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.

Problem 1: Longest Increasing Subsequence (LIS)

Link: UVa 231 **Difficulty:** Intermediate

C++ Solution with Explanation Comments:

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

int main() {
    int n;
    while (cin >> n) {
        vector<int> seq(n);
        for (int i = 0; i < n; i++) cin >> seq[i];
        vector<int> dp(n,1);
        int ans = 1;
        for (int i = 1; i < n; i++) {
            for (int j = 0; j < i; j++) {
                if (seq[i] > seq[j]) dp[i] = max(dp[i], dp[j]+1);
            }
            ans = max(ans, dp[i]);
        }
        cout << ans << endl;
    }
    return 0;
}
```

Explanation Comments: - DP array `dp[i]` stores LIS ending at index i. - Check all previous elements for increasing sequence. - Classic O(n^2) LIS DP solution.

Problem 2: 0-1 Knapsack

Link: UVa 624 **Difficulty:** Intermediate

C++ Solution with Explanation Comments:

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

int main() {
    int W, n;
    while (cin >> W >> n) {
        vector<int> w(n), v(n);
        for (int i = 0; i < n; i++) cin >> w[i] >> v[i];
        vector<int> dp(W+1, 0);
        for (int i = 0; i < n; i++) {
            for (int j = W; j >= w[i]; j--) {
                dp[j] = max(dp[j], dp[j-w[i]] + v[i]);
            }
        }
        cout << dp[W] << endl;
    }
    return 0;
}
```

Explanation Comments: - DP array `dp[j]` stores max value with weight j. - Iterate backward to avoid using same item twice. - Demonstrates standard 0-1 Knapsack using space-optimized DP.

Problem 3: Minimum Path Sum in Grid

Link: LeetCode 64 **Difficulty:** Intermediate

C++ Solution with Explanation Comments:

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

int main() {
```

```
    int n, m; cin >> n >> m;
    vector<vector<int>> grid(n, vector<int>(m));
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            cin >> grid[i][j];
    vector<vector<int>> dp(n, vector<int>(m,0));
    dp[0][0] = grid[0][0];
    for (int i = 1; i < n; i++) dp[i][0] = dp[i-1][0] + grid[i][0];
    for (int j = 1; j < m; j++) dp[0][j] = dp[0][j-1] + grid[0][j];
    for (int i = 1; i < n; i++) {
        for (int j = 1; j < m; j++) {
            dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j];
        }
    }
    cout << dp[n-1][m-1] << endl;
    return 0;
}
```

Explanation Comments: - DP for grid: min path sum from top-left to bottom-right. - Transition: `dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j]`. - Demonstrates tabulation and grid-based DP.

Problem 4: Coin Change

Link: CSES Coin Combinations **Difficulty:** Intermediate

C++ Solution with Explanation Comments:

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

int main() {
    int n, x; cin >> n >> x;
    vector<int> coins(n);
    for (int i = 0; i < n; i++) cin >> coins[i];
    vector<long long> dp(x+1,0);
    dp[0] = 1;
    for (int i = 0; i < n; i++) {
        for (int j = coins[i]; j <= x; j++) {
            dp[j] += dp[j - coins[i]];
        }
    }
    cout << dp[x] << endl;
```

```
    return 0;
}
```

Explanation Comments: - `dp[j]` stores number of ways to make sum j. - Iterate over coins to update combination counts. - Classic unbounded knapsack problem.

End of Week 7 - Focus on understanding DP state definition. - Practice memoization and tabulation techniques. - Gradually move from 1D to 2D DP and more complex states.