- Graph input (directed/undirected)
- DFS
- BFS
- Dijkstra (مع استرجاع المسار)
- Bellman-Ford (مع كشف الـ negative cycle)
- Floyd-Warshall
- Topological Sort
- Union-Find (DSU)
- 0-1 BFS

```
// ======================
// Graph Template by Reem
// ======================

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

#define int long long
#define pb push_back
#define pii pair<int, int>
#define all(v) v.begin(), v.end()

const int INF = 1e18;
const int N = 1e5 + 5;

int n, m;
vector<vector<pii>> adj(N); // adjacency list: {to, weight}
vector<int> dist, parent, color;
vector<bool> visited;
vector<int> topo_order;
bool hasCycle = false;
int par[N], sz[N];

// ---------------------- Graph Input ----------------------
// Reads edges and builds the graph (directed/undirected, weighted/unweighted)
void readGraph(bool directed = false, bool weighted = false) {
    cin >> n >> m;
    adj.assign(n + 1, {});
    for (int i = 0; i < m; ++i) {
        int u, v, w = 1;
        if (weighted) cin >> u >> v >> w;
        else cin >> u >> v;
        adj[u].pb({v, w});
        if (!directed) adj[v].pb({u, w});
    }
}

// ---------------------- DFS ----------------------
// Basic Depth-First Search
void dfs(int u) {
    visited[u] = true;
    for (auto [v, _] : adj[u])
        if (!visited[v])
            dfs(v);
}

// ---------------------- BFS ----------------------
// Breadth-First Search for shortest path in unweighted graph
void bfs(int src) {
```

```
    queue<int> q;
    dist.assign(n + 1, INF);
    dist[src] = 0;
    q.push(src);
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (auto [v, _] : adj[u]) {
            if (dist[v] == INF) {
                dist[v] = dist[u] + 1;
                q.push(v);
            }
        }
    }
}

// ---------------------- Dijkstra ----------------------
// Shortest path with non-negative weights
void dijkstra(int src) {
    dist.assign(n + 1, INF);
    parent.assign(n + 1, -1);
    priority_queue<pii, vector<pii>, greater<>> pq;
    dist[src] = 0;
    pq.push({0, src});
    while (!pq.empty()) {
        auto [d, u] = pq.top(); pq.pop();
        if (d > dist[u]) continue;
        for (auto [v, w] : adj[u]) {
            if (dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
                parent[v] = u;
                pq.push({dist[v], v});
            }
        }
    }
}

// Returns the path from source to destination
vector<int> getPath(int dest) {
    vector<int> path;
    for (int cur = dest; cur != -1; cur = parent[cur]) path.pb(cur);
    reverse(all(path));
    return path;
}


// ---------------------- Bellman-Ford ----------------------
// Shortest paths + detects negative weight cycles
bool bellman_ford(int src) {
    dist.assign(n + 1, INF);
    dist[src] = 0;
    for (int i = 1; i < n; ++i)
        for (int u = 1; u <= n; ++u)
            for (auto [v, w] : adj[u])
                if (dist[u] < INF && dist[v] > dist[u] + w)
                    dist[v] = dist[u] + w;
    for (int u = 1; u <= n; ++u)
        for (auto [v, w] : adj[u])
            if (dist[u] < INF && dist[v] > dist[u] + w)
                return false; // negative cycle
    return true;
}

// ---------------------- Floyd-Warshall ----------------------
// All-pairs shortest paths (small graphs only)
```

```
vector<vector<int>> floyd() {
    vector<vector<int>> dp(n + 1, vector<int>(n + 1, INF));
    for (int i = 1; i <= n; ++i) dp[i][i] = 0;
    for (int u = 1; u <= n; ++u)
        for (auto [v, w] : adj[u])
            dp[u][v] = min(dp[u][v], w);
    for (int k = 1; k <= n; ++k)
        for (int i = 1; i <= n; ++i)
            for (int j = 1; j <= n; ++j)
                dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j]);
    return dp;
}

// ---------------------- Topological Sort ----------------------
// Valid for Directed Acyclic Graph (DAG)
vector<int> topo_sort() {
    vector<int> in(n + 1, 0), res;
    for (int u = 1; u <= n; ++u)
        for (auto [v, _] : adj[u])
            in[v]++;
    queue<int> q;
    for (int i = 1; i <= n; ++i)
        if (in[i] == 0) q.push(i);
    while (!q.empty()) {
        int u = q.front(); q.pop();
        res.pb(u);
        for (auto [v, _] : adj[u])
            if (--in[v] == 0)
                q.push(v);
    }
    return res;
}

// ---------------------- Cycle Detection ----------------------
// For undirected graphs
bool dfs_cycle(int u, int p) {
    visited[u] = true;
    for (auto [v, _] : adj[u]) {
        if (!visited[v]) {
            if (dfs_cycle(v, u)) return true;
        } else if (v != p) return true;
    }
    return false;
}

bool isCyclicUndirected() {
    visited.assign(n + 1, false);
    for (int i = 1; i <= n; i++)
        if (!visited[i] && dfs_cycle(i, -1))
            return true;
    return false;
}

// For directed graphs
bool dfs_directed_cycle(int u, vector<int>& state) {
    state[u] = 1;
    for (auto [v, _] : adj[u]) {
        if (state[v] == 1) return true;
        if (state[v] == 0 && dfs_directed_cycle(v, state)) return true;
    }
    state[u] = 2;
    return false;
}
```

```
bool isCyclicDirected() {
    vector<int> state(n + 1, 0);
    for (int i = 1; i <= n; i++)
        if (state[i] == 0 && dfs_directed_cycle(i, state))
            return true;
    return false;
}

// ---------------------- DAG Check ----------------------
bool isDAG() {
    return !isCyclicDirected();
}

// ---------------------- Check if Tree ----------------------
// Must be undirected, connected, and have n-1 edges
int countEdges() {
    int total = 0;
    for (int u = 1; u <= n; u++)
        total += adj[u].size();
    return total / 2;
}

void dfs_simple(int u) {
    visited[u] = true;
    for (auto [v, _] : adj[u])
        if (!visited[v])
            dfs_simple(v);
}

int connectedComponents() {
    visited.assign(n + 1, false);
    int cnt = 0;
    for (int i = 1; i <= n; i++)
        if (!visited[i]) {
            dfs_simple(i);
            cnt++;
        }
    return cnt;
}

bool isTree() {
    return connectedComponents() == 1 && countEdges() == n - 1;
}

// ---------------------- Bipartite Check (Coloring) ----------------------
bool dfs_color(int u, int c) {
    color[u] = c;
    for (auto [v, _] : adj[u]) {
        if (color[v] == -1) {
            if (!dfs_color(v, c ^ 1)) return false;
        } else if (color[v] == c) {
            return false;
        }
    }
    return true;
}

bool isBipartite() {
    color.assign(n + 1, -1);
    for (int i = 1; i <= n; ++i)
        if (color[i] == -1 && !dfs_color(i, 0))
            return false;
```

```
    return true;
}

// isBipartite check code (BFS)
int bi[N];
bool isBipartite(int src){// two sets, not have edges between every node in one set
    // *notes -> any cycle in graph have (odd) nodes so it not bipartite
    // -> it can be biColored
    clr(bi, OO);
    queue<int> q;
    bi[src] = 1;
    q.push(src);
    while(!q.empty()){
        int node = q.front();
        q.pop();
        for(auto child : adj[node]){
            if(bi[child] == OO) {
                q.push(child);
                bi[child] = 3 - bi[node]; // 1 for set1, 2 for set2 other is not declared
            }else if(bi[child] == bi[node]){
                return false;
            }
        }
    }
    return true;
}

// ---------------------- Union-Find (DSU) ----------------------
// checks if two nodes are in the same group, helps detect cycles and build minimum spanning trees

void initDSU(int size) {
    for (int i = 0; i <= size; ++i) par[i] = i, sz[i] = 1;
}

int find(int u) {
    if (u == par[u]) return u;
    return par[u] = find(par[u]);
}
bool sameSet(int x, int y){ // O(n)
    return getRoot(x) == getRoot(y);
}

bool unite(int u, int v) {
    u = find(u); v = find(v);
    if (u == v) return false;
    if (sz[u] < sz[v]) swap(u, v);
    par[v] = u;
    sz[u] += sz[v];
    return true;
}
int getSize(int x){
    int leader = getRoot(x);
    return setSize[leader];
}

// ---------------------- 0-1 BFS ----------------------
// Fast BFS for edges with weights 0 or 1
void zero_one_bfs(int src) {
    dist.assign(n + 1, INF);
    deque<int> dq;
    dist[src] = 0;
    dq.push_front(src);
    while (!dq.empty()) {
```

```
        int u = dq.front(); dq.pop_front();
        for (auto [v, w] : adj[u]) {
            if (dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
                if (w == 1) dq.push_back(v);
                else dq.push_front(v);
            }
        }
    }
}

// ---------------------- Main ----------------------
int32_t main() {
    ios::sync_with_stdio(0); cin.tie(0);
    // readGraph(false, true);
    return 0;
}
```