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.

Week 7: Dynamic Programming (DP)

Topics: - Introduction to DP: memoization & tabulation - Classic problems: Fibonacci, Knapsack, LIS - Grid DP, state compression

Weekly Tips: - Identify overlapping subproblems and optimal substructure. - Start with recursive solution, then memoize or tabulate. - Practice simple to complex DP to build intuition.

Week 8: Advanced Graph Algorithms

Topics: - Minimum Spanning Trees: Prim, Kruskal - Bellman-Ford for negative weights - Floyd-Warshall for all-pairs shortest paths - Strongly Connected Components (Kosaraju, Tarjan)

Weekly Tips: - MST: Focus on edge selection and cycle prevention. - Bellman-Ford: Detect negative cycles. - Floyd-Warshall: Use DP-like approach for all-pairs shortest path. - SCC: Identify components and condensation graph.

Week 9: Greedy & Interval Problems

Topics: - Activity Selection Problem - Interval Scheduling - Interval Covering - Fractional Knapsack

Weekly Tips: - Always sort intervals by finishing time for scheduling problems. - Greedy works when local optimum leads to global optimum. - Pay attention to edge cases where intervals overlap. - Fractional Knapsack can be solved using sorting by value/weight ratio.

Week 10: Advanced Dynamic Programming

Topics: - DP on Trees - Bitmask DP - Sequence DP with constraints (e.g., subsequences, partitions) - Optimization techniques: prefix sums, cumulative DP

Weekly Tips: - DP on trees: use DFS and store DP for subtrees. - Bitmask DP: useful for problems with small n (<=20) subsets. - Sequence DP: carefully define states and transitions. - Optimize using cumulative sums, monotonic queues when possible.

Week 11: Network Flow & Matching

Topics: - Max Flow (Ford-Fulkerson, Edmonds-Karp) - Min Cut - Bipartite Matching (Hungarian Algorithm, Hopcroft-Karp) - Flow-based problem solving

Weekly Tips: - Max Flow: Understand residual graph and augmenting paths. - Min Cut: Relates to Max Flow by MFMC theorem. - Bipartite Matching: Use flow or DFS-based approaches. - Practice transforming problems into flow networks.

Problem 1: Max Flow (Edmonds-Karp)

Link: CSES Flight Routes **Difficulty:** Advanced

C++ Solution with Explanation Comments:

```cpp
#include <bits/stdc++.h>
using namespace std;
const int INF = 1e9;
int main(){
    int n,m; cin>>n>>m;
    vector<vector<int>> capacity(n,vector<int>(n,0));
    vector<vector<int>> adj(n);
    for(int i=0;i<m;i++){
        int u,v,c; cin>>u>>v>>c; u--;v--;
        capacity[u][v]+=c; // handle multiple edges
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    int s=0,t=n-1;
    int flow=0;
    while(true){
        vector<int> parent(n,-1); parent[s]=-2;
        queue<pair<int,int>> q; q.push({s,INF});
        while(!q.empty() && parent[t]==-1){
            auto [cur,f] = q.front(); q.pop();
            for(int next:adj[cur]){
                if(parent[next]==-1 && capacity[cur][next]>0){
                    parent[next]=cur;
                    int new_f = min(f, capacity[cur][next]);
                    if(next==t){ f=new_f; break; }
                    q.push({next,new_f});
                }
            }
        }
        if(parent[t]==-1) break;
```

```
        flow += f;
        int cur=t;
        while(cur!=s){
            int prev=parent[cur];
            capacity[prev][cur]-=f;
            capacity[cur][prev]+=f;
            cur=prev;
        }
    }
    cout<<flow<<endl;
}
```

Explanation Comments: - BFS-based Edmonds-Karp to find augmenting paths. - Update residual capacities after each path. - Total flow accumulates until no path exists.

Problem 2: Bipartite Matching (Hopcroft-Karp)

Link: CSES Room Assignments **Difficulty:** Advanced

C++ Solution with Explanation Comments:

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    int n,m; cin>>n>>m;
    vector<vector<int>> adj(n);
    for(int i=0;i<m;i++){ int u,v; cin>>u>>v; u--;v--; adj[u].push_back(v); }
    vector<int> match(m,-1);
    int result=0;
    function<bool(int,vector<bool>&)> dfs = [&](int u, vector<bool> &visited){
        for(int v:adj[u]){
            if(!visited[v]){
                visited[v]=true;
                if(match[v]==-1 || dfs(match[v],visited)){
                    match[v]=u; return true;
                }
            }
        }
        return false;
    };
    for(int u=0;u<n;u++){
        vector<bool> visited(m,false);
        if(dfs(u,visited)) result++;
    }
```

```
    cout<<result<<endl;
}
```

Explanation Comments: - DFS-based maximum bipartite matching. - Try to find augmenting path for each unmatched node. - Efficient for standard bipartite matching problems.

End of Week 11 - Understand residual graphs for max flow. - Convert matching or allocation problems into flow networks. - Practice small to large networks for efficiency understanding.