Week 1: Basics & Implementation

Topics: - Input/Output, Loops, Conditionals - Arrays, Strings, Basic Math - Simple sorting

Weekly Tips: - Focus on writing clean, readable code. - Always test edge cases (0, 1, negative numbers, large numbers). - Use online judge IDE or local compiler to verify behavior.

Week 2: Ad-hoc & Simulation

Topics: - Simulation - Ad-hoc logic problems - Greedy basics

Weekly Tips: - Think step by step, simulate processes on paper first. - Carefully read problem constraints to optimize loops. - Greedy approach works if problem guarantees local optimality leads to global optimality.

Week 3: Sorting & Searching

Topics: - Sorting algorithms: QuickSort, MergeSort, STL sort - Binary Search & Ternary Search - Two-pointer technique

Weekly Tips: - Always check if STL sort suffices before implementing manually. - Binary search can be applied to sorted arrays or answer space. - Two-pointer technique is useful for finding pairs, sums, or sliding windows.

Week 4: Strings & Pattern Matching

Topics: - String searching: KMP, Rabin-Karp - Palindromes & substrings - Prefix/Suffix techniques

Weekly Tips: - Understand failure function in KMP for linear-time matching. - Use rolling hash for fast substring comparison. - Practice manipulating strings efficiently with STL.

Week 5: Recursion & Backtracking

Topics: - Recursion basics - Backtracking: N-Queens, subsets, combinations - Depth-First Search (DFS) for combinatorial problems

Weekly Tips: - Draw recursion trees to understand problem flow. - Watch stack usage and avoid unnecessary deep recursion. - Memoization can be applied to optimize repetitive recursive calls.

Week 6: Graph Theory Basics

Topics: - Graph representation: adjacency list & matrix - BFS & DFS traversal - Connected components - Shortest paths (Dijkstra, BFS for unweighted)

Weekly Tips: - Always check graph type: directed, undirected, weighted, unweighted. - Use visited array to avoid revisiting nodes. - For unweighted shortest paths, BFS is sufficient.

Problem 1: Counting Rooms

Link: [Kattis Counting Rooms] **Difficulty:** Beginner/Intermediate

C++ Solution with Explanation Comments:

```
#include <iostream>
#include <vector>
using namespace std;

int n, m;
vector<string> grid;
vector<vector<bool>> visited;

void dfs(int x, int y) {
    if (x < 0 || y < 0 || x >= n || y >= m) return;
    if (grid[x][y] == '#' || visited[x][y]) return;
    visited[x][y] = true;
    dfs(x+1, y);
    dfs(x-1, y);
    dfs(x, y+1);
    dfs(x, y-1);
}

int main() {
    cin >> n >> m;
    grid.resize(n);
    visited.assign(n, vector<bool>(m, false));
    for (int i = 0; i < n; i++) cin >> grid[i];

    int rooms = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (grid[i][j] == '.' && !visited[i][j]) {
                dfs(i,j);
                rooms++;
            }
```

```
        }
    }
    cout << rooms << endl;
    return 0;
}
```

Explanation Comments: - DFS traversal to mark connected '.' cells. - Increment `rooms` for each new unvisited component. - Classic connected components counting.

Problem 2: Shortest Reach

Link: [HackerRank BFS: Shortest Reach]() **Difficulty:** Intermediate

C++ Solution with Explanation Comments:

```cpp
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

int main() {
    int t; cin >> t;
    while (t--) {
        int n, m; cin >> n >> m;
        vector<vector<int>> adj(n+1);
        for (int i = 0; i < m; i++) {
            int u,v; cin >> u >> v;
            adj[u].push_back(v);
            adj[v].push_back(u);
        }
        int s; cin >> s;
        vector<int> dist(n+1, -1);
        queue<int> q;
        dist[s] = 0;
        q.push(s);
        while (!q.empty()) {
            int u = q.front(); q.pop();
            for (int v : adj[u]) {
                if (dist[v] == -1) {
                    dist[v] = dist[u] + 6;
                    q.push(v);
                }
            }
        }
        for (int i = 1; i <= n; i++) {
```

```
            if (i != s) cout << dist[i] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Explanation Comments: - BFS used to find shortest distance in unweighted graph. - Distance incremented by 6 per edge as per problem statement. - Queue ensures level-order traversal.

Problem 3: Flight Routes

Link: [CSES Flight Routes] **Difficulty:** Intermediate

C++ Solution with Explanation Comments:

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

int main() {
    int n, m; cin >> n >> m;
    vector<vector<pair<int,int>>> adj(n+1);
    for (int i = 0; i < m; i++) {
        int u,v,w; cin >> u >> v >> w;
        adj[u].push_back({v,w});
    }
    int start = 1;
    vector<long long> dist(n+1, 1e18);
    dist[start] = 0;
    priority_queue<pair<long long,int>, vector<pair<long long,int>>,
greater<pair<long long,int>>> pq;
    pq.push({0,start});
    while (!pq.empty()) {
        auto [d,u] = pq.top(); pq.pop();
        if (d != dist[u]) continue;
        for (auto [v,w] : adj[u]) {
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                pq.push({dist[v],v});
            }
        }
    }
    for (int i = 1; i <= n; i++) cout << dist[i] << " ";
```

```
    cout << endl;
    return 0;
}
```

Explanation Comments: - Implements Dijkstra's algorithm using priority queue. - Tracks shortest paths from starting node. - Efficiently handles weighted graphs.

Problem 4: Bipartite Check

Link: [Kattis Bipartite] **Difficulty:** Intermediate

C++ Solution with Explanation Comments:

```
#include <iostream>
#include <vector>
using namespace std;

bool dfs(int u, int c, vector<int>& color, vector<vector<int>>& adj) {
    color[u] = c;
    for (int v : adj[u]) {
        if (color[v] == 0) {
            if (!dfs(v, 3-c, color, adj)) return false;
        } else if (color[v] == c) return false;
    }
    return true;
}

int main() {
    int n, m; cin >> n >> m;
    vector<vector<int>> adj(n+1);
    for (int i = 0; i < m; i++) {
        int u,v; cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    vector<int> color(n+1,0);
    bool ok = true;
    for (int i = 1; i <= n; i++) {
        if (color[i] == 0 && !dfs(i,1,color,adj)) { ok = false; break; }
    }
    cout << (ok ? "Bipartite" : "Not Bipartite") << endl;
    return 0;
}
```

Explanation Comments: - DFS-based coloring to check bipartite property. - Assigns alternating colors; conflict indicates non-bipartite. - Demonstrates recursive traversal with state tracking.

End of Week 6 - Practice BFS/DFS on grids and graphs. - Learn to distinguish when BFS or DFS is more suitable. - Focus on shortest path, connected components, and bipartite checking.