Notebook - Maratona de Programação

Cabo HDMI, VGA, USB

Contents

1 Math

1.1 Matrix exponentiation

Complexity: O(n*n*n log(b)) to raise an nxn matrix to the power of b.
Computes powers of matrices efficiently.

```cpp
// TITLE: Matrix exponentiation
// COMPLEXITY: O(n*n*n log(b)) to raise an nxn matrix to the power of b.
// DESCRIPTION: Computes powers of matrices efficiently.

struct Matrix {
    vector<vi> m;
    int r, c;

    Matrix(vector<vi> mat) {
        m = mat;
        r = mat.size();
        c = mat[0].size();
    }

    Matrix(int row, int col, bool ident=false) {
        r = row; c = col;
        m = vector<vi>(r, vi(c, 0));
        if(ident) {
            for(int i = 0; i < min(r, c); i++) {
                m[i][i] = 1;
            }
        }
    }

    Matrix operator*(const Matrix &o) const {
        assert(c == o.r); // garantir que da pra multiplicar
        vector<vi> res(r, vi(o.c, 0));

        for(int i = 0; i < r; i++) {
            for(int k = 0; k < c; k++) {
                for(int j = 0; j < o.c; j++) {
                    res[i][j] = (res[i][j] + m[i][k]*o.m[k][j]) % MOD;
                }
            }
        }

        return Matrix(res);
    }
};

Matrix fpow(Matrix b, int e, int n) {
    if(e == 0) return Matrix(n, n, true); // identidade
    Matrix res = fexp(b, e/2, n);
    res = (res * res);
    if(e%2) res = (res * b);

    return res;
}
```

1.2 Fast Fourier Transform

Complexity: O(n log(n))
Multiply polynomials quickly

```cpp
// TITLE: Fast Fourier Transform
// COMPLEXITY: O(n log(n))
// DESCRIPTION: Multiply polynomials quickly

typedef double ld;
typedef long long ll;

struct num{
    ld x, y;
    num() { x = y = 0; }
    num(ld x, ld y) : x(x), y(y) {}
};

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

int base = 1;
vector<num> roots = {{0, 0}, {1, 0}};
vector<int> rev = {0, 1};
const ld PI = acos(-1);

void ensure_base(int nbase){
    if(nbase <= base)
        return;

    rev.resize(1 << nbase);
    for(int i = 0; i < (1 << nbase); i++)
        rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (nbase - 1));

    roots.resize(1 << nbase);

    while(base < nbase){
        ld angle = 2*PI / (1 << (base + 1));
        for(int i = 1 << (base - 1); i < (1 << base); i++){
            roots[i << 1] = roots[i];
            ld angle_i = angle * (2 * i + 1 - (1 << base));
            roots[(i << 1) + 1] = num(cos(angle_i), sin(angle_i));
        }
        base++;
    }
}

void fft(vector<num> &a, int n = -1){
    if(n == -1)
        n = a.size();

    assert((n & (n-1)) == 0);
    int zeros = __builtin_ctz(n);
    ensure_base(zeros);
    int shift = base - zeros;
    for(int i = 0; i < n; i++)
        if(i < (rev[i] >> shift))
            swap(a[i], a[rev[i] >> shift]);

    for(int k = 1; k < n; k <<= 1)
        for(int i = 0; i < n; i += 2 * k)
            for(int j = 0; j < k; j++){
                num z = a[i+j+k] * roots[j+k];
                a[i+j+k] = a[i+j] - z;
                a[i+j] = a[i+j] + z;
            }
}

vector<num> fa, fb;
vector<ll> multiply(vector<ll> &a, vector<ll> &b){
    int need = a.size() + b.size() - 1;
    int nbase = 0;
    while((1 << nbase) < need) nbase++;
```

```
    ensure_base(nbase);
    int sz = 1 << nbase;
    if(sz > (int) fa.size())
        fa.resize(sz);

    for(int i = 0; i < sz; i++){
        int x = (i < (int) a.size() ? a[i] : 0);
        int y = (i < (int) b.size() ? b[i] : 0);
        fa[i] = num(x, y);
    }
    fft(fa, sz);
    num r(0, -0.25 / sz);
    for(int i = 0; i <= (sz >> 1); i++){
        int j = (sz - i) & (sz - 1);
        num z = (fa[j] * fa[j] - conj(fa[i] * fa[i])) * r;
        if(i != j) {
            fa[j] = (fa[i] * fa[i] - conj(fa[j] * fa[j])) * r;
        }
        fa[i] = z;
    }
    fft(fa, sz);
    vector<ll> res(need);
    for(int i = 0; i < need; i++)
        res[i] = round(fa[i].x);

    return res;
}


vector<ll> multiply_mod(vector<ll> &a, vector<ll> &b, int m, int eq = 0){
    int need = a.size() + b.size() - 1;
    int nbase = 0;
    while((1 << nbase) < need) nbase++;
    ensure_base(nbase);
    int sz = 1 << nbase;
    if(sz > (int) fa.size())
        fa.resize(sz);

    for(int i=0;i<(int)a.size();i++){
        int x = (a[i] % m + m) % m;
        fa[i] = num(x & ((1 << 15) - 1), x >> 15);
    }
    fill(fa.begin() + a.size(), fa.begin() + sz, num {0, 0});
    fft(fa, sz);
    if(sz > (int) fb.size())
        fb.resize(sz);
    if(eq)
        copy(fa.begin(), fa.begin() + sz, fb.begin());
    else{
        for(int i = 0; i < (int) b.size(); i++){
            int x = (b[i] % m + m) % m;
            fb[i] = num(x & ((1 << 15) - 1), x >> 15);
        }
        fill(fb.begin() + b.size(), fb.begin() + sz, num {0, 0});
        fft(fb, sz);
    }
    ld ratio = 0.25 / sz;
    num r2(0, -1);
    num r3(ratio, 0);
    num r4(0, -ratio);
    num r5(0, 1);
    for(int i=0;i<=(sz >> 1);i++) {
        int j = (sz - i) & (sz - 1);
        num a1 = (fa[i] + conj(fa[j]));
        num a2 = (fa[i] - conj(fa[j])) * r2;
        num b1 = (fb[i] + conj(fb[j])) * r3;
        num b2 = (fb[i] - conj(fb[j])) * r4;
        if(i != j){
            num c1 = (fa[j] + conj(fa[i]));
            num c2 = (fa[j] - conj(fa[i])) * r2;
            num d1 = (fb[j] + conj(fb[i])) * r3;
            num d2 = (fb[j] - conj(fb[i])) * r4;
            fa[i] = c1 * d1 + c2 * d2 * r5;
            fb[i] = c1 * d2 + c2 * d1;
        }
        fa[j] = a1 * b1 + a2 * b2 * r5;
        fb[j] = a1 * b2 + a2 * b1;
    }
    fft(fa, sz);
    fft(fb, sz);
    vector<ll> res(need);
    for(int i=0;i<need;i++){
        ll aa = round(fa[i].x);
        ll bb = round(fb[i].x);
        ll cc = round(fa[i].y);
        res[i] = (aa + ((bb % m) << 15) + ((cc % m) << 30)) % m;
    }
    return res;
}
```

2 Graph

2.1 Dfs tree

Complexity: O(E + V)

```
// TITLE: Dfs tree
// COMPLEXITY: O(E + V)
// DESCRIPION: Create dfs tree from graph

int desce[mxN], sobe[mxN];
int backedges[mxN], vis[mxN];
int pai[mxN], h[mxN];

void dfs(int a, int p) {
    if(vis[a]) return;
    pai[a] = p;
    h[a] = h[p]+1;
    vis[a] = 1;

    for(auto b : g[a]) {
        if (p == b) continue;
        if (vis[b]) continue;
        dfs(b, a);
        backedges[a] += backedges[b];
    }
    for(auto b : g[a]) {
        if(h[b] > h[a]+1)
            desce[a]++;
        else if(h[b] < h[a]-1)
            sobe[a]++;
    }
    backedges[a] += sobe[a] - desce[a];
}
```

2.2 Bellman Ford

Complexity: O(n * m) | n = |nodes|, m = |edges|
Finds shortest paths from a starting node to all nodes of the graph. Detects negative cycles, if they exist.

```
// TITLE: Bellman Ford
// COMPLEXITY: O(n * m) | n = |nodes|, m = |edges|
```

~~~
// DESCRIPTION: Finds shortest paths from a starting node to all nodes of the graph. Detects negative cycles, if they exist.

// a and b vertices, c cost
// [{a, b, c}, {a, b, c}]
vector<tuple<int, int, int>> edges;
int N;

void bellman_ford(int x){
    for (int i = 0; i < N; i++){
        dist[i] = oo;
    }
    dist[x] = 0;

    for (int i = 0; i < N - 1; i++){
        for (auto [a, b, c]: edges){
            if (dist[a] == oo) continue;
            dist[b]= min(dist[b], dist[a] + w);
        }
    }
}
// return true if has cycle
bool check_negative_cycle(int x){
    for (int i = 0; i < N; i++){
        dist[i] = oo;
    }
    dist[x] = 0;

    for (int i = 0; i < N - 1; i++){
        for (auto [a, b, c]: edges){
            if (dist[a] == oo) continue;
            dist[b]= min(dist[b], dist[a] + w);
        }
    }

    for (auto [a, b, c]: edges){
        if (dist[a] == oo) continue;
        if (dist[a] + w < dist[b]){
            return true;
        };
    }
    return false;
}
```
~~~

2.3 Floyd Warshall

Complexity: O(V*V*V)
Finds shortest distances between all pairs of vertices

```
// TITLE: Floyd Warshall
// COMPLEXITY: O(V*V*V)
// DESCRIPTION: Finds shortest distances between all pairs of vertices

for(int k=0;k<n;k++) {

        for(int i=0;i<n;i++) {
            for(int j=0;j<n;j++) {
                graph[i][j]=min(graph[i][j],
                graph[i][k] + graph[k][j]);
            }
        }
    }
```

2.4 2SAT

Complexity: O(n+m), n = number of variables, m = number of conjunctions (ands).
Finds an assignment that makes a certain boolean formula true, or determines that such an assignment does not exist.

```
// TITLE: 2SAT
// COMPLEXITY: O(n+m), n = number of variables, m = number of conjunctions (ands).
// DESCRIPTION: Finds an assignment that makes a certain boolean formula true, or determines that such an assignment does not exist.

struct twosat {
    vi vis, degin;
    stack<int> tout;
    vector<vi> g, gi, con, sccg;
    vi repr, conv;
    int gsize;
    void dfs1(int a) {
        if (vis[a]) return;
        vis[a]=true;

        for(auto& b : g[a]) {
            dfs1(b);
        }

        tout.push(a);
    }

    void dfs2(int a, int orig) {
        if (vis[a]) return;
        vis[a]=true;

        repr[a]=orig;
        sccg[orig].pb(a);
        for(auto& b : gi[a]) {
            if (vis[b]) {
                if (repr[b] != orig) {
                    con[repr[b]].pb(orig);
                    degin[orig]++;
                }
                continue;
            }
            dfs2(b, orig);
        }

    }
    // if s1 = 1 and s2 = 1 this adds a \/ b to the graph
    void addedge(int a, int s1,
                 int b, int s2) {
        g[2*a+(!s1)].pb(2*b+s2);
        gi[2*b+s2].pb(2*a+(!s1));

        g[2*b+(!s2)].pb(2*a+s1);
        gi[2*a+s1].pb(2*b+(!s2));
    }


    twosat(int nvars) {
        gsize=2*nvars;
        g.assign(gsize, vi());
        gi.assign(gsize, vi());
        con.assign(gsize, vi());
        sccg.assign(gsize, vi());
        repr.assign(gsize, -1);
        vis.assign(gsize, 0);
        degin.assign(gsize, 0);
    }

    // returns empty vector if the formula is not satisfiable.
    vi run() {
        vi vals(gsize/2, -1);
        rep(i,0,gsize) dfs1(i);
```

```
        vis.assign(gsize,0);
        while(!tout.empty()) {
            int cur = tout.top();tout.pop();
            if (vis[cur]) continue;
            dfs2(cur,cur);
            conv.pb(cur);
        }

        rep(i, 0, gsize/2) {
            if (repr[2*i] == repr[2*i+1]) {
                return {};
            }
        }

        queue<int> q;
        for(auto& v : conv) {
            if (degin[v] == 0) q.push(v);
        }

        while(!q.empty()) {
            int cur=q.front(); q.pop();
            for(auto guy : sccg[cur]) {
                int s = guy%2;
                int idx = guy/2;
                if (vals[idx] != -1) continue;
                if (s) {
                    vals[idx] = false;
                } else {
                    vals[idx]=true;
                }
            }
            for (auto& b : con[cur]) {
                if(--degin[b] == 0) q.push(b);
            }
        }

        return vals;
    }
};
```

2.5 Dominator tree

Complexity: O(E + V)

```
// TITLE: Dominator tree
// COMPLEXITY: O(E + V)
// DESCRIPION: Builds dominator tree

vector<int> g[mxN];
vector<int> S, gt[mxN], T[mxN];
int dsu[mxN], label[mxN];
int sdom[mxN], idom[mxN], id[mxN];
int dfs_time = 0;

vector<int> bucket[mxN];
vector<int> down[mxN];

void prep(int a)
{
    S.pb(a);
    id[a] = ++dfs_time;
    label[a] = sdom[a] = dsu[a] = a;

    for (auto b: g[a]) {
        if (!id[b]) {
            prep(b);
            down[a].pb(b);
        }
        gt[b].pb(a);
    }
}

int fnd(int a, int flag = 0)
{
    if (a == dsu[a]) return a;
    int p = fnd(dsu[a], 1);
    int b = label[ dsu[a] ];
    if (id [ sdom[b] ] < id[ sdom[ label[a] ] ]) {
        label[a] = b;
    }
    dsu[a] = p;
    return (flag ? p: label[a]);
}

void build_dominator_tree(int root)
{
    prep(root);
    reverse(all(S));

    int w;
    for (int a: S) {
        for (int b: gt[a]) {
            w = fnd(b);
            if (id[ sdom[w] ] < id[ sdom[a] ]) {
                sdom[a] = sdom[w];
            }
        }
        gt[a].clear();
        if (a != root) {
            bucket[ sdom[a] ].pb(a);
        }
        for (int b: bucket[a]) {
            w = fnd(b);
            if (sdom[w] == sdom[b]) {
                idom[b] = sdom[b];
            }
            else {
                idom[b] = w;
            }
        }
        bucket[a].clear();
        for (int b: down[a]) {
            dsu[b] = a;
        }
        down[a].clear();
    }
    reverse(all(S));
    for (int a: S) {
        if (a != root) {
            if (idom[a] != sdom[a]) {
                idom[a] = idom[ idom[a] ];
            }
            T[ idom[a] ].pb(a);
        }
    }
    S.clear();
}
```

2.6 Kth Ancestor

Complexity: O(n * log(n))
Preprocess, then find in log n

```
// TITLE: Kth Ancestor
// COMPLEXITY: O(n * log(n))
// DESCRIPTION: Preprocess, then find in log n

const int LOG_N = 30;
int get_kth_ancestor(vector<vector<int>> & up, int v,
     int k)
{
    for (int j = 0; j < LOG_N; j++) {
        if (k & ((int)1 << j)) {
```

```
            v = up[v][j];
        }
    }
    return v;
}

void solve()
{
    vector<vector<int>> up(n, vector<int>(LOG_N));

    for (int i = 0; i < n; i++) {
        up[i][0] = parents[i];
        for (int j = 1; j < LOG_N; j++) {
            up[i][j] = up[up[i][j-1]][j-1];
        }
    }
    cout << get_kth_ancestor(up, x, k) << endl;

}
```

2.7 Topological Sort

Complexity: O(N + M), N: Vertices, M: Arestas
Retorna no do grapho em ordem topologica, se a quantidade de nos retornada nao for igual a quantidade de nos e impossivel

```
// TITLE: Topological Sort
// COMPLEXITY: O(N + M), N: Vertices, M: Arestas
// DESCRIPTION: Retorna no do grapho em ordem
    topologica, se a quantidade de nos retornada nao
    for igual a quantidade de nos e impossivel

typedef vector<vector<int>> Adj_List;
typedef vector<int> Indegree_List; // How many nodes
    depend on him
typedef vector<int> Order_List;    // The order in
    which the nodes appears

Order_List kahn(Adj_List adj, Indegree_List indegree)
{
    queue<int> q;
    // priority_queue<int> q; // If you want in
    lexicografic order
    for (int i = 0; i < indegree.size(); i++) {
        if (indegree[i] == 0)
            q.push(i);
    }
    vector<int> order;

    while (not q.empty()) {
        auto a = q.front();
        q.pop();

        order.push_back(a);
        for (auto b: adj[a]) {
            indegree[b]--;
            if (indegree[b] == 0)
                q.push(b);
        }
    }
    return order;
}

int32_t main()
{

    Order_List = kahn(adj, indegree);
    if (Order_List.size() != N) {
        cout << "IMPOSSIBLE" << endl;
    }
    return 0;
}
```

2.8 Dkistra

Complexity: O(E + V.log(v))

```
// TITLE: Dkistra
// COMPLEXITY: O(E + V.log(v))
// DESCRIPTION: Finds to shortest path from start

int dist[mxN];
bool vis[mxN];
vector<pair<int, int>> g[mxN];

void dikstra(int start)
{
    fill(dist, dist + mxN, oo);
    fill(vis, vis + mxN, 0);
    priority_queue<pair<int, int>> q;
    dist[start] = 0;
    q.push({0, start});

    while(!q.empty()) {
        auto [d, a] = q.top();
        q.pop();
        if (vis[a]) continue;
        vis[a] = true;
        for (auto [b, w]: g[a]) {
            if (dist[a] + w < dist[b]) {
                dist[b] = dist[a] + w;
                q.push({-dist[b], b});
            }
        }
    }
}
```

2.9 Dinic Min cost

Complexity: O(V*V*E), Bipartite is O(sqrt(V) E)
Gives you the max_flow with the min cost

```
// TITLE: Dinic Min cost
// COMPLEXITY: O(V*V*E), Bipartite is O(sqrt(V) E)
// DESCRIPTION: Gives you the max_flow with the min
    cost

// Edge structure
struct Edge
{
    int from, to;
    int flow, capacity;
    int cost;

    Edge(int from_, int to_, int flow_, int capacity_
    , int cost_)
        : from(from_), to(to_), flow(flow_), capacity
    (capacity_), cost(cost_)
    {}
};

struct Dinic
{
    vector<vector<int>> graph;
    vector<Edge> edges;
    vector<int> dist;
    vector<bool> inqueue;
    int size;
    int cost = 0;

```

```cpp
    Dinic(int n)
    {
        graph.resize(n);
        dist.resize(n);
        inqueue.resize(n);
        size = n;
        edges.clear();
    }

    void add_edge(int from, int to, int capacity, int cost)
    {
        edges.emplace_back(from, to, 0, capacity, cost);
        graph[from].push_back(edges.size() - 1);

        edges.emplace_back(to, from, 0, 0, -cost);
        graph[to].push_back(edges.size() - 1);
    }

    int get_max_flow(int source, int sink)
    {
        int max_flow = 0;
        vector<int> next(size);
        while(spfa(source, sink)) {
            next.assign(size, 0);
            for (int f = dfs(source, sink, next, oo); f != 0; f = dfs(source, sink, next, oo)) {
                max_flow += f;
            }
        }
        return max_flow;
    }

    bool spfa(int source, int sink)
    {
        dist.assign(size, oo);
        inqueue.assign(size, false);
        queue<int> q;
        q.push(source);
        dist[source] = 0;
        inqueue[source] = true;

        while(!q.empty()) {
            int a = q.front();
            q.pop();
            inqueue[a] = false;

            for (int & b: graph[a]) {
                auto edge = edges[b];
                int cap = edge.capacity - edge.flow;
                if (cap > 0 && dist[edge.to] > dist[edge.from] + edge.cost) {
                    dist[edge.to] = dist[edge.from] + edge.cost;
                    if (not inqueue[edge.to]) {
                        q.push(edge.to);
                        inqueue[edge.to] = true;
                    }
                }
            }
        }
        return dist[sink] != oo;
    }

    int dfs(int curr, int sink, vector<int> & next, int flow)
    {
        if (curr == sink) return flow;
        int num_edges = graph[curr].size();

        for (; next[curr] < num_edges; next[curr]++) {
            int b = graph[curr][next[curr]];
            auto & edge = edges[b];
            auto & rev_edge = edges[b^1];

            int cap = edge.capacity - edge.flow;
            if (cap > 0 && (dist[edge.from] + edge.cost == dist[edge.to])) {
                int bottle_neck = dfs(edge.to, sink, next, min(flow, cap));
                if (bottle_neck > 0) {
                    edge.flow += bottle_neck;
                    rev_edge.flow -= bottle_neck;
                    cost += edge.cost * bottle_neck;
                    return bottle_neck;
                }
            }
        }
        return 0;
    }

    vector<pair<int, int>> mincut(int source, int sink)
    {
        vector<pair<int, int>> cut;
        spfa(source, sink);
        for (auto & e: edges) {
            if (e.flow == e.capacity && dist[e.from] != oo && level[e.to] == oo && e.capacity > 0) {
                cut.emplace_back(e.from, e.to);
            }
        }
        return cut;
    }
};

// Example on how to use
void solve()
{

    int N = 10;

    int source = 8;
    int sink = 9;

    Dinic flow(N);
    flow.add_edge(8, 0, 4, 0);
    flow.add_edge(8, 1, 3, 0);
    flow.add_edge(8, 2, 2, 0);
    flow.add_edge(8, 3, 1, 0);

    flow.add_edge(0, 6, oo, 3);
    flow.add_edge(0, 7, oo, 2);
    flow.add_edge(0, 5, oo, 0);

    flow.add_edge(1, 4, oo, 0);

    flow.add_edge(4, 9, oo, 0);
    flow.add_edge(5, 9, oo, 0);
    flow.add_edge(6, 9, oo, 0);
    flow.add_edge(7, 9, oo, 0);

    int ans = flow.get_max_flow(source, sink);
    debug(ans);
    debug(flow.cost);
}

int32_t main()
{
    solve();
}
```

2.10 Kosaraju

Complexity: O(V+E)
Find the strongly connected components of a graph

```
// TITLE: Kosaraju
// COMPLEXITY: O(V+E)
// DESCRIPTION: Find the strongly connected components of a graph

int n,m;
vector<vi> g, gi, scc;
vi vis, order, p;

void dfs1(int a) {
    if(vis[a]) return;
    vis[a]=true;
    for(auto& b:g[a]) {
        dfs1(b);
    }
    order.pb(a);
}

void dfs2(int a, int orig) {
    if (vis[a]) return;
    vis[a]=true;
    p[a]=orig;

    for(auto& b:gi[a]) {
        if (vis[b] && p[b] != orig)
            scc[p[b]].pb(orig);
        dfs2(b,orig);
    }
}

void solve() {
    cin>>n>>m;

    g.assign(n, vi());
    gi.assign(n, vi());
    scc.assign(n, vi());
    vis.assign(n, 0);
    p.assign(n, 0);
    rep(i, 0, m) {
        int a,b;cin>>a>>b;a--;b--;
        g[a].pb(b);
        gi[b].pb(a);
    }

    rep(i,0,n)dfs1(i);
    vis.assign(n,0);
    for(int i=n-1; i>=0;i--) dfs2(order[i],order[i]);

    vis.assign(n,0);
}
```

2.11 Dinic

Complexity: O(V*V*E), Bipartite is O(sqrt(V) E)
Dinic

```
// TITLE: Dinic
// COMPLEXITY: O(V*V*E), Bipartite is O(sqrt(V) E)
// DESCRIPTION: Dinic

const int oo = 0x3f3f3f3f3f3f3f3f;
// Edge structure
struct Edge
{
    int from, to;
    int flow, capacity;

    Edge(int from_, int to_, int flow_, int capacity_)
        : from(from_), to(to_), flow(flow_), capacity(capacity_)
    {}
};

struct Dinic
{
    vector<vector<int>> graph;
    vector<Edge> edges;
    vector<int> level;
    int size;

    Dinic(int n)
    {
        graph.resize(n);
        level.resize(n);
        size = n;
        edges.clear();
    }

    void add_edge(int from, int to, int capacity)
    {
        edges.emplace_back(from, to, 0, capacity);
        graph[from].push_back(edges.size() - 1);

        edges.emplace_back(to, from, 0, 0);
        graph[to].push_back(edges.size() - 1);
    }

    int get_max_flow(int source, int sink)
    {
        int max_flow = 0;
        vector<int> next(size);
        while(bfs(source, sink)) {
            next.assign(size, 0);
            for (int f = dfs(source, sink, next, oo); f != 0; f = dfs(source, sink, next, oo)) {
                max_flow += f;
            }
        }
        return max_flow;
    }

    bool bfs(int source, int sink)
    {
        level.assign(size, -1);
        queue<int> q;
        q.push(source);
        level[source] = 0;

        while(!q.empty()) {
            int a = q.front();
            q.pop();

            for (int & b: graph[a]) {
                auto edge = edges[b];
                int cap = edge.capacity - edge.flow;
                if (cap > 0 && level[edge.to] == -1) {
                    level[edge.to] = level[a] + 1;
                    q.push(edge.to);
                }
            }
        }
        return level[sink] != -1;
    }

    int dfs(int curr, int sink, vector<int> & next, int flow)
    {
        if (curr == sink) return flow;
```

```
        int num_edges = graph[curr].size();

        for (; next[curr] < num_edges; next[curr]++) {
            int b = graph[curr][next[curr]];
            auto & edge = edges[b];
            auto & rev_edge = edges[b^1];

            int cap = edge.capacity - edge.flow;
            if (cap > 0 && (level[curr] + 1 == level[edge.to])) {
                int bottle_neck = dfs(edge.to, sink, next, min(flow, cap));
                if (bottle_neck > 0) {
                    edge.flow += bottle_neck;
                    rev_edge.flow -= bottle_neck;
                    return bottle_neck;
                }
            }
        }
        return 0;
    }

    vector<pair<int, int>> mincut(int source, int sink)
    {
        vector<pair<int, int>> cut;
        bfs(source, sink);
        for (auto & e: edges) {
            if (e.flow == e.capacity && level[e.from] != -1 && level[e.to] == -1 && e.capacity > 0) {
                cut.emplace_back(e.from, e.to);
            }
        }
        return cut;
    }
};

// Example on how to use
void solve()
{
    int n, m;
    cin >> n >> m;
    int N = n + m + 2;

    int source = N - 2;
    int sink = N - 1;

    Dinic flow(N);

    for (int i = 0; i < n; i++) {
        int q; cin >> q;
        while(q--) {
            int b; cin >> b;
            flow.add_edge(i, n + b - 1, 1);
        }
    }
    for (int i =0; i < n; i++) {
        flow.add_edge(source, i, 1);
    }
    for (int i =0; i < m; i++) {
        flow.add_edge(i + n, sink, 1);
    }

    cout << m - flow.get_max_flow(source, sink) << endl;

    // Getting participant edges
    for (auto & edge: flow.edges) {
        if (edge.capacity == 0) continue; // This means is a reverse edge
        if (edge.from == source || edge.to == source) continue;
        if (edge.from == sink   || edge.to == sink) continue;
        if (edge.flow == 0) continue; // Is not participant

        cout << edge.from + 1 << " " << edge.to -n + 1 << endl;
    }
}
```

3 Segtree

3.1 Standard SegTree

Complexity: O(log(n)) query and update
Sum segment tree with point update.

```
// TITLE: Standard SegTree
// COMPLEXITY: O(log(n)) query and update
// DESCRIPTION: Sum segment tree with point update.

using type = int;

type iden = 0;
vector<type> seg;
int segsize;

type func(type a, type b)
{
    return a + b;
}

// query do intervalo [l, r)
type query(int l, int r, int no = 0, int lx = 0, int rx = segsize)
{
    // l lx rx r
    if (r <= lx or rx <= l)
        return iden;
    if (l <= lx and rx <= r)
        return seg[no];

    int mid = lx + (rx - lx) / 2;
    return func(query(l, r, 2 * no + 1, lx, mid),
                query(l, r, 2 * no + 2, mid, rx));
}

void update(int dest, type val, int no = 0, int lx = 0, int rx = segsize)
{
    if (dest < lx or dest >= rx)
        return;
    if (rx - lx == 1)
    {
        seg[no] = val;
        return;
    }

    int mid = lx + (rx - lx) / 2;
    update(dest, val, 2 * no + 1, lx, mid);
    update(dest, val, 2 * no + 2, mid, rx);
    seg[no] = func(seg[2 * no + 1], seg[2 * no + 2]);
}

signed main()
{
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);
    int n;
```

```
    cin >> n;
    segsize = n;
    if (__builtin_popcount(n) != 1)
    {
        segsize = 1 + (int)log2(segsize);
        segsize = 1 << segsize;
    }
    seg.assign(2 * segsize - 1, iden);

    rep(i, 0, n)
    {
        int x;
        cin >> x;
        update(i, x);
    }
}
```

3.2 Persistent sum segment tree

Complexity: O(log(n)) query and update, O(k log(n)) memory, n = number of elements, k = number of operations
Sum segment tree which preserves its history.

```
// TITLE: Persistent sum segment tree
// COMPLEXITY: O(log(n)) query and update, O(k log(n)
    ) memory, n = number of elements, k = number of
    operations
// DESCRIPTION: Sum segment tree which preserves its
    history.

int segsize;

struct node {
    int val;
    int lx, rx;
    node *l=0, *r=0;

    node() {}
    node(int val, int lx, int rx, node *l, node *r) :
    val(val), lx(lx),rx(rx),l(l),r(r) {}
};

node* build(vi& arr, int lx=0, int rx=segsize) {
    if (rx - lx == 1) {
        if (lx < (int)arr.size()) {
            return new node(arr[lx], lx, rx, 0, 0);
        }

        return new node(0,lx,rx,0,0);
    }

    int mid = (lx+rx)/2;
    auto nol = build(arr, lx, mid);
    auto nor = build(arr, mid, rx);
    return new node(nol->val + nor->val, lx, rx, nol,
     nor);
}

node* update(int idx, int val, node *no) {
    if (idx < no->lx or idx >= no->rx) return no;
    if (no->rx - no->lx == 1) {
        return new node(val+no->val, no->lx, no->rx,
    no->l, no->r);
    }

    auto nol = update(idx, val, no->l);
    auto nor = update(idx, val, no->r);
    return new node(nol->val + nor->val, no->lx, no->
    rx, nol, nor);
}

int query(int l, int r, node *no) {
    if (r <= no->lx or no->rx <= l) return 0;
    if (l <= no->lx and no->rx <= r) return no->val;

    return query(l,r,no->l) + query(l,r,no->r);
}
```

3.3 Set and update lazy seg

Complexity: O(log(n)) query and update
Sum segtree with set and update

```
// TITLE: Set and update lazy seg
// COMPLEXITY: O(log(n)) query and update
// DESCRIPTION: Sum segtree with set and update

vector<int> lazy, opvec;
vector<int> seg;

constexpr int SET = 30;
constexpr int ADD = 31;

int segsize;

void propagate(int no, int lx, int rx) {
    if (lazy[no] == -1) return;

    if (rx-lx == 1) {
        if(opvec[no] == SET) seg[no] = lazy[no];
        else seg[no] += lazy[no];

        lazy[no]=-1;
        opvec[no]=-1;
        return;
    }

    if(opvec[no] == SET) {
        seg[no] = (rx-lx) * lazy[no];
        lazy[2*no+1] = lazy[no];
        lazy[2*no+2] = lazy[no];

        opvec[2*no+1] = SET;
        opvec[2*no+2] = SET;

        lazy[no] = -1;
        opvec[no]=-1;
        return;
    }

    seg[no] += (rx-lx) * lazy[no];
    if (lazy[2*no+1] == -1) {
        lazy[2*no+1] = 0;
        opvec[2*no+1] = ADD;
    }
    if (lazy[2*no+2] == -1) {
        lazy[2*no+2] = 0;
        opvec[2*no+2] = ADD;
    }
    lazy[2*no+1] += lazy[no];
    lazy[2*no+2] += lazy[no];

    lazy[no] = -1;
    opvec[no]=-1;
}

void update(int l, int r, int val, int op, int no=0,
    int lx=0, int rx=segsize) {
    propagate(no, lx, rx);
    if (r <= lx or l >= rx) return;
    if (lx >= l and rx <= r) {
        lazy[no] = val;
        opvec[no] = op;
```

```
        propagate(no, lx, rx);
        return;
    }

    int mid = (rx+lx)/2;
    update(l, r, val, op, 2*no+1, lx, mid);
    update(l, r, val, op, 2*no+2, mid, rx);
    seg[no] = seg[2*no+1]+seg[2*no+2];
}

int query(int l, int r, int no=0, int lx=0, int rx=segsize) {
    propagate(no, lx, rx);
    if (r <= lx or l >= rx) return 0;
    if (lx >= l and rx <= r) return seg[no];

    int mid = (rx+lx)/2;
    return
        query(l,r,2*no+1,lx,mid) +
        query(l,r,2*no+2, mid, rx);
}
```

3.4 Binary Indexed Tree

Complexity: O(log(n)) query and update
Range sum queries with point update. One-indexed.

```
// TITLE: Binary Indexed Tree
// COMPLEXITY: O(log(n)) query and update
// DESCRIPTION: Range sum queries with point update. One-indexed.

struct BIT{
    #define lowbit(x) ( x & -x )
    int n;
    vi b;

    BIT( int n ) : n(n) , b(n+1 , 0){};
    BIT( vi &c ){
        n = c.size() , b = c;
        for( int i = 1 , fa = i + lowbit(i) ; i <= n ;
        i ++ , fa = i + lowbit(i) )
            if( fa <= n ) b[fa] += b[i];
    }
    void add( int i , int y ){
        for( ; i <= n ; i += lowbit(i) ) b[i] += y;
    }

    int calc( int i ){
        int sum = 0;
        for( ; i ; i -= lowbit(i) ) sum += b[i];
        return sum;
    }
};
```

3.5 Lazy SegTree

Complexity: O(log(n)) query and update
Sum segment tree with range sum update.

```
// TITLE: Lazy SegTree
// COMPLEXITY: O(log(n)) query and update
// DESCRIPTION: Sum segment tree with range sum update.
vector<int> seg, lazy;
int segsize;

// change 0s to -1s if update is
// set instead of add. also,
// remove the +=s
void prop(int no, int lx, int rx) {
    if (lazy[no] == 0) return;

    seg[no]+=(rx-lx)*lazy[no];
    if(rx-lx>1) {
        lazy[2*no+1] += lazy[no];
        lazy[2*no+2] += lazy[no];
    }

    lazy[no]=0;
}

void update(int l, int r, int val,int no=0, int lx=0, int rx=segsize) {
    // l r lx rx
    prop(no, lx, rx);
    if (r <= lx or rx <= l) return;
    if (l <= lx and rx <= r) {
        lazy[no]=val;
        prop(no,lx,rx);
        return;
    }

    int mid=lx+(rx-lx)/2;
    update(l,r,val,2*no+1,lx,mid);
    update(l,r,val,2*no+2,mid,rx);
    seg[no] =seg[2*no+1]+seg[2*no+2];
}

int query(int l,int r,int no=0,int lx=0, int rx=segsize) {
    prop(no,lx,rx);
    if (r <= lx or rx <= l) return 0;
    if (l <= lx and rx <= r) return seg[no];

    int mid=lx+(rx-lx)/2;
    return query(l,r,2*no+1, lx, mid)+
           query(l,r,2*no+2,mid,rx);
}

signed main() {
    ios_base::sync_with_stdio(0);cin.tie(0);cout.tie(0);

    int n;cin>>n;
    segsize=n;
    if(__builtin_popcount(n) != 1) {
        segsize=1+(int)log2(segsize);
        segsize= 1<<segsize;
    }

    seg.assign(2*segsize-1, 0);
    // use -1 instead of 0 if
    // update is set instead of add
    lazy.assign(2*segsize-1, 0);
}
```

4 Set

4.1 Set

Complexity: Insertion Log(n)
Keeps elements sorted, remove duplicates, upper_bound, lower_bound, find, count

```
// TITLE: Set
// COMPLEXITY: Insertion Log(n)
// Description: Keeps elements sorted, remove duplicates, upper_bound, lower_bound, find, count
```

```cpp

int main() {
  set<int> set1;

  set1.insert(1);       // O(log(n))
  set1.erase(1);        // O(log(n))

  set1.upper_bound(1);  // O(log(n))
  set1.lower_bound(1);  // O(log(n))
  set1.find(1);         // O(log(n))
  set1.count(1);        // O(log(n))

  set1.size();          // O(1)
  set1.empty();         // O(1)

  set1.clear()          // O(1)
  return 0;
}
```

4.2 Multiset

Complexity: O(log(n))
Same as set but you can have multiple elements with same values

```cpp
// TITLE: Multiset
// COMPLEXITY: O(log(n))
// DESCRIPTION: Same as set but you can have multiple elements with same values

int main() {
  multiset<int> set1;
}
```

4.3 Ordered Set

Complexity: log n
Worst set with adtional operations

```cpp
// TITLE: Ordered Set
// COMPLEXITY: log n
// DESCRIPTION: Worst set with adtional operations


#include <bits/extc++.h>
using namespace __gnu_pbds; // or pb_ds;
template<typename T, typename B = null_type>
using ordered_set = tree<T, B, less<T>, rb_tree_tag, tree_order_statistics_node_update>;

int32_t main() {
    ordered_set<int> oset;

    oset.insert(5);
    oset.insert(1);
    oset.insert(2);
    // o_set = {1, 2, 5}
    5 == *(oset.find_by_order(2)); // Like an array index
    2 == oset.order_of_key(4); // How many elements are strictly less than 4
}
```

5 Misc

5.1 Template

Complexity: O(1)
Standard template for competitions

```cpp
// TITLE: Template
// COMPLEXITY: O(1)
// DESCRIPTION: Standard template for competitions

#include <bits/stdc++.h>

#define int long long
#define endl '\n'
#define pb push_back
#define eb emplace_back
#define all(x) (x).begin(), (x).end()
#define rep(i, a, b) for(int i=(int)(a);i < (int)(b);i++)
#define debug(var) cout << #var << ": " << var << endl
#define pii pair<int, int>
#define vi vector<int>

int MAX = 2e5;
int MOD=1e9+7;
int oo=0x3f3f3f3f3f3f3f3f;

using namespace std;

void solve()
{

}

signed main()
{
    ios_base::sync_with_stdio(0);cin.tie(0);cout.tie(0);
    int t=1;
    // cin>>t;
    while(t--) solve();
}
```

6 String

6.1 String hash

Complexity: O(n) preprocessing, O(1) query
Computes the hash of arbitrary substrings of a given string s.

```cpp
// TITLE: String hash
// COMPLEXITY: O(n) preprocessing, O(1) query
// DESCRIPTION: Computes the hash of arbitrary substrings of a given string s.
int m1, m2;
int n; string s;

struct Hash {
    const int P = 31;
    int n; string s;
    vector<int> h, hi, p, p2, h2, hi2;
    Hash() {}
    Hash(string s):
    s(s), n(s.size()), h(n), hi(n), p(n), h2(n), hi2(n), p2(n) {
        for (int i=0;i<n;i++) p[i] = (i ? P*p[i-1]:1) % m1;
```

```
        for (int i=0;i<n;i++) p2[i] = (i ? P*p2[i-1]:1) % m2;

        for (int i=0;i<n;i++)
            h[i] = (s[i] + (i ? h[i-1]:0) * P) % m1;
        for (int i=0;i<n;i++)
            h2[i] = (s[i] + (i ? h2[i-1]:0) * P) % m2;

        for (int i=n-1;i>=0;i--)
            hi[i] = (s[i] + (i+1<n ? hi[i+1]:0) * P) % m1;
        for (int i=n-1;i>=0;i--)
            hi2[i] = (s[i] + (i+1<n ? hi2[i+1]:0) * P) % m2;
    }
    int gethash(int l, int r) {
        int hash = (h[r] - (l ? h[l-1]*p[r-l+1]%m1 : 0));
        int hash2 = (h2[r] - (l ? h2[l-1]*p2[r-l+1]%m2 : 0));
        hash = hash < 0 ? hash + m1 : hash;
        hash2 = hash2 < 0 ? hash2 + m2 : hash2;
        return (hash << 30) ^ hash2;
    }
    int gethashi(int l, int r) {
        int hash = (hi[l] - (r+1 < n ? hi[r+1]*p[r-l+1] % m1 : 0));
        int hash2 = (hi2[l] - (r+1 < n ? hi2[r+1]*p2[r-l+1] % m2 : 0));
        hash= hash < 0 ? hash + m1 : hash;
        hash2= hash2 < 0 ? hash2 + m2 : hash2;
        return (hash << 30) ^ hash2;
    }
};

void solve()
{
    srand(time(0));
    m1 = rand()/10 + 1e9;
    m2 = rand()/10 + 1e9;
    Hash hasher(s);
}
```

6.2 Suffix Array

Complexity: O(n log(n)), contains big constant (around 25). Computes a sorted array of the suffixes of a string.

```
// TITLE: Suffix Array
// COMPLEXITY: O(n log(n)), contains big constant (around 25).
// DESCRIPTION: Computes a sorted array of the suffixes of a string.

void countingsort(vi& p, vi& c) {
    int n=p.size();
    vi count(n,0);
    rep(i,0,n) count[c[i]]++;

    vi psum(n); psum[0]=0;
    rep(i,1,n) psum[i]=psum[i-1]+count[i-1];

    vi ans(n);
    rep(i,0,n)
        ans[psum[c[p[i]]]++]=p[i];

    p = ans;
}

vi sfa(string s) {
    s += "$";

    int n=s.size();
    vi p(n);
    vi c(n);
    {
        vector<pair<char, int>> a(n);
        rep(i,0,n) a[i]={s[i],i};
        sort(all(a));

        rep(i,0,n) p[i]=a[i].second;

        c[p[0]]=0;
        rep(i,1,n) {
            if(s[p[i]] == s[p[i-1]]) {
                c[p[i]]=c[p[i-1]];
            }
            else c[p[i]]=c[p[i-1]]+1;
        }
    }

    for(int k=0; (1<<k) < n; k++) {
        rep(i, 0, n)
            p[i] = (p[i] - (1<<k) + n) % n;

        countingsort(p,c);

        vi nc(n);
        nc[p[0]]=0;
        rep(i,1,n) {
            pii prev = {c[p[i-1]], c[(p[i-1]+(1<<k))%n]};
            pii cur = {c[p[i]], c[(p[i]+(1<<k))%n]};

            if (prev == cur)
                nc[p[i]]=nc[p[i-1]];
            else nc[p[i]]=nc[p[i-1]]+1;
        }
        c=nc;
    }

    return p;
}
```

6.3 Z function

Complexity: O(n)
z[i] = largest m such that s[0..m]=s[i..i+m]

```
// TITLE: Z function
// COMPLEXITY: O(n)
// DESCRIPTION: z[i] = largest m such that s[0..m]=s[i..i+m]

vector<int> Z(string s) {
    int n = s.size();
    vector<int> z(n);
    int x = 0, y = 0;
    for (int i = 1; i < n; i++) {
        z[i] = max(0, min(z[i - x], y - i + 1));
        while (i + z[i] < n and s[z[i]] == s[i + z[i]]) {
            x = i; y = i + z[i]; z[i]++;
        }
    }
    return z;
}
```

7 Geometry

7.1 Point structure

Complexity: Does not apply
Basic 2d point functionality

```
// TITLE: Point structure
// COMPLEXITY: Does not apply
// DESCRIPTION: Basic 2d point functionality

// Point/vector structure definition and sorting

#define T int
float EPS = 1e-6;
bool eq(T a, T b){ return abs(a-b)<=EPS; }

struct point{
    T x, y;
    point(t x=0, t y=0): x(x), y(y){}

    point operator+(const point &o) const{ return {x + o.x, y + o.y}; }
    point operator-(const point &o) const{ return {x - o.x, y + o.y}; }
    point operator*(T k) const{ return {x*k, y*k}; }
    point operator/(T k) const{ return {x/k, y/k}; }
    T operator*(const point &o) const{ return x*o.x + y*o.y; }
    T operator^(const point &o) const{ return x*o.y - y*o.x; }
    bool operator<(const point &o) const{ return (eq(x, o.x) ? y < o.y : x < o.x); }
    bool operator==(const point &o) const{ return eq(x, o.x) and eq(y, o.t); }

    friend ostream& operator<<(ostream& os, point p){
        return os << "(" << p.x << "," << p.y << ")";
    }
};

int ret[2][2] = {{3, 2}, {4, 1}};
inline int quad(point p){
    return ret[p.x >= 0][p.y >= 0];
}

bool comp(point a, point b){
    int qa = quad(a), qb = quad(b);
    return (qa == qb ? (a ^ b) > 0 : qa < qb);
}
```

7.2 Lattice Points

Complexity: N
Points with integer coordinate

```
// TITLE: Lattice Points
// COMPLEXITY: N
// DESCRIPTION: Points with integer coordinate

// Picks theorem
// A = area
// i = points_inside
// b = points in boundary including vertices
// A = i + b/2 - 1

void solve()
{
  int n; cin >> n;
  vector<Point> points(n);
  for (int i = 0; i < n; i++) {
    points[i].read();
  }

  // Calculatting points on boundary
  int B = 0;
  for (int i =0; i < n; i++) {
    int j = (i + 1) % n;
    Point p = points[j] - points[i];
    B += __gcd(abs(p.x), abs(p.y)); // Unsafe for 0
  }
  // Calculating Area
  int a2 = 0;
  for (int i= 0; i < n; i++) {
    int j = (i + 1) % n;
    a2 += points[i] * points[j];
  }
  a2 = abs(a2);
  // Picks theorem
  int I = (a2 - B + 2)/2;
  cout << I << " " << B << endl;
}
```

7.3 Convex Hull

Complexity: N
Gives you the convex hull of a set of points

```
// TITLE: Convex Hull
// COMPLEXITY: N
// DESCRIPTION: Gives you the convex hull of a set of points


struct Point
{
  int x, y;

  void read()
  {
    cin >> x >> y;
  }

  Point operator- (const Point & b) const
  {
    Point p;
    p.x = x - b.x;
    p.y = y - b.y;
    return p;
  }

  void operator-= (const Point & b)
  {
    x -= b.x;
    y -= b.y;
  }

  int operator* (const Point & b) const
  {
    return x * b.y - b.x * y;
  }

  bool operator< (const Point & b) const
  {
    return make_pair(x, y) < make_pair(b.x, b.y);
  }

};

int triangle(const Point & a, const Point & b, const Point & c)
{
  return (b - a) * (c - a);
```

```
}

vector<Point> convex_hull(vector<Point> points)
{
  vector<Point> hull;
  sort(all(points));

  for (int z = 0; z < 2; z++) {
    int s = hull.size();
    for (int i = 0; i < points.size(); i++) {
        while(hull.size() >= s + 2) {
            auto a = hull.end()[-2];
            auto b = hull.end()[-1];
            if (triangle(a, b, points[i]) <= 0) {
                break;
            }
            hull.pop_back();
        }
        hull.push_back(points[i]);
    }
    hull.pop_back();
    reverse(all(points));
  }
  return hull;
}
```

7.4 Line Intersegment

Complexity: O(1)
Check if two half segments intersect with which other

```
// TITLE: Line Intersegment
// COMPLEXITY: O(1)
// DESCRIPTION: Check if two half segments intersect
    with which other

struct Point
{
  int x, y;

  void read()
  {
    cin >> x >> y;
  }

  Point operator- (const Point & b) const
  {
    Point p;
    p.x = x - b.x;
    p.y = y - b.y;
    return p;
  }

  void operator-= (const Point & b)
  {
    x -= b.x;
    y -= b.y;
  }

  int operator* (const Point & b) const
  {
    return x * b.y - b.x * y;
  }

};

int triangle(const Point & a, const Point & b, const
    Point & c)
{
  return (b - a) * (c - a);
}

bool intersect(const Point & p1, const Point & p2,
    const Point & p3, const Point & p4) {
  bool ans = true;
  int s1 = triangle(p1, p2, p3);
  int s2 = triangle(p1, p2, p4);

  if (s1 == 0 && s2 == 0) {
    int a_min_x = min(p1.x, p2.x);
    int a_max_x = max(p1.x, p2.x);
    int a_min_y = min(p1.y, p2.y);
    int a_max_y = max(p1.y, p2.y);

    int b_min_x = min(p3.x, p4.x);
    int b_max_x = max(p3.x, p4.x);
    int b_min_y = min(p3.y, p4.y);
    int b_max_y = max(p3.y, p4.y);
    if (a_min_x > b_max_x || a_min_y > b_max_y) {
      ans = false;
    }
    if (b_min_x > a_max_x || b_min_y > a_max_y) {
      ans = false;
    }
    return ans;
  }
  int s3 = triangle(p3, p4, p1);
  int s4 = triangle(p3, p4, p2);

  if ((s1 < 0) && (s2 < 0)) ans = false;
  if ((s1 > 0) && (s2 > 0)) ans = false;
  if ((s3 < 0) && (s4 < 0)) ans = false;
  if ((s3 > 0) && (s4 > 0)) ans = false;
  return ans;
}
```

8 Algorithms

8.1 Sparse table

Complexity: O(n log(n)) preprocessing, O(1) query
Computes the minimum of a half open interval.

```
// TITLE: Sparse table
// COMPLEXITY: O(n log(n)) preprocessing, O(1) query
// DESCRIPTION: Computes the minimum of a half open
    interval.

struct sptable {
    vector<vi> table;

    int ilog(int x) {
        return (__builtin_clzll(1ll) -
    __builtin_clzll(x));
    }

    sptable(vi& vals) {
        int n = vals.size();
        int ln= ilog(n)+1;
        table.assign(ln, vi(n));

        rep(i,0,n) table[0][i]=vals[i];

        rep(k, 1, ln) {
            rep(i,0,n) {
                table[k][i] = min(table[k-1][i],
                table[k-1][min(i + (1<<(k-1)), n-1)])
    ;
            }
        }
    }

```

```
    // returns minimum of vals in range [a, b)
    int getmin(int a, int b) {
        int k = ilog(b-a);
        return min(table[k][a], table[k][b-(1<<k)]);
    }
};
```

9 Parser

9.1 Parsing Functions

Complexity:

```
// TITLE: Parsing Functions

vector<string> split_string(const string & s, const
    string & sep = " ") {
    int w = sep.size();
    vector<string> ans;
    string curr;

    auto add = [&](string a) {
        if (a.size() > 0) {
            ans.push_back(a);
        }
    };

    for (int i = 0; i + w < s.size(); i++) {
        if (s.substr(i, w) == sep) {
            i += w-1;
            add(curr);
            curr.clear();
            continue;
        }
        curr.push_back(s[i]);
    }
    add(curr);
    return ans;
}

vector<int> parse_vector_int(string & s)
{
    vector<int> nums;
    for (string x: split_string(s)) {
        nums.push_back(stoi(x));
    }
    return nums;
}

vector<float> parse_vector_float(string & s)
{
    vector<float> nums;
    for (string x: split_string(s)) {
        nums.push_back(stof(x));
    }
    return nums;
}

void solve()
{
    cin.ignore();
    string s;
    getline(cin, s);

    auto nums = parse_vector_float(s);
    for (auto x: nums) {
        cout << x << endl;
    }
}
```