Contents

1 Vim

1.1 vim config

```
set nu
set relativenumber

set tabstop=4
set softtabstop=4
set shiftwidth=4
set expandtab
set smartindent

let mapleader=" "

vnoremap jk <esc>
inoremap jk <esc>

nnoremap <leader>3 <esc>:w <cr><cmd>!g++ -ggdb -fsanitize=address,
    undefined -std=c++17 -DLOCAL -O2 %:~ -o %:p:r && %:~:r < %:~:h/inp
     <cr>
inoremap <leader>3 <esc>:w <cr><cmd>!g++ -ggdb -fsanitize=address,
    undefined -std=c++17 -DLOCAL -O2 %:~ -o %:p:r && %:~:r < %:~:h/inp
     <cr>
vnoremap <leader>3 <esc>:w <cr><cmd>!g++ -ggdb -fsanitize=address,
    undefined -std=c++17 -DLOCAL -O2 %:~ -o %:p:r && %:~:r < %:~:h/inp
     <cr>
```

2 Graph

2.1 Kruskal

```cpp
struct Edge {
    int u, v, w;
    bool operator < (Edge const &other)
        return w < other.w;
} edges[N]; vector<Edge> result;
// -1 -> not exitst mst
ll kruskal(int n, int m) { // O(m*log(n))
    result.clear();
    ll total_weight = 0;
    sort(edges, edges + m);
    dsu d(n);
    int nodes = 0;
    for (auto &e : edges) {
        if (!d.same(e.u, e.v)) {
            total_weight += e.w;
            result.push_back(e);
            d.merge(e.u, e.v);
            nodes++;
        }
        if (nodes == n - 1) break;
    }
    if (nodes != n - 1) return -1;
    return total_weight;
}
```

2.2 SPFA

```cpp
const ll INF = 4e18;
ll d[N], cnt[N]; bool inq[N];
// false -> exist negative cycle
bool spfa(int s, int n) { // O(n*m)
    fill(d, d + n, INF);
    fill(cnt, cnt + n, 0);
    fill(inq, inq + n, 0);
    queue<int> q; q.push(s);
    d[s] = 0; inq[s] = true;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        inq[u] = false;
        for (auto &e : adj[u]) {
            auto [v, w] = e.first;
            if (d[u] + w < d[v]) {
                d[v] = d[u] + w;
                if (!inq[v]) {
                    q.push(v); inq[v] = true;
                    cnt[v]++;
                    if (cnt[v] > n) return false;
                }
            }
        }
    }
    return true;
}
```

2.3 Bellman Ford

```cpp
struct Edge { ll u, v, w; };
const ll INF = 4e18;
vector<Edge> edges; vector<int> neg_path;
ll d[N], p[N];
```

```
// false -> exist negative cycle
bool bellman_ford(int s, int n) { // O(n*m)
    fill(d, d + n, INF);
    fill(p, p + n, -1);
    neg_path.clear();
    d[s] = 0; int x = -1;
    for (int i = 0; i < n; i++) {
        x = -1;
        for (auto &e : edges)
            if (d[e.u] < INF)
                if (d[e.v] > d[e.u] + e.w) {
                    d[e.v] = max(-INF, d[e.u] + e.w);
                    p[e.v] = e.u;
                    x = e.v;
                }
        if (x == -1) return false;
    }
    int y = x;
    for (int i = 0; i < n; i++) y = p[y];
    for (int cur = y; ; cur = p[cur]) {
        neg_path.push_back(cur);
        if (cur == y && int(neg_path.size()) > 1) break;
    }
    reverse(neg_path.begin(), neg_path.end());
    return true;
}
```

2.4 Dijkstra

```
const ll INF = 4e18;
ll d[N], p[N], cnt[N];
void dij(int s, int n) { // O(m*log(n))
    fill(d, d + n, INF);
    fill(p, p + n, -1);
    fill(cnt, cnt + n, 0);
    priority_queue<pair<ll, ll>, vector<pair<ll, ll>>, greater<pair<ll
        , ll>>> q;
    q.push({0, s});
    d[s] = 0; cnt[s] = 1;
    while (!q.empty()) {
        auto [d_u, u] = q.top(); q.pop();
        if (d_u != d[u]) continue;
        for (auto &e : adj[u]) {
            auto [v, w] = e;
            if (d[u] + w < d[v]) {
                d[v] = d[u] + w;
                p[v] = u;
                cnt[v] = cnt[u];
                q.push({d[v], v});
            }
            else if (d[u] + w == d[v])
                cnt[v] = (cnt[v] + cnt[u]);
        }
    }
}
```

2.5 Articulation Points

```
vector<int> tin, low, art_points;
int timer;
void dfs(int u, int p = -1) {
    vis[u] = true;
    tin[u] = low[u] = timer++;
    int children = 0;
    for (int &v : adj[u]) {
        if (v == p) continue;
        if (vis[v]) low[u] = min(low[u], tin[v]);
        else {
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            if (low[v] >= tin[u] && p != -1)
                art_points.push_back(u);
            children++;
        }
    }
    if (p == -1 && children > 1)
        art_points.push_back(u);
}
void find_art_points(int n) {
    art_points.clear();
    tin.assign(n, -1);
    low.assign(n, -1);
    timer = 0;
    for (int u = 0; u < n; u++)
        if (!vis[u]) dfs(u);
    sort(art_points.begin(), art_points.end());
    art_points.erase(unique(art_points.begin(), art_points.end()),
        art_points.end());
}
```

2.6 Bridge

```
vector<pair<int, int>> bridges;
vector<int> tin, low;
int timer;
void dfs(int u, int p = -1) {
    vis[u] = true;
    tin[u] = low[u] = timer++;
    for (auto &v : adj[u]) {
        if (v == p) continue;
        if (vis[v])
            low[u] = min(low[u], tin[v]);
        else {
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            if (low[v] > tin[u])
                bridges.push_back({u, v});
        }
    }
}
void find_bridges(int n) {
    bridges.clear();
    tin.assign(n, -1);
    low.assign(n, -1);
    timer = 0;
    for (int u = 0; u < n; u++)
        if (!vis[u]) dfs(u);
}
```

2.7 SCC

```
vector<int> order, component, root_nodes, roots(N), adj_scc[N];
void dfs1(int u) {
    vis[u] = true;
    for (auto &v : adj[u])
        if (!vis[v]) dfs1(v);
    order.push_back(u);
}
void dfs2(int u) {
    vis[u] = true;
    component.push_back(u);
    for (auto &v : adj_rev[u])
        if (!vis[v]) dfs2(v);
}
void scc(int n) {
    order.clear();
    root_nodes.clear();
    for (int u = 0; u < n; u++)
        if (!vis[u]) dfs1(u);
    reverse(order.begin(), order.end());
    fill(vis, vis + n, false);
    for (auto &u : order) {
        if (!vis[u]) {
            dfs2(u);
            int root = component.front();
            for (auto &v : component)
                roots[v] = root;
            root_nodes.push_back(root);
            adj_scc[root].clear();
            component.clear();
        }
    }
    for (int u = 0; u < n; u++) {
        for (auto &v : adj[u]) {
            int root_u = roots[u], root_v = roots[v];
            if (root_u != root_v)
                adj_scc[root_u].push_back(root_v);
        }
    }
}
```

2.8 Directed Cycle

```
vector<int> color, parent, cycle;
int cycle_start, cycle_end;
bool dfs(int u) {
    color[u] = 1;
    for (auto &v : adj[u]) {
        if (color[v] == 0) {
            parent[v] = u;
            if (dfs(v)) return true;
        }
        else if (color[v] == 1) {
            cycle_end = u;
            cycle_start = v;
            return true;
        }
    }
    color[u] = 2;
    return false;
}
bool find_cycle(int n) {
    cycle.clear();
    color.assign(n, 0);
    parent.assign(n, -1);
    cycle_start = -1;
    for (int u = 0; u < n; u++)
        if (color[u] == 0 && dfs(u)) break;
    if (cycle_start == -1) return false;
    cycle.push_back(cycle_start);
    for (int u = cycle_end; u != cycle_start; u = parent[u])
        cycle.push_back(u);
    cycle.push_back(cycle_start);
    reverse(cycle.begin(), cycle.end());
    return true;
}
```

2.9 Undirected Cycle

```
vector<int> parent, cycle;
int cycle_start, cycle_end;
bool dfs(int u, int p) {
    vis[u] = true;
    for (auto &v : adj[u]) {
        if(v == p) continue;
        if (vis[v]) {
            cycle_end = u;
            cycle_start = v;
            return true;
        }
        parent[v] = u;
        if (dfs(v, parent[v])) return true;
    }
    return false;
}
bool find_cycle(int n) {
    cycle.clear();
    parent.assign(n, -1);
    cycle_start = -1;
    for (int u = 0; u < n; u++)
        if (!vis[u] && dfs(u, parent[u]))
            break;
    if (cycle_start == -1) return false;
    cycle.push_back(cycle_start);
    for (int u = cycle_end; u != cycle_start; u = parent[u])
        cycle.push_back(u);
    cycle.push_back(cycle_start);
    return true;
}
```