ICPC Library

Keigo Habara

2019年12月4日

1 Template

```cpp
#ifndef TEMPLATE
#define TEMPLATE

// #include <bits/stdc++.h>
using namespace std;

#define SZ(x) (int)(x.size())
#define REP(i, n) for(int i = 0; i < n; i++)
#define FOR(i, a, b) for(int i = a; i < b; i++)

using ll = long long;
using ld = long double;
using P = pair<int, int>;
using vi = vector<int>;
using vvi = vector<vi>;
using vll = vector<ll>;
using vvll = vector<vll>;
const double eps = 1e-8;
const int MOD = 1000000007;

//int main() {
// cin.tie(0);
// ios::sync_with_stdio(false);
// cout << fixed << setprecision(10);
//
//}

#endif
```

2 Graph

2.1 Dijkstra

```cpp
#include "template.cpp"

template<typename T>
vector<T> dijkstra(const Graph<T> &g, int s) {
    const auto INF = numeric_limits<T>::max();
    vector<T> d(g.size(), INF);

    using Pi = pair<T, int>;
    priority_queue<Pi, vector<Pi>, greater<Pi>> que;
    d[s] = 0;
    que.emplace(d[s], s);
    while (!que.empty()) {
        T cost;
        int v;
        tie(cost, v) = que.top();
        que.pop();
        if (d[v] < cost) continue;
        for (auto &e : g[v]) {
            auto nxt = cost + e.cost;
            if (d[e.to] > nxt) {
                d[e.to] = nxt;
                que.emplace(nxt, e.to);
            }
        }
    }
    return d;
}
```

2.2 Dinic

```cpp
#include "template.cpp"

template<typename T>
struct Dinic {
    const T INF;

    struct edge {
        int to;
        T cap;
        int rev;
        bool isrev;
    };

    vector<vector<edge>> g;
    vector<int> level, iter;

    Dinic(int V) : INF(numeric_limits<T>::max()), g(V) {}

    void add_edge(int from, int to, T cap) {
        g[from].push_back({to, cap, (int)g[to].size(), false});
        g[to].push_back({from, 0, (int)g[from].size()-1, true});
    }

    bool bfs(int s, int t) {
        level.assign(g.size(), -1);
        queue<int> que;
        level[s] = 0;
        que.push(s);
        while (!que.empty()) {
            int v = que.front();
            que.pop();
            for (auto &e : g[v]) {
                if (e.cap > 0 && level[e.to] == -1) {
                    level[e.to] = level[v] + 1;
                    que.push(e.to);
                }
            }
        }
        return level[t] != -1;
    }

    T dfs(int v, const int t, T flow) {
        if (v == t) return flow;
        for (int &i = iter[v]; i < g[v].size(); i++) {
            edge &e = g[v][i];
            if (e.cap > 0 && level[v] < level[e.to]) {
                T d = dfs(e.to, t, min(flow, e.cap));
                if (d > 0) {
                    e.cap -= d;
                    g[e.to][e.rev].cap += d;
                    return d;
                }
            }
        }
        return 0;
    }

    T max_flow(int s, int t) {
        T flow = 0;
        while (bfs(s, t)) {
            iter.assign(g.size(), 0);
            T f = 0;
            while((f = dfs(s, t, INF)) > 0) flow += f;
        }
        return flow;
    }
};
```

2.3 Hopcroft Karp

```cpp
#include "template.cpp"

struct HopcroftKarp {
    vector<vector<int>> g;
    vector<int> d, mch;
    vector<bool> used, vv;

    HopcroftKarp(int n, int m) : g(n), mch(m, -1), used(n) {}

    void add_edge(int u, int v) {
        g[u].push_back(v);
    }

    void bfs() {
        d.assign(g.size(), -1);
        queue<int> que;
        for (int i = 0; i < g.size(); i++) {
            if (!used[i]) {
                que.emplace(i);
                d[i] = 0;
            }
        }

        while (!que.empty()) {
            int a = que.front();
            que.pop();
            for (auto &b : g[a]) {
                int c = mch[b];
                if (c >= 0 && d[c] == -1) {
                    d[c] = d[a] + 1;
                    que.emplace(c);
                }
            }
        }
    }

    bool dfs(int a) {
        vv[a] = true;
        for (auto &b : g[a]) {
            int c = mch[b];
            if (c < 0 || (!vv[c] && d[c] == d[a] + 1 && dfs(c))) {
                mch[b] = a;
                used[a] = true;
                return (true);
            }
        }
        return (false);
    }

    int bipartite_matching() {
        int ret = 0;
        while (true) {
            bfs();
            vv.assign(g.size(), false);
            int flow = 0;
            for (int i = 0; i < g.size(); i++) {
                if (!used[i] && dfs(i)) ++flow;
            }
            if (flow == 0) return ret;
            ret += flow;
        }
    }
};
```

2.4 Kruskal

```cpp
#include "template.cpp"

#include "../structure/union_find.cpp"

template<typename T>
T kruskal(vector<edge<T>> &es, int V) {

    UnionFind uf(V);
    T ret = 0;

    // sort destructively
    sort(es.begin(), es.end(), [](edge<T> &a,edge<T> &b){
            return a.cost < b.cost;
            });
    for (auto &e : es) {
        if (!uf.issame(e.src, e.to)) {
            ret += e.cost;
            uf.merge(e.src, e.to);
        }
    }

    // // sort only the order to check
    // vector<int> ord(es.size());
    // iota(ord.begin(), ord.end(), 0);
    // sort(ord.begin(), ord.end(), [&](int i,int j){
    // return es[i].cost < es[j].cost;
    // });
    // for (auto i : ord) {
    // auto &e = es[i];
    // if (!uf.issame(e.src, e.to)) {
    // ret += e.cost;
    // uf.merge(e.src, e.to);
    // }
    // }

    return ret;
}
```

2.5 LCA

```
#include "../template.cpp"

struct LCA {
    int n, log2_n;
    vector<int> depth;
    vector<vector<int>> par;

    void dfs(const vector<vector<int>>& G, int v, int p, int d) {
        depth[v] = d;
        par[0][v] = p;
        for (auto to : G[v]) {
            if (to != p) dfs(G, to, v, d+1);
        }
    }

    LCA(const vector<vector<int>>& G, int root=0) :
        n(G.size()), log2_n(log2(n)), depth(n),
        par(log2_n+1, vector<int>(n,-1)) {

            dfs(G, root, -1, 0);

            for (int k = 0; k < log2_n; ++k) {
                for (int v = 0; v < n; ++v) {
                    if (par[k][v] != -1) {
                        par[k+1][v] = par[k][par[k][v]];
                    }
                }
            }
        }

    int query(int u, int v) {
        if (depth[u] > depth[v]) swap(u, v);

        // align the depth of u and v
        for (int k = 0; k <= log2_n; ++k) {
            if ((depth[v] - depth[u]) >> k & 1) {
                v = par[k][v];
            }
        }
        if (u == v) return u;

        // go back until u and v's parents do not match
        for (int k = log2_n; k >= 0; --k) {
            if (par[k][u] != par[k][v]) {
                u = par[k][u];
                v = par[k][v];
            }
        }
        return par[0][u];
    }
};
```

2.6 Lowlink

```
#include "./template.cpp"

template<typename T>
struct LowLink {
    const int inf = 1000000000;
    int sz;
    std::vector<int> pre, low;
    std::vector<bool> sel;
    std::vector<std::pair<int, int>> bridge;
    std::vector<int> articulation;

    LowLink(const Graph<T> &g) {
        sz = g.size();
        pre.resize(sz, inf);
        low.resize(sz, inf);
        sel.resize(sz, false);
        int cnt = 0;
        dfs(g, 0, -1, cnt);
    }

    void dfs(const Graph<T> &g, int now, int prev, int &cnt) {
        if(pre[now] != inf) {
            low[prev] = min(low[prev], pre[now]);
            return;
        }
        pre[now] = cnt;
        low[now] = cnt;
        cnt++;
        for(int i=0;i<(int)(g[now].size());++i) {
            int nxt = g[now][i].to;
            //if g is an undirected graph
            if(nxt == prev) continue;
            dfs(g, nxt, now, cnt);
        }
        if(prev != -1) low[prev] = min(low[prev], low[now]);
        if(prev != -1 && pre[prev] < low[now]) {
            bridge.emplace_back(make_pair(prev, now));
        }
    }

    void get_articulation(const Graph<T> &g, int now, int prev) {
        sel[now] = true;
        int art = 0;
        for(int i=0;i<(int)(g[now].size());++i) {
            int nxt = g[now][i].to;
            //cout << now << ":" << nxt << endl;
            if(sel[nxt]) continue;
            // if g is an undirected graph
            if(nxt == prev) continue;
            if(now == 0 || pre[now] <= low[nxt]) art++;
            get_articulation(g, nxt, now);
        }
        if((now == 0 && art >= 2) || (now != 0 && art >= 1)) {
            articulation.push_back(now);
        }
    }
};
```

2.7 Maximum Clique

```cpp
#include "template.cpp"

int maximum_clique(const vector<vector<bool>>& G) {
    // G: 隣接行列, 無向グラフ
    int n = G.size();
    vector<int> deg(n);
    int M = 0;
    for (int i = 0; i < n; ++i) {
        for (int j = i+1; j < n; ++j) {
            ++deg[i], ++deg[j], ++M;
        }
    }
    vector<vector<bool>> g = G;
    vector<bool> used(n);

    int lim = sqrt(2*M), ret = 0;

    for (int t = 0; t < n; ++t) {
        int u = -1;
        for (int i = 0; i < n; ++i) {
            if (!used[i] && deg[i] < lim) {
                u = i;
                used[u] = true;
                break;
            }
        }

        vector<int> neighbor;
        if (u != -1) neighbor.push_back(u);
        for (int v = 0; v < n; ++v) if (!used[v]) {
            if (u == -1 || g[u][v]) {
                neighbor.push_back(v);
            }
        }

        int sz = neighbor.size();
        vector<int> bit(sz);
        for(int i = 0; i < sz; i++) {
            for(int j = i+1; j < sz; j++) {
                if(!g[neighbor[i]][neighbor[j]]) {
                    bit[i] |= 1 << j;
                    bit[j] |= 1 << i;
                }
            }
        }

        vector<int> dp(1<<sz);
        dp[0] = 1;
        for (int s = 1; s < 1<<sz; ++s) {
            int i = __builtin_ffs(s) - 1;

            if (dp[s] = dp[s & ~(1<<i)] && (bit[i] & s) == 0) {
                ret = max(ret, __builtin_popcount(s));
            }
        }

        if (u == -1) break;

        for (auto v : neighbor) {
            --deg[v], --deg[u];
            g[u][v] = g[v][u] = false;
        }
    }

    return ret;
}
```

2.8 Primal Dual

```cpp
#include "template.cpp"

template<typename flow_t, typename cost_t>
struct PrimalDual {
    const cost_t INF;

    struct edge {
        int to;
        flow_t cap;
        cost_t cost;
        int rev;
    };
    vector<vector<edge>> g;
    vector<cost_t> h, d;
    vector<int> prevv, preve;

    PrimalDual(int V) : g(V), INF(numeric_limits< cost_t >::max()) {}

    void add_edge(int from, int to, flow_t cap, cost_t cost) {
        g[from].push_back({to, cap, cost, (int)g[to].size()});
        g[to].push_back({from, 0, -cost, (int)g[from].size()-1});
    }

    cost_t min_cost_flow(int s, int t, flow_t f) {
        int V = (int)g.size();
        cost_t ret = 0;
        using Pi = pair<cost_t, int>;
        priority_queue<Pi, vector<Pi>, greater<Pi>> que;
        h.assign(V, 0);
        preve.assign(V, -1);
        prevv.assign(V, -1);

        while (f > 0) {
            d.assign(V, INF);
            que.emplace(0, s);
            d[s] = 0;
            while (!que.empty()) {
                Pi p = que.top(); que.pop();
                if (d[p.second] < p.first) continue;
                for (int i = 0; i < g[p.second].size(); i++) {
                    edge &e = g[p.second][i];
                    cost_t nextCost = d[p.second] + e.cost +
                        h[p.second] - h[e.to];
                    if (e.cap > 0 && d[e.to] > nextCost) {
                        d[e.to] = nextCost;
                        prevv[e.to] = p.second, preve[e.to] = i;
                        que.emplace(d[e.to], e.to);
                    }
                }
            }
            if (d[t] == INF) return -1;
            for (int v = 0; v < V; v++) h[v] += d[v];
            flow_t addflow = f;
            for (int v = t; v != s; v = prevv[v]) {
                addflow = min(addflow, g[prevv[v]][preve[v]].cap);
            }
            f -= addflow;
            ret += addflow * h[t];
            for (int v = t; v != s; v = prevv[v]) {
                edge &e = g[prevv[v]][preve[v]];
                e.cap -= addflow;
                g[v][e.rev].cap += addflow;
            }
        }
        return ret;
    }
};
```

2.9 SCC

```
#include "template.cpp"

template<typename T>
struct SCC {
    int sz, cnt, num;
    vi post, comp;
    vector<pair<int, int>> vp;
    vector<bool> sel;
    Graph<T> revg;

    SCC(const Graph<T> &g) {
        sz = g.size();
        cnt = 0;
        num = 0;
        post.resize(sz, -1);
        comp.resize(sz, -1);
        sel.resize(sz, false);
        revg.resize(sz);
    }

    void build(const Graph<T> &g) {
        for(int i=0;i<sz;++i) {
            if(sel[i]) continue;
            sel[i] = true;
            dfs1(g, i);
        }

        rev(g, revg);

        for(int i=0;i<sz;++i) {
            vp.emplace_back(make_pair(post[i], i));
        }
        sort(vp.begin(), vp.end());
        reverse(vp.begin(), vp.end());
        sel.clear();
        sel.resize(sz, false);
        for(int i=0;i<sz;++i) {
            if(sel[vp[i].second]) continue;
            sel[vp[i].second] = true;
            comp[vp[i].second] = num;
            dfs2(revg, vp[i].second);
            num++;
        }
    }

    vi get_comp() {return comp;}

    Graph<T> build_graph(const Graph<T> &g) {
        build(g);
        vector<set<int>> s(sz);
        Graph<T> res(sz);
        for(int i=0;i<sz;++i) {
            for(int j=0;j<(int)(g[i].size());++j) {
                s[comp[i]].insert(comp[g[i][j].to]);
            }
        }
        for(int i=0;i<sz;++i) {
            for(auto j: s[i]) {
                if(i != j) res[i].push_back(edge<int>(i, j, 1));
            }
        }
        return res;
    }

    void dfs1(const Graph<T> &g, int now) {
        for(int i=0;i<(int)(g[now].size());++i) {
            int nxt = g[now][i].to;
            if(sel[nxt]) continue;
            sel[nxt] = true;
            dfs1(g, nxt);
        }
        post[now] = cnt;
        cnt++;
    }

    void rev(const Graph<T> &g, Graph<T> &revg) {
        for(int i=0;i<sz;++i) {
            for(int j=0;j<(int)(g[i].size());++j) {
                revg[g[i][j].to].push_back({
                        g[i][j].to, g[i][j].src, g[i][j].cost});
            }
        }
    }

    void dfs2(const Graph<T> &revg, int now) {
        for(int i=0;i<(int)(revg[now].size());++i) {
            int nxt = revg[now][i].to;
            if(sel[nxt]) continue;
            sel[nxt] = true;
            comp[nxt] = num;
            dfs2(revg, nxt);
        }
    }
};
```

2.10 Topological Sort

```
#include "template.cpp"

void topological_sort(const vector<vector<int>>& G, vector<int>& ord)
{
    int n = G.size();
    vector<int> num(n, 0);
    ord.assign(n, 0);
    for (int i = 0; i < n; ++i) {
        for (auto u : G[i]) {
            ++num[u];
        }
    }
    stack<int> st;
    for(int i = 0; i < n; ++i) {
        if (num[i] == 0) {
            st.push(i);
        }
    }
    for (int k = 0; !st.empty(); ++k) {
        int i = st.top(); st.pop();
        ord[k] = i;
        for (auto u : G[i]) {
            if (--num[u] == 0) {
                st.push(u);
            }
        }
    }
}
```

2.11 Warshall Floyd

```
#include "template.cpp"

template<typename T>
void warshall_floyd(vector<vector<T>> &g) {
    const auto INF = numeric_limits<T>::max();
    int n = g.size();
    for(int k = 0; k < n; k++) {
        for(int i = 0; i < n; i++) {
            for(int j = 0; j < n; j++) {
                if(g[i][k] == INF || g[k][j] == INF) continue;
                g[i][j] = min(g[i][j], g[i][k] + g[k][j]);
            }
        }
    }
}
```

3 Number

3.1 Mod

```
#include "../template.cpp"

ll powm(ll a, ll n, ll m) {
    ll ret = 1;
    while (n > 0) {
        if (n & 1) (ret *= a) %= m;
        (a *= a) %= m;
        n >>= 1;
    }
    return ret;
}

ll invm(ll a, ll m) {
    return powm(a, m-2, m);
}
```

3.2 ExtendedGCD

```
#include "../template.cpp"

ll extended_gcd(ll a, ll b, ll& x, ll& y) {
    // solve ax + by = gcd(a, b)
    if (b == 0) { x = 1; y = 0; return a; }
    ll X, Y;
    ll g = extended_gcd(b, a % b, X, Y);
    x = Y; y = X - a/b * Y;
    return g;
}
```

3.3 Combination

```
#include "mod.cpp"

vector<ll> fact;
void init_fact(int n, ll m) {
    fact.assign(n+1, 1);
    for (int i = 2; i <= n; ++i) {
        (fact[i] = fact[i-1] * i) %= m;
    }
}

// require init_fact(GREATER THAN OR EQUAL TO n, m)
ll C(ll n, ll r, ll m) {
    return (fact[n] * invm((fact[r] * fact[n-r]) % m, m)) % m;
}

// Stirling number
// Stirling(n, k) := the number of cases
// to split n balls(distinguished)
// into k boxes(not distinguished)
// s.t. each box contains at least one ball.
//
// require init_fact(GREATER THAN OR EQUAL TO k, m)
ll Stirling(ll n, ll k, ll m) {
    ll ret = 0;
    for (ll l = 0; l <= k; ++l) {
        ll tmp = (C(k, l, m) * powm((k-l) % m, n, m)) % m;
        if (l & 1) tmp = (-tmp + m) % m;
        (ret += tmp) %= m;
    }
    return (ret *= invm(fact[k], m)) %= m;
}

```

```
// Bell number
// Bell(n, k) := the number of cases
// to split n balls(distinguished)
// into k boxes(not distinguished)
//
// require init_fact(GREATER THAN OR EQUAL TO k, m)
ll Bell(ll n, ll k, ll m) {
    ll ret = 0;
    for (ll l = 0; l <= k; ++l) {
        (ret += Stirling(n, l, m)) %= m;
    }
    return ret;
}

// Partition function
// Partition[k][n] := the number of cases
// to split n balls(not distinguished)
// into k boxes(not distinguished)
vector<vector<ll>> Part;
void init_partition(ll k, ll n, ll m) {
    Part.assign(k+1, vector<ll>(n+1, 0));
    Part[0][0] = 1;
    for (int i = 1; i <= k; ++i) {
        for (int j = 0; j <= n; ++j) {
            if (j-i >= 0) {
                Part[i][j] = (Part[i-1][j] + Part[i][j-i]) % m;
            } else {
                Part[i][j] = Part[i-1][j];
            }
        }
    }
}
```

4 String

4.1 Rolling Hash

```
#include "../template.cpp"

struct RollingHash {
    const int base = 9973;
    const int mod[2] = {999999937, 1000000007};
    vector<int> s;
    vector<ll> hash[2], pow[2];

    RollingHash(const vector<int> &cs) : s(cs) {
        int n = s.size();
        for (int id = 0; id < 2; ++id) {
            hash[id].assign(n+1, 0);
            pow[id].assign(n+1, 1);
            for (int i = 0; i < n; ++i) {
                hash[id][i+1] = (hash[id][i] * base + s[i]) % mod[id];
                pow[id][i+1] = pow[id][i] * base % mod[id];
            }
        }
    }

    // get hash of s[l:r)
    ll get(int l, int r, int id = 0) {
        ll res = hash[id][r] - hash[id][l] * pow[id][r-l] % mod[id];
        if (res < 0) res += mod[id];
        return res;
    }
};
```

4.2 Z Algorithm

```
#include "../template.cpp"

// GET A[i]: the longest common prefix size of S and S[i:n-1]
template<typename S>
void z_algorithm(const S& s, vector<int>& A) {
    int n = s.size();
    A.resize(n);
    A[0] = n;
    int i = 1, j = 0;
    while (i < n) {
        while (i+j < n && s[j] == s[i+j]) ++j;
        A[i] = j;
        if (j == 0) { ++i; continue; }
        int k = 1;
        while (i+k < n && k+A[k] < j) { A[i+k] = A[k]; ++k; }
        i += k; j -= k;
    }
}
```

5 Structure

5.1 Segment Tree

```
#include "../template.cpp"

template<typename M>
struct SegmentTree {
    int sz;
    vector<M> data;

    // RMQ
    const M e = numeric_limits<M>::max();
    const function<M(M,M)> f = [](M a,M b){ return min(a,b); };

    SegmentTree(int n) {
        sz = 1;
        while (sz < n) sz <<= 1;
        data.assign(2*sz, e);
    }

    void update(int k, const M &x) {
        k += sz;
        data[k] = x;
        while (k >>= 1) {
            data[k] = f(data[2*k], data[2*k+1]);
        }
    }

    M query(int a, int b, int k, int l, int r) {
        if (r <= a || b <= l) {
            return e;
        } else if (a <= l && r <= b) {
            return data[k];
        } else {
            return f(query(a,b,2*k, l,(l+r)/2),
                     query(a,b,2*k+1,(l+r)/2,r));
        }
    }

    M query(int a, int b) {
        // return f[a,b)
        return query(a, b, 1, 0, sz);
    }

    M operator[](int k) {
        return data[k + sz];
    }
};
```

5.2 Lazy Segment Tree

```
#include "../template.cpp"

template<typename M, typename OM = M>
struct LazySegmentTree {
    int sz;
    vector<M> data;
    vector<OM> lazy;

    // RangeSumRangeAdd
    const function<M(M,M)> f = [](M a,M b){ return a+b; };
    const function<M(M,OM,int)> g = [](M a,OM b,int l){ return a+b*l; };
    const function<OM(OM,OM)> h = [](OM a,OM b){ return a+b; };
    const M e = 0;
    const OM oe = 0;

    LazySegmentTree(int n) {
        sz = 1;
        while (sz < n) sz <<= 1;
        data.assign(2*sz, e);
        lazy.assign(2*sz, oe);
    }

    void propagate(int k, int len) {
        if (lazy[k] == oe) return;
        if (k < sz) {
            lazy[2*k ] = h(lazy[2*k ], lazy[k]);
            lazy[2*k+1] = h(lazy[2*k+1], lazy[k]);
        }
        data[k] = g(data[k], lazy[k], len);
        lazy[k] = oe;
    }

    M update(int a, int b, const OM &x, int k, int l, int r) {
        propagate(k, r - l);
        if (r <= a || b <= l) {
            return data[k];
        } else if (a <= l && r <= b) {
            lazy[k] = h(lazy[k], x);
            propagate(k, r - l);
            return data[k];
        } else {
            return data[k] = f(
                update(a, b, x, 2*k, l, (l+r)/2),
                update(a, b, x, 2*k+1, (l+r)/2, r));
        }
    }

    void update(int a, int b, const OM &x) {
        // update [a, b) with x.
        update(a, b, x, 1, 0, sz);
    }

    M query(int a, int b, int k, int l, int r) {
        propagate(k, r - l);
        if (r <= a || b <= l) {
            return e;
        } else if (a <= l && r <= b) {
            return data[k];
        } else {
            return f(
                query(a, b, 2*k, l, (l+r)/2),
                query(a, b, 2*k+1, (l+r)/2, r));
        }
    }

    M query(int a, int b) {
        // return f[a, b).
        return query(a, b, 1, 0, sz);
    }
};
```

5.3 Union Find

```
#include "../template.cpp"

struct UnionFind
{
    vector<int> par, sz;
    UnionFind(int n) : par(n), sz(n, 1) {
        for (int i = 0; i < n; ++i) par[i] = i;
    }
    int root(int x) {
        if (par[x] == x) return x;
        return par[x] = root(par[x]);
    }
    void merge(int x, int y) {
        x = root(x);
        y = root(y);
        if (x == y) return;
        if (sz[x] < sz[y]) swap(x, y);
        par[y] = x;
        sz[x] += sz[y];
        sz[y] = 0;
    }
    bool issame(int x, int y) {
        return root(x) == root(y);
    }
    int size(int x) {
        return sz[root(x)];
    }
};
```

5.4 Weighted Union Find

```
#include "../template.cpp"

template<typename A>
struct WeightedUnionFind
{
    vector<int> par, sz;
    vector<A> data; // data[x]: diff from root to x
    WeightedUnionFind(int n, A e=0) :
        par(n), sz(n, 1), data(n, e) {
        for (int i = 0; i < n; ++i) par[i] = i;
    }

    int root(int x) {
        if (par[x] == x) return x;
        int r = root(par[x]);
        data[x] += data[par[x]];
        return par[x] = r;
    }

    A weight(int x) {
        root(x);
        return data[x];
    }

    A diff(int x, int y) {
        // diff from x to y
        return data[y] - data[x];
    }

    void merge(int x, int y, A w) {
        // merge so that "diff from x to y" will be w.
        w += weight(x); w -= weight(y);
        x = root(x); y = root(y);
        if (x == y) return;
        if (sz[x] < sz[y]) swap(x, y), w = -w;
        par[y] = x;
        sz[x] += sz[y];
        sz[y] = 0;
        data[y] = w;
    }

    bool issame(int x, int y) {
        return root(x) == root(y);
    }
};
```

6 Vimrc

```
syntax enable
set number
set autoindent
set expandtab
set tabstop=4
set shiftwidth=4
```

7 Emacs

```
(package-initialize)

(setq inhibit-startup-message t)

(setq make-backup-files nil)

(setq delete-auto-save-files t)

(setq-default tab-width 4 indent-tabs-mode nil)

(setq eol-mnemonic-dos "(CRLF)")
(setq eol-mnemonic-mac "(CR)")
(setq eol-mnemonic-unix "(LF)")

(tool-bar-mode -1)
(menu-bar-mode -1)

(column-number-mode t)
(global-linum-mode t)
(blink-cursor-mode t)
(show-paren-mode 1)
(setq mouse-wheel-scroll-amount ’(1 ((shift) . 5)))
(load-theme ’monokai t)
```