Graphs on Grid – Contest Template

```
// ================================================
// Prepared by Reem Elsayed Ghareeb
// ================================================
// Common movement arrays + templates for BFS/DFS/grid pathfinding

// --------------------
// 📌 MOVEMENT ARRAYS
// --------------------

// Knight moves (8 directions - like chess knight)
int dxK[] = {-2, -2, -1, -1, 2, 2, 1, 1};
int dyK[] = {1, -1, 2, -2, 1, -1, 2, -2};

// 4 directions: up, down, left, right
int dx4[] = {1, -1, 0, 0};
int dy4[] = {0, 0, 1, -1};
char dir4[] = {'D', 'U', 'R', 'L'}; // Optional: use when reconstructing path

// 8 directions: includes diagonals
int dx8[] = {1, -1, 0, 0, 1, 1, -1, -1};
int dy8[] = {0, 0, 1, -1, 1, -1, 1, -1};

// ------------------------------
//  1. Shortest Path in a Maze
// ------------------------------
// BFS to find shortest path in unweighted grid

int n, m;
vector<string> grid;
vector<vector<int>> dist;

bool isValid(int x, int y) {
    return x >= 0 && x < n && y >= 0 && y < m && grid[x][y] != '#' && dist[x][y] == -1;
}

void bfsMaze(int sx, int sy) {
    dist.assign(n, vector<int>(m, -1));
    queue<pair<int, int>> q;
    q.push({sx, sy});
    dist[sx][sy] = 0;
    while (!q.empty()) {
        auto [x, y] = q.front(); q.pop();
        for (int d = 0; d < 4; ++d) {
            int nx = x + dx4[d], ny = y + dy4[d];
            if (isValid(nx, ny)) {
                dist[nx][ny] = dist[x][y] + 1;
                q.push({nx, ny});
            }
        }
    }
}

// -------------------------------------
//  2. Multi-Source BFS (fire, zombies)
// -------------------------------------
// Start BFS from multiple starting points (e.g., fire, virus)

void multiSourceBFS(vector<pair<int, int>> sources) {
```

```
    dist.assign(n, vector<int>(m, -1));
    queue<pair<int, int>> q;
    for (auto [x, y] : sources) {
        dist[x][y] = 0;
        q.push({x, y});
    }
    while (!q.empty()) {
        auto [x, y] = q.front(); q.pop();
        for (int d = 0; d < 4; ++d) {
            int nx = x + dx4[d], ny = y + dy4[d];
            if (isValid(nx, ny)) {
                dist[nx][ny] = dist[x][y] + 1;
                q.push({nx, ny});
            }
        }
    }
}

// ------------------------------------------------
//  3. Count Connected Components in a Grid
// ------------------------------------------------
// DFS to count separate "islands" or connected groups

void dfsComponent(int x, int y) {
    if (!isValid(x, y)) return;
    dist[x][y] = 1;
    for (int d = 0; d < 4; ++d)
        dfsComponent(x + dx4[d], y + dy4[d]);
}

int countComponents() {
    int count = 0;
    dist.assign(n, vector<int>(m, -1));
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
            if (grid[i][j] != '#' && dist[i][j] == -1) {
                dfsComponent(i, j);
                count++;
            }
    return count;
}

// ---------------------------------------
//  4. Boundary Flood Fill (e.g. water)
// ---------------------------------------
// Spread from boundaries inward (use for ocean flow, etc.)

void floodFromBoundary() {
    dist.assign(n, vector<int>(m, -1));
    queue<pair<int, int>> q;
    for (int i = 0; i < n; i++) {
        if (grid[i][0] != '#') { dist[i][0] = 0; q.push({i, 0}); }
        if (grid[i][m - 1] != '#') { dist[i][m - 1] = 0; q.push({i, m - 1}); }
    }
    for (int j = 0; j < m; j++) {
        if (grid[0][j] != '#') { dist[0][j] = 0; q.push({0, j}); }
        if (grid[n - 1][j] != '#') { dist[n - 1][j] = 0; q.push({n - 1, j}); }
    }
    while (!q.empty()) {
        auto [x, y] = q.front(); q.pop();
        for (int d = 0; d < 4; ++d) {
            int nx = x + dx4[d], ny = y + dy4[d];
            if (isValid(nx, ny)) {
```

```
                dist[nx][ny] = dist[x][y] + 1;
                q.push({nx, ny});
            }
        }
    }
}

// ----------------------------------------
//  5. Grid with Walls or Weighted Cells
// ----------------------------------------
// Use Dijkstra if grid has weights other than 1

vector<vector<int>> weight;

void dijkstraOnGrid(int sx, int sy) {
    dist.assign(n, vector<int>(m, INF));
    priority_queue<pair<int, pair<int, int>>, vector<pair<int, pair<int, int>>>, greater<>> pq;
    dist[sx][sy] = 0;
    pq.push({0, {sx, sy}});

    while (!pq.empty()) {
        auto [cost, pos] = pq.top(); pq.pop();
        int x = pos.first, y = pos.second;
        if (cost > dist[x][y]) continue;

        for (int d = 0; d < 4; ++d) {
            int nx = x + dx4[d], ny = y + dy4[d];
            if (nx >= 0 && nx < n && ny >= 0 && ny < m && grid[nx][ny] != '#') {
                int w = weight[nx][ny];
                if (dist[nx][ny] > dist[x][y] + w) {
                    dist[nx][ny] = dist[x][y] + w;
                    pq.push({dist[nx][ny], {nx, ny}});
                }
            }
        }
    }
}
```