Contents

1 A 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;
//#pragma GCC optimize("O3,unroll-loops")
//#pragma GCC target("avx2,bmi,bmi2,lzcnt")
//mt19937
     rng(chrono::system_clock::now().time_since_epoch().count());
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
struct custom_hash {
  static uint64_t splitmix64(uint64_t x) {
    x += 0x9e3779b97f4a7c15;       ///Random
    x=(x^(x>>30))*0xbf58476d1ce4e5b9; ///Random
    x=(x^(x>>27))*0x94d049bb133111eb; ///Random
    return x^(x>>31);
  }
  const uint64_t FIXED_RANDOM = chrono::
    steady_clock::now().time_since_epoch().count();
  size_t operator()(uint64_t x) const {
    return splitmix64(x + FIXED_RANDOM);
  }
  size_t operator()(pair<int, int> x) const {
    return splitmix64((uint64_t(x.first)<<32) +
          x.second + FIXED_RANDOM);
  }
};
gp_hash_table<pair<int,int>,int,custom_hash> ht;
namespace my_gcc_ints {
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wpedantic"
    using int128 = __int128;
#pragma GCC diagnostic pop
}
# stresstester GENERATOR SOL1 SOL2 ITERATIONS
for i in $(seq 1 "$4") ; do
    echo -en "\rAttempt $i/$4"
    $1 > in.txt
    $2 < in.txt > out1.txt
    $3 < in.txt > out2.txt
    diff -y out1.txt out2.txt > diff.txt
    if [ $? -ne 0 ] ; then
        echo -e "\nTestcase Found:"; cat in.txt
        echo -e "\nOutputs:"; cat diff.txt
        exit
    fi
done
```

2 DP

2.1 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];
}
```

2.2 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;
}
```

2.3 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);
    }
};
```

2.4 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;
}
```

3 DS

3.1 Heavy light decomposition

```
int value[N],Tree[N];
int parent[N], depth[N], heavy[N], head[N], pos[N];
int cur_pos,n;
vector<int > adj[N];
int dfs(int v) {
    int size = 1,max_c_size = 0;
    for (int c : adj[v]) {
        if (c != parent[v]) {
            parent[c] = v, depth[c] = depth[v] + 1;
            int c_size = dfs(c);
            size += c_size;
            if (c_size > max_c_size)
                max_c_size = c_size, heavy[v] = c;
        }
    }
    return size;
}
void update(int idx, int x,int nn){
  while(idx<=nn){
    Tree[idx]+=x; idx+=(idx&-idx);
  }
}
void decompose(int v, int h) {
    head[v] = h, pos[v] = cur_pos;
    update(cur_pos,value[v],n+1);
    cur_pos++;
    if (heavy[v] != -1) decompose(heavy[v], h);
    for (int c : adj[v]) {
        if (c != parent[v] && c != heavy[v])
            decompose(c, c);
    }
}
int query_bit(int idx){
  int sum=0;
  while(idx>0){
    sum+=Tree[idx]; idx-=(idx&-idx);
  }
  return sum;
}
void init_hld(int root,int n){
    memset(Tree, 0,sizeof Tree);
    memset(heavy,-1,sizeof heavy);
    cur_pos = 1; parent[root]=-1;
    assert(dfs(root)==n);
    decompose(root, root);
}
int segment_tree_query(int x,int y){
    if(y<x) swap(x,y);
    return query_bit(y)-query_bit(x-1);
}
int query_hld(int a, int b) {
    int res = 0;
    for (; head[a] != head[b]; b = parent[head[b]]) {
        if (depth[head[a]] > depth[head[b]])
            swap(a, b);
        int cur_heavy_path_max =
              segment_tree_query(pos[head[b]], pos[b]);
        res += cur_heavy_path_max;
    }
    if (depth[a] > depth[b])
        swap(a, b);
    /// now a is the lca or quert(a,b)
    int last_heavy_path_max = segment_tree_query(pos[a],
         pos[b]);
    res += last_heavy_path_max;
    return res;
}
```

3.2 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,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], 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){
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();
```

3.3 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];}
```

3.4 bitset

```
int LEN;// length of Bitset array t
struct Bitset{
ull t[N/64+5]; Bitset(){memset(t,0,sizeof t);}
void set(int p){ t[p>>6]|=1llu<<(p&63); }
void shift(){ ull last=0llu;
  for(int i=0;i<LEN;i++){
    ull curr=t[i]>>63llu; (t[i]<<=1)|=last; last =curr;
  }
}
int count(){ int ret=0;
  for(int i=0;i<LEN;i++) ret+=__builtin_popcountll(t[i]);
  return ret;
}
Bitset &operator = (Bitset const&b){
  memcpy(t,b.t,sizeof (t)); return *this;
}
  Bitset &operator |=(Bitset &b){
    for(int i=0;i<LEN;i++)t[i]|=b.t[i]; return *this;
  }
  Bitset &operator &=(Bitset &b){
    for(int i=0;i<LEN;i++) t[i]&=b.t[i]; return *this;
  }
  Bitset &operator ^=(Bitset &b){
    for(int i=0;i<LEN;i++) t[i]^=b.t[i]; return *this;
  }
};
Bitset operator-(const Bitset &a,const Bitset &b){
  Bitset tmp; ull last=0;
  for(int i=0;i<LEN;i++){ ull curr=(a.t[i]< b.t[i] + last);
    tmp.t[i]=a.t[i]-b.t[i]-last; last = curr;
  }
  return tmp;
}
// https://loj.ac/p/6564 , string s,t, m_old = 0
// lcs formula: let x = m_old | Occurance set of s [char or t]
// m_new = ((x - ((m_old<<1)+1)) ^x)& x; finally m_new.count()
```

3.5 centroid decomposition

```
set<int> g[N];
int par[N],sub[N],level[N],ans[N]; int DP[LOGN][N];
int n,m; int nn;
void dfs1(int u,int p){
  sub[u]=1; nn++;
  for(auto it=g[u].begin();it!=g[u].end();it++) if(*it!=p){
    dfs1(*it,u); sub[u]+=sub[*it];}
}
int dfs2(int u,int p){
  for(auto it=g[u].begin();it!=g[u].end();it++)
    if(*it!=p && sub[*it]>nn/2)
      return dfs2(*it,u);
  return u;
}
void decompose(int root,int p){
  nn=0; dfs1(root,root); int centroid = dfs2(root,root);
  if(p==-1)p=centroid; par[centroid]=p;
 for(auto it=g[centroid].begin();it!=g[centroid].end();it++){
    g[*it].erase(centroid); decompose(*it,centroid); }
  g[centroid].clear();
}
```

3.6 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;
    }
};
```

3.7 link cut tree

```
const int MOD = 998244353;
int sum(int a, int b) {
    return a+b >= MOD ? a+b-MOD : a+b;
}
int mul(int a, int b) {
    return (a*1LL*b)%MOD;
}
typedef pair< int , int >Linear;
Linear compose(const Linear &p, const Linear &q) {
    return Linear(mul(p.first, q.first), sum(mul(q.second,
         p.first), p.second));
}
struct SplayTree {
    struct Node {
        int ch[2] = {0, 0}, p = 0;
        long long self = 0, path = 0;    // Path aggregates
        long long sub = 0, vir = 0;      // Subtree aggregates
        bool flip = 0;                   // Lazy tags
        int size = 1;
        Linear _self{1, 0}, _path_shoja{1, 0}, _path_ulta{1, 0};
    };
    vector<Node> T;
    SplayTree(int n) : T(n + 1) {
        T[0].size = 0;
    }
    void push(int x) {
        if (!x || !T[x].flip) return;
        int l = T[x].ch[0], r = T[x].ch[1];
        T[l].flip ^= 1, T[r].flip ^= 1;
        swap(T[x].ch[0], T[x].ch[1]);
        T[x].flip = 0;
        swap(T[x]._path_shoja, T[x]._path_ulta);
    }
    void pull(int x) {
        int l = T[x].ch[0], r = T[x].ch[1]; push(l); push(r);
        T[x].size = T[l].size + T[r].size + 1;
        T[x].path = T[l].path + T[x].self + T[r].path;
        T[x].sub = T[x].vir + T[l].sub + T[r].sub + T[x].self;
        T[x]._path_shoja = compose(T[r]._path_shoja,
             compose(T[x]._self, T[l]._path_shoja));
        T[x]._path_ulta = compose(T[l]._path_ulta,
             compose(T[x]._self, T[r]._path_ulta));
    }
    void set(int x, int d, int y) {
        T[x].ch[d] = y; T[y].p = x; pull(x);
    }
    void splay(int x) {
        auto dir = [&](int x) {
            int p = T[x].p; if (!p) return -1;
            return T[p].ch[0] == x ? 0 : T[p].ch[1] == x ? 1 :
                -1;
        };
        auto rotate = [&](int x) {
            int y = T[x].p, z = T[y].p, dx = dir(x), dy =
                dir(y);
            set(y, dx, T[x].ch[!dx]);
            set(x, !dx, y);
            if (~dy) set(z, dy, x);
            T[x].p = z;
```

```
        };
        for (push(x); ~dir(x); ) {
            int y = T[x].p, z = T[y].p;
            push(z); push(y); push(x);
            int dx = dir(x), dy = dir(y);
            if (~dy) rotate(dx != dy ? x : y);
            rotate(x);
        }
    }
    int KthNext(int x, int k) {
        assert(k > 0);
        splay(x);
        x = T[x].ch[1];
        if (T[x].size < k) return -1;
        while (true) {
            push(x);
            int l = T[x].ch[0], r = T[x].ch[1];
            if (T[l].size+1 == k) return x;
            if (k <= T[l].size) x = l;
            else k -= T[l].size+1, x = r;
        }
    }
};
struct LinkCut : SplayTree {
    LinkCut(int n) : SplayTree(n) {}
    int access(int x) {
        int u = x, v = 0;
        for (; u; v = u, u = T[u].p) {
            splay(u);
            int& ov = T[u].ch[1];
            T[u].vir += T[ov].sub;
            T[u].vir -= T[v].sub;
            ov = v; pull(u);
        }
        splay(x);
        return v;
    }
    void reroot(int x) {
        access(x); T[x].flip ^= 1; push(x);
    }
    ///makes v parent of u (optional: u must be a root)
    void Link(int u, int v) {
        reroot(u); access(v);
        T[v].vir += T[u].sub;
        T[u].p = v; pull(v);
    }
    ///removes edge between u and v
    void Cut(int u, int v) {
        int _u = FindRoot(u);
        reroot(u); access(v);
        T[v].ch[0] = T[u].p = 0; pull(v);
        reroot(_u);
    }
    // Rooted tree LCA. Returns 0 if u and v arent connected.
    int LCA(int u, int v) {
        if (u == v) return u;
        access(u); int ret = access(v);
        return T[u].p ? ret : 0;
    }
    // Query subtree of u where v is outside the subtree.
    long long Subtree(int u, int v) {
        int _v = FindRoot(v);
        reroot(v); access(u);
        long long ans = T[u].vir + T[u].self;
        reroot(_v);
        return ans;
    }
    // Query path [u..v]
    long long Path(int u, int v) {
        int _u = FindRoot(u);
        reroot(u); access(v);
        long long ans = T[v].path;
        reroot(_u);
        return ans;
    }
    Linear _Path(int u, int v) {
        reroot(u); access(v); return T[v]._path_shoja;
    }
    // Update vertex u with value v
    void Update(int u, long long v) {
        access(u); T[u].self = v; pull(u);
    }
    // Update vertex u with value v
    void _Update(int u, Linear v) {
        access(u); T[u]._self = v; pull(u);
    }
    int FindRoot(int u) {
        access(u);
        while (T[u].ch[0]) {
            u = T[u].ch[0];
            push(u);
        }
        access(u);
        return u;
    }
    ///k-th node (0-indexed) on the path from u to v
    int KthOnPath(int u, int v, int k) {
        if (u == v) return k == 0 ? u : -1;
        int _u = FindRoot(u);
        reroot(u); access(v);
        int ans = KthNext(u, k);
        reroot(_u);
        return ans;
    }
};
int main() {
    cin >> n >> q;
    LinkCut lct(n);
    for (int i = 1; i <= n; i++) {
        Linear l;
        cin >> l.first >> l.second;
        lct._Update(i, l);
    }
    for (int i = 1; i < n; i++) {
        int u, v;
        cin >> u >> v;
        lct.Link(u+1, v+1);
    }
    while (q--) {
        int op;
        cin >> op;
        if (op == 0) {
            int u, v, w, x;
            cin >> u >> v >> w >> x;
            lct.Cut(u+1, v+1);
            lct.Link(w+1, x+1);
        } else if (op == 1) {
            int p; Linear l;
            cin >> p >> l.first >> l.second;
            lct._Update(p+1, l);
        } else {
            int u, v, x;
            cin >> u >> v >> x;
            Linear l = lct._Path(u+1, v+1);
            cout << sum(mul(l.first, x), l.second) << "\n";
        }
    }
    return 0;
}
```

3.8 sparse table 2d

```
const int N=500; const int K = 8 ; /// k >= ceil(lg22(n)) +1
int arr[N][N]; int st[K+1][K+1][N][N]; int lg2[N+1];
void ini(){ lg2[1] = 0;
    for (int i = 2; i <= N; i++) lg2[i] = lg2[i/2] + 1; }
int f(int i,int j){ return max(i,j); }
void pre( int n,int m){
    for(int x=0;x<=K;x++){
        for(int y=0;y<=K;y++){
            for(int i=0;i<n;i++){
                for(int j=0;j<m;j++){
if(i+(1<<x)>n or j+(1<<y) > m) continue;
if(x>0) st[x][y][i][j] = f(st[x-1][y][i][j] ,
                          st[x-1][y][i+(1<<(x-1))][j]);
else if(y>0) st[x][y][i][j] = f(st[x][y-1][i][j] ,
                          st[x][y-1][i][j+(1<<(y-1))]);
else st[x][y][i][j] = f(arr[i][j] , arr[i][j]);
                } } } } }
int getf( int R1 ,int C1 , int R2 , int C2){
    if(R1>R2) swap(R1,R2); if(C1>C2) swap(C1,C2);
    int x = lg2[R2 - R1 + 1]; int y = lg2[C2 - C1 + 1];
    return f(f(f(st[x][y][R1][C1],st[x][y][R2-(1<<x)+1][C1]),
st[x][y][R1][C2-(1<<y)+1]),st[x][y][R2-(1<<x)+1][C2-(1<<y)+1]);
}
```

3.9 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);
    }
    // keys(l) < keys(r)
    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);
    }
};
```

3.10 wavelet tree

```
#include <bits/stdc++.h>
using namespace std;
#define fo(i,n) for(i=0;i<n;i++)
#define ll long long
#define pb push_back
#define mp make_pair
typedef pair<int, int> pii;
typedef pair<ll, ll> pl;
typedef vector<int>  vi;
const int N = 3e5, M = N;
//=======================
const int MAX = 1e6;
int a[N];
struct wavelet_tree{
#define vi vector<int>
#define pb push_back
    int lo, hi;
    wavelet_tree *l=0, *r=0;
    vi b;
    vi c; // c holds the prefix sum of elements
    //nos are in range [x,y]
    //array indices are [from, to)
    wavelet_tree(int *from, int *to, int x, int y){
        lo = x, hi = y;
        if( from >= to)
            return;
        if( hi == lo ){
            b.reserve(to-from+1);
            b.pb(0);
            c.reserve(to-from+1);
```

```
        c.pb(0);
        for(auto it = from; it != to; it++){
            b.pb(b.back() + 1);
            c.pb(c.back()+*it);
        }
        return ;
    }
    int mid = (lo+hi)/2;
    auto f = [mid](int x){
        return x <= mid;
    };
    b.reserve(to-from+1);
    b.pb(0);
    c.reserve(to-from+1);
    c.pb(0);
    for(auto it = from; it != to; it++){
        b.pb(b.back() + f(*it));
        c.pb(c.back() + *it);
    }
    //see how lambda function is used here
    auto pivot = stable_partition(from, to, f);
    l = new wavelet_tree(from, pivot, lo, mid);
    r = new wavelet_tree(pivot, to, mid+1, hi);
}
// swap a[i] with a[i+1] , if a[i]!=a[i+1] call
    swapadjacent(i)
void swapadjacent(int i){
    if(lo == hi)
        return ;
    b[i]= b[i-1] + b[i+1] - b[i];
    c[i] = c[i-1] + c[i+1] - c[i];
    if( b[i+1]-b[i] == b[i] - b[i-1]){
        if(b[i]-b[i-1])
            return this->l->swapadjacent(b[i]);
        else
            return this->r->swapadjacent(i-b[i]);
    }
    else
        return ;
}
//kth smallest element in [l, r]
int kth(int l, int r, int k){
    if(l > r)
        return 0;
    if(lo == hi)
        return lo;
    int inLeft = b[r] - b[l-1];
    int lb = b[l-1]; //amt of nos in first (l-1) nos that
        go in left
    int rb = b[r]; //amt of nos in first (r) nos that go in
        left
    if(k <= inLeft)
        return this->l->kth(lb+1, rb, k);
    return this->r->kth(l-lb, r-rb, k-inLeft);
}
//count of nos in [l, r] Less than or equal to k
int LTE(int l, int r, int k){
    if(l > r or k < lo)
        return 0;
    if(hi <= k)
        return r - l + 1;
    int lb = b[l-1], rb = b[r];
    return this->l->LTE(lb+1, rb, k) + this->r->LTE(l-lb,
        r-rb, k);
}
//count of nos in [l, r] equal to k
int count(int l, int r, int k){
    if(l > r or k < lo or k > hi)
        return 0;
    if(lo == hi)
        return r - l + 1;
    int lb = b[l-1], rb = b[r], mid = (lo+hi)/2;
    if(k <= mid)
        return this->l->count(lb+1, rb, k);
    return this->r->count(l-lb, r-rb, k);
}
//sum of nos in [l ,r] less than or equal to k
int sumk(int l, int r, int k){
    if(l > r or k < lo)
        return 0;
        if(hi <= k)
            return c[r] - c[l-1];
        int lb = b[l-1], rb = b[r];
        return this->l->sumk(lb+1, rb, k) + this->r->sumk(l-lb,
            r-rb, k);
    }
    ~wavelet_tree(){
        if(l)
        delete l;
        if(r)
        delete r;
    }
};
int main(){
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    srand(time(NULL));
    int i,n,k,j,q,l,r;
    cin >> n;
    fo(i, n) cin >> a[i+1];
    wavelet_tree T(a+1, a+n+1, 1, MAX);
    cin >> q;
    while(q--){
        int x;
        cin >> x;
        cin >> l >> r >> k;
        if(x == 0){
            //kth smallest
            cout << "Kth smallest: ";
            cout << T.kth(l, r, k) << endl;
        }
        if(x == 1){
            //less than or equal to K
            cout << "LTE: ";
            cout << T.LTE(l, r, k) << endl;
        }
        if(x == 2){
            //count occurence of K in [l, r]
            cout << "Occurence of K: ";
            cout << T.count(l, r, k) << endl;
        }
        if(x == 3){
            //sum of elements less than or equal to K in [l, r]
            cout << "Sum: ";
            cout << T.sumk(l, r, k) << endl;
        }
    }
    return 0;
}
```

4 Game

4.1 HackenBush

```
/* tree case: g[u] = for all v : XOR[ g[v] + 1]
 lose if no moves available
1. Colon Principle: Grundy number of a tree is the
        xor of Grundy number of child subtrees.
2. Fusion Principle: Consider a pair of adjacent
    vertices u, v that has another path (i.e.,
    they are in a cycle). Then,we can contract u
    and v without changing Grundy number.
We first decompose graph into two-edge connected
components.Then, by contracting each components by
using Fusion Principle,we obtain a tree (and many
self loops) that has the same Grundy number to the
original graph. By using Colon Principle, we can
compute the Grundy number. O(m + n). */
struct hackenbush {
  int n; vector<vector<int>> adj;
  hackenbush(int n) : n(n), adj(n) { }
  void add_edge(int u, int v) {
    adj[u].push_back(v);
    if(u!=v) adj[v].push_back(u);
  }
  int grundy(int r) {
    vector<int> num(n), low(n); int t = 0;
    function<int(int, int)> dfs=[&](int p,int u) {
      num[u] = low[u] = ++t; int ans = 0;
      for (int v : adj[u]) {
        if (v == p) { p += 2 * n; continue; }
        if (num[v] == 0) {
          int res = dfs(u, v);
          low[u] = min(low[u], low[v]);
          if (low[v] > num[u]) ans ^= (1+res)^1;
          else ans ^= res;        // non bridge
        }
        else low[u] = min(low[u], num[v]);
      }
      if (p > n) p -= 2 * n;
      for (int v : adj[u])
        if (v != p && num[u] <= num[v]) ans ^= 1;
      return ans;
    };
    return dfs(-1, r);
  }
};
```

5 Geo

5.1 3dGeo

```
int dcmp(double x) { return abs(x) < EPS ? 0 : (x<0 ? -1 : 1);}
double degreeToRadian(double rad) { return rad*PI/180; }
struct Point {
    double x, y, z; Point() : x(0), y(0), z(0) {}
    Point(double X, double Y, double Z) : x(X), y(Y), z(Z) {}
    Point operator + (const Point& u) const {
        return Point(x + u.x, y + u.y, z + u.z); }
    Point operator - (const Point& u) const {
        return Point(x - u.x, y - u.y, z - u.z); }
    Point operator * (const double u) const {
        return Point(x * u, y * u, z * u); }
    Point operator / (const double u) const {
        return Point(x / u, y / u, z / u); }
};
double dot(Point a, Point b){ return a.x*b.x+a.y*b.y+a.z*b.z; }
Point cross(Point a, Point b) { return
Point(a.y*b.z - a.z*b.y, a.z*b.x - a.x*b.z, a.x*b.y - a.y*b.x);
}
double length(Point a) { return sqrt(dot(a, a)); }
double distance(Point a, Point b) { return length(a-b); }
Point unit(const Point &p) { return p/length(p); }
// Rotate p around axis x, with angle radians.
Point rotate(Point p, Point axis, double angle) {
    axis = unit(axis); Point comp1 = p * cos(angle);
    Point comp2 = axis * (1-cos(angle)) * dot(axis, p);
    Point comp3 = cross(axis, p) * sin(angle);
    return comp1 + comp2 + comp3;
}
struct Line {Point a, v;}; ///a+tv
// returns the distance from point a to line l
double distancePointLine(Point p, Line l) {
    return length(cross(l.v, p - l.a)) / length(l.v); }
/// distance from Line ab to Line cd
double distanceLineLine(Line a, Line b) {
    Point cr = cross(a.v, b.v); double crl = length(cr);
    if (dcmp(crl) == 0) return distancePointLine(a.a, b);
    return abs(dot(cr, a.a-b.a))/crl; }
struct Plane {
    Point normal; /// Normal = (A, B, C)
    double d;     /// dot(Normal) = d <--> Ax + By + Cz = d
    Point P;      /// anyPoint on the plane, optional
    Plane(Point normal, double d) {
        double len = length(normal); assert(dcmp(len) > 0);
        normal = normal / len; d = d / len;
        if  (dcmp(normal.x))   P = Point(d/normal.x, 0, 0);
        else if (dcmp(normal.y)) P = Point(0, d/normal.y, 0);
        else                   P = Point(0, 0, d/normal.z);
    }
    ///Plane given by three Non-Collinear Points
    Plane(Point a, Point b, Point c) {
        normal = unit(cross(b-a, c-a)); d = dot(normal,a); P=a;
    }
    bool onPlane(Point a) { return dcmp(dot(normal,a)-d)== 0;}
    double distance(Point a) { return abs(dot(normal, a) - d);}
    double isParallel(Line l){return dcmp(dot(l.v,normal))==0;}
//return t st l.a + t*l.v is a point on the plane, check
//parallel first
```

```
    double intersectLine(Line l) {
        return dot(P-l.a, normal)/dot(l.v, normal); } };
```

5.2 Circle Cover

```
///Check if the all of the area of circ(O, R) in
///Circ(OO, RR) is covered by some other circle
bool CoverCircle(PT O, double R, vector<PT> &cen,
      vector<double> &rad, PT OO, double RR) {
  int n = cen.size();
  vector<pair<double, double>> arcs;
  for (int i=0; i<n; i++) {
    PT P = cen[i]; double r = rad[i];
    if (i!=0 && R + sqrt(dist2(O, P))<r) return 1;
    if (i==0 && r + sqrt(dist2(O, P))<R) return 1;
    vector<PT> inter =
          CircleCircleIntersection(O, P, R, r);
    if (inter.size() <= 1) continue;
    PT X = inter[0], Y = inter[1];
    if (cross(O, X, Y) < 0) swap(X, Y);
    if (!(cross(O, X, P) >= 0 &&
        cross(O, Y, P) <= 0)) swap(X, Y);
    if (i==0)    swap(X, Y);
    X = X-O; Y=Y-O;
    double ll = atan2(X.y, X.x);
    double rr = atan2(Y.y, Y.x);
    if (rr < ll) rr += 2*PI;
    arcs.emplace_back(ll, rr);
  }
  if (arcs.empty()) return false;
  sort(arcs.begin(), arcs.end());
  double st = arcs[0].ff, en = arcs[0].ss,ans = 0;
  for (int i=1; i<arcs.size(); i++) {
    if (arcs[i].first <= en + EPS)
      en = max(en, arcs[i].second);
    else st = arcs[i].first, en = arcs[i].second;
    ans = max(ans, en-st);
  }
  return ans >= 2*PI;
}
```

5.3 Circle Union Area

```
struct Point {
  LD x,y ;
  LD operator*(const Point &a)const {
    return x*a.y-y*a.x;}
  LD operator/(const Point &a)const {
    return sqrt((a.x-x)*(a.x-x)+(a.y-y)*(a.y-y));}
}po[N];
LD r[N];
int sgn(LD x) {return fabs(x)<EPS?0:(x>0.0?1:-1);}
pair<LD,bool> ARG[2*N] ;
LD cir_union(Point c[],LD r[],int n) {
  LD sum = 0.0 , sum1 = 0.0 ,d,p1,p2,p3 ;
  for(int i = 0 ; i < n ; i++) {
    bool f = 1 ;
    for(int j = 0 ; f&&j<n ; j++)
      if(i!=j && sgn(r[j]-r[i]-c[i]/c[j])!=-1)f=0;
    if(!f) swap(r[i],r[--n]),swap(c[i--],c[n]);
  }
  for(int i = 0; i < n; i++) {
    int k = 0, cnt = 0;
    for(int j = 0; j < n; j++) {
      if(i!=j&&sgn((d=c[i]/c[j])-r[i]-r[j])<=0){
        p3=acos((r[i]*r[i]+d*d-r[j]*r[j])/
                          (2.0*r[i]*d));
        p2=atan2(c[j].y-c[i].y,c[j].x-c[i].x);
        p1 = p2-p3; p2 = p2+p3;
        if(sgn(p1+PI)==-1) p1+=2*PI,cnt++;
        if(sgn(p2-PI)==1) p2-=2*PI,cnt++;
        ARG[k++] = make_pair(p1,0);
        ARG[k++] = make_pair(p2,1);
      }
    }
    if(k) {
      sort(ARG,ARG+k) ;
      p1 = ARG[k-1].first-2*PI;
      p3 = r[i]*r[i] ;
      for(int j = 0 ; j < k ; j++) {
        p2 = ARG[j].first;
        if(cnt==0) {
          sum+=(p2-p1-sin(p2-p1))*p3 ;
          sum1+=(c[i]+Point(cos(p1),sin(p1))*
                r[i])*(c[i]+
                Point(cos(p2),sin(p2))*r[i]);
        }
        p1 = p2;
        ARG[j].second ? cnt--:cnt++;
      }
    }
    else sum += 2*PI*r[i]*r[i];
  }
  return (sum+fabs(sum1))*0.5 ;
}
```

5.4 anachor-geo

```
typedef double Tf;
typedef Tf Ti;    /// use long long for exactness
const Tf PI = acos(-1), EPS = 1e-9;
int dcmp(Tf x) { return abs(x) < EPS ? 0 : (x<0 ? -1 : 1);}
struct Point;
Ti dot(Point a, Point b) { return a.x * b.x + a.y * b.y; }
Ti cross(Point a, Point b) { return a.x * b.y - a.y * b.x; }
Tf length(Point a) { return sqrt(dot(a, a)); }
Ti sqLength(Point a) { return dot(a, a); }
Tf distance(Point a, Point b) {return length(a-b);}
Tf angle(Point u) { return atan2(u.y, u.x); }
// returns angle between oa, ob in (-PI, PI]
Tf angleBetween(Point a, Point b);
// Rotate a ccw by rad radians
Point rotate(Point a, Tf rad) {
  static_assert(is_same<Tf, Ti>::value);
  return Point(a.x * cos(rad) - a.y * sin(rad), a.x * sin(rad)
       + a.y * cos(rad));
}
// rotate a ccw by angle th with cos(th) = co && sin(th) = si
Point rotatePrecise(Point a, Tf co, Tf si) {
  static_assert(is_same<Tf, Ti>::value);
  return Point(a.x * co - a.y * si, a.y * co + a.x * si);
}
Point rotate90(Point a) { return Point(-a.y, a.x); }
// scales vector a by s such that length of a becomes s
Point scale(Point a, Tf s) {
  static_assert(is_same<Tf, Ti>::value);
  return a / length(a) * s;
}
// returns an unit vector perpendicular to vector a
Point normal(Point a) {
  static_assert(is_same<Tf, Ti>::value);
  Tf l = length(a);
  return Point(-a.y / l, a.x / l);
}
// returns 1 if c is left of ab, 0 if on ab && -1 if right of ab
int orient(Point a, Point b, Point c) {
  return dcmp(cross(b - a, c - a));
}
bool half(Point p){   // returns true for point above x axis
     or on negative x axis
  return p.y > 0 || (p.y == 0 && p.x < 0);
}
bool polarComp(Point p, Point q){ //to be used in sort()
     function
  return make_tuple(half(p), 0) < make_tuple(half(q), cross(p,
       q));
}
struct Segment;//Includes two points
typedef Segment Line;
struct Circle {
    Point o;
    Tf r;
    Circle(Point o = Point(0, 0), Tf r = 0) : o(o), r(r) {}
    // returns true if point p is in || on the circle
    bool contains(Point p);
    // returns a point on the circle rad radians away from +X
         CCW
    Point point(Tf rad) {
      static_assert(is_same<Tf, Ti>::value);
      return Point(o.x + cos(rad) * r, o.y + sin(rad) * r);
    }
    // area of a circular sector with central angle rad
    Tf area(Tf rad = PI + PI) { return rad * r * r / 2; }
    // area of the circular sector cut by a chord with central
         angle alpha
    Tf sector(Tf alpha) { return r * r * 0.5 * (alpha -
         sin(alpha)); }
};
namespace Linear {
  // returns true if point p is on segment s
  bool onSegment(Point p, Segment s) {
    return dcmp(cross(s.a - p, s.b - p)) == 0 && dcmp(dot(s.a
         - p, s.b - p)) <= 0;
  }
  // returns true if segment p && q touch or intersect
  bool segmentsIntersect(Segment p, Segment q) {
    if(onSegment(p.a, q) || onSegment(p.b, q)) return true;
    if(onSegment(q.a, p) || onSegment(q.b, p)) return true;

    Ti c1 = cross(p.b - p.a, q.a - p.a);
    Ti c2 = cross(p.b - p.a, q.b - p.a);
    Ti c3 = cross(q.b - q.a, p.a - q.a);
    Ti c4 = cross(q.b - q.a, p.b - q.a);
    return dcmp(c1) * dcmp(c2) < 0 && dcmp(c3) * dcmp(c4) < 0;
  }
  bool linesParallel(Line p, Line q) {
    return dcmp(cross(p.b - p.a, q.b - q.a)) == 0;
  }
  // lines are represented as a ray from a point: (point,
       vector)
  // returns false if two lines (p, v) && (q, w) are parallel
       or collinear
  // true otherwise, intersection point is stored at o via
       reference
  bool lineLineIntersection(Point p, Point v, Point q, Point
       w, Point& o) {
    static_assert(is_same<Tf, Ti>::value);
    if(dcmp(cross(v, w)) == 0) return false;
    Point u = p - q;
    o = p + v * (cross(w,u)/cross(v,w));
    return true;
  }
  // returns false if two lines p && q are parallel or collinear
  // true otherwise, intersection point is stored at o via
       reference
  bool lineLineIntersection(Line p, Line q, Point& o) {
    return lineLineIntersection(p.a, p.b - p.a, q.a, q.b -
         q.a, o);
  }
  // returns the distance from point a to line l
  Tf distancePointLine(Point p, Line l) {
    return abs(cross(l.b - l.a, p - l.a) / length(l.b - l.a));
  }
  // returns the shortest distance from point a to segment s
  Tf distancePointSegment(Point p, Segment s) {
    if(s.a == s.b) return length(p - s.a);
    Point v1 = s.b - s.a, v2 = p - s.a, v3 = p - s.b;
    if(dcmp(dot(v1, v2)) < 0)  return length(v2);
    else if(dcmp(dot(v1, v3)) > 0) return length(v3);
    else return abs(cross(v1, v2) / length(v1));
  }
  // returns the shortest distance from segment p to segment q
  Tf distanceSegmentSegment(Segment p, Segment q) {
    if(segmentsIntersect(p, q)) return 0;
    Tf ans = distancePointSegment(p.a, q);
    ans = min(ans, distancePointSegment(p.b, q));
    ans = min(ans, distancePointSegment(q.a, p));
    ans = min(ans, distancePointSegment(q.b, p));
    return ans;
  }
  // returns the projection of point p on line l
  Point projectPointLine(Point p, Line l) {
    static_assert(is_same<Tf, Ti>::value);
    Point v = l.b - l.a;
    return l.a + v * ((Tf) dot(v, p - l.a) / dot(v, v));
```

```
  }
} // namespace Linear

typedef vector<Point> Polygon;
namespace Polygonal {
  // returns the signed area of polygon p of n vertices
  Tf signedPolygonArea(Polygon p);
  // returns the longest line segment of l that is inside or on
      the
  // simply polygon p. O(n lg n). TESTED: TIMUS 1955
  Tf longestSegInPoly(Line l, const Polygon &p) {
    using Linear::lineLineIntersection;
    int n = p.size();
    vector<pair<Tf, int>> ev;
    for(int i=0; i<n; ++i) {
      Point a = p[i], b = p[(i + 1) % n], z = p[(i - 1 + n) %
          n];
      int ora = orient(l.a, l.b, a), orb = orient(l.a, l.b, b),
          orz = orient(l.a, l.b, z);
      if(!ora) {
        Tf d = dot(a - l.a, l.b - l.a);
        if(orz && orb) {
          if(orz != orb) ev.emplace_back(d, 0);
        }
        else if(orz) ev.emplace_back(d, orz);
        else if(orb) ev.emplace_back(d, orb);
      }
      else if(ora == -orb) {
        Point ins;
        lineLineIntersection(l, Line(a, b), ins);
        ev.emplace_back(dot(ins - l.a, l.b - l.a), 0);
      }
    }
    sort(ev.begin(), ev.end());

    Tf ret = 0, cur = 0, pre = 0;
    bool active = false;
    int sign = 0;
    for(auto &qq : ev) {
      int tp = qq.second;
      Tf d = qq.first;
      if(sign) {
        cur += d - pre;
        ret = max(ret, cur);
        if(tp != sign) active = !active;
        sign = 0;
      }
      else {
        if(active) cur += d - pre, ret = max(ret, cur);
        if(tp == 0) active = !active;
        else sign = tp;
      }
      pre = d;
      if(!active) cur = 0;
    }

    ret /= length(l.b - l.a);
    return ret;
  }
} // namespace Polygonal

namespace Convex {
  ///Tested on Kattis::fenceortho
  void rotatingCalipersGetRectangle(Point* p, int n, Tf& area,
      Tf& perimeter) {
    using Linear::distancePointLine;
    static_assert(is_same<Tf, Ti>::value);
    p[n] = p[0];
    int l = 1, r = 1, j = 1;
    area = perimeter = 1e100;

    for(int i = 0; i < n; i++) {
      Point v = (p[i + 1] - p[i]) / length(p[i + 1] - p[i]);
      while(dcmp(dot(v, p[r % n] - p[i]) - dot(v, p[(r + 1) %
          n] - p[i])) < 0) r++;
      while(j < r || dcmp(cross(v, p[j % n] - p[i]) - cross(v,
          p[(j + 1) % n] - p[i])) < 0) j++;
      while(l < j || dcmp(dot(v, p[l % n] - p[i]) - dot(v, p[(l
          + 1) % n] - p[i])) > 0) l++;
      Tf w = dot(v, p[r % n] - p[i]) - dot(v, p[l % n] - p[i]);
      Tf h = distancePointLine(p[j % n], Line(p[i], p[i + 1]));
      area = min(area, w * h);
      perimeter = min(perimeter, 2 * w + 2 * h);
    }
  }

  // returns the left side of polygon u after cutting it by ray
      a->b
  Polygon cutPolygon(Polygon u, Point a, Point b) {
    using Linear::lineLineIntersection, Linear::onSegment;

    Polygon ret;
    int n = u.size();
    for(int i = 0; i < n; i++) {
      Point c = u[i], d = u[(i + 1) % n];
      if(dcmp(cross(b-a, c-a)) >= 0) ret.push_back(c);
      if(dcmp(cross(b-a, d-c)) != 0) {
        Point t;
        lineLineIntersection(a, b - a, c, d - c, t);
        if(onSegment(t, Segment(c, d))) ret.push_back(t);
      }
    }
    return ret;
  }

  // returns true if point p is in or on triangle abc
  bool pointInTriangle(Point a, Point b, Point c, Point p) {
    return dcmp(cross(b - a, p - a)) >= 0
      && dcmp(cross(c - b, p - b)) >= 0
      && dcmp(cross(a - c, p - c)) >= 0;
  }

  // Tested : https://www.spoj.com/problems/INOROUT
  // pt must be in ccw order with no three collinear points
  // returns inside = -1, on = 0, outside = 1
  int pointInConvexPolygon(const Polygon &pt, Point p) {
    int n = pt.size();
    assert(n >= 3);

    int lo = 1, hi = n - 1;
    while(hi - lo > 1) {
      int mid = (lo + hi) / 2;
      if(dcmp(cross(pt[mid] - pt[0], p - pt[0])) > 0) lo = mid;
      else hi = mid;
    }

    bool in = pointInTriangle(pt[0], pt[lo], pt[hi], p);
    if(!in) return 1;

    if(dcmp(cross(pt[lo] - pt[lo - 1], p - pt[lo - 1])) == 0)
        return 0;
    if(dcmp(cross(pt[hi] - pt[lo], p - pt[lo])) == 0) return 0;
    if(dcmp(cross(pt[hi] - pt[(hi + 1) % n], p - pt[(hi + 1) %
        n])) == 0) return 0;
    return -1;
  }

  // Extreme Point for a direction is the farthest point in
      that direction
  // O'Rourke, page 270, http://crtl-i.com/PDF/comp_c.pdf
  // also https://codeforces.com/blog/entry/48868
  // poly is a convex polygon, sorted in CCW, doesn't contain
      redundant points
  // u is the direction for extremeness
  int extremePoint(const Polygon &poly, Point u = Point(0, 1))
      {
    int n = (int) poly.size();
    int a = 0, b = n;
    while(b - a > 1) {
      int c = (a + b) / 2;
      if(dcmp(dot(poly[c] - poly[(c + 1) % n], u)) >= 0 &&
          dcmp(dot(poly[c] - poly[(c - 1 + n) % n], u)) >= 0) {
        return c;
      }

      bool a_up = dcmp(dot(poly[(a + 1) % n] - poly[a], u)) >=
          0;
      bool c_up = dcmp(dot(poly[(c + 1) % n] - poly[c], u)) >=
          0;
      bool a_above_c = dcmp(dot(poly[a] - poly[c], u)) > 0;

      if(a_up && !c_up) b = c;
      else if(!a_up && c_up) a = c;
      else if(a_up && c_up) {
        if(a_above_c) b = c;
        else a = c;
      }
      else {
        if(!a_above_c) b = c;
        else a = c;
      }
    }

    if(dcmp(dot(poly[a] - poly[(a + 1) % n], u)) > 0 &&
        dcmp(dot(poly[a] - poly[(a - 1 + n) % n], u)) > 0)
      return a;
    return b % n;
  }

  // For a convex polygon p and a line l, returns a list of
      segments
  // of p that are touch or intersect line l.
  // the i'th segment is considered (p[i], p[(i + 1) modulo
      |p|])
  // #1 If a segment is collinear with the line, only that is
      returned
  // #2 Else if l goes through i'th point, the i'th segment is
      added
  // If there are 2 or more such collinear segments for #1,
  // any of them (only one, not all) should be returned (not
      tested)
  // Complexity: O(lg |p|)
  vector<int> lineConvexPolyIntersection(const Polygon &p,
      Line l) {
    assert((int) p.size() >= 3);
    assert(l.a != l.b);

    int n = p.size();
    vector<int> ret;

    Point v = l.b - l.a;
    int lf = extremePoint(p, rotate90(v));
    int rt = extremePoint(p, rotate90(v) * Ti(-1));
    int olf = orient(l.a, l.b, p[lf]);
    int ort = orient(l.a, l.b, p[rt]);

    if(!olf || !ort) {
      int idx = (!olf ? lf : rt);
      if(orient(l.a, l.b, p[(idx - 1 + n) % n]) == 0)
        ret.push_back((idx - 1 + n) % n);
      else ret.push_back(idx);
      return ret;
    }
    if(olf == ort) return ret;

    for(int i=0; i<2; ++i) {
      int lo = i ? rt : lf;
      int hi = i ? lf : rt;
      int olo = i ? ort : olf;

      while(true) {
        int gap = (hi - lo + n) % n;
        if(gap < 2) break;

        int mid = (lo + gap / 2) % n;
        int omid = orient(l.a, l.b, p[mid]);
        if(!omid) {
          lo = mid;
          break;
        }
        if(omid == olo) lo = mid;
        else hi = mid;
      }
      ret.push_back(lo);
    }
    return ret;
  }
  // Tested : https://toph.co/p/cover-the-points
  // Calculate [ACW, CW] tangent pair from an external point
  constexpr int CW = -1, ACW = 1;
  bool isGood(Point u, Point v, Point Q, int dir) { return
      orient(Q, u, v) != -dir; }
  Point better(Point u, Point v, Point Q, int dir) { return
      orient(Q, u, v) == dir ? u : v; }
  Point pointPolyTangent(const Polygon &pt, Point Q, int dir,
      int lo, int hi) {
```

```
    while(hi - lo > 1) {
      int mid = (lo + hi) / 2;
      bool pvs = isGood(pt[mid], pt[mid - 1], Q, dir);
      bool nxt = isGood(pt[mid], pt[mid + 1], Q, dir);

      if(pvs && nxt) return pt[mid];
      if(!(pvs || nxt)) {
        Point p1 = pointPolyTangent(pt, Q, dir, mid + 1, hi);
        Point p2 = pointPolyTangent(pt, Q, dir, lo, mid - 1);
        return better(p1, p2, Q, dir);
      }

      if(!pvs) {
        if(orient(Q, pt[mid], pt[lo]) == dir)    hi = mid - 1;
        else if(better(pt[lo], pt[hi], Q, dir) == pt[lo]) hi =
             mid - 1;
        else lo = mid + 1;
      }
      if(!nxt) {
        if(orient(Q, pt[mid], pt[lo]) == dir)    lo = mid + 1;
        else if(better(pt[lo], pt[hi], Q, dir) == pt[lo]) hi =
             mid - 1;
        else lo = mid + 1;
      }
    }

    Point ret = pt[lo];
    for(int i = lo + 1; i <= hi; i++) ret = better(ret, pt[i],
         Q, dir);
    return ret;
  }
  // [ACW, CW] Tangent
  pair<Point, Point> pointPolyTangents(const Polygon &pt,
       Point Q) {
    int n = pt.size();
    Point acw_tan = pointPolyTangent(pt, Q, ACW, 0, n - 1);
    Point cw_tan = pointPolyTangent(pt, Q, CW, 0, n - 1);
    return make_pair(acw_tan, cw_tan);
  }
}

namespace Circular {
  // Extremely inaccurate for finding near touches
  // compute intersection of line l with circle c
  // The intersections are given in order of the ray (l.a, l.b)
  vector<Point> circleLineIntersection(Circle c, Line l) {
    static_assert(is_same<Tf, Ti>::value);
    vector<Point> ret;
    Point b = l.b - l.a, a = l.a - c.o;

    Tf A = dot(b, b), B = dot(a, b);
    Tf C = dot(a, a) - c.r * c.r, D = B*B - A*C;
    if (D < -EPS) return ret;

    ret.push_back(l.a + b * (-B - sqrt(D + EPS)) / A);
    if (D > EPS)
      ret.push_back(l.a + b * (-B + sqrt(D)) / A);
    return ret;
  }

  // signed area of intersection of circle(c.o, c.r) &&
  // triangle(c.o, s.a, s.b) [cross(a-o, b-o)/2]
  Tf circleTriangleIntersectionArea(Circle c, Segment s) {
    using Linear::distancePointSegment;
    Tf OA = length(c.o - s.a);
    Tf OB = length(c.o - s.b);

    // sector
    if(dcmp(distancePointSegment(c.o, s) - c.r) >= 0)
      return angleBetween(s.a-c.o, s.b-c.o) * (c.r * c.r) / 2.0;

    // triangle
    if(dcmp(OA - c.r) <= 0 && dcmp(OB - c.r) <= 0)
      return cross(c.o - s.b, s.a - s.b) / 2.0;

    // three part: (A, a) (a, b) (b, B)
    vector<Point> Sect = circleLineIntersection(c, s);
    return circleTriangleIntersectionArea(c, Segment(s.a,
         Sect[0]))
      + circleTriangleIntersectionArea(c, Segment(Sect[0],
           Sect[1]))
      + circleTriangleIntersectionArea(c, Segment(Sect[1],
           s.b));
  }
  // area of intersecion of circle(c.o, c.r) && simple
       polyson(p[])
  // Tested : https://codeforces.com/gym/100204/problem/F -
       Little Mammoth
  Tf circlePolyIntersectionArea(Circle c, Polygon p) {
    Tf res = 0;
    int n = p.size();
    for(int i = 0; i < n; ++i)
      res += circleTriangleIntersectionArea(c, Segment(p[i],
           p[(i + 1) % n]));
    return abs(res);
  }

  // locates circle c2 relative to c1
  // interior        (d < R - r)    ----> -2
  // interior tangents (d = R - r)  ----> -1
  // concentric   (d = 0)
  // secants      (R - r < d < R + r) ----> 0
  // exterior tangents (d = R + r)  ----> 1
  // exterior       (d > R + r)     ----> 2
  int circleCirclePosition(Circle c1, Circle c2) {
    Tf d = length(c1.o - c2.o);
    int in = dcmp(d - abs(c1.r - c2.r)), ex = dcmp(d - (c1.r +
         c2.r));
    return in < 0 ? -2 : in == 0 ? -1 : ex == 0 ? 1 : ex > 0 ?
         2 : 0;
  }

  // compute the intersection points between two circles c1 &&
       c2
  vector<Point> circleCircleIntersection(Circle c1, Circle c2)
       {
    static_assert(is_same<Tf, Ti>::value);

    vector<Point> ret;
    Tf d = length(c1.o - c2.o);
    if(dcmp(d) == 0) return ret;
    if(dcmp(c1.r + c2.r - d) < 0) return ret;
    if(dcmp(abs(c1.r - c2.r) - d) > 0) return ret;

    Point v = c2.o - c1.o;
    Tf co = (c1.r * c1.r + sqLength(v) - c2.r * c2.r) / (2 *
         c1.r * length(v));
    Tf si = sqrt(abs(1.0 - co * co));
    Point p1 = scale(rotatePrecise(v, co, -si), c1.r) + c1.o;
    Point p2 = scale(rotatePrecise(v, co, si), c1.r) + c1.o;

    ret.push_back(p1);
    if(p1 != p2) ret.push_back(p2);
    return ret;
  }

  // intersection area between two circles c1, c2
  Tf circleCircleIntersectionArea(Circle c1, Circle c2) {
    Point AB = c2.o - c1.o;
    Tf d = length(AB);
    if(d >= c1.r + c2.r) return 0;
    if(d + c1.r <= c2.r) return PI * c1.r * c1.r;
    if(d + c2.r <= c1.r) return PI * c2.r * c2.r;

    Tf alpha1 = acos((c1.r * c1.r + d * d - c2.r * c2.r) /
         (2.0 * c1.r * d));
    Tf alpha2 = acos((c2.r * c2.r + d * d - c1.r * c1.r) /
         (2.0 * c2.r * d));
    return c1.sector(2 * alpha1) + c2.sector(2 * alpha2);
  }

  // returns tangents from a point p to circle c
  vector<Point> pointCircleTangents(Point p, Circle c) {
    static_assert(is_same<Tf, Ti>::value);

    vector<Point> ret;
    Point u = c.o - p;
    Tf d = length(u);
    if(d < c.r) ;
    else if(dcmp(d - c.r) == 0) {
      ret = { rotate(u, PI / 2) };
    }
    else {
      Tf ang = asin(c.r / d);
      ret = { rotate(u, -ang), rotate(u, ang) };
    }
    return ret;
  }

  // returns the points on tangents that touches the circle
  vector<Point> pointCircleTangencyPoints(Point p, Circle c) {
    static_assert(is_same<Tf, Ti>::value);

    Point u = p - c.o;
    Tf d = length(u);
    if(d < c.r) return {};
    else if(dcmp(d - c.r) == 0) return {c.o + u};
    else {
      Tf ang = acos(c.r / d);
      u = u / length(u) * c.r;
      return { c.o + rotate(u, -ang), c.o + rotate(u, ang) };
    }
  }

  // for two circles c1 && c2, returns two list of points a && b
  // such that a[i] is on c1 && b[i] is c2 && for every i
  // Line(a[i], b[i]) is a tangent to both circles
  // CAUTION: a[i] = b[i] in case they touch | -1 for c1 = c2
  int circleCircleTangencyPoints(Circle c1, Circle c2,
       vector<Point> &a, vector<Point> &b) {
    a.clear(), b.clear();
    int cnt = 0;
    if(dcmp(c1.r - c2.r) < 0) {
      swap(c1, c2); swap(a, b);
    }
    Tf d2 = sqLength(c1.o - c2.o);
    Tf rdif = c1.r - c2.r, rsum = c1.r + c2.r;
    if(dcmp(d2 - rdif * rdif) < 0) return 0;
    if(dcmp(d2) == 0 && dcmp(c1.r - c2.r) == 0) return -1;

    Tf base = angle(c2.o - c1.o);
    if(dcmp(d2 - rdif * rdif) == 0) {
      a.push_back(c1.point(base));
      b.push_back(c2.point(base));
      cnt++;
      return cnt;
    }

    Tf ang = acos((c1.r - c2.r) / sqrt(d2));
    a.push_back(c1.point(base + ang));
    b.push_back(c2.point(base + ang));
    cnt++;
    a.push_back(c1.point(base - ang));
    b.push_back(c2.point(base - ang));
    cnt++;

    if(dcmp(d2 - rsum * rsum) == 0) {
      a.push_back(c1.point(base));
      b.push_back(c2.point(PI + base));
      cnt++;
    }
    else if(dcmp(d2 - rsum * rsum) > 0) {
      Tf ang = acos((c1.r + c2.r) / sqrt(d2));
      a.push_back(c1.point(base + ang));
      b.push_back(c2.point(PI + base + ang));
      cnt++;
      a.push_back(c1.point(base - ang));
      b.push_back(c2.point(PI + base - ang));
      cnt++;
    }
    return cnt;
  }
} // namespace Circular

namespace EnclosingCircle{
  // returns false if points are collinear, true otherwise
  // circle p touch each arm of triangle abc
  bool inscribedCircle(Point a, Point b, Point c, Circle &p) {
    using Linear::distancePointLine;
    static_assert(is_same<Tf, Ti>::value);
    if(orient(a, b, c) == 0) return false;
    Tf u = length(b - c);
    Tf v = length(c - a);
```

```
    Tf w = length(a - b);
    p.o = (a * u + b * v + c * w) / (u + v + w);
    p.r = distancePointLine(p.o, Line(a, b));
    return true;
  }

  // set of points A(x, y) such that PA : QA = rp : rq
Circle apolloniusCircle(Point P, Point Q, Tf rp, Tf rq) {
  static_assert(is_same<Tf, Ti>::value);
  rq*=rq; rp*=rp; Tf a = rq-rp; assert(dcmp(a));
  Tf g = (rq*P.x-rp*Q.x)/a, h = (rq*P.y-rp*Q.y)/a;
  Tf c =  (rq * P.x * P.x - rp * Q.x * Q.x
      + rq * P.y * P.y - rp * Q.y * Q.y)/a;
  Point o(g, h); Tf R = sqrt(g * g + h * h - c);
  return Circle(o, R);
}
  // returns false if points are collinear
  bool circumscribedCircle(Point a, Point b, Point c, Circle
       &p) {
    using Linear::lineLineIntersection;
    if(orient(a, b, c) == 0) return false;
    Point d = (a + b) / 2, e = (a + c) / 2;
    Point vd = rotate90(b-a), ve = rotate90(a-c);
    bool f = lineLineIntersection(d,vd,e,ve,p.o);
    if(f) p.r = length(a - p.o);
    return f;
  }

  // Following three methods implement Welzl's algorithm for
  // the smallest enclosing circle problem: Given a set of
  // points, find out the minimal circle that covers them all.
  // boundary(p) determines (if possible) a circle that goes
  // through the points in p. Ideally |p| <= 3.
  // welzl() is a recursive helper function doing the most part
  // of Welzl's algorithm. Call minidisk with the set of points
  // Randomized Complexity: O(CN) with C~10 (practically lesser)
  // TESTED: ICPC Dhaka 2019 - I (CodeMarshal)

  Circle boundary(const vector<Point> &p) {
    Circle ret;
    int sz = p.size();
    if(sz == 0)      return Circle({0, 0}, 0);
    else if(sz == 1) ;
    else if(sz == 2) ret.o = (p[0] + p[1]) / 2, ret.r =
         length(p[0] - p[1]) / 2;
    else if(!circumscribedCircle(p[0], p[1], p[2], ret)) ret.r
         = 0;
    return ret;
  }
  Circle welzl(const vector<Point> &p, int fr, vector<Point>
       &b) {
    if(fr >= (int) p.size() || b.size() == 3) return
         boundary(b);

    Circle c = welzl(p, fr + 1, b);
    if(!c.contains(p[fr])) {
      b.push_back(p[fr]);
      c = welzl(p, fr + 1, b);
      b.pop_back();
    }
    return c;
  }
  Circle minidisk(vector<Point> p) {
    random_shuffle(p.begin(), p.end());
    vector<Point> q;
    return welzl(p, 0, q);
  }
}
// Given a bunch of segments. Check if any two intersect.
// Sweep Line. O(n lg n). TESTED: CF 1359F
namespace IntersectingSegments {
  Tf yvalSegment(const Line &s, Tf x) {
    if(dcmp(s.a.x - s.b.x) == 0) return s.a.y;
    return s.a.y + (s.b.y - s.a.y) * (x - s.a.x) / (s.b.x -
         s.a.x);
  }
  struct SegCompare {
    bool operator () (const Segment &p, const Segment &q)
         const {
      Tf x = max(min(p.a.x, p.b.x), min(q.a.x, q.b.x));
      return dcmp(yvalSegment(p, x) - yvalSegment(q, x)) < 0;
    }
  };
  multiset&lt;Segment, SegCompare&gt; st;
  typedef multiset&lt;Segment, SegCompare&gt;::iterator iter;
  iter prev(iter it) {
    return it == st.begin() ? st.end() : --it;
  }
  iter next(iter it) {
    return it == st.end() ? st.end() : ++it;
  }
  struct Event {
    Tf x;
    int tp, id;
    Event(Ti x, int tp, int id) : x(x), tp(tp), id(id) { }
    bool operator < (const Event &p) const {
      if(dcmp(x - p.x)) return x < p.x;
      return tp > p.tp;
    }
  };
  bool anyIntersection(const vector&lt;Segment&gt; &v) {
    using Linear::segmentsIntersect;
    vector<Event> ev;
    for(int i=0; i<(int) v.size(); ++i) {
      ev.push_back(Event(min(v[i].a.x, v[i].b.x), +1, i));
      ev.push_back(Event(max(v[i].a.x, v[i].b.x), -1, i));
    }
    sort(ev.begin(), ev.end());
    st.clear();
    vector<iter> where(v.size());
    for(auto &cur : ev) {
      int id = cur.id;
      if(cur.tp == 1) {
        iter nxt = st.lower_bound(v[id]);
        iter pre = prev(nxt);
        if(pre != st.end() && segmentsIntersect(*pre, v[id]))
            return true;
        if(nxt != st.end() && segmentsIntersect(*nxt, v[id]))
            return true;
        where[id] = st.insert(nxt, v[id]);
      }
      else {
        iter nxt = next(where[id]);
        iter pre = prev(where[id]);
        if(pre != st.end() && nxt != st.end() &&
            segmentsIntersect(*pre, *nxt))
          return true;
        st.erase(where[id]);
      }
    }
    return false;
  }
}
```

5.5 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(p2d 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;
}
```

5.6 geo-formulae

Triangle Centers and Radii

- Incenter: $(\frac{ax_1+bx_2+cx_3}{a+b+c}, \frac{ay_1+by_2+cy_3}{a+b+c})$.
- Inradius: $\sqrt{\frac{(s-a)(s-b)(s-c)}{s}}$.
- Excenter: $(\frac{-ax_1+bx_2+cx_3}{-a+b+c}, \frac{-ay_1+by_2+cy_3}{-a+b+c})$.
- Exradius: $\sqrt{\frac{s(s-b)(s-c)}{(s-a)}}$.
- Circumcenter: $(\frac{x_1 sin2A+x_2 sin2B+x_3 sin2C}{sin2A+sin2B+sin2C}, \frac{y_1 sin2A+y_2 sin2B+y_3 sin2C}{sin2A+sin2B+sin2C})$.
- Circumradius: $\frac{abc}{\sqrt{(a+b+c)(b+c-a)(c+a-b)(a+b-c)}}$.

5.7 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;
```

5.8 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;
}
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];
        }
    }
}
```

5.9 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);
}
```

5.10 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()){
            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;
}
```

6 Graph

6.1 DMST with solution

```
// not tested yet
const int INF = 1029384756;
#define MAXN 1000
#define FOR(i,x) for(auto i :x )
struct edge_t {
    int u,v,w;
    set< pair<int,int> > add, sub;
    edge_t() : u(-1), v(-1), w(0) {}
    edge_t(int _u, int _v, int _w) {
        u = _u;
        v = _v;
        w = _w;
        add.insert({u, v});
    }
    edge_t& operator += (const edge_t& obj) {
        w += obj.w;
        for (auto it : obj.add) {
            if (!sub.count(it)) add.insert(it);
            else sub.erase(it);
        }
        for (auto it : obj.sub) {
            if (!add.count(it)) sub.insert(it);
            else add.erase(it);
        }
        return *this;
    }
    edge_t& operator -= (const edge_t& obj) {
        w -= obj.w;
        for (auto it : obj.sub) {
            if (!sub.count(it)) add.insert(it);
            else sub.erase(it);
        }
        for (auto it : obj.add) {
            if (!add.count(it)) sub.insert(it);
            else add.erase(it);
        }
        return *this;
    }
} eg[MAXN*MAXN],prv[MAXN],EDGE_INF(-1,-1,INF);
int N,M;
int cid,incyc[MAXN],contracted[MAXN];
vector<int> E[MAXN];
edge_t dmst(int rt) {
    edge_t cost;
    for (int i=0; i<N; i++) {
        contracted[i] = incyc[i] = 0;
        prv[i] = EDGE_INF;
    }
    cid = 0;
    int u,v;
    while (true) {
        for (v=0; v<N; v++) {
            if (v != rt && !contracted[v] && prv[v].w == INF)
                break;
        }
        if (v >= N) break; // end
        for (int i=0; i<M; i++) {
            if (eg[i].v == v && eg[i] .w < prv[v].w)
                prv[v] = eg[i];
        }
        if (prv[v].w == INF) // not connected
            return EDGE_INF;
        cost += prv[v];
        for (u=prv[v].u; u!=v && u!=-1; u=prv[u].u);
        if (u == -1) continue;
        incyc[v] = ++cid;
        for (u=prv[v].u; u!=v; u=prv[u].u) {
            contracted[u] = 1;
            incyc[u] = cid;
        }
        for (int i=0; i<M; i++) {
            if (incyc[eg[i].u] != cid && incyc[eg[i].v] ==
                cid) {
                eg[i] -= prv[eg[i].v];
            }
        }
        for (int i=0; i<M; i++) {
            if (incyc[eg[i].u] == cid) eg[i].u = v;
            if (incyc[eg[i].v] == cid) eg[i].v = v;
            if (eg[i].u == eg[i].v) eg[i--] = eg[--M];
        }
        for (int i=0; i<N; i++) {
            if (contracted[i]) continue;
            if (prv[i].u>=0 && incyc[prv[i].u] == cid)
                prv[i].u = v;
        }
        prv[v] = EDGE_INF;
    }
    return cost;
}
#define F first
#define S second
void solve() {
    edge_t cost = dmst(0);
    for (auto it : cost.add) { // find a solution
        E[it.F].push_back(it.S);
        prv[it.S] = edge_t(it.F,it.S,0);
    }
}
```

6.2 DMST

```
// tested on https://lightoj.com/problem/teleport
const int inf = 1e9 ;
struct edge {
    int u, v, w;
    edge() {}
    edge(int a,int b,int c) : u(a), v(b), w(c) {}
    bool operator < (const edge& o) const {
        if (u == o.u)
            if (v == o.v)return w < o.w;
            else return v < o.v;
        return u < o.u;
    }
};
int dmst(vector<edge> &edges, int root, int n) {
    int ans = 0;
    int cur_nodes = n;
    while (true) {
        vector<int> lo(cur_nodes, inf), pi(cur_nodes, inf);
        for (int i = 0; i < edges.size(); ++i) {
            int u = edges[i].u, v = edges[i].v, w = edges[i].w;
            if (w < lo[v] and u != v) {
                lo[v] = w;
                pi[v] = u;
            }
        }
        lo[root] = 0;
        for (int i = 0; i < lo.size(); ++i) {
            if (i == root) continue;
            if (lo[i] == inf) return -1;
        }
        int cur_id = 0;
        vector<int> id(cur_nodes, -1), mark(cur_nodes, -1);
        for (int i = 0; i < cur_nodes; ++i) {
            ans += lo[i];
            int u = i;
            while (u != root and id[u] < 0 and mark[u] != i) {
                mark[u] = i;
                u = pi[u];
            }
            if (u != root and id[u] < 0) { // Cycle
                for (int v = pi[u]; v != u; v = pi[v]) id[v] =
                    cur_id;
                id[u] = cur_id++;
            }
        }
        if (cur_id == 0) break;
        for (int i = 0; i < cur_nodes; ++i)
            if (id[i] < 0) id[i] = cur_id++;
        for (int i = 0; i < edges.size(); ++i) {
            int u = edges[i].u, v = edges[i].v, w = edges[i].w;
            edges[i].u = id[u];
            edges[i].v = id[v];
            if (id[u] != id[v]) edges[i].w -= lo[v];
        }
        cur_nodes = cur_id;
        root = id[root];
    }
    return ans;
}
```

6.3 Flow With Demands

Finding an arbitrary flow Consider flow networks, where we additionally require the flow of each edge to have a certain amount, i.e. we bound the flow from below by a **demand** function $d(e)$:

$$d(e) \leq f(e) \leq c(e)$$

So next each edge has a minimal flow value, that we have to pass along the edge.

We make the following changes in the network. We add a new source s' and a new sink t', a new edge from the source s' to every other vertex, a new edge for every vertex to the sink t', and one edge from t to s. Additionally we define the new capacity function c' as:

- $c'((s', v)) = \sum_{u \in V} d((u, v))$ for each edge (s', v).
- $c'((v, t')) = \sum_{w \in V} d((v, w))$ for each edge (v, t').
- $c'((u, v)) = c((u, v)) - d((u, v))$ for each edge (u, v) in the old network.
- $c'((t, s)) = \infty$

If the new network has a saturating flow (a flow where each edge outgoing from s' is completely filled, which is equivalent to every edge incoming to t' is completely filled), then the network with demands has a valid flow, and the actual flow can be easily reconstructed from the new network. Otherwise there doesn't exist a flow that satisfies all conditions. Since a saturating flow has to be a maximum flow, it can be found by any maximum flow algorithm.

Minimal flow Note that along the edge (t, s) (from the old sink to the old source) with the capacity ∞ flows the entire flow of the corresponding old network. I.e. the capacity of this edge effects the flow value of the old network. By giving this edge a sufficient large capacity (i.e. ∞), the flow of the old network is unlimited. By limiting this edge by smaller capacities, the flow value will decrease. However if we limit this edge by a too small value, than the network will not have a saturated solution, e.g. the corresponding solution for the original network will not satisfy the demand of the edges. Obviously here can use a binary search to find the lowest value with which all constraints are still satisfied. This gives the minimal flow of the original network.

6.4 articulation-vertex

```
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);
    }
}
```

6.5 bellman-ford

```
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;
    }
}
```

6.6 bridge

```
const int vmax = 2e5+10, emax = 2e5+10;
namespace Bridge {///edge, nodes, comps 1 indexed
  vector<int> adj[vmax]; /// edge-id
  pair<int, int> edges[emax]; /// (u, v)
  bool isBridge[emax];
  int visited[vmax]; ///0-unvis,1-vising,2-vis
  int st[vmax], low[vmax], clk = 0, edgeId = 0;
  /// For bridge tree components
  int who[vmax], compId = 0;
  vector<int> stk;
  /// For extra end time calc
  int en[vmax];
  void dfs(int u, int parEdge) {
    visited[u] = 1; low[u] = st[u] = ++clk;
    stk.push_back(u);
    for (auto e : adj[u]) {
      if (e == parEdge) continue;
      int v=edges[e].first^edges[e].second^u;
      if (visited[v] == 1) {
        low[u] = min(low[u], st[v]);
      } else if(visited[v] == 0){
        dfs(v, e); low[u] = min(low[u], low[v]);
      }
    }
    visited[u] = 2;
    if(st[u] == low[u]){/// found
      ++compId; int cur;
      do{
        cur = stk.back(); stk.pop_back();
        who[cur] = compId;
      }while(cur != u);
      if(parEdge != -1){isBridge[parEdge] = true;}
    }
    en[u] = clk;
  }
  void clearAll(int n){
    for(int i = 0; i<=n; i++) {
      adj[i].clear(); visited[i] = st[i] = 0; }
    for(int i = 0; i<=edgeId; i++) isBridge[i]=0;
    clk = compId = edgeId = 0;
  }
  void findBridges(int n){
    for(int i = 1; i<=n; i++){
      if(visited[i] == 0) dfs(i, -1); }
  }
  bool isReplacable(int eid, int u, int v){
    if(!isBridge[eid]) return true;
    int a=edges[eid].first,b=edges[eid].second;
    if(st[a] > st[b]) swap(a, b);
    return (st[b] <= st[u] && st[u] <= en[b])
    != (st[b] <= st[v] && st[v] <= en[b]);
  }
  void addEdge(int u, int v){
    edgeId++; edges[edgeId] = {u, v};
    adj[u].emplace_back(edgeId);
    adj[v].emplace_back(edgeId);
  }
}
```

6.7 edmond-blossom

```
/***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;
```

6.8 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 << " ";
    }
}
```

6.9 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.
}
```

6.10 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;
    }
};
```

6.11 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;
}
```

6.12 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;
}
```

6.13 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);
    }
}
```

6.14 scc + 2 Sat

```
namespace SCC { //Everything 0-indexed.
const int N = 2e6+7; int which[N], vis[N], cc;
vector<int> adj[N], adjr[N]; vector<int> order;
void addEdge(int u, int v) {
  adj[u].push_back(v); adjr[v].push_back(u);
}
void dfs1(int u){
  if (vis[u]) return; vis[u] = true;
  for(int v: adj[u]) dfs1(v); order.push_back(u);
}
void dfs2(int u, int id) {
  if(vis[u]) return; vis[u] = true;
  for(int v: adjr[u]) dfs2(v, id); which[u] = id;
}
int last = 0;
void findSCC(int n) {
  cc=0,last=n; order.clear(); fill(vis, vis+n, 0);
  for(int i=0; i<n; i++) if(!vis[i]) dfs1(i);
  reverse(order.begin(), order.end());
  fill(vis, vis+n, 0);
  for (int u: order) {
    if (vis[u])  continue; dfs2(u, cc); ++cc;
  }
}
void clear() {
  for (int i=0; i<last; i++)
    adj[i].clear(), adjr[i].clear();
} }
struct TwoSat {
  int n; int vars = 0; vector<bool> ans;
  TwoSat(int n) : n(n), ans(n) {
    SCC::clear(); vars = 2*n;
  }
  void implies(int x, int y) {
    SCC::addEdge(x, y); SCC::addEdge(y^1, x^1);
  }
  void OR(int x, int y) {
    SCC::addEdge(x^1, y); SCC::addEdge(y^1, x);
  }
  void XOR(int x, int y) {
    implies(x, y^1); implies(x^1, y);
  }
  void atmostOne(vector<int> v) {
    int k = v.size();
    for (int i=0; i<k; i++) {
      if (i+1<k)   implies(vars+2*i, vars+2*i+2);
      implies(v[i], vars+2*i);
      if (i>0)   implies(v[i], vars+2*i-1);
```

```
    }
    vars += 2*k;
  }
  bool solve() {
    SCC::findSCC(vars); ans.resize(vars/2);
    for (int i=0; i<vars; i+=2) {
      if (SCC::which[i]==SCC::which[i+1])return 0;
      if (i<2*n)
        ans[i/2] = SCC::which[i]>SCC::which[i+1];
    }
    return true;
  }
};
```

7 Math

7.1 BerleKampMassey

```
const int MOD = 998244353;
vector<LL> berlekampMassey(vector<LL> s) {
  if (s.empty()) return {};
  int n = s.size(), L = 0, m = 0;
  vector<LL> C(n), B(n), T;
  C[0] = B[0] = 1; LL b = 1;
  for (int i = 0; i < n; ++i) {
    ++m; LL d = s[i] % MOD;
    for (int j = 1; j <= L; ++j) d = (d + C[j] * s[i - j]) %
         MOD;
    if (!d) continue;
    T = C; LL coeff = d * bigMod(b, -1) % MOD;
    for (int j = m; j < n; ++j) C[j] = (C[j] - coeff * B[j -
         m]) % MOD;
    if (2*L > i) continue;
    L = i+1-L, B = T, b = d, m = 0;
  }
  C.resize(L + 1), C.erase(C.begin());
  for (LL &x : C) x = (MOD - x) % MOD;
  return C;
}
```

7.2 FloorSum

```
LL mod(LL a, LL m) {
    LL ans = a%m;
    return ans < 0 ? ans+m : ans;
}
///Sum(floor((ax+b)/m)) for i=0 to n-1, (n,m >= 0)
LL floorSum(LL n, LL m, LL a, LL b) {
    LL ra = mod(a, m), rb = mod(b, m), k = (ra*n+rb);
    LL ans = ((a-ra)/m) * n*(n-1)/2 + ((b-rb)/m) * n;
    if (k < m) return ans;
    return ans + floorSum(k/m, ra, m, k%m);
}
```

7.3 Stern Brocot Tree

```
//finds x/y with min y st: L <= (x/y) < R
pair<LL,LL>solve(LD L, LD R){
  pair<LL, LL> l(0, 1), r(1, 1);
  if(L==0.0) return l; // corner case
  while(true) {
    pair<int, int> m(l.x+r.x, l.y+r.y);
    if(m.x<L*m.y){ // move to the right
      LL kl=1, kr=1;
      while(l.x+kr*r.x <= L*(l.y+kr*r.y)) kr*=2;
      while(kl!=kr){
        LL km = (kl+kr)/2;
        if(l.x+km*r.x < L*(l.y+km*r.y)) kl=km+1;
        else kr=km;
      }
      l={l.x+(kl-1)*r.x,l.y+(kl-1)*r.y};
    }
    else if(m.x>=R*m.y){//move to left
      LL kl=1, kr=1;
      while(r.x+kr*l.x>=R*(r.y+kr*l.y)) kr*=2;
      while(kl!=kr){
        LL km = (kl+kr)/2;
        if(r.x+km*l.x>=R*(r.y+km*l.y)) kl = km+1;
        else kr = km;
      }
      r={r.x+(kl-1)*l.x,r.y+(kl-1)*l.y};
```

```
    }
    else return m;
  }
}
```

7.4 Sum Of Kth Power

```
LL mod; LL S[105][105];
// Find 1^k+2^k+...+n^k % mod
void solve() {
  LL n, k;
/* x^k = sum (i=1 to k) Stirling2(k, i) * i! * ncr(x, i)
sum (x = 0 to n) x^k
= sum (i=0 to k) Stirling2(k, i) * i! * sum (x=0 to n)ncr(x, i)
= sum (i=0 to k) Stirling2(k, i) * i! * ncr(n + 1, i + 1)
= sum (i=0 to k) Stirling2(k, i) * i! * (n + 1)! /(i+1)!/(n-i)!
= sum (i=0 to k) Stirling2(k, i) * (n - i + 1) *
    (n - i + 2) * ... (n + 1) / (i + 1) */
  S[0][0] = 1 % mod;
  for (int i = 1; i <= k; i++) {
    for (int j = 1; j <= i; j++) {
if (i == j) S[i][j] = 1 % mod;
else S[i][j] = (j * S[i - 1][j] + S[i - 1][j - 1]) % mod;
    }
  }
  LL ans = 0;
  for (int i = 0; i <= k; i++) {
    LL fact = 1, z = i + 1;
    for (LL j = n - i + 1; j <= n + 1; j++) {
      LL mul = j;
      if (mul % z == 0) {
        mul /= z; z /= z;
      }
      fact = (fact * mul) % mod;
    }
    ans = (ans + S[k][i] * fact) % mod;
  }
}
```

7.5 combination-generator

```
bool next_combination(vector<int>& a, int n) {
    int k = (int)a.size();
    for (int i = k - 1; i >= 0; i--) {
        if (a[i] < n - k + i + 1) {
            a[i]++;
            for (int j = i + 1; j < k; j++)
                a[j] = a[j - 1] + 1;
            return true;
        }
    }
    return false;
}
vector<int> ans;
void gen(int n, int k, int idx, bool rev) {
    if (k > n || k < 0)
        return;
    if (!n) {
        for (int i = 0; i < idx; ++i) {
            if (ans[i])
                cout << i + 1;
        }
        cout << "\n";
        return;
    }
    ans[idx] = rev;
    gen(n - 1, k - rev, idx + 1, false);
    ans[idx] = !rev;
    gen(n - 1, k - !rev, idx + 1, true);
}
void all_combinations(int n, int k) {
    ans.resize(n);
    gen(n, k, 0, false);
}
```

7.6 continued-fractions

```
auto fraction(int p, int q) {
    vector<int> a;
    while(q) {
        a.push_back(p / q);
        tie(p, q) = make_pair(q, p % q);
    }
    return a;
}
auto convergents(vector<int> a) {
    vector<int> p = {0, 1};
    vector<int> q = {1, 0};
    for(auto it: a) {
        p.push_back(p[p.size() - 1] * it + p[p.size() - 2]);
        q.push_back(q[q.size() - 1] * it + q[q.size() - 2]);
    }
    return make_pair(p, q);
}
```

7.7 crt anachor

```
/// Chinese remainder theorem (special case): find z st z%m1 =
     r1, z%m2 = r2.
/// z is unique modulo M = lcm(m1, m2). Returns (z, M). On
     failure, M = -1.
PLL CRT(LL m1, LL r1, LL m2, LL r2) {
  LL s, t;
  LL g = egcd(m1, m2, s, t);
  if (r1%g != r2%g) return PLL(0, -1);
  LL M = m1*m2;
  LL ss = ((s*r2)%m2)*m1;
  LL tt = ((t*r1)%m1)*m2;
  LL ans = ((ss+tt)%M+M)%M;
  return PLL(ans/g, M/g);
}
// expected: 23 105
//           11 12
PLL ans = CRT({3,5,7}, {2,3,2});
cout << ans.first << " " << ans.second << endl;
ans = CRT({4,6}, {3,5});
cout << ans.first << " " << ans.second << endl;
```

7.8 discrete-root

```
#define MAX 100000
int prime[MAX+1],Phi[MAX+1];
vector<int> pr;
void sieve(){
    for (int i=2; i <= N; ++i) {
        if (prime[i] == 0) {
            prime[i] = i;
            pr.push_back(i);
        }
        for (int j=0; j < (int)pr.size() && pr[j] <= prime[i]
             && i*pr[j] <= N; ++j) {
            prime[i * pr[j]] = pr[j];
        }
    }
}
void PhiWithSieve(){
    int i;
    for(i=2; i<=MAX; i++){
        if(prime[i]==i){
            Phi[i]=i-1;
        }
        else if((i/prime[i])%prime[i]==0){
            Phi[i]=Phi[i/prime[i]]*prime[i];
        }
        else{
            Phi[i]=Phi[i/prime[i]]*(prime[i]-1);
        }
    }
}
int powmod (int a, int b, int p) {
    int res = 1;
    while (b)
        if (b & 1)
            res = int (res * 1ll * a % p), --b;
        else
            a = int (a * 1ll * a % p), b >>= 1;
    return res;
}
int PrimitiveRoot(int p){
```

```
    vector<int>fact;
    int phi=Phi[p];
    int n=phi;
    while(n>1){
        if(prime[n]==n){
            fact.push_back(n);
            n=1;
        }
        else{
            int f=prime[n];
            while(n%f==0){
                n=n/f;
            }
            fact.push_back(f);
        }
    }
    int res;
    for(res=p-1; res>1; --res){
        for(n=0; n<fact.size(); n++){
            if(powmod(res,phi/fact[n],p)==1){
                break;
            }
        }
        if(n>=fact.size()) return res;
    }
    return -1;
}
int DiscreteLog(int a, int b, int m) {
    a %= m, b %= m;
    int n = sqrt(m) + 1;
    map<int, int> vals;
    for (int p = 1; p <= n; ++p)
        vals[powmod(a,(int) (1ll* p * n) %m , m)] = p;
    for (int q = 0; q <= n; ++q) {
        int cur = (powmod(a, q, m) * 1ll * b) % m;
        if (vals.count(cur)) {
            int ans = vals[cur] * n - q;
            return ans;
        }
    }
    return -1;
}
vector<int> DiscreteRoot(int n,int a,int k){
    int g = PrimitiveRoot(n);
    vector<int> ans;
    int any_ans = DiscreteLog(powmod(g,k,n),a,n);
    if (any_ans == -1){
        return ans;
    }
    int delta = (n-1) / gcd(k, n-1);
    for (int cur = any_ans % delta; cur < n-1; cur += delta)
        ans.push_back(powmod(g, cur, n));
    sort(ans.begin(), ans.end());
    return ans;
}
```

7.9 fast-fourier-transform

```
using cd = complex<double>;
const double PI = acos(-1);
typedef long long ll;
void fft(vector<cd>& a, bool invert){
    int n = a.size();
    for(int i = 1, j = 0; i < n; i++){
        int bit = n>>1;
        for(; j&bit; bit>>=1){
            j^=bit;
        }
        j ^= bit;
        if(i < j)
            swap(a[i], a[j]);
    }
    for(int len = 2; len <= n; len <<= 1){
        double ang = 2*PI/len*(invert ? -1 : 1);
        cd wlen(cos(ang), sin(ang));
        for(int i = 0; i < n; i += len){
            cd w(1);
            for(int j = 0; j < len/2; j++){
                cd u = a[i+j], v = a[i+j+len/2]*w;
                a[i+j] = u+v;
                a[i+j+len/2] = u-v;
                w *= wlen;
            }
        }
    }
    if(invert){
        for(cd &x: a)
            x /= n;
    }
}
vector<int> multiply(vector<int> const& a, vector<int>
     const&b){
    vector<cd> fa(a.begin(), a.end());
    vector<cd> fb(b.begin(), b.end());
    int n = 1;
    while(n < a.size()+b.size())
        n <<= 1;
    fa.resize(n);
    fb.resize(n);
    fft(fa, false);
    fft(fb, false);
    for(int i = 0; i < n; i++)
        fa[i] *= fb[i];
    fft(fa, true);
    vector<int> result(n);
    for(int i = 0; i < n; i++)
        result[i] = round(fa[i].real());
    return result;
}
/*Number Theoretic Transformation
ll int gcd(ll int a,ll int b){
    if(b==0) return a;
    else return gcd(b,a%b);
}
ll int egcd(ll int a, ll int b, ll int & x, ll int & y) {
    if (a == 0) {
        x = 0;
        y = 1;
        return b;
    }
    ll int x1, y1;
    ll int d = egcd(b % a, a, x1, y1);
    x = y1 - (b / a) * x1;
    y = x1;
    return d;
}
ll int ModuloInverse(ll int a,ll int n){
    ll int x,y;
    x=gcd(a,n);
    a=a/x;
    n=n/x;
    ll int res = egcd(a,n,x,y);
    x=(x%n+n)%n;
    return x;
}
const int mod = 998244353;
const int root = 15311432;
const int root_1 = 469870224;
const int root_pw = 1 << 23;
998244353 = 119 * 2^23 + 1 , primitive root = 3
985661441 = 235 * 2^22 + 1 , primitive root = 3
1012924417 = 483 * 2^21 + 1 , primitive root = 5
void fft(vector<int> & a, bool invert) {
    int n = a.size();
    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j & bit; bit >>= 1)
            j ^= bit;
        j ^= bit;
        if (i < j)
            swap(a[i], a[j]);
    }
    for (int len = 2; len <= n; len <<= 1) {
        int wlen = invert ? root_1 : root;
        for (int i = len; i < root_pw; i <<= 1)
            wlen = (int)(1LL * wlen * wlen % mod);
        for (int i = 0; i < n; i += len) {
            int w = 1;
            for (int j = 0; j < len / 2; j++) {
                int u = a[i+j], v = (int)(1LL * a[i+j+len/2] * w
                     % mod);
                a[i+j] = u + v < mod ? u + v : u + v - mod;
                a[i+j+len/2] = u - v >= 0 ? u - v : u - v + mod;
                w = (int)(1LL * w * wlen % mod);
            }
        }
    }
    if (invert) {
        int n_1 = (int) ModuloInverse(n, mod);
        for (int & x : a)
            x = (int)(1LL * x * n_1 % mod);
    }
}
vector<int> multiply(vector<int> const& a, vector<int> const&b){
    vector<int> fa(a.begin(), a.end());
    vector<int> fb(b.begin(), b.end());
    int n = 1;
    while(n < a.size()+b.size())
        n <<= 1;
    fa.resize(n);
    fb.resize(n);
    fft(fa, false);
    fft(fb, false);
    for(int i = 0; i < n; i++)
        fa[i] = (int) (1LL*fa[i]*fb[i]%mod);
    fft(fa, true);
    vector<int> result(n);
    for(int i = 0; i < n; i++)
        result[i] = fa[i];
    return result;
}
*/
```

7.10 fast-walsh-hadamard

```
void FWHT(vector<LL> &p, bool inv) {
  int n = p.size(); assert((n&(n-1))==0);
  for (int len=1; 2*len<=n; len <<= 1) {
    for (int i = 0; i < n; i += len+len){
      for (int j = 0; j < len; j++) {
        LL u = p[i+j], v = p[i+len+j];
///XOR  p[i+j]=u+v; p[i+len+j]=u-v;
////OR   if(!inv)   p[i+j]=v, p[i+len+j]=u+v;
////OR   else       p[i+j]=-u+v, p[i+len+j]=u;
///AND  if(!inv)   p[i+j]=u+v, p[i+len+j]=u;
///AND  else       p[i+j]=v, p[i+len+j]=u-v;
        }
      }
  }
///XOR if(inv) for(int i=0;i<n;i++) p[i]/=n;
}
vector<LL> convo(vector<LL> a, vector<LL> b) {
  int n = 1, sz = max(a.size(), b.size());
  while(n<sz) n*=2;
  a.resize(n); b.resize(n); vector<LL>res(n, 0);
  FWHT(a, 0); FWHT(b, 0);
  for(int i=0;i<n;i++) res[i] = a[i] * b[i];
  FWHT(res, 1);
  return res;
}
```

7.11 find-root

```
double sqrt_newton(double n) {
    const double eps = 1E-15;
    double x = 1;
    for (;;) {
        double nx = (x + n / x) / 2;
        if (abs(x - nx) < eps)
            break;
        x = nx;
    }
    return x;
}
int isqrt_newton(int n) {
    int x = 1;
    bool decreased = false;
    for (;;) {
```

```
        int nx = (x + n / x) >> 1;
        if (x == nx || nx > x && decreased)
            break;
        decreased = nx < x;
        x = nx;
    }
    return x;
}
```

7.12 formulas

Binomial Coefficient List

- $\sum_{m=0}^{n} \binom{m}{k} = \binom{n+1}{k+1}$.
- $\sum_{k=0}^{m} \binom{n+k}{k} = \binom{n+m+1}{m}$.
- $\binom{n}{0}^2 + \binom{n}{1}^2 + \cdots + \binom{n}{n}^2 = \binom{2n}{n}$.
- $1\binom{n}{1} + 2\binom{n}{2} + \cdots + n\binom{n}{n} = n2^{n-1}$
- $\binom{n}{0} + \binom{n-1}{1} + \cdots + \binom{n-k}{k} + \cdots + \binom{0}{n} = F_{n+1}$

Fibonacci Numbers

- $F_{n-1}F_{n+1} - {F_n}^2 = (-1)^n$.
- $F_{n+k} = F_k F_{n+1} + F_{k-1} F_n$.
- $GCD(F_m, F_n) = F_{GCD(m,n)}$.

7.13 integer-factorization

```
long long pollards_p_minus_1(long long n) {
    int B = 10; long long g = 1;
    while (B <= 1000000 && g < n) {
        long long a = 2 + rand() % (n - 3); g = gcd(a, n);
        if (g > 1) return g;
        // compute a^M
        for (int p : primes) {
            if (p >= B) continue; long long p_power = 1;
            while (p_power * p <= B) p_power *= p;
            a = power(a, p_power, n); g = gcd(a - 1, n);
            if (g > 1 && g < n) return g;
        }
        B *= 2;
    }
    return 1;
}
long long mult(long long a, long long b, long long mod) {
    long long result = 0;
    while (b) {
        if (b & 1) result = (result + a) % mod;
        a = (a + a) % mod; b >>= 1;
    }
    return result;
}
long long f(long long x, long long c, long long mod) {
    return (mult(x, x, mod) + c) % mod;
}
long long rho(long long n, long long x0=2, long long c=1) {
    long long x = x0; long long y = x0; long long g = 1;
    while (g == 1) {
        x = f(x, c, n); y = f(y, c, n); y = f(y, c, n);
        g = gcd(abs(x - y), n);
    }
    return g;
}
long long brent(long long n, long long x0=2, long long c=1) {
    long long x = x0; long long g = 1; long long q = 1;
    long long xs, y; int m = 128; int l = 1;
    while (g == 1) {
        y = x; int k = 0;
        for (int i = 1; i < l; i++) x = f(x, c, n);
        while (k < l && g == 1) {
            xs = x;
            for (int i = 0; i < m && i < l - k; i++) {
                x = f(x, c, n); q = mult(q, abs(y - x), n);
            }
            g = gcd(q, n); k += m;
        }
        l *= 2;
    }
    if (g == n) {
        do {
            xs = f(xs, c, n); g = gcd(abs(xs - y), n);
        } while (g == 1);
    }
    return g;
}
```

7.14 integration-simpson

```
const int N = 1000 * 1000; // number of steps (already
    multiplied by 2)
double simpson_integration(double a, double b){
    double h = (b - a) / N;
    double s = f(a) + f(b); // a = x_0 and b = x_2n
    for (int i = 1; i <= N - 1; ++i) { // Refer to final
        Simpson's formula
        double x = a + h * i;
        s += f(x) * ((i & 1) ? 4 : 2);
    }
    s *= h / 3;
    return s;
}
```

7.15 linear-diophantine-equation-gray-code

```
int gcd(int a, int b, int& x, int& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    int x1, y1;
    int d = gcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a / b);
    return d;
}
bool find_any_solution(int a, int b, int c, int &x0, int &y0,
    int &g) {
    g = gcd(abs(a), abs(b), x0, y0);
    if (c % g) {
        return false;
    }
    x0 *= c / g;
    y0 *= c / g;
    if (a < 0) x0 = -x0;
    if (b < 0) y0 = -y0;
    return true;
}
void shift_solution(int & x, int & y, int a, int b, int cnt) {
    x += cnt * b;
    y -= cnt * a;
}
int find_all_solutions(int a, int b, int c, int minx, int
    maxx, int miny, int maxy) {
    int x, y, g;
    if (!find_any_solution(a, b, c, x, y, g))
        return 0;
    a /= g;
    b /= g;
    int sign_a = a > 0 ? +1 : -1;
    int sign_b = b > 0 ? +1 : -1;
    shift_solution(x, y, a, b, (minx - x) / b);
    if (x < minx)
        shift_solution(x, y, a, b, sign_b);
    if (x > maxx)
        return 0;
    int lx1 = x;
    shift_solution(x, y, a, b, (maxx - x) / b);
    if (x > maxx)
        shift_solution(x, y, a, b, -sign_b);
    int rx1 = x;
    shift_solution(x, y, a, b, -(miny - y) / a);
    if (y < miny)
        shift_solution(x, y, a, b, -sign_a);
    if (y > maxy)
        return 0;
    int lx2 = x;
    shift_solution(x, y, a, b, -(maxy - y) / a);
    if (y > maxy)
        shift_solution(x, y, a, b, sign_a);
    int rx2 = x;
    if (lx2 > rx2)
        swap(lx2, rx2);
    int lx = max(lx1, lx2);
    int rx = min(rx1, rx2);
    if (lx > rx)
        return 0;
    return (rx - lx) / abs(b) + 1;
}
int g (int n) {
    return n ^ (n >> 1);
}
int rev_g (int g) {
  int n = 0;
  for (; g; g >>= 1)
    n ^= g;
  return n;
}
```

7.16 linear-equation-system

```
const double EPS = 1e-9;
const int INF = 2; // it doesn't actually have to be infinity
    or a big number
int gauss (vector < vector<double> > a, vector<double> & ans) {
    int n = (int) a.size();
    int m = (int) a[0].size() - 1;
    vector<int> where (m, -1);
    for (int col=0, row=0; col<m && row<n; ++col) {
        int sel = row;
        for (int i=row; i<n; ++i)
            if (abs (a[i][col]) > abs (a[sel][col]))
                sel = i;
        if (abs (a[sel][col]) < EPS)
            continue;
        for (int i=col; i<=m; ++i)
            swap (a[sel][i], a[row][i]);
        where[col] = row;
        for (int i=0; i<n; ++i)
            if (i != row) {
                double c = a[i][col] / a[row][col];
                for (int j=col; j<=m; ++j)
                    a[i][j] -= a[row][j] * c;
            }
        ++row;
    }
    ans.assign (m, 0);
    for (int i=0; i<m; ++i)
        if (where[i] != -1)
            ans[i] = a[where[i]][m] / a[where[i]][i];
    for (int i=0; i<n; ++i) {
        double sum = 0;
        for (int j=0; j<m; ++j)
            sum += ans[j] * a[i][j];
        if (abs (sum - a[i][m]) > EPS)
            return 0;
    }
    for (int i=0; i<m; ++i)
        if (where[i] == -1)
            return INF;
    return 1;
}
```

7.17 matrix-determinant

```
const double EPS = 1E-9;
int n;
vector < vector<double> > a (n, vector<double> (n));
double det = 1;
for (int i=0; i<n; ++i) {
    int k = i;
    for (int j=i+1; j<n; ++j)
        if (abs (a[j][i]) > abs (a[k][i]))
            k = j;
    if (abs (a[k][i]) < EPS) {
        det = 0;
        break;
    }
    swap (a[i], a[k]);
    if (i != k)
        det = -det;
    det *= a[i][i];
    for (int j=i+1; j<n; ++j)
        a[i][j] /= a[i][i];
    for (int j=0; j<n; ++j)
        if (j != i && abs (a[j][i]) > EPS)
            for (int k=i+1; k<n; ++k)
                a[j][k] -= a[i][k] * a[j][i];
}
cout << det;
```

7.18 matrix-rank

```
const double EPS = 1E-9;
int compute_rank(vector<vector<double>> A) {
    int n = A.size(); int m = A[0].size(); int rank = 0;
    vector<bool> row_selected(n, false);
    for (int i = 0; i < m; ++i) {
        int j;
        for (j = 0; j < n; ++j) {
            if (!row_selected[j] && abs(A[j][i]) > EPS) break;
        }
        if (j != n) {
            ++rank; row_selected[j] = true;
            for (int p = i + 1; p < m; ++p) A[j][p] /= A[j][i];
            for (int k = 0; k < n; ++k) {
                if (k != j && abs(A[k][i]) > EPS) {
                    for (int p = i + 1; p < m; ++p)
                        A[k][p] -= A[j][p] * A[k][i];
                } } } }
    return rank;
}
```

7.19 nCr mod p^a

```
LL F[1000009];
void pre(LL mod,LL pp){ // mod is pp^a, pp is prime
    F[0] = 1;
    REPL(i,1,mod){ // we keep in F factorial with
    // the terms which are coprime with pp
        if(i%pp!= 0) F[i]=(F[i-1]*i)%mod;
        else F[i]=F[i-1];
    }
}
LL fact2(LL nn,LL mod){
    LL cycle = nn/mod;
    LL n2=nn%mod;
    return (bigmod(F[mod],cycle,mod)*F[n2])%mod;
}
// returns highest power of pp that divides N and the coprime
// with pp part of N! %mod
PLL fact(LL N,LL pp,LL mod){
    LL nn = N; LL ord = 0;
    while(nn > 0){nn /= pp;ord += nn;}
    LL ans = 1; nn = N;
    while(nn > 0){ ans =(ans*fact2(nn,mod))%mod;
        nn/=pp;}
    return MP(ord,ans);
}
LL ncrp(ULL n,ULL r,LL prm,LL pr){ //ncr mod prm^pr
  LL mod=bigmod(prm,pr,INF),temp;
  pre(mod,prm);
PLL x=fact(n,prm,mod),y=fact(r,prm,mod),z=fact(n-r,prm,mod);
  LL guun=x.second*modInverse(y.second,mod,prm);
  guun%=mod;guun*=modInverse(z.second,mod,prm);
  guun%=mod;
  LL guun2=x.first-y.first-z.first;
  guun*=bigmod(prm,guun2,mod);
  guun%=mod;
  return guun;
}
```

7.20 primality-test

```
using u64 = uint64_t;
using u128 = __uint128_t;
u64 binpower(u64 base, u64 e, u64 mod) {
    u64 result = 1; base %= mod;
    while (e) {
        if (e & 1) result = (u128)result * base % mod;
        base = (u128)base * base % mod; e >>= 1;
    }
    return result;
}
bool check_composite(u64 n, u64 a, u64 d, int s) {
    u64 x = binpower(a, d, n);
    if (x == 1 || x == n - 1) return false;
    for (int r = 1; r < s; r++) {
        x = (u128)x * x % n;
        if (x == n - 1) return false;
    }
    return true;
};
// returns true if n is prime, else returns false.
bool MillerRabin(u64 n) {
    if (n < 2) return false;
    int r = 0; u64 d = n - 1;
    while ((d & 1) == 0) {
        d >>= 1; r++;
    }
    for (int a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37})
        {
        if (n == a) return true;
        if (check_composite(n, a, d, r)) return false;
    }
    return true;
}
bool probablyPrimeFermat(int n, int iter=5) {
    if (n < 4) return n == 2 || n == 3;
    for (int i = 0; i < iter; i++) {
        int a = 2 + rand() % (n - 3);
        if (binpower(a, n - 1, n) != 1)
            return false;
    }
    return true;
}
```

7.21 prime counting function

```
#define MAXN 500
#define MAXM 100010
#define MAXP 666666
#define MAX 10000010
#define ll long long int
#define chkbit(ar,i) (((ar[(i)>>6])&(1 << (((i) >> 1) & 31))))
#define setbit(ar, i) (((ar[(i) >> 6]) |= (1 <<(((i)>>1)&31))))
#defin isprime(x) (((x)&&((x)&1)&&(!chkbit(ar,(x))))||((x)==2))
namespace pcf{
    long long dp[MAXN][MAXM];
    unsigned int ar[(MAX>>6)+5] = {0};
    int len=0, primes[MAXP], counter[MAX];
    void Sieve(){ setbit(ar,0), setbit(ar,1);
        for (int i=3;(i*i)<MAX;i++,i++){
            if(!chkbit(ar, i)){ int k=i<<1;
                for(int j=(i*i);j<MAX;j+=k) setbit(ar,j);
            }
        }
        for(int i=1;i<MAX;i++){ counter[i]=counter[i - 1];
            if(isprime(i)) primes[len++]=i, counter[i]++;
        }
    }
    void init(){
        Sieve();
        for(int n=0;n<MAXN;n++){
            for(int m=0;m<MAXM;m++){
        if(!n) dp[n][m]=m;
        else dp[n][m]=dp[n-1][m]-dp[n-1][m/primes[n-1]];
            }
        }
    }
    ll phi(ll m,int n){
        if(n==0) return m;    if(primes[n-1]>=m) return 1;
        if(m<MAXM && n<MAXN) return dp[n][m];
        return phi(m,n-1) - phi(m/primes[n-1],n-1);
    }
    ll Lehmer(long long m){
        if(m<MAX) return counter[m];
        ll w,res=0; int i,a,s,c,x,y;
        s=sqrt(0.9+m), y=c=cbrt(0.9+m);
        a=counter[y], res=phi(m,a)+a-1;
        for(i=a;primes[i]<=s;i++)
            res=res-Lehmer(m/primes[i])+Lehmer(primes[i])-1;
        return res;
    }
}
```

8 String

8.1 Hashing

```
ll fmod(ll a, ll b, int md=mods[0]) {
  unsigned long long x = (long long) a * b;
  unsigned xh = (unsigned) (x >> 32), xl = (unsigned) x, d, m;
  asm(
    "div %4; \n\t"
    : "=a" (d), "=d" (m)
    : "d" (xh), "a" (xl), "r" (md)
  );
  return m;
}
void Build1(const string &str) {
  for(ll i = str.size() - 1; i >= 0; i--){
    hsh[i] = fmod(hsh[i + 1],bases[j],mods[j])+str[i];
    if (hsh[i] > mods[j]) hsh[i] -= mods[j];
  }
}
ll getSingleHash(ll i, ll j) {
  assert(i <= j);
  ll tmp1 = (hsh[i] - fmod(hsh[j+1],pwbase[0][j-i+1]));
  if(tmp1 < 0) tmp1 += mods[0]; return tmp1;
}
```

8.2 aho-corasick

```
const int K = 26;
struct Vertex {
    int next[K]; bool leaf = false; int p = -1; char pch;
    int link = -1; int go[K];
    Vertex(int p=-1, char ch='$') : p(p), pch(ch) {
        fill(begin(next), end(next), -1);
        fill(begin(go), end(go), -1);
    }
};
vector<Vertex> t(1);
void add_string(string const& s) { int v = 0;
    for (char ch : s) {
        int c = ch - 'a';
        if (t[v].next[c] == -1) {
            t[v].next[c] = t.size(); t.emplace_back(v, ch);
        }
        v = t[v].next[c];
    }
    t[v].leaf = true;
}
int go(int v, char ch);
int get_link(int v) {
    if (t[v].link == -1) {
        if (v == 0 || t[v].p == 0) t[v].link = 0;
        else    t[v].link = go(get_link(t[v].p), t[v].pch);
    }
    return t[v].link;
```

```
}
int go(int v, char ch) {
    int c = ch - 'a';
    if (t[v].go[c] == -1) {
        if (t[v].next[c] != -1) t[v].go[c] = t[v].next[c];
        else  t[v].go[c] = v == 0 ? 0 : go(get_link(v), ch);
    }
    return t[v].go[c]; }
```

8.3 manacher

```
char s[MAX]; vector<int> d1(n); vector<int> d2(n);
for (int i = 0, l = 0, r = -1; i < n; i++){
    int k = (i > r) ? 1 : min(d1[l + r - i], r - i + 1);
    while (0 <= i - k && i + k < n && s[i - k] ==s[i + k])
        { k++; }
    d1[i] = k--;
    if (i + k > r)    { l = i - k; r = i + k; }
}
for (int i = 0, l = 0, r = -1; i < n; i++){
    int k = (i > r) ? 0 : min(d2[l + r - i + 1],r - i + 1);
    while (0 <= i-k-1 && i+k < n && s[i-k-1] == s[i + k])
        { k++; }
    d2[i] = k--;
    if (i + k > r) { l = i - k - 1; r = i + k ; }
}
```

8.4 palindromic tree

```
struct node {
    int next[26]; int len; int sufflink; int num; };
int len; char s[MAXN]; node tree[MAXN];
int num;// node 1 - root with len -1, node 2 - root with len 0
int suff;// max suffix palindrome
bool addLetter(int pos) {
    int cur = suff, curlen = 0; int let = s[pos] - 'a';
    while (true) {
        curlen = tree[cur].len;
        if (pos-1-curlen >= 0 && s[pos - 1 - curlen] == s[pos])
            break;
        cur = tree[cur].sufflink;
    }
    if (tree[cur].next[let]) {
        suff = tree[cur].next[let]; return false;
    }
    num++; suff = num; tree[num].len = tree[cur].len + 2;
    tree[cur].next[let] = num;
    if (tree[num].len == 1) { tree[num].sufflink = 2;
        tree[num].num = 1;  return true;
    }
    while (true) {
        cur = tree[cur].sufflink; curlen = tree[cur].len;
        if (pos-1-curlen>=0 && s[pos- 1 - curlen] == s[pos]) {
            tree[num].sufflink = tree[cur].next[let]; break;
        }
    }
    tree[num].num=1+tree[tree[num].sufflink].num; return true;
}
void initTree() {
    num = 2; suff = 2; // memset tree must
    tree[1].len = -1; tree[1].sufflink = 1;
    tree[2].len = 0; tree[2].sufflink = 1;
}
int main() { gets(s); len = strlen(s); initTree();
    for (int i = 0; i < len; i++) { addLetter(i);
     ans += tree[suff].num; }
    cout << ans << endl;  return 0;
}
```

8.5 suffix array da

```
/* sa => ith smallest suffix of the string
rak => rak[i] indicates the position of suffix(i) in the suffix
array; height => height[i] indicates the LCP of i-1 and i th
suffix; LCP of suffix(i) & suffix(j) = { L = rak[i], R = rak[j]
 ,  min(height[L+1, R]);}*/
const int maxn = 5e5+5;
int wa[maxn],wb[maxn],wv[maxn],wc[maxn];
int r[maxn],sa[maxn],rak[maxn], height[maxn],dp[maxn][22],
                                  jump[maxn], SIGMA = 0 ;
int cmp(int *r,int a,int b,int l)
                        {return r[a]==r[b]&&r[a+l]==r[b+l];}
void da(int *r,int *sa,int n,int m){
    int i,j,p,*x=wa,*y=wb,*t;
    for( i=0;i<m;i++) wc[i]=0;
    for( i=0;i<n;i++) wc[x[i]=r[i]] ++;
    for( i=1;i<m;i++) wc[i] += wc[i-1];
    for( i= n-1;i>=0;i--)sa[--wc[x[i]]] = i;
    for( j= 1,p=1;p<n;j*=2,m=p){
        for(p=0,i=n-j;i<n;i++)y[p++] = i;
        for(i=0;i<n;i++)if(sa[i] >= j) y[p++] = sa[i] - j;
        for(i=0;i<n;i++)wv[i] = x[y[i]];
        for(i=0;i<m;i++) wc[i] = 0;
        for(i=0;i<n;i++) wc[wv[i]] ++;
        for(i=1;i<m;i++) wc[i] += wc[i-1];
        for(i=n-1;i>=0;i--) sa[--wc[wv[i]]] = y[i];
        for(t=x,x=y,y=t,p=1,x[sa[0]] = 0,i=1;i<n;i++)
                  x[sa[i]]= cmp(y,sa[i-1],sa[i],j) ? p-1:p++;
    }
}
void calheight(int *r,int *sa,int n){
    int i,j,k=0;
    for(i=1;i<=n;i++) rak[sa[i]] = i;
    for(i=0;i<n;height[rak[i++]] = k ) {
        for(k?k--:0, j=sa[rak[i]-1] ; r[i+k] == r[j+k] ; k ++);
    }
}
void initRMQ(int n){
    for(int i= 0;i<=n;i++) dp[i][0] = height[i];
    for(int j= 1; (1<<j) <= n; j ++ ){
        for(int i = 0; i + (1<<j) - 1 <= n ; i ++ ) {
            dp[i][j] = min(dp[i][j-1] ,dp[i+(1<<(j-1))][j-1]);
        }
    }
    for(int i = 1;i <= n;i ++ ) {
        int k = 0; while((1 << (k+1)) <= i) k++; jump[i] = k;
    }
}
int askRMQ(int L,int R){
    int k = jump[R-L+1];
    return min(dp[L][k], dp[R - (1<<k) + 1][k]);
}
int main(){
    scanf("%s",s); int n = strlen(s);
    for(int i = 0; i < n; i ++) {
        r[i] = s[i]-'a' + 1; SIGMA = max(SIGMA, r[i]);
    }
    r[n] = 0; da(r,sa,n+1,SIGMA + 1);
    calheight(r,sa,n);
    /* don't forget SIGMA + 1. It will ruin you.*/ }
```

8.6 suffix-automaton

```
class SuffixAutomaton{
bool complete; int last;
set<char> alphabet;
struct state{
    int len, link, endpos, first_pos,snas,height;
    long long substrings,sublen;
    bool is_clone;
    map<char, int> next;
    vector<int> inv_link;
    state(int leng=0,int li=0){
        len=leng;  link=li;
        first_pos=-1; substrings=0;
        sublen=0; // length of all substrings
        snas=0; // shortest_non_appearing_string
        endpos=1;  is_clone=false; height=0;
    }
};
vector<state> st;
void process(int node){
    map<char, int> ::iterator mit;
    st[node].substrings=1;
    st[node].snas=st.size();
    if((int) st[node].next.size()<(int) alphabet.size())
        st[node].snas=1;
for(mit=st[node].next.begin(); mit!=st[node].next.end();++mit){
    if(st[mit->second].substrings==0) process(mit->second);
st[node].height=max(st[node].height,1+st[mit->second].height);
    st[node].substrings=
              st[node].substrings+st[mit->second].substrings;
    st[node].sublen=st[node].sublen
    +st[mit->second].sublen+st[mit->second].substrings;
    st[node].snas=min(st[node].snas,
                             1+st[mit->second].snas);
    }
    if(st[node].link!=-1)
        st[st[node].link].inv_link.push_back(node);
}
void set_suffix_links(int node){
    int i;
    for(i=0; i<st[node].inv_link.size(); i++){
        set_suffix_links(st[node].inv_link[i]);
        st[node].endpos=
        st[node].endpos+st[st[node].inv_link[i]].endpos; }
}
voi output_all_occurrences(int v,int P_length,vector<int>&pos){
    if (!st[v].is_clone)
        pos.push_back(st[v].first_pos - P_length + 1);
    for (int u : st[v].inv_link)
        output_all_occurrences(u, P_length, pos);
}
void kth_smallest(int node,int k,vector<char> &str){
    if(k==0) return;
    map<char, int> ::iterator mit;
for(mit=st[node].next.begin(); mit!=st[node].next.end();++mit){
if(st[mit->second].substrings<k)k=k-st[mit->second].substrings;
        else{
            str.push_back(mit->first);
            kth_smallest(mit->second,k-1,str);
            return;
        }
    }
}
int find_occurrence_index(int node,int index,vector<char>&str){
    if(index==str.size()) return node;
    if(!st[node].next.count(str[index])) return -1;
    else return find_occurrence_index(st[node].next[str[index]]
                                          ,index+1,str);
}
void klen_smallest(int node,int k,vector<char> &str){
    if(k==0) return;
    map<char, int> ::iterator mit;
    for(mit=st[node].next.begin(); mit!=st[node].next.end();
++mit){ if(st[mit->second].height>=k-1){
            str.push_back(mit->first);
            klen_smallest(mit->second,k-1,str);
            return;
        }
    }
}
void minimum_non_existing_string(int node,vector<char> &str){
    map<char, int> ::iterator mit;
    set<char>::iterator sit;
    for(mit=st[node].next.begin(),sit=alphabet.begin();
                        sit!=alphabet.end(); ++sit,++mit){
        if(mit==st[node].next.end()||mit->first!=(*sit)){
            str.push_back(*sit);
            return;
        }
        else if(st[node].snas==1+st[mit->second].snas){
            str.push_back(*sit);
            minimum_non_existing_string(mit->second,str);
            return;
        }
    }
}
void find_substrings(int node,int index,vector<char> &str,
vector<pair<long long,long long> > &sub_info){
    sub_info.push_back(make_pair(st[node].substrings,
               st[node].sublen+st[node].substrings*index));
    if(index==str.size()) return;
    if(st[node].next.count(str[index])){ find_substrings(
        st[node].next[str[index]],index+1,str,sub_info);return;
    }
    else{
```

```
        sub_info.push_back(make_pair(0,0));
    }
}
void check(){
    if(!complete){
        process(0);
        set_suffix_links(0);
        int i;
        complete=true;
    }
}
public:
    SuffixAutomaton(set<char> &alpha){
        st.push_back(state(0,-1));
        last=0;
        complete=false;
        set<char>::iterator sit;
        for(sit=alpha.begin(); sit!=alpha.end(); sit++)
            alphabet.insert(*sit);
        st[0].endpos=0;
    }
    void sa_extend(char c){
        int cur = st.size();
        st.push_back(state(st[last].len + 1));
        st[cur].first_pos=st[cur].len-1;
        int p = last;
        while (p != -1 && !st[p].next.count(c)){
            st[p].next[c] = cur;
            p = st[p].link;
        }
        if (p == -1){
            st[cur].link = 0;
        }
        else{
            int q = st[p].next[c];
            if (st[p].len + 1 == st[q].len){
                st[cur].link = q;
            }
            else{
                int clone = st.size();
                st.push_back(state(st[p].len + 1,st[q].link));
                st[clone].next = st[q].next;
                st[clone].is_clone=true;
                st[clone].endpos=0;
                st[clone].first_pos=st[q].first_pos;
                while (p != -1 && st[p].next[c] == q){
                    st[p].next[c] = clone; p = st[p].link;
                }
                st[q].link = st[cur].link = clone;
            }
        }
        last = cur;
        complete=false;
    }
    ~SuffixAutomaton(){
        int i;
        for(i=0; i<st.size(); i++){
            st[i].next.clear();
            st[i].inv_link.clear();
        }
        st.clear();
        alphabet.clear();
    }
    void kth_smallest(int k,vector<char> &str){
        check();
        kth_smallest(0,k,str);
    }
    int FindFirstOccurrenceIndex(vector<char> &str){
        check();
        int ind=find_occurrence_index(0,0,str);
        if(ind==0) return -1;
        else if(ind==-1) return st.size();
        else return st[ind].first_pos+1-(int) str.size();
    }
void FindAllOccurrenceIndex(vector<char> &str,vector<int>&pos){
        check();
        int ind=find_occurrence_index(0,0,str);
        if(ind!=-1) output_all_occurrences(ind,str.size(),pos);
    }
    int Occurrences(vector<char> &str){
        check();
        int ind=find_occurrence_index(0,0,str);
        if(ind==0) return 1;
        else if(ind==-1) return 0;
        else return st[ind].endpos;
    }
    void klen_smallest(int k,vector<char> &str){
        check();
        if(st[0].height>=k) klen_smallest(0,k,str);
    }
    void minimum_non_existing_string(vector<char> &str){
        check();
        int ind=find_occurrence_index(0,0,str);
        if(ind!=-1) minimum_non_existing_string(ind,str);
    }
};
```

8.7 z-algorithm

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