll& b) { return a >= b; }
bool le(const ll& a, const ll& b) { return a <= b; }
bool eq(const ll& a, const ll& b) { return a == b; }
bool gt(const ll& a, const ll& b) { return a > b; }
bool lt(const ll& a, const ll& b) { return a < b; }
int sgn(const ll& a) { return a >= 0 ? a ? 1 : 0 : -1; }
struct pt {
    ll x, y;
    pt() { }
    pt(ll _x, ll _y) : x(_x), y(_y) { }
    pt operator-(const pt& p) const {
        return pt(x - p.x, y - p.y);
    }
    ll cross(const pt& p) const {
        return x * p.y - y * p.x;
    }
    ll cross(const pt& a, const pt& b) const {
        return (a - *this).cross(b - *this);
    }
    ll dot(const pt& p) const {
        return x * p.x + y * p.y;
    }
    ll dot(const pt& a, const pt& b) const {
        return (a - *this).dot(b - *this);
    }
    ll sqrLength() const {
        return this->dot(*this);
    }
    bool operator==(const pt& p) const {
        return eq(x, p.x) && eq(y, p.y);
    }
};
const pt inf_pt = pt(1e18, 1e18);
struct QuadEdge {
    pt origin;
    QuadEdge* rot = nullptr;
    QuadEdge* onext = nullptr;
    bool used = false;
    QuadEdge* rev() const {
        return rot->rot;
    }
    QuadEdge* lnext() const {
        return rot->rev()->onext->rot;
    }
    QuadEdge* oprev() const {
        return rot->onext->rot;
    }
    pt dest() const {
        return rev()->origin;
    }
};
QuadEdge* make_edge(pt from, pt to) {
    QuadEdge* e1 = new QuadEdge;
    QuadEdge* e2 = new QuadEdge;
    QuadEdge* e3 = new QuadEdge;
    QuadEdge* e4 = new QuadEdge;
    e1->origin = from;
    e2->origin = to;
    e3->origin = e4->origin = inf_pt;
    e1->rot = e3;
    e2->rot = e4;
    e3->rot = e2;
    e4->rot = e1;
    e1->onext = e1;
    e2->onext = e2;
    e3->onext = e4;
    e4->onext = e3;
    return e1;
}
void splice(QuadEdge* a, QuadEdge* b) {
    swap(a->onext->rot->onext, b->onext->rot->onext);
    swap(a->onext, b->onext);
}
void delete_edge(QuadEdge* e) {
    splice(e, e->oprev());
    splice(e->rev(), e->rev()->oprev());
    delete e->rev()->rot;
    delete e->rev();
    delete e->rot;
    delete e;
}
QuadEdge* connect(QuadEdge* a, QuadEdge* b) {
    QuadEdge* e = make_edge(a->dest(), b->origin);
    splice(e, a->lnext());
    splice(e->rev(), b);
    return e;
}
bool left_of(pt p, QuadEdge* e) {
    return gt(p.cross(e->origin, e->dest()), 0);
}
bool right_of(pt p, QuadEdge* e) {
    return lt(p.cross(e->origin, e->dest()), 0);
}
template <class T>
T det3(T a1, T a2, T a3, T b1, T b2, T b3, T c1, T c2, T
    c3) {
    return a1 * (b2 * c3 - c2 * b3) - a2 * (b1 * c3 - c1
        * b3) +
          a3 * (b1 * c2 - c1 * b2);
}
bool in_circle(pt a, pt b, pt c, pt d) {
// If there is __int128, calculate directly.
// Otherwise, calculate angles.
#if defined(__LP64__) || defined(_WIN64)
    __int128 det = -det3<__int128>(b.x, b.y,
        b.sqrLength(), c.x, c.y,
                              c.sqrLength(), d.x, d.y,
                                  d.sqrLength());
    det += det3<__int128>(a.x, a.y, a.sqrLength(), c.x,
        c.y, c.sqrLength(), d.x,
                      d.y, d.sqrLength());
    det -= det3<__int128>(a.x, a.y, a.sqrLength(), b.x,
        b.y, b.sqrLength(), d.x,
                      d.y, d.sqrLength());
    det += det3<__int128>(a.x, a.y, a.sqrLength(), b.x,
        b.y, b.sqrLength(), c.x,
                      c.y, c.sqrLength());
    return det > 0;
#else
    auto ang = [](pt l, pt mid, pt r) {
        ll x = mid.dot(l, r);
        ll y = mid.cross(l, r);
```

```
        long double res = atan2((long double)x, (long
            double)y);
        return res;
    };
    long double kek = ang(a, b, c) + ang(c, d, a) -
        ang(b, c, d) - ang(d, a, b);
    if (kek > 1e-8)
        return true;
    else
        return false;
#endif
}
pair<QuadEdge*, QuadEdge*> build_tr(int l, int r,
    vector<pt>& p) {
    if (r - l + 1 == 2) {
        QuadEdge* res = make_edge(p[l], p[r]);
        return make_pair(res, res->rev());
    }
    if (r - l + 1 == 3) {
        QuadEdge *a = make_edge(p[l], p[l + 1]), *b =
            make_edge(p[l + 1], p[r]);
        splice(a->rev(), b);
        int sg = sgn(p[l].cross(p[l + 1], p[r]));
        if (sg == 0)
            return make_pair(a, b->rev());
        QuadEdge* c = connect(b, a);
        if (sg == 1)
            return make_pair(a, b->rev());
        else
            return make_pair(c->rev(), c);
    }
    int mid = (l + r) / 2;
    QuadEdge *ldo, *ldi, *rdo, *rdi;
    tie(ldo, ldi) = build_tr(l, mid, p);
    tie(rdi, rdo) = build_tr(mid + 1, r, p);
    while (true) {
        if (left_of(rdi->origin, ldi)) {
            ldi = ldi->lnext();
            continue;
        }
        if (right_of(ldi->origin, rdi)) {
            rdi = rdi->rev()->onext;
            continue;
        }
        break;
    }
    QuadEdge* basel = connect(rdi->rev(), ldi);
    auto valid = [&basel](QuadEdge* e) { return
        right_of(e->dest(), basel); };
    if (ldi->origin == ldo->origin)
        ldo = basel->rev();
    if (rdi->origin == rdo->origin)
        rdo = basel;
    while (true) {
        QuadEdge* lcand = basel->rev()->onext;
        if (valid(lcand)) {
            while (in_circle(basel->dest(),
                basel->origin, lcand->dest(),
                          lcand->onext->dest())) {
                QuadEdge* t = lcand->onext;
                delete_edge(lcand);
                lcand = t;
            }
        }
        QuadEdge* rcand = basel->oprev();
        if (valid(rcand)) {
            while (in_circle(basel->dest(),
                basel->origin, rcand->dest(),
                          rcand->oprev()->dest())) {
                QuadEdge* t = rcand->oprev();
                delete_edge(rcand);
                rcand = t;
            }
        }
        if (!valid(lcand) && !valid(rcand))
            break;
        if (!valid(lcand) ||
            (valid(rcand) && in_circle(lcand->dest(),
                lcand->origin,
                                rcand->origin,
                                    rcand->dest())))
            basel = connect(rcand, basel->rev());
        else
            basel = connect(basel->rev(), lcand->rev());
    }
    return make_pair(ldo, rdo);
}
vector<tuple<pt, pt, pt>> delaunay(vector<pt> p) {
    sort(p.begin(), p.end(), [](const pt& a, const pt&
        b) {
        return lt(a.x, b.x) || (eq(a.x, b.x) && lt(a.y,
            b.y));
    });
    auto res = build_tr(0, (int)p.size() - 1, p);
    QuadEdge* e = res.first;
    vector<QuadEdge*> edges = {e};
    while (lt(e->onext->dest().cross(e->dest(),
        e->origin), 0))
        e = e->onext;
    auto add = [&p, &e, &edges]() {
        QuadEdge* curr = e;
        do {
            curr->used = true;
            p.push_back(curr->origin);
            edges.push_back(curr->rev());
            curr = curr->lnext();
        } while (curr != e);
    };
    add();
    p.clear();
    int kek = 0;
    while (kek < (int)edges.size()) {
        if (!(e = edges[kek++])->used)
            add();
    }
    vector<tuple<pt, pt, pt>> ans;
    for (int i = 0; i < (int)p.size(); i += 3) {
        ans.push_back(make_tuple(p[i], p[i + 1], p[i +
            2]));
    }
    return ans;
}
```

### 3.3 half-plane-intersection

```
class HalfPlaneIntersection{
    static double eps, inf;
public:
    struct Point{
        double x, y;
        explicit Point(double x = 0, double y = 0) :
            x(x), y(y) {}
        // Addition, substraction, multiply by constant,
            cross product.
        friend Point operator + (const Point& p, const
            Point& q){
            return Point(p.x + q.x, p.y + q.y);
        }
        friend Point operator - (const Point& p, const
            Point& q){
            return Point(p.x - q.x, p.y - q.y);
        }
        friend Point operator * (const Point& p, const
            double& k){
            return Point(p.x * k, p.y * k);
        }
        friend double cross(const Point& p, const Point&
            q){
            return p.x * q.y - p.y * q.x;
        }
    };
// Basic half-plane struct.
    struct Halfplane{
        // 'p' is a passing point of the line and 'pq'
            is the direction vector of the line.
        Point p, pq;
        double angle;
        Halfplane() {}
        Halfplane(const Point& a, const Point& b) :
            p(a), pq(b - a){
            angle = atan2l(pq.y, pq.x);
        }
        // Check if point 'r' is outside this half-plane.
        // Every half-plane allows the region to the
            LEFT of its line.
        bool out(const Point& r){
            return cross(pq, r - p) < -eps;
        }
        // Comparator for sorting.
        // If the angle of both half-planes is equal,
            the leftmost one should go first.
        bool operator < (const Halfplane& e) const{
            if (fabsl(angle - e.angle) < eps) return
                cross(pq, e.p - p) < 0;
            return angle < e.angle;
        }
        // We use equal comparator for std::unique to
            easily remove parallel half-planes.
        bool operator == (const Halfplane& e) const{
            return fabsl(angle - e.angle) < eps;
        }
        // Intersection point of the lines of two
            half-planes. It is assumed they're never
            parallel.
        friend Point inter(const Halfplane& s, const
            Halfplane& t){
            double alpha = cross((t.p - s.p), t.pq) /
                cross(s.pq, t.pq);
            return s.p + (s.pq * alpha);
        }
    };
    static vector<Point> hp_intersect(vector<Halfplane>&
        H){
        Point box[4] =  // Bounding box in CCW order{
            Point(inf, inf),
            Point(-inf, inf),
            Point(-inf, -inf),
            Point(inf, -inf)
        };
        for(int i = 0; i<4; i++) // Add bounding box
            half-planes.{
            Halfplane aux(box[i], box[(i+1) % 4]);
            H.push_back(aux);
        }
        // Sort and remove duplicates
        sort(H.begin(), H.end());
        H.erase(unique(H.begin(), H.end()), H.end());
        deque<Halfplane> dq;
        int len = 0;
```

```
        for(int i = 0; i < int(H.size()); i++){
            // Remove from the back of the deque while
                last half-plane is redundant
            while (len > 1 && H[i].out(inter(dq[len-1],
                dq[len-2]))){
                dq.pop_back();
                --len;
            }
            // Remove from the front of the deque while
                first half-plane is redundant
            while (len > 1 && H[i].out(inter(dq[0],
                dq[1]))){
                dq.pop_front();
                --len;
            }
            // Add new half-plane
            dq.push_back(H[i]);
            ++len;
        }
        // Final cleanup: Check half-planes at the front
            against the back and vice-versa
        while (len > 2 && dq[0].out(inter(dq[len-1],
            dq[len-2]))){
            dq.pop_back();
            --len;
        }
        while (len > 2 && dq[len-1].out(inter(dq[0],
            dq[1]))){
            dq.pop_front();
            --len;
        }
        // Report empty intersection if necessary
        if (len < 3) return vector<Point>();
        // Reconstruct the convex polygon from the
            remaining half-planes.
        vector<Point> ret(len);
        for(int i = 0; i+1 < len; i++){
            ret[i] = inter(dq[i], dq[i+1]);
        }
        ret.back() = inter(dq[len-1], dq[0]);
        return ret;
    }
};
double HalfPlaneIntersection::eps=1e-9;
double HalfPlaneIntersection::inf=1e9;
```

## 3.4 heart-of-geometry-2d

```
typedef double ftype;
const double EPS = 1E-9;
struct pt{
    ftype x, y;
    int id;
    pt() {}
    pt(ftype _x, ftype _y):x(_x), y(_y) {}
    pt operator+(const pt & p) const{
        return pt(x + p.x, y + p.y);
    }
    pt operator-(const pt & p) const{
        return pt(x - p.x, y - p.y);
    }
    ftype cross(const pt & p) const{
        return x * p.y - y * p.x;
    }
    ftype dot(const pt & p) const{
        return x * p.x + y * p.y;
    }
    ftype cross(const pt & a, const pt & b) const{
        return (a - *this).cross(b - *this);
    }
    ftype dot(const pt & a, const pt & b) const{
        return (a - *this).dot(b - *this);
    }
    ftype sqrLen() const{
        return this->dot(*this);
    }
    bool operator<(const pt& p) const{
        return x < p.x - EPS || (abs(x - p.x) < EPS && y
            < p.y - EPS);
    }
    bool operator==(const pt& p) const{
        return abs(x-p.x)<EPS && abs(y-p.y)<EPS;
    }
};
int sign(double x) { return (x > EPS) - (x < -EPS); }
inline int orientation(pt a, pt b, pt c) { return
    sign(a.cross(b,c)); }
bool is_point_on_seg(pt a, pt b, pt p) {
    if (fabs(b.cross(p,a)) < EPS) {
        if (p.x < min(a.x, b.x) - EPS || p.x > max(a.x,
            b.x) + EPS) return false;
        if (p.y < min(a.y, b.y) - EPS || p.y > max(a.y,
            b.y) + EPS) return false;
        return true;
    }
    return false;
}
bool is_point_on_polygon(vector<pt> &p, const pt& z) {
    int n = p.size();
    for (int i = 0; i < n; i++) {
        if (is_point_on_seg(p[i], p[(i + 1) % n], z))
            return 1;
    }
    return 0;
}
int winding_number(vector<pt> &p, const pt& z) { // O(n)
    if (is_point_on_polygon(p, z)) return 1e9;
    int n = p.size(), ans = 0;
    for (int i = 0; i < n; ++i) {
        int j = (i + 1) % n;
        bool below = p[i].y < z.y;
        if (below != (p[j].y < z.y)) {
            auto orient = orientation(z, p[j], p[i]);
            if (orient == 0) return 0;
            if (below == (orient > 0)) ans += below ? -1
                : 1;
        }
    }
    return ans;
}
double dist_sqr(pt a,pt b){
    return ((a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y));
}
double dist(pt a, pt b){
    return sqrt((a.x-b.x)*(a.x-b.x) +
        (a.y-b.y)*(a.y-b.y));
}
double angle(pt a,pt b,pt c){
    if(b==a || b==c) return 0;
    double A2 = dist_sqr(b,c);
    double C2 = dist_sqr(a,b);
    double B2 = dist_sqr(c,a);
    double A = sqrt(A2), C = sqrt(C2);
    double ans = (A2 + C2 - B2)/(A*C*2);
    if(ans<-1) ans=acos(-1);
    else if(ans>1) ans=acos(1);
    else ans = acos(ans);
    return ans;
}
bool cmp(pt a, pt b){
    return a.x < b.x || (a.x == b.x && a.y < b.y);
}
bool ccw(pt a, pt b, pt c, bool include_collinear=false)
    {
    int o = orientation(a, b, c);
    return o > 0 || (include_collinear && o == 0);
}
bool cw(pt a, pt b, pt c, bool include_collinear=false) {
    int o = orientation(a, b, c);
    return o < 0 || (include_collinear && o == 0);
}
bool collinear(pt a, pt b, pt c) { return orientation(a,
    b, c) == 0; }
double area(pt a, pt b, pt c){
    return (a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y))/2;
}
struct cmp_x{
    bool operator()(const pt & a, const pt & b) const{
        return a.x < b.x || (a.x == b.x && a.y < b.y);
    }
};
struct cmp_y{
    bool operator()(const pt & a, const pt & b) const{
        return a.y < b.y || (a.y == b.y && a.x < b.x);
    }
};
struct circle : pt {
    ftype r;
};
bool insideCircle(circle c, pt p){
    return dist_sqr(c,p) <= c.r*c.r + EPS;
}
struct line {
    ftype a, b, c;
    line() {}
    line(pt p, pt q){
        a = p.y - q.y;
        b = q.x - p.x;
        c = -a * p.x - b * p.y;
        norm();
    }
    void norm(){
        double z = sqrt(a * a + b * b);
        if (abs(z) > EPS)
            a /= z, b /= z, c /= z;
    }
    line getParallel(pt p){
        line ans = *this;
        ans.c = -(ans.a*p.x+ans.b*p.y);
        return ans;
    }
    ftype getValue(pt p){
        return a*p.x+b*p.y+c;
    }
    line getPerpend(pt p){
        line ans;
        ans.a=this->b;
        ans.b=-(this->a);
        ans.c = -(ans.a*p.x+ans.b*p.y);
        return ans;
    }
    //dist formula is wrong but don't change
    double dist(pt p) const { return a * p.x + b * p.y +
        c; }
};
double sqr (double a) {
    return a * a;
```

```cpp
}
double det(double a, double b, double c, double d) {
    return a*d - b*c;
}
bool intersect(line m, line n, pt & res) {
    double zn = det(m.a, m.b, n.a, n.b);
    if (abs(zn) < EPS)
        return false;
    res.x = -det(m.c, m.b, n.c, n.b) / zn;
    res.y = -det(m.a, m.c, n.a, n.c) / zn;
    return true;
}
bool parallel(line m, line n) {
    return abs(det(m.a, m.b, n.a, n.b)) < EPS;
}
bool equivalent(line m, line n) {
    return abs(det(m.a, m.b, n.a, n.b)) < EPS
        && abs(det(m.a, m.c, n.a, n.c)) < EPS
        && abs(det(m.b, m.c, n.b, n.c)) < EPS;
}
double det(double a, double b, double c, double d){
    return a * d - b * c;
}
inline bool betw(double l, double r, double x){
    return min(l, r) <= x + EPS && x <= max(l, r) + EPS;
}
inline bool intersect_1d(double a, double b, double c,
     double d){
    if (a > b)
        swap(a, b);
    if (c > d)
        swap(c, d);
    return max(a, c) <= min(b, d) + EPS;
}
bool intersect_segment(pt a, pt b, pt c, pt d, pt& left,
     pt& right){
    if (!intersect_1d(a.x, b.x, c.x, d.x) ||
         !intersect_1d(a.y, b.y, c.y, d.y))
        return false;
    line m(a, b);
    line n(c, d);
    double zn = det(m.a, m.b, n.a, n.b);
    if (abs(zn) < EPS) {
        if (abs(m.dist(c)) > EPS || abs(n.dist(a)) > EPS)
            return false;
        if (b < a)
            swap(a, b);
        if (d < c)
            swap(c, d);
        left = max(a, c);
        right = min(b, d);
        return true;
    } else {
        left.x = right.x = -det(m.c, m.b, n.c, n.b) / zn;
        left.y = right.y = -det(m.a, m.c, n.a, n.c) / zn;
        return betw(a.x, b.x, left.x) && betw(a.y, b.y,
             left.y) &&
               betw(c.x, d.x, left.x) && betw(c.y, d.y,
                   left.y);
    }
}
void tangents (pt c, double r1, double r2, vector<line>
     & ans) {
    double r = r2 - r1;
    double z = sqr(c.x) + sqr(c.y);
    double d = z - sqr(r);
    if (d < -EPS) return;
    d = sqrt (abs (d));
    line l;
    l.a = (c.x * r + c.y * d) / z;
    l.b = (c.y * r - c.x * d) / z;
    l.c = r1;
    ans.push_back (l);
}
vector<line> tangents (circle a, circle b) {
    vector<line> ans;
    for (int i=-1; i<=1; i+=2)
        for (int j=-1; j<=1; j+=2)
            tangents (b-a, a.r*i, b.r*j, ans);
    for (size_t i=0; i<ans.size(); ++i)
        ans[i].c -= ans[i].a * a.x + ans[i].b * a.y;
    return ans;
}
class pointLocationInPolygon{
    bool lexComp(const pt & l, const pt & r){
        return l.x < r.x || (l.x == r.x && l.y < r.y);
    }
    int sgn(ftype val){
        return val > 0 ? 1 : (val == 0 ? 0 : -1);
    }
    vector<pt> seq;
    int n;
    pt translate;
    bool pointInTriangle(pt a, pt b, pt c, pt point){
        ftype s1 = abs(a.cross(b, c));
        ftype s2 = abs(point.cross(a, b)) +
            abs(point.cross(b, c)) + abs(point.cross(c,
            a));
        return s1 == s2;
    }
public:
    pointLocationInPolygon(){
    }
    pointLocationInPolygon(vector<pt> & points){
        prepare(points);
    }
    void prepare(vector<pt> & points){
        seq.clear();
        n = points.size();
        int pos = 0;
        for(int i = 1; i < n; i++){
            if(lexComp(points[i], points[pos]))
                pos = i;
        }
        translate.x=points[pos].x;
        translate.y=points[pos].y;
        rotate(points.begin(), points.begin() + pos,
            points.end());
        n--;
        seq.resize(n);
        for(int i = 0; i < n; i++)
            seq[i] = points[i + 1] - points[0];
    }
    bool pointInConvexPolygon(pt point){
        point.x-=translate.x;
        point.y-=translate.y;
        if(seq[0].cross(point) != 0 &&
            sgn(seq[0].cross(point)) !=
            sgn(seq[0].cross(seq[n - 1])))
            return false;
        if(seq[n - 1].cross(point) != 0 && sgn(seq[n -
            1].cross(point)) != sgn(seq[n -
            1].cross(seq[0])))
            return false;
        if(seq[0].cross(point) == 0)
            return seq[0].sqrLen() >= point.sqrLen();
        int l = 0, r = n - 1;
        while(r - l > 1){
            int mid = (l + r)/2;
            int pos = mid;
            if(seq[pos].cross(point) >= 0)l = mid;
            else r = mid;
        }
        int pos = l;
        return pointInTriangle(seq[pos], seq[pos + 1],
             pt(0, 0), point);
    }
    ~pointLocationInPolygon(){
        seq.clear();
    }
};
class Minkowski{
    static void reorder_polygon(vector<pt> & P){
        size_t pos = 0;
        for(size_t i = 1; i < P.size(); i++){
            if(P[i].y < P[pos].y || (P[i].y == P[pos].y
                && P[i].x < P[pos].x))
                pos = i;
        }
        rotate(P.begin(), P.begin() + pos, P.end());
    }
public:
    static vector<pt> minkowski(vector<pt> P, vector<pt>
        Q){
        // the first vertex must be the lowest
        reorder_polygon(P);
        reorder_polygon(Q);
        // we must ensure cyclic indexing
        P.push_back(P[0]);
        P.push_back(P[1]);
        Q.push_back(Q[0]);
        Q.push_back(Q[1]);
        // main part
        vector<pt> result;
        size_t i = 0, j = 0;
        while(i < P.size() - 2 || j < Q.size() - 2){
            result.push_back(P[i] + Q[j]);
            auto cross = (P[i + 1] - P[i]).cross(Q[j + 1]
                - Q[j]);
            if(cross >= 0)
                ++i;
            if(cross <= 0)
                ++j;
        }
        return result;
    }
};
vector<pt> circle_line_intersections(circle cir,line l){
    double r = cir.r, a = l.a, b = l.b, c = l.c +
        l.a*cir.x + l.b*cir.y;
    vector<pt> ans;
    double x0 = -a*c/(a*a+b*b), y0 = -b*c/(a*a+b*b);
    if (c*c > r*r*(a*a+b*b)+EPS);
    else if (abs (c*c - r*r*(a*a+b*b)) < EPS){
        pt p;
        p.x=x0;
        p.y=y0;
        ans.push_back(p);
    }
    else{
        double d = r*r - c*c/(a*a+b*b);
        double mult = sqrt (d / (a*a+b*b));
        double ax, ay, bx, by;
        ax = x0 + b * mult;
        bx = x0 - b * mult;
        ay = y0 - a * mult;
        by = y0 + a * mult;
        pt p;
        p.x = ax;
        p.y = ay;
        ans.push_back(p);
```

```
        p.x = bx;
        p.y = by;
        ans.push_back(p);
    }
    for(int i=0;i<ans.size();i++){
        ans[i] = ans[i] + cir;
    }
    return ans;
}
double circle_polygon_intersection(circle c,vector<pt>
     &V){
    int n = V.size();
    double ans = 0;
    for(int i=0; i<n; i++){
        line l(V[i],V[(i+1)%n]);
        vector<pt> lpts = circle_line_intersections(c,l);
        int sz=lpts.size();
        for(int j=sz-1; j>=0; j--){
            if(!is_point_on_seg(V[i],V[(i+1)%n],lpts[j])){
                swap(lpts.back(),lpts[j]);
                lpts.pop_back();
            }
        }
        lpts.push_back(V[i]);
        lpts.push_back(V[(i+1)%n]);
        sort(lpts.begin(),lpts.end());
        sz=lpts.size();
        if(V[(i+1)%n]<V[i])
            reverse(lpts.begin(),lpts.end());
        for(int j=1; j<sz; j++){
            if(insideCircle(c,lpts[j-1])
               &&insideCircle(c,lpts[j]))
                ans = ans + area(lpts[j-1],lpts[j],c);
            else{
                double ang = angle(lpts[j-1],c,lpts[j]);
                double aa = c.r*c.r*ang/2;
                if(ccw(lpts[j-1],lpts[j],c))
                    ans = ans+aa;
                else
                    ans = ans-aa;
            }
        }
    }
    ans = abs(ans);
    return ans;
}
void convex_hull(vector<pt>& a, bool include_collinear =
     false) {
    pt p0 = *min_element(a.begin(), a.end(), [](pt a, pt
         b) {
        return make_pair(a.y, a.x) < make_pair(b.y, b.x);
    });
    sort(a.begin(), a.end(), [&p0](const pt& a, const
         pt& b) {
        int o = orientation(p0, a, b);
        if (o == 0)
            return (p0.x-a.x)*(p0.x-a.x) +
                (p0.y-a.y)*(p0.y-a.y)
                < (p0.x-b.x)*(p0.x-b.x) +
                  (p0.y-b.y)*(p0.y-b.y);
        return o < 0;
    });
    if (include_collinear) {
        int i = (int)a.size()-1;
        while (i >= 0 && collinear(p0, a[i], a.back()))
            i--;
        reverse(a.begin()+i+1, a.end());
    }
    vector<pt> st;
    for (int i = 0; i < (int)a.size(); i++) {
        while (st.size() > 1 && !cw(st[st.size()-2],
             st.back(), a[i], include_collinear))
            st.pop_back();
        st.push_back(a[i]);
    }
    a = st;
    int m = a.size();
    for(int i = 0;i<m-1-i;i++){
        swap(a[i],a[m-1-i]);
    }
}
double mindist;
pair<int,pair<int, int> > best_pair;
void upd_ans(const pt & a, const pt & b,const pt & c){
    double distC = sqrt((a.x - b.x)*(a.x - b.x) + (a.y -
         b.y)*(a.y - b.y));
    double distA = sqrt((c.x - b.x)*(c.x - b.x) + (c.y -
         b.y)*(c.y - b.y));
    double distB = sqrt((a.x - c.x)*(a.x - c.x) + (a.y -
         c.y)*(a.y - c.y));
    if (distA + distB + distC < mindist){
        mindist = distA + distB + distC;
        best_pair = make_pair(a.id,make_pair(b.id,c.id));
    }
}
vector<pt> t;
//Min possible triplet distance
void rec(int l, int r){
    if (r - l <= 3 &&r - l >=2){
        for (int i = l; i < r; ++i){
            for (int j = i + 1; j < r; ++j){
                for(int k=j+1;k<r;k++){
                    upd_ans(a[i],a[j],a[k]);
                }
            }
        }
        sort(a.begin() + l, a.begin() + r, cmp_y());
        return;
    }
    int m = (l + r) >> 1;
    int midx = a[m-1].x;
    /*
     * Got WA in a team contest
     * for putting midx = a[m].x;
     * Don't know why. Maybe due to
     * floating point numbers.
     */
    rec(l, m);
    rec(m, r);
    merge(a.begin() + l, a.begin() + m, a.begin() + m,
         a.begin() + r, t.begin(), cmp_y());
    copy(t.begin(), t.begin() + r - l, a.begin() + l);
    int tsz = 0;
    for (int i = l; i < r; ++i){
        if (abs(a[i].x - midx) < mindist/2){
            for (int j = tsz - 1; j >= 0 && a[i].y -
                 t[j].y < mindist/2; --j){
                if(i+1<r) upd_ans(a[i], a[i+1], t[j]);
                if(j>0) upd_ans(a[i], t[j-1], t[j]);
            }
            t[tsz++] = a[i];
        }
    }
}
```

## 3.5 intersecting-segments-pair

```
const double EPS = 1E-9;
struct pt {
    double x, y;
};
struct seg {
    pt p, q;
    int id;
    double get_y(double x) const {
        if (abs(p.x - q.x) < EPS)
            return p.y;
        return p.y + (q.y - p.y) * (x - p.x) / (q.x -
             p.x);
    }
};
bool intersect1d(double l1, double r1, double l2, double
     r2) {
    if (l1 > r1)
        swap(l1, r1);
    if (l2 > r2)
        swap(l2, r2);
    return max(l1, l2) <= min(r1, r2) + EPS;
}
int vec(const pt& a, const pt& b, const pt& c) {
    double s = (b.x - a.x) * (c.y - a.y) - (b.y - a.y) *
         (c.x - a.x);
    return abs(s) < EPS ? 0 : s > 0 ? +1 : -1;
}
bool intersect(const seg& a, const seg& b){
    return intersect1d(a.p.x, a.q.x, b.p.x, b.q.x) &&
           intersect1d(a.p.y, a.q.y, b.p.y, b.q.y) &&
           vec(a.p, a.q, b.p) * vec(a.p, a.q, b.q) <= 0 &&
           vec(b.p, b.q, a.p) * vec(b.p, b.q, a.q) <= 0;
}
bool operator<(const seg& a, const seg& b){
    double x = max(min(a.p.x, a.q.x), min(b.p.x, b.q.x));
    return a.get_y(x) < b.get_y(x) - EPS;
}
struct event {
    double x;
    int tp, id;
    event() {}
    event(double x, int tp, int id) : x(x), tp(tp),
         id(id) {}
    bool operator<(const event& e) const {
        if (abs(x - e.x) > EPS)
            return x < e.x;
        return tp > e.tp;
    }
};
set<seg> s;
vector<set<seg>::iterator> where;
set<seg>::iterator prev(set<seg>::iterator it) {
    return it == s.begin() ? s.end() : --it;
}
set<seg>::iterator next(set<seg>::iterator it) {
    return ++it;
}
pair<int, int> solve(const vector<seg>& a) {
    int n = (int)a.size();
    vector<event> e;
    for (int i = 0; i < n; ++i) {
        e.push_back(event(min(a[i].p.x, a[i].q.x), +1,
             i));
        e.push_back(event(max(a[i].p.x, a[i].q.x), -1,
             i));
    }
    sort(e.begin(), e.end());
    s.clear();
    where.resize(a.size());
    for (size_t i = 0; i < e.size(); ++i) {
        int id = e[i].id;
        if (e[i].tp == +1) {
```

```
            set<seg>::iterator nxt =
                s.lower_bound(a[id]), prv = prev(nxt);
            if (nxt != s.end() && intersect(*nxt, a[id]))
                return make_pair(nxt->id, id);
            if (prv != s.end() && intersect(*prv, a[id]))
                return make_pair(prv->id, id);
            where[id] = s.insert(nxt, a[id]);
        } else {
            set<seg>::iterator nxt = next(where[id]), prv
                = prev(where[id]);
            if (nxt != s.end() && prv != s.end() &&
                intersect(*nxt, *prv))
                return make_pair(prv->id, nxt->id);
            s.erase(where[id]);
        }
    }
    return make_pair(-1, -1);
}
```

## 3.6 point-location

```
typedef long long ll;
bool ge(const ll& a, const ll& b) { return a >= b; }
bool le(const ll& a, const ll& b) { return a <= b; }
bool eq(const ll& a, const ll& b) { return a == b; }
bool gt(const ll& a, const ll& b) { return a > b; }
bool lt(const ll& a, const ll& b) { return a < b; }
int sgn(const ll& x) { return le(x, 0) ? eq(x, 0) ? 0 :
    -1 : 1; }
struct pt {
    ll x, y;
    pt() {}
    pt(ll _x, ll _y) : x(_x), y(_y) {}
    pt operator-(const pt& a) const { return pt(x - a.x,
        y - a.y); }
    ll dot(const pt& a) const { return x * a.x + y *
        a.y; }
    ll dot(const pt& a, const pt& b) const { return (a -
        *this).dot(b - *this); }
    ll cross(const pt& a) const { return x * a.y - y *
        a.x; }
    ll cross(const pt& a, const pt& b) const { return (a
        - *this).cross(b - *this); }
    bool operator==(const pt& a) const { return a.x == x
        && a.y == y; }
};
struct Edge {
    pt l, r;
};
bool edge_cmp(Edge* edge1, Edge* edge2){
    const pt a = edge1->l, b = edge1->r;
    const pt c = edge2->l, d = edge2->r;
    int val = sgn(a.cross(b, c)) + sgn(a.cross(b, d));
    if (val != 0)
        return val > 0;
    val = sgn(c.cross(d, a)) + sgn(c.cross(d, b));
    return val < 0;
}
enum EventType { DEL = 2, ADD = 3, GET = 1, VERT = 0 };
struct Event {
    EventType type;
    int pos;
    bool operator<(const Event& event) const { return
        type < event.type; }
};
vector<Edge*> sweepline(vector<Edge*> planar, vector<pt>
    queries){
    using pt_type = decltype(pt::x);
    // collect all x-coordinates
    auto s =
        set<pt_type, std::function<bool(const pt_type&,
            const pt_type&)>>(lt);
    for (pt p : queries)
        s.insert(p.x);
    for (Edge* e : planar) {
        s.insert(e->l.x);
        s.insert(e->r.x);
    }
    // map all x-coordinates to ids
    int cid = 0;
    auto id =
        map<pt_type, int, std::function<bool(const
            pt_type&, const pt_type&)>>(
            lt);
    for (auto x : s)
        id[x] = cid++;
    // create events
    auto t = set<Edge*, decltype(*edge_cmp)>(edge_cmp);
    auto vert_cmp = [](const pair<pt_type, int>& l,
                       const pair<pt_type, int>& r) {
        if (!eq(l.first, r.first))
            return lt(l.first, r.first);
        return l.second < r.second;
    };
    auto vert = set<pair<pt_type, int>,
         decltype(vert_cmp)>(vert_cmp);
    vector<vector<Event>> events(cid);
    for (int i = 0; i < (int)queries.size(); i++) {
        int x = id[queries[i].x];
        events[x].push_back(Event{GET, i});
    }
    for (int i = 0; i < (int)planar.size(); i++) {
        int lx = id[planar[i]->l.x], rx =
            id[planar[i]->r.x];
        if (lx > rx) {
            swap(lx, rx);
            swap(planar[i]->l, planar[i]->r);
        }
        if (lx == rx) {
            events[lx].push_back(Event{VERT, i});
        } else {
            events[lx].push_back(Event{ADD, i});
            events[rx].push_back(Event{DEL, i});
        }
    }
    // perform sweep line algorithm
    vector<Edge*> ans(queries.size(), nullptr);
    for (int x = 0; x < cid; x++) {
        sort(events[x].begin(), events[x].end());
        vert.clear();
        for (Event event : events[x]) {
            if (event.type == DEL) {
                t.erase(planar[event.pos]);
            }
            if (event.type == VERT) {
                vert.insert(make_pair(
                    min(planar[event.pos]->l.y,
                        planar[event.pos]->r.y),
                    event.pos));
            }
            if (event.type == ADD) {
                t.insert(planar[event.pos]);
            }
            if (event.type == GET) {
                auto jt = vert.upper_bound(
                    make_pair(queries[event.pos].y,
                        planar.size()));
                if (jt != vert.begin()) {
                    --jt;
                    int i = jt->second;
                    if (ge(max(planar[i]->l.y,
                        planar[i]->r.y),
                           queries[event.pos].y)) {
                        ans[event.pos] = planar[i];
                        continue;
                    }
                }
                Edge* e = new Edge;
                e->l = e->r = queries[event.pos];
                auto it = t.upper_bound(e);
                if (it != t.begin())
                    ans[event.pos] = *(--it);
                delete e;
            }
        }
        for (Event event : events[x]) {
            if (event.type != GET)
                continue;
            if (ans[event.pos] != nullptr &&
                eq(ans[event.pos]->l.x,
                   ans[event.pos]->r.x))
                continue;
            Edge* e = new Edge;
            e->l = e->r = queries[event.pos];
            auto it = t.upper_bound(e);
            delete e;
            if (it == t.begin())
                e = nullptr;
            else
                e = *(--it);
            if (ans[event.pos] == nullptr) {
                ans[event.pos] = e;
                continue;
            }
            if (e == nullptr)
                continue;
            if (e == ans[event.pos])
                continue;
            if (id[ans[event.pos]->r.x] == x) {
                if (id[e->l.x] == x) {
                    if (gt(e->l.y, ans[event.pos]->r.y))
                        ans[event.pos] = e;
                }
            } else {
                ans[event.pos] = e;
            }
        }
    }
    return ans;
}
struct DCEL {
    struct Edge {
        pt origin;
        Edge* nxt = nullptr;
        Edge* twin = nullptr;
        int face;
    };
    vector<Edge*> body;
};
vector<pair<int, int>> point_location(DCEL planar,
     vector<pt> queries){
    vector<pair<int, int>> ans(queries.size());
    vector<Edge*> planar2;
    map<intptr_t, int> pos;
    map<intptr_t, int> added_on;
    int n = planar.body.size();
    for (int i = 0; i < n; i++) {
```

```
        if (planar.body[i]->face >
            planar.body[i]->twin->face)
            continue;
        Edge* e = new Edge;
        e->l = planar.body[i]->origin;
        e->r = planar.body[i]->twin->origin;
        added_on[(intptr_t)e] = i;
        pos[(intptr_t)e] =
            lt(planar.body[i]->origin.x,
               planar.body[i]->twin->origin.x)
              ? planar.body[i]->face
              : planar.body[i]->twin->face;
        planar2.push_back(e);
    }
    auto res = sweepline(planar2, queries);
    for (int i = 0; i < (int)queries.size(); i++) {
        if (res[i] == nullptr) {
            ans[i] = make_pair(1, -1);
            continue;
        }
        pt p = queries[i];
        pt l = res[i]->l, r = res[i]->r;
        if (eq(p.cross(l, r), 0) && le(p.dot(l, r), 0)) {
            ans[i] = make_pair(0,
               added_on[(intptr_t)res[i]]);
            continue;
        }
        ans[i] = make_pair(1, pos[(intptr_t)res[i]]);
    }
    for (auto e : planar2)
        delete e;
    return ans;
}
```

## 3.7 vertical-decomposition

```
typedef double dbl;
const dbl eps = 1e-9;
inline bool eq(dbl x, dbl y){
    return fabs(x - y) < eps;
}
inline bool lt(dbl x, dbl y){
    return x < y - eps;
}
inline bool gt(dbl x, dbl y){
    return x > y + eps;
}
inline bool le(dbl x, dbl y){
    return x < y + eps;
}
inline bool ge(dbl x, dbl y){
    return x > y - eps;
}
struct pt{
    dbl x, y;
    inline pt operator - (const pt & p)const{
        return pt{x - p.x, y - p.y};
    }
    inline pt operator + (const pt & p)const{
        return pt{x + p.x, y + p.y};
    }
    inline pt operator * (dbl a)const{
        return pt{x * a, y * a};
    }
    inline dbl cross(const pt & p)const{
        return x * p.y - y * p.x;
    }
    inline dbl dot(const pt & p)const{
        return x * p.x + y * p.y;
    }
    inline bool operator == (const pt & p)const{
        return eq(x, p.x) && eq(y, p.y);
    }
};
struct Line{
    pt p[2];
    Line(){}
    Line(pt a, pt b):p{a, b}{}
    pt vec()const{
        return p[1] - p[0];
    }
    pt& operator [](size_t i){
        return p[i];
    }
};
inline bool lexComp(const pt & l, const pt & r){
    if(fabs(l.x - r.x) > eps){
        return l.x < r.x;
    }
    else return l.y < r.y;
}
vector<pt> interSegSeg(Line l1, Line l2){
    if(eq(l1.vec().cross(l2.vec()), 0)){
        if(!eq(l1.vec().cross(l2[0] - l1[0]), 0))
            return {};
        if(!lexComp(l1[0], l1[1]))
            swap(l1[0], l1[1]);
        if(!lexComp(l2[0], l2[1]))
            swap(l2[0], l2[1]);
        pt l = lexComp(l1[0], l2[0]) ? l2[0] : l1[0];
        pt r = lexComp(l1[1], l2[1]) ? l1[1] : l2[1];
        if(l == r)
            return {l};
        else return lexComp(l, r) ? vector<pt>{l, r} :
            vector<pt>();
    }
    else{
        dbl s = (l2[0] - l1[0]).cross(l2.vec()) /
            l1.vec().cross(l2.vec());
        pt inter = l1[0] + l1.vec() * s;
        if(ge(s, 0) && le(s, 1) && le((l2[0] -
            inter).dot(l2[1] - inter), 0))
            return {inter};
        else
            return {};
    }
}
inline char get_segtype(Line segment, pt other_point){
    if(eq(segment[0].x, segment[1].x))
        return 0;
    if(!lexComp(segment[0], segment[1]))
        swap(segment[0], segment[1]);
    return (segment[1] - segment[0]).cross(other_point -
        segment[0]) > 0 ? 1 : -1;
}
dbl union_area(vector<tuple<pt, pt, pt> > triangles){
    vector<Line> segments(3 * triangles.size());
    vector<char> segtype(segments.size());
    for(size_t i = 0; i < triangles.size(); i++){
        pt a, b, c;
        tie(a, b, c) = triangles[i];
        segments[3 * i] = lexComp(a, b) ? Line(a, b) :
            Line(b, a);
        segtype[3 * i] = get_segtype(segments[3 * i], c);
        segments[3 * i + 1] = lexComp(b, c) ? Line(b, c)
            : Line(c, b);
        segtype[3 * i + 1] = get_segtype(segments[3 * i
            + 1], a);
        segments[3 * i + 2] = lexComp(c, a) ? Line(c, a)
            : Line(a, c);
        segtype[3 * i + 2] = get_segtype(segments[3 * i
            + 2], b);
    }
    vector<dbl> k(segments.size()), b(segments.size());
    for(size_t i = 0; i < segments.size(); i++){
        if(segtype[i]){
            k[i] = (segments[i][1].y - segments[i][0].y)
                / (segments[i][1].x - segments[i][0].x);
            b[i] = segments[i][0].y - k[i] *
                segments[i][0].x;
        }
    }
    dbl ans = 0;
    for(size_t i = 0; i < segments.size(); i++){
        if(!segtype[i])
            continue;
        dbl l = segments[i][0].x, r = segments[i][1].x;
        vector<pair<dbl, int> > evts;
        for(size_t j = 0; j < segments.size(); j++){
            if(!segtype[j] || i == j)
                continue;
            dbl l1 = segments[j][0].x, r1 =
                segments[j][1].x;
            if(ge(l1, r) || ge(l, r1))
                continue;
            dbl common_l = max(l, l1), common_r = min(r,
                r1);
            auto pts = interSegSeg(segments[i],
                segments[j]);
            if(pts.empty()){
                dbl yl1 = k[j] * common_l + b[j];
                dbl yl = k[i] * common_l + b[i];
                if(lt(yl1, yl) == (segtype[i] == 1)){
                    int evt_type = -segtype[i] *
                        segtype[j];
                    evts.emplace_back(common_l, evt_type);
                    evts.emplace_back(common_r, -evt_type);
                }
            }
            else if(pts.size() == 1u){
                dbl yl = k[i] * common_l + b[i], yl1 =
                    k[j] * common_l + b[j];
                int evt_type = -segtype[i] * segtype[j];
                if(lt(yl1, yl) == (segtype[i] == 1)){
                    evts.emplace_back(common_l, evt_type);
                    evts.emplace_back(pts[0].x, -evt_type);
                }
                yl = k[i] * common_r + b[i], yl1 = k[j] *
                    common_r + b[j];
                if(lt(yl1, yl) == (segtype[i] == 1)){
                    evts.emplace_back(pts[0].x, evt_type);
                    evts.emplace_back(common_r, -evt_type);
                }
            }
            else{
                if(segtype[j] != segtype[i] || j > i){
                    evts.emplace_back(common_l, -2);
                    evts.emplace_back(common_r, 2);
                }
            }
        }
        evts.emplace_back(l, 0);
        sort(evts.begin(), evts.end());
        size_t j = 0;
        int balance = 0;
        while(j < evts.size()){
```

```cpp
            size_t ptr = j;
            while(ptr < evts.size() && eq(evts[j].first,
                evts[ptr].first)){
                balance += evts[ptr].second;
                ++ptr;
            }
            if(!balance && !eq(evts[j].first, r)){
                dbl next_x = ptr == evts.size() ? r :
                    evts[ptr].first;
                ans -= segtype[i] * (k[i] * (next_x +
                    evts[j].first) + 2 * b[i]) * (next_x
                    - evts[j].first);
            }
            j = ptr;
        }
    }
    return ans/2;
}
pair<dbl,dbl> union_perimeter(vector<tuple<pt, pt, pt> >
    triangles){
    //Same as before
    pair<dbl,dbl> ans = make_pair(0,0);
    for(size_t i = 0; i < segments.size(); i++){
        //Same as before
        double
            dist=(segments[i][1].x-segments[i][0].x)*(segments[i][1].x-segments[i][0].x)+(segments[i][1].y-segments[i][0].y)*(segments[i][1].y-segments[i][0].y);
        dist=sqrt(dist);
        while(j < evts.size()){
            size_t ptr = j;
            while(ptr < evts.size() && eq(evts[j].first,
                evts[ptr].first)){
                balance += evts[ptr].second;
                ++ptr;
            }
            if(!balance && !eq(evts[j].first, r)){
                dbl next_x = ptr == evts.size() ? r :
                    evts[ptr].first;
                ans.first += dist * (next_x -
                    evts[j].first) / (r-l);
                if(eq(segments[i][1].y,segments[i][0].y))
                    ans.second+=(next_x - evts[j].first);
            }
            j = ptr;
        }
    }
    return ans;
}
```

# 4 Graph

## 4.1 articulation-vertex

```cpp
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of graph
vector<bool> visited;
vector<int> tin, low;
int timer;
void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    int children=0;
    for (int to : adj[v]) {
        if (to == p) continue;
        if (visited[to]) {
            low[v] = min(low[v], tin[to]);
        } else {
            dfs(to, v);
            low[v] = min(low[v], low[to]);
            if (low[to] >= tin[v] && p!=-1)
                IS_CUTPOINT(v);
            ++children;
        }
    }
    if(p == -1 && children > 1)
        IS_CUTPOINT(v);
}
void find_cutpoints() {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs (i);
    }
}
```

## 4.2 bellman-ford

```cpp
struct Edge {
    int a, b, cost;
};
int n, m;
vector<Edge> edges;
const int INF = 1000000000;
void solve(){
    vector<int> d(n);
    vector<int> p(n, -1);
    int x;
    for (int i = 0; i < n; ++i) {
        x = -1;
        for (Edge e : edges) {
            if (d[e.a] + e.cost < d[e.b]) {
                d[e.b] = d[e.a] + e.cost;
                p[e.b] = e.a;
                x = e.b;
            }
        }
    }
    if (x == -1) {
        cout << "No negative cycle found.";
    } else {
        for (int i = 0; i < n; ++i)
            x = p[x];
        vector<int> cycle;
        for (int v = x;; v = p[v]) {
            cycle.push_back(v);
            if (v == x && cycle.size() > 1)
                break;
        }
        reverse(cycle.begin(), cycle.end());
        cout << "Negative cycle: ";
        for (int v : cycle)
            cout << v << ' ';
        cout << endl;
    }
}
```

## 4.3 edmond-blossom

```cpp
/***Copied from
    https://codeforces.com/blog/entry/49402***/
/*
GETS:
V->number of vertices
E->number of edges
pair of vertices as edges (vertices are 1..V)
GIVES:
output of edmonds() is the maximum matching
match[i] is matched pair of i (-1 if there isn't a
    matched pair)
 */
const int M=500;
struct struct_edge
{
    int v;
    struct_edge* n;
};
typedef struct_edge* edge;
struct_edge pool[M*M*2];
edge top=pool,adj[M];
int V,E,match[M],qh,qt,q[M],father[M],base[M];
bool inq[M],inb[M],ed[M][M];
void add_edge(int u,int v)
{
    top->v=v,top->n=adj[u],adj[u]=top++;
    top->v=u,top->n=adj[v],adj[v]=top++;
}
int LCA(int root,int u,int v)
{
    static bool inp[M];
    memset(inp,0,sizeof(inp));
    while(1)
    {
        inp[u=base[u]]=true;
        if (u==root) break;
        u=father[match[u]];
    }
    while(1)
    {
        if (inp[v=base[v]]) return v;
        else v=father[match[v]];
    }
}
void mark_blossom(int lca,int u)
{
    while (base[u]!=lca)
    {
        int v=match[u];
        inb[base[u]]=inb[base[v]]=true;
        u=father[v];
        if (base[u]!=lca) father[u]=v;
    }
}
void blossom_contraction(int s,int u,int v)
{
    int lca=LCA(s,u,v);
    memset(inb,0,sizeof(inb));
    mark_blossom(lca,u);
    mark_blossom(lca,v);
    if (base[u]!=lca)
        father[u]=v;
    if (base[v]!=lca)
        father[v]=u;
    for (int u=0; u<V; u++)
        if (inb[base[u]])
        {
            base[u]=lca;
            if (!inq[u])
                inq[q[++qt]=u]=true;
        }
}
int find_augmenting_path(int s)
{
    memset(inq,0,sizeof(inq));
    memset(father,-1,sizeof(father));
    for (int i=0; i<V; i++) base[i]=i;
    inq[q[qh=qt=0]=s]=true;
    while (qh<=qt)
    {
```

```
        int u=q[qh++];
        for (edge e=adj[u]; e; e=e->n)
        {
            int v=e->v;
            if (base[u]!=base[v]&&match[u]!=v)
                if ((v==s)||(match[v]!=-1 &&
                    father[match[v]]!=-1))
                    blossom_contraction(s,u,v);
                else if (father[v]==-1)
                {
                    father[v]=u;
                    if (match[v]==-1)
                        return v;
                    else if (!inq[match[v]])
                        inq[q[++qt]=match[v]]=true;
                }
        }
    }
    return -1;
}
int augment_path(int s,int t)
{
    int u=t,v,w;
    while (u!=-1)
    {
        v=father[u];
        w=match[v];
        match[v]=u;
        match[u]=v;
        u=w;
    }
    return t!=-1;
}
int edmonds()//Gives number of matchings
{
    int matchc=0;
    memset(match,-1,sizeof(match));
    for (int u=0; u<V; u++)
        if (match[u]==-1)
            matchc+=augment_path(u,find_augmenting_path(u));
    return matchc;
}
//To add edge add_edge(u-1,v-1);
    ed[u-1][v-1]=ed[v-1][u-1]=true;
```

## 4.4 euler-path

```
int main() {
    int n;
    vector<vector<int>> g(n, vector<int>(n));
    // reading the graph in the adjacency matrix
    vector<int> deg(n);
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j)
            deg[i] += g[i][j];
    }
    int first = 0;
    while (first < n && !deg[first])
        ++first;
    if (first == n) {
        cout << -1;
        return 0;
    }
    int v1 = -1, v2 = -1;
    bool bad = false;
    for (int i = 0; i < n; ++i) {
        if (deg[i] & 1) {
            if (v1 == -1)
                v1 = i;
            else if (v2 == -1)
                v2 = i;
            else
                bad = true;
        }
    }
    if (v1 != -1)
        ++g[v1][v2], ++g[v2][v1];
    stack<int> st;
    st.push(first);
    vector<int> res;
    while (!st.empty()) {
        int v = st.top();
        int i;
        for (i = 0; i < n; ++i)
            if (g[v][i])
                break;
        if (i == n) {
            res.push_back(v);
            st.pop();
        } else {
            --g[v][i];
            --g[i][v];
            st.push(i);
        }
    }
    if (v1 != -1) {
        for (size_t i = 0; i + 1 < res.size(); ++i) {
            if ((res[i] == v1 && res[i + 1] == v2) ||
                (res[i] == v2 && res[i + 1] == v1)) {
                vector<int> res2;
                for (size_t j = i + 1; j < res.size();
                    ++j)
                    res2.push_back(res[j]);
                for (size_t j = 1; j <= i; ++j)
                    res2.push_back(res[j]);
                res = res2;
                break;
            }
        }
    }
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (g[i][j])
                bad = true;
        }
    }
    if (bad) {
        cout << -1;
    } else {
        for (int x : res)
            cout << x << " ";
    }
}
```

## 4.5 hopcraft-karp

```
/** Source:
    https://iq.opengenus.org/hopcroft-karp-algorithm/
    **/
// A class to represent Bipartite graph for
// Hopcroft Karp implementation
class BGraph{
    // m and n are number of vertices on left
    // and right sides of Bipartite Graph
    int m, n;
    // adj[u] stores adjacents of left side
    // vertex 'u'. The value of u ranges from 1 to m.
    // 0 is used for dummy vertex
    std::list<int> *adj;
    // pointers for hopcroftKarp()
    int *pair_u, *pair_v, *dist;
public:
    BGraph(int m, int n);   // Constructor
    void addEdge(int u, int v); // To add edge
    // Returns true if there is an augmenting path
    bool bfs();
    // Adds augmenting path if there is one beginning
    // with u
    bool dfs(int u);
    // Returns size of maximum matching
    int hopcroftKarpAlgorithm();
};
// Returns size of maximum matching
int BGraph::hopcroftKarpAlgorithm(){
    // pair_u[u] stores pair of u in matching on left
        side of Bipartite Graph.
    // If u doesn't have any pair, then pair_u[u] is NIL
    pair_u = new int[m + 1];
    // pair_v[v] stores pair of v in matching on right
        side of Biparite Graph.
    // If v doesn't have any pair, then pair_u[v] is NIL
    pair_v = new int[n + 1];
    // dist[u] stores distance of left side vertices
    dist = new int[m + 1];
    // Initialize NIL as pair of all vertices
    for (int u = 0; u <= m; u++)
        pair_u[u] = NIL;
    for (int v = 0; v <= n; v++)
        pair_v[v] = NIL;
    // Initialize result
    int result = 0;
    // Keep updating the result while there is an
    // augmenting path possible.
    while (bfs()){
        // Find a free vertex to check for a matching
        for (int u = 1; u <= m; u++)
            // If current vertex is free and there is
            // an augmenting path from current vertex
            // then increment the result
            if (pair_u[u] == NIL && dfs(u))
                result++;
    }
    return result;
}
// Returns true if there is an augmenting path
    available, else returns false
bool BGraph::bfs(){
    std::queue<int> q; //an integer queue for bfs
    // First layer of vertices (set distance as 0)
    for (int u = 1; u <= m; u++){
        // If this is a free vertex, add it to queue
        if (pair_u[u] == NIL){
            // u is not matched so distance is 0
            dist[u] = 0;
            q.push(u);
        }
        // Else set distance as infinite so that this
            vertex is considered next time for
            availibility
        else
            dist[u] = INF;
    }
    // Initialize distance to NIL as infinite
    dist[NIL] = INF;
    // q is going to contain vertices of left side only.
    while (!q.empty()){
        // dequeue a vertex
        int u = q.front();
        q.pop();
```

```
        // If this node is not NIL and can provide a
           shorter path to NIL then
        if (dist[u] < dist[NIL]){
            // Get all the adjacent vertices of the
               dequeued vertex u
            std::list<int>::iterator it;
            for (it = adj[u].begin(); it != adj[u].end();
                 ++it){
                int v = *it;
                // If pair of v is not considered so far
                // i.e. (v, pair_v[v]) is not yet
                   explored edge.
                if (dist[pair_v[v]] == INF){
                    // Consider the pair and push it to
                       queue
                    dist[pair_v[v]] = dist[u] + 1;
                    q.push(pair_v[v]);
                }
            }
        }
    }
    // If we could come back to NIL using alternating
       path of distinct
    // vertices then there is an augmenting path
       available
    return (dist[NIL] != INF);
}
// Returns true if there is an augmenting path beginning
   with free vertex u
bool BGraph::dfs(int u){
    if (u != NIL){
        std::list<int>::iterator it;
        for (it = adj[u].begin(); it != adj[u].end();
             ++it){
            // Adjacent vertex of u
            int v = *it;
            // Follow the distances set by BFS search
            if (dist[pair_v[v]] == dist[u] + 1){
                // If dfs for pair of v also returnn true
                   then
                if (dfs(pair_v[v]) == true){ // new
                     matching possible, store the matching
                    pair_v[v] = u;
                    pair_u[u] = v;
                    return true;
                }
            }
        }
        // If there is no augmenting path beginning with
           u then.
        dist[u] = INF;
        return false;
    }
    return true;
}
// Constructor for initialization
BGraph::BGraph(int m, int n){
    this->m = m;
    this->n = n;
    adj = new std::list<int>[m + 1];
}
// function to add edge from u to v
void BGraph::addEdge(int u, int v){
    adj[u].push_back(v); // Add v to us list.
}
```

## 4.6 hungerian-algorithm

```
class HungarianAlgorithm{
    int N,inf,n,max_match;
    int *lx,*ly,*xy,*yx,*slack,*slackx,*prev;
    int **cost;
    bool *S,*T;
    void init_labels(){
        for(int x=0;x<n;x++) lx[x]=0;
        for(int y=0;y<n;y++) ly[y]=0;
        for (int x = 0; x < n; x++)
            for (int y = 0; y < n; y++)
                lx[x] = max(lx[x], cost[x][y]);
    }
    void update_labels(){
        int x, y, delta = inf; //init delta as infinity
        for (y = 0; y < n; y++) //calculate delta using
             slack
            if (!T[y])
                delta = min(delta, slack[y]);
        for (x = 0; x < n; x++) //update X labels
            if (S[x]) lx[x] -= delta;
        for (y = 0; y < n; y++) //update Y labels
            if (T[y]) ly[y] += delta;
        for (y = 0; y < n; y++) //update slack array
            if (!T[y])
                slack[y] -= delta;
    }
    void add_to_tree(int x, int prevx)
//x - current vertex,prevx - vertex from X before x in
   the alternating path,
//so we add edges (prevx, xy[x]), (xy[x], x){
        S[x] = true; //add x to S
        prev[x] = prevx; //we need this when augmenting
        for (int y = 0; y < n; y++) //update slacks,
             because we add new vertex to S
            if (lx[x] + ly[y] - cost[x][y] < slack[y]){
                slack[y] = lx[x] + ly[y] - cost[x][y];
                slackx[y] = x;
            }
    }
    void augment() //main function of the algorithm{
        if (max_match == n) return; //check wether
             matching is already perfect
        int x, y, root; //just counters and root vertex
        int q[N], wr = 0, rd = 0; //q - queue for bfs,
             wr,rd - write and read
//pos in queue
        //memset(S, false, sizeof(S)); //init set S
        for(int i=0;i<n;i++) S[i]=false;
        //memset(T, false, sizeof(T)); //init set T
        for(int i=0;i<n;i++) T[i]=false;
        //memset(prev, -1, sizeof(prev)); //init set
             prev - for the alternating tree
        for(int i=0;i<n;i++) prev[i]=-1;
        for (x = 0; x < n; x++) //finding root of the
             tree{
            if (xy[x] == -1){
                q[wr++] = root = x;
                prev[x] = -2;
                S[x] = true;
                break;
            }
        }
        for (y = 0; y < n; y++) //initializing slack
             array{
            slack[y] = lx[root] + ly[y] - cost[root][y];
            slackx[y] = root;
        }
        while (true) //main cycle{
            while (rd < wr) //building tree with bfs
                 cycle{
                x = q[rd++]; //current vertex from X part
                for (y = 0; y < n; y++) //iterate through
                     all edges in equality graph{
                    if (cost[x][y] == lx[x] + ly[y] &&
                        !T[y]){
                        if (yx[y] == -1) break; //an
                             exposed vertex in Y found, so
//augmenting path exists!
                        T[y] = true; //else just add y to
                             T,
                        q[wr++] = yx[y]; //add vertex
                             yx[y], which is matched
//with y, to the queue
                        add_to_tree(yx[y], x); //add edges
                             (x,y) and (y,yx[y]) to the tree
                    }
                }
                if (y < n) break; //augmenting path found!
            }
            if (y < n) break; //augmenting path found!
            update_labels(); //augmenting path not found,
                 so improve labeling
            wr = rd = 0;
            for (y = 0; y < n; y++){
                //in this cycle we add edges that were
                     added to the equality graph as a
//result of improving the labeling, we add edge
     (slackx[y], y) to the tree if
//and only if !T[y] && slack[y] == 0, also with this
     edge we add another one
//(y, yx[y]) or augment the matching, if y was exposed
                if (!T[y] && slack[y] == 0){
                    if (yx[y] == -1) //exposed vertex in Y
                         found - augmenting path exists!{
                        x = slackx[y];
                        break;
                    }
                    else{
                        T[y] = true; //else just add y to
                             T,
                        if (!S[yx[y]]){
                            q[wr++] = yx[y]; //add vertex
                                 yx[y], which is matched
                                 with
//y, to the queue
                            add_to_tree(yx[y], slackx[y]);
                                 //and add edges (x,y) and
                                 (y,
//yx[y]) to the tree
                        }
                    }
                }
            }
            if (y < n) break; //augmenting path found!
        }
        if (y < n) //we found augmenting path!{
            max_match++; //increment matching
//in this cycle we inverse edges along augmenting path
            for (int cx = x, cy = y, ty; cx != -2; cx =
                 prev[cx], cy = ty){
                ty = xy[cx];
                yx[cy] = cx;
                xy[cx] = cy;
            }
            augment(); //recall function, go to step 1 of
                 the algorithm
        }
    }//end of augment() function
public:
    HungarianAlgorithm(int vv,int inf=1000000000){
```

```
        N=vv;
        n=N;
        max_match=0;
        this->inf=inf;
        lx=new int[N];
        ly=new int[N];//labels of X and Y parts
        xy=new int[N];//xy[x] - vertex that is matched
            with x,
        yx=new int[N];//yx[y] - vertex that is matched
            with y
        slack=new int[N];//as in the algorithm
            description
        slackx=new int[N];//slackx[y] such a vertex,
            that l(slackx[y]) + l(y) - w(slackx[y],y) =
            slack[y]
        prev=new int[N];//array for memorizing
            alternating paths
        S=new bool[N];
        T=new bool[N];//sets S and T in algorithm
        cost=new int*[N];//cost matrix
        for(int i=0; i<N; i++){
            cost[i]=new int[N];
        }
    }
    ~HungarianAlgorithm(){
        delete []lx;
        delete []ly;
        delete []xy;
        delete []yx;
        delete []slack;
        delete []slackx;
        delete []prev;
        delete []S;
        delete []T;
        int i;
        for(i=0; i<N; i++){
            delete [](cost[i]);
        }
        delete []cost;
    }
    void setCost(int i,int j,int c){
        cost[i][j]=c;
    }
    int* matching(bool first=true){
        int *ans;
        ans=new int[N];
        for(int i=0;i<N;i++){
            if(first) ans[i]=xy[i];
            else ans[i]=yx[i];
        }
        return ans;
    }
    int hungarian(){
        int ret = 0; //weight of the optimal matching
        max_match = 0; //number of vertices in current
            matching
        for(int x=0;x<n;x++) xy[x]=-1;
        for(int y=0;y<n;y++) yx[y]=-1;
        init_labels(); //step 0
        augment(); //steps 1-3
        for (int x = 0; x < n; x++) //forming answer
            there
            ret += cost[x][xy[x]];
        return ret;
    }
};
```

## 4.7 max-flow-dinic

```
#include<bits/stdc++.h>
#include<vector>
using namespace std;
#define MAX 100
#define HUGE_FLOW 1000000000
#define BEGIN 1
#define DEFAULT_LEVEL 0
struct FlowEdge {
    int v, u;
    long long cap, flow = 0;
    FlowEdge(int v, int u, long long cap) : v(v), u(u),
        cap(cap) {}
};
struct Dinic {
    const long long flow_inf = 1e18;
    vector<FlowEdge> edges;
    vector<vector<int>> adj;
    int n, m = 0;
    int s, t;
    vector<int> level, ptr;
    queue<int> q;
    Dinic(int n, int s, int t) : n(n), s(s), t(t) {
        adj.resize(n);
        level.resize(n);
        ptr.resize(n);
    }
    void add_edge(int v, int u, long long cap) {
        edges.emplace_back(v, u, cap);
        edges.emplace_back(u, v, 0);
        adj[v].push_back(m);
        adj[u].push_back(m + 1);
        m += 2;
    }
    bool bfs() {
        while (!q.empty()) {
            int v = q.front();
            q.pop();
            for (int id : adj[v]) {
                if (edges[id].cap - edges[id].flow < 1)
                    continue;
                if (level[edges[id].u] != -1)
                    continue;
                level[edges[id].u] = level[v] + 1;
                q.push(edges[id].u);
            }
        }
        return level[t] != -1;
    }
    long long dfs(int v, long long pushed) {
        if (pushed == 0)
            return 0;
        if (v == t)
            return pushed;
        for (int& cid = ptr[v]; cid <
            (int)adj[v].size(); cid++) {
            int id = adj[v][cid];
            int u = edges[id].u;
            if (level[v] + 1 != level[u] || edges[id].cap
                - edges[id].flow < 1)
                continue;
            long long tr = dfs(u, min(pushed,
                edges[id].cap - edges[id].flow));
            if (tr == 0)
                continue;
            edges[id].flow += tr;
            edges[id ^ 1].flow -= tr;
            return tr;
        }
        return 0;
    }
    long long flow() {
        long long f = 0;
        while (true) {
            fill(level.begin(), level.end(), -1);
            level[s] = 0;
            q.push(s);
            if (!bfs())
                break;
            fill(ptr.begin(), ptr.end(), 0);
            while (long long pushed = dfs(s, flow_inf)) {
                f += pushed;
            }
        }
        return f;
    }
};
int main(){
    return 0;
}
```

## 4.8 min-cost-max-flow

```
struct Edge{
    int from, to, capacity, cost;
};
vector<vector<int>> adj, cost, capacity;
const int INF = 1e9;
void shortest_paths(int n, int v0, vector<int>& d,
     vector<int>& p) {
    d.assign(n, INF);
    d[v0] = 0;
    vector<bool> inq(n, false);
    queue<int> q;
    q.push(v0);
    p.assign(n, -1);
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        inq[u] = false;
        for (int v : adj[u]) {
            if (capacity[u][v] > 0 && d[v] > d[u] +
                 cost[u][v]) {
                d[v] = d[u] + cost[u][v];
                p[v] = u;
                if (!inq[v]) {
                    inq[v] = true;
                    q.push(v);
                }
            }
        }
    }
}
int min_cost_flow(int N, vector<Edge> edges, int K, int
     s, int t) {
    adj.assign(N, vector<int>());
    cost.assign(N, vector<int>(N, 0));
    capacity.assign(N, vector<int>(N, 0));
    for (Edge e : edges) {
        adj[e.from].push_back(e.to);
        adj[e.to].push_back(e.from);
        cost[e.from][e.to] = e.cost;
        cost[e.to][e.from] = -e.cost;
        capacity[e.from][e.to] = e.capacity;
    }
    int flow = 0;
    int cost = 0;
    vector<int> d, p;
    while (flow < K) {
        shortest_paths(N, s, d, p);
        if (d[t] == INF)
            break;
```

```
        // find max flow on that path
        int f = K - flow;
        int cur = t;
        while (cur != s) {
            f = min(f, capacity[p[cur]][cur]);
            cur = p[cur];
        }
        // apply flow
        flow += f;
        cost += f * d[t];
        cur = t;
        while (cur != s) {
            capacity[p[cur]][cur] -= f;
            capacity[cur][p[cur]] += f;
            cur = p[cur];
        }
    }
    if (flow < K)
        return -1;
    else
        return cost;
}
```

## 4.9 online-bridge

```
vector<int> par, dsu_2ecc, dsu_cc, dsu_cc_size;
int bridges;
int lca_iteration;
vector<int> last_visit;
void init(int n) {
    par.resize(n);
    dsu_2ecc.resize(n);
    dsu_cc.resize(n);
    dsu_cc_size.resize(n);
    lca_iteration = 0;
    last_visit.assign(n, 0);
    for (int i=0; i<n; ++i) {
        dsu_2ecc[i] = i;
        dsu_cc[i] = i;
        dsu_cc_size[i] = 1;
        par[i] = -1;
    }
    bridges = 0;
}
int find_2ecc(int v) {
    if (v == -1)
        return -1;
    return dsu_2ecc[v] == v ? v : dsu_2ecc[v] =
        find_2ecc(dsu_2ecc[v]);
}
int find_cc(int v) {
    v = find_2ecc(v);
    return dsu_cc[v] == v ? v : dsu_cc[v] =
        find_cc(dsu_cc[v]);
}
void make_root(int v) {
    v = find_2ecc(v);
    int root = v;
    int child = -1;
    while (v != -1) {
        int p = find_2ecc(par[v]);
        par[v] = child;
        dsu_cc[v] = root;
        child = v;
        v = p;
    }
    dsu_cc_size[root] = dsu_cc_size[child];
}
void merge_path (int a, int b) {
    ++lca_iteration;
    vector<int> path_a, path_b;
    int lca = -1;
    while (lca == -1) {
        if (a != -1) {
            a = find_2ecc(a);
            path_a.push_back(a);
            if (last_visit[a] == lca_iteration){
                lca = a;
                break;
                }
            last_visit[a] = lca_iteration;
            a = par[a];
        }
        if (b != -1) {
            b = find_2ecc(b);
            path_b.push_back(b);
            if (last_visit[b] == lca_iteration){
                lca = b;
                break;
                }
            last_visit[b] = lca_iteration;
            b = par[b];
        }
    }
    for (int v : path_a) {
        dsu_2ecc[v] = lca;
        if (v == lca)
            break;
        --bridges;
    }
    for (int v : path_b) {
        dsu_2ecc[v] = lca;
        if (v == lca)
            break;
        --bridges;
    }
}
void add_edge(int a, int b) {
    a = find_2ecc(a);
    b = find_2ecc(b);
    if (a == b)
        return;
    int ca = find_cc(a);
    int cb = find_cc(b);
    if (ca != cb) {
        ++bridges;
        if (dsu_cc_size[ca] > dsu_cc_size[cb]) {
            swap(a, b);
            swap(ca, cb);
        }
        make_root(a);
        par[a] = dsu_cc[a] = b;
        dsu_cc_size[cb] += dsu_cc_size[a];
    } else {
        merge_path(a, b);
    }
}
```

# 5 header

```
#define FastIO ios::sync_with_stdio(false);
    cin.tie(0);cout.tie(0)

#include <ext/pb_ds/assoc_container.hpp> // Common file
using namespace __gnu_pbds;

/*
find_by_order(k) --> returns iterator to the kth largest
    element counting from 0
order_of_key(val) --> returns the number of items in a
    set that are strictly smaller than our item
*/
typedef tree<
int,
null_type,
less<int>,
rb_tree_tag,
tree_order_statistics_node_update>
ordered_set;

#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
gp_hash_table<int, int> table;
struct custom_hash {
    static uint64_t splitmix64(uint64_t x) {
        // http://xorshift.di.unimi.it/splitmix64.c
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }
    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM =
            chrono::steady_clock::now().time_since_epoch().co
        return splitmix64(x + FIXED_RANDOM);
    }
};
gp_hash_table<long long, int, custom_hash>
    safe_hash_table;
```