Bitmask_DP.cpp

```cpp
/**
 *    Author: devesh95
 *
 *    Topic: Bitmask DP Examples (Easy to Hard)
 *
 *    Description:
 *    This file contains 20 different dynamic programming problems solved using bitmasking.
 *    The examples are arranged roughly from easier (simple subset enumeration) to
 *    harder (assignment, TSP, graph problems, etc.). Each example includes detailed
 *    comments explaining the DP state, recurrence (DP equation), and input/output processing.
 *
 *    Compilation:
 *         g++ -std=c++17 -O2 -Wall Bitmask_DP_Examples.cpp -o bitmask_dp
 *
 *    Execution:
 *         ./bitmask_dp
 */

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

#define int long long
#define INF 1000000000  // Use a large value

// ---------------------------------------------------------------------
// 1. Enumerate All Subsets
// ---------------------------------------------------------------------
/*
   Problem:
     Given n, enumerate all 2^n subsets.

   Explanation:
     This example does not use a DP recurrence but uses bitmask enumeration.

   (No DP equation)
*/
void solve_enumerate_subsets() {
    cout << "\n----- 1. Enumerate All Subsets -----\n";
    int n;
    cout << "Enter n (number of elements): ";
    cin >> n;
    cout << "All subsets (each as bitmask):\n";
    int total = 1 << n;
    for (int mask = 0; mask < total; mask++) {
        // Print only n bits
        string bits = bitset<16>(mask).to_string().substr(16 - n);
        cout << bits << "\n";
    }
}

// ---------------------------------------------------------------------
// 2. Sum of All Subsets
// --------------------------------------------------------------------
/*
   Problem:
     Given an array of n numbers, compute and display the sum of each subset.

   Explanation:
     We enumerate each subset and compute its sum.

   (No DP equation)
*/
void solve_sum_of_subsets() {
    cout << "\n----- 2. Sum of All Subsets -----\n";
    int n;
    cout << "Enter n (number of elements): ";
    cin >> n;
    vector<int> arr(n);
    cout << "Enter " << n << " numbers:\n";
    for (int i = 0; i < n; i++) cin >> arr[i];

    int total = 1 << n;
    for (int mask = 0; mask < total; mask++) {
        int sum = 0;
        cout << "Subset (mask " << bitset<16>(mask).to_string().substr(16 - n) << "): ";
        for (int i = 0; i < n; i++) {
            if(mask & (1 << i)){
                sum += arr[i];
                cout << arr[i] << " ";
            }
        }
        cout << "=> Sum: " << sum << "\n";
    }
}

// --------------------------------------------------------------------
// 3. Count Subsets with Given Sum
// --------------------------------------------------------------------
/*
   Problem:
     Given an array and a target sum S, count the number of subsets that sum to S.

   DP Equation / Recurrence:
     (Brute-force via bitmask enumeration; no memoized recurrence)

     For each subset represented by mask, compute:
         if (sum(mask) == S) then count++
*/
void solve_count_subsets_with_sum() {
    cout << "\n----- 3. Count Subsets with Given Sum -----\n";
    int n, S;
    cout << "Enter n (number of elements) and target sum S: ";
    cin >> n >> S;
    vector<int> arr(n);
    cout << "Enter " << n << " numbers:\n";
    for (int i = 0; i < n; i++) cin >> arr[i];

    int total = 1 << n;
    int count = 0;
    for (int mask = 0; mask < total; mask++) {
        int sum = 0;
        for (int i = 0; i < n; i++) {
            if(mask & (1 << i))
                sum += arr[i];
        }
        if(sum == S) count++;
    }
    cout << "Number of subsets with sum " << S << ": " << count << "\n";
}

// --------------------------------------------------------------------
// 4. Maximum Sum Subset
// --------------------------------------------------------------------
/*
   Problem:
     Given an array of non-negative numbers, find the subset with the maximum sum.

   Explanation:
     (Trivially, the maximum sum is the sum of all elements if all are non-negative.)

   (No DP equation; simple enumeration.)
*/
void solve_max_sum_subset() {
    cout << "\n----- 4. Maximum Sum Subset -----\n";
    int n;
    cout << "Enter n (number of elements): ";
    cin >> n;
    vector<int> arr(n);
    cout << "Enter " << n << " non-negative numbers:\n";
    for (int i = 0; i < n; i++) cin >> arr[i];

    int total = 1 << n;
    int maxSum = 0;
    for (int mask = 0; mask < total; mask++) {
        int sum = 0;
        for (int i = 0; i < n; i++) {
            if(mask & (1 << i))
                sum += arr[i];
        }
        maxSum = max(maxSum, sum);
    }
    cout << "Maximum subset sum: " << maxSum << "\n";
}

// --------------------------------------------------------------------
// 5. Assignment Problem (Minimum Cost Matching)
// --------------------------------------------------------------------
/*
   Problem:
     Given an n x n cost matrix, assign each job to a worker so that the total cost is minimized.

   DP Equation:
     Let dp[mask] be the minimum cost when jobs represented by mask are assigned.
     Transition:
         dp[mask | (1<<j)] = min(dp[mask | (1<<j)], dp[mask] + cost[popcount(mask)][j])
*/
void solve_assignment_problem() {
    cout << "\n----- 5. Assignment Problem (Min Cost Matching) -----\n";
    int n;
    cout << "Enter n (number of jobs/workers): ";
    cin >> n;
    vector<vector<int>> cost(n, vector<int>(n));
    cout << "Enter the cost matrix (n x n):\n";
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            cin >> cost[i][j];

    int N = 1 << n;
    vector<int> dp(N, INT_MAX);
    dp[0] = 0;
    for (int mask = 0; mask < N; mask++) {
        int i = __builtin_popcount(mask); // number of jobs assigned so far
        for (int j = 0; j < n; j++) {
            if (!(mask & (1 << j))) {
                dp[mask | (1 << j)] = min(dp[mask | (1 << j)], dp[mask] + cost[i][j]);
            }
        }
    }
    cout << "Minimum assignment cost: " << dp[N - 1] << "\n";
}

// --------------------------------------------------------------------
// 6. Traveling Salesman Problem (TSP)
// --------------------------------------------------------------------
/*
   Problem:
     Given n cities and a cost (distance) matrix, find the minimum cost to visit all cities
     starting from 0 and returning to 0.

   DP Equation:
     Let dp[mask][i] be the minimum cost to reach city i with visited set = mask.
     Transition:
         dp[mask | (1<<j)][j] = min(dp[mask | (1<<j)][j], dp[mask][i] + dist[i][j])
*/
void solve_tsp() {
    cout << "\n----- 6. Traveling Salesman Problem (TSP) -----\n";
    int n;
    cout << "Enter number of cities: ";
    cin >> n;
    vector<vector<int>> dist(n, vector<int>(n));
    cout << "Enter the distance matrix:\n";
    for (int i = 0; i < n; i++)
```

```cpp
        for (int j = 0; j < n; j++)
            cin >> dist[i][j];

    int N = 1 << n;
    vector<vector<int>> dp(N, vector<int>(n, INT_MAX));
    dp[1][0] = 0; // Starting at city 0 (mask 1 means only city0 visited)
    for (int mask = 1; mask < N; mask++) {
        for (int i = 0; i < n; i++) {
            if (mask & (1 << i)) {
                for (int j = 0; j < n; j++) {
                    if (!(mask & (1 << j)) && dist[i][j] < INT_MAX) {
                        dp[mask | (1 << j)][j] = min(dp[mask | (1 << j)][j], dp[mask][i] + dist[i][j]);
                    }
                }
            }
        }
    }
    int ans = INT_MAX;
    for (int i = 0; i < n; i++) {
        ans = min(ans, dp[N - 1][i] + dist[i][0]);
    }
    cout << "Minimum TSP cost: " << ans << "\n";
}

// -------------------------------------------------------------------
// 7. Counting Hamiltonian Paths in a DAG
// -------------------------------------------------------------------
/*
   Problem:
     Given a directed acyclic graph (DAG) with n nodes, count the number of Hamiltonian paths.

   DP Equation:
     Let dp[mask][i] be the number of ways to reach node i having visited nodes in mask.
     Transition:
         dp[mask | (1<<v)][v] += dp[mask][u], for each edge (u -> v) where v is not in mask.
*/
void solve_count_hamiltonian_paths() {
    cout << "\n----- 7. Counting Hamiltonian Paths in a DAG -----\n";
    int n, m;
    cout << "Enter number of nodes and edges: ";
    cin >> n >> m;
    vector<vector<int>> graph(n);
    cout << "Enter directed edges (u v) (0-indexed):\n";
    for (int i = 0; i < m; i++){
        int u, v;
        cin >> u >> v;
        graph[u].push_back(v);
    }
    int N = 1 << n;
    vector<vector<int>> dp(N, vector<int>(n, 0));
    for (int i = 0; i < n; i++)
        dp[1 << i][i] = 1;
    for (int mask = 0; mask < N; mask++){
        for (int u = 0; u < n; u++){
            if(mask & (1 << u)){
                for (int v : graph[u]){
                    if(!(mask & (1 << v))){
                        dp[mask | (1 << v)][v] += dp[mask][u];
                    }
                }
            }
        }
    }
    int total = 0;
    for (int i = 0; i < n; i++){
        total += dp[N - 1][i];
    }
    cout << "Total Hamiltonian paths in the DAG: " << total << "\n";
}

// --------------------------------------------------------------------
// 8. Maximum Independent Set (Graph)
// --------------------------------------------------------------------
/*
   Problem:
     Given an undirected graph with n vertices (n small), find the size of the maximum independent set.

   Explanation:
     Enumerate all subsets and check if the subset forms an independent set.

   (No DP recurrence; brute-force bitmask enumeration)
*/
void solve_max_independent_set() {
    cout << "\n----- 8. Maximum Independent Set -----\n";
    int n, m;
    cout << "Enter number of vertices and edges: ";
    cin >> n >> m;
    vector<vector<bool>> adj(n, vector<bool>(n, false));
    cout << "Enter " << m << " edges (u v) (0-indexed):\n";
    for (int i = 0; i < m; i++){
        int u, v;
        cin >> u >> v;
        adj[u][v] = adj[v][u] = true;
    }
    int N = 1 << n;
    int maxSize = 0;
    for (int mask = 0; mask < N; mask++){
        bool valid = true;
        int count = 0;
        for (int i = 0; i < n && valid; i++){
            if(mask & (1 << i)){
                count++;
                for (int j = i+1; j < n; j++){
                    if(mask & (1 << j)){
                        if(adj[i][j]){
                            valid = false;
                            break;
                        }
                    }
                }
            }
        }
        if(valid)
            maxSize = max(maxSize, count);
    }
    cout << "Size of maximum independent set: " << maxSize << "\n";
}

// ---------------------------------------------------------------------
// 9. Maximum Clique (Graph)
// ---------------------------------------------------------------------
/*
   Problem:
     Given an undirected graph with n vertices (n small), find the size of the maximum clique.

   Explanation:
     Enumerate all subsets and check if they form a clique.

   (No DP recurrence; brute-force enumeration)
*/
void solve_max_clique() {
    cout << "\n----- 9. Maximum Clique -----\n";
    int n, m;
    cout << "Enter number of vertices and edges: ";
    cin >> n >> m;
    vector<vector<bool>> adj(n, vector<bool>(n, false));
    // Mark self-loops for convenience.
    for (int i = 0; i < n; i++) adj[i][i] = true;
    cout << "Enter " << m << " edges (u v) (0-indexed):\n";
    for (int i = 0; i < m; i++){
        int u, v;
        cin >> u >> v;
        adj[u][v] = adj[v][u] = true;
    }
    int N = 1 << n;
    int maxClique = 0;
    for (int mask = 0; mask < N; mask++){
        vector<int> nodes;
        bool clique = true;
        for (int i = 0; i < n; i++){
            if(mask & (1 << i))
                nodes.push_back(i);
        }
        for (int i = 0; i < (int)nodes.size() && clique; i++){
            for (int j = i+1; j < (int)nodes.size(); j++){
                if(!adj[nodes[i]][nodes[j]]){
                    clique = false;
                    break;
                }
            }
        }
        if(clique)
            maxClique = max(maxClique, (int)nodes.size());
    }
    cout << "Size of maximum clique: " << maxClique << "\n";
}

// ----------------------------------------------------------------------
// 10. Minimum Vertex Cover (Graph)
// ----------------------------------------------------------------------
/*
   Problem:
     Given an undirected graph with n vertices (n small), find the size of the minimum vertex cover.

   Explanation:
     A vertex cover is a set of vertices such that every edge is incident to at least one vertex in the set.
     (Brute-force enumeration via bitmask)

   (No DP recurrence; relation: |MIS| + |MinVertexCover| = n)
*/
void solve_min_vertex_cover() {
    cout << "\n----- 10. Minimum Vertex Cover -----\n";
    int n, m;
    cout << "Enter number of vertices and edges: ";
    cin >> n >> m;
    vector<vector<bool>> adj(n, vector<bool>(n, false));
    cout << "Enter " << m << " edges (u v) (0-indexed):\n";
    for (int i = 0; i < m; i++){
        int u, v;
        cin >> u >> v;
        adj[u][v] = adj[v][u] = true;
    }
    int N = 1 << n;
    int minCover = n;
    for (int mask = 0; mask < N; mask++){
        bool cover = true;
        for (int u = 0; u < n && cover; u++){
            for (int v = u+1; v < n && cover; v++){
                if(adj[u][v]){
                    // At least one of u or v must be in the cover.
                    if (!(mask & (1 << u)) && !(mask & (1 << v)))
                        cover = false;
                }
            }
        }
        if(cover){
            int cnt = __builtin_popcount(mask);
            minCover = min(minCover, cnt);
        }
    }
```

```cpp
    cout << "Minimum vertex cover size: " << minCover << "\n";
}

// --------------------------------------------------------------------
// 11. Set Cover Problem
// --------------------------------------------------------------------
/*
   Problem:
     Given a universe of m elements and n subsets (each represented as a bitmask),
     find the minimum number of subsets required to cover the entire universe.

   Explanation:
     We enumerate over all subset selections and check if their union covers the universe.

   (No DP recurrence; brute-force enumeration)
*/
void solve_set_cover() {
    cout << "\n----- 11. Set Cover Problem -----\n";
    int m, n;
    cout << "Enter size of universe (m) and number of subsets (n): ";
    cin >> m >> n;
    vector<int> subsets(n);
    cout << "Enter each subset as a bitmask (integer between 0 and " << ((1 << m) - 1) << "):\n";
    for (int i = 0; i < n; i++){
        cin >> subsets[i];
    }
    int full = (1 << m) - 1;
    int N = 1 << n;
    int ans = INT_MAX;
    for (int mask = 0; mask < N; mask++){
        int cover = 0, count = 0;
        for (int i = 0; i < n; i++){
            if(mask & (1 << i)){
                cover |= subsets[i];
                count++;
            }
        }
        if(cover == full)
            ans = min(ans, count);
    }
    if(ans == INT_MAX)
        cout << "No cover found.\n";
    else
        cout << "Minimum number of subsets to cover the universe: " << ans << "\n";
}

// --------------------------------------------------------------------
// 12. Count Perfect Matchings in a Bipartite Graph
// --------------------------------------------------------------------
/*
   Problem:
     Given a bipartite graph with n workers and n jobs, count the number of perfect matchings.

   DP Equation:
     Let dp[mask] be the number of ways to assign jobs corresponding to the bitmask.
     Transition:
         dp[mask | (1 << j)] += dp[mask] for each unassigned job j that can be matched.
*/
void solve_count_perfect_matchings() {
    cout << "\n----- 12. Count Perfect Matchings in a Bipartite Graph -----\n";
    int n;
    cout << "Enter n (number of workers/jobs): ";
    cin >> n;
    vector<vector<bool>> adj(n, vector<bool>(n, false));
    cout << "Enter the " << n << "x" << n << " bipartite adjacency matrix (0/1):\n";
    for (int i = 0; i < n; i++){
        for (int j = 0; j < n; j++){
            int temp;
            cin >> temp;
            adj[i][j] = (temp == 1);
        }
    }
    int N = 1 << n;
    vector<int> dp(N, 0);
    dp[0] = 1;
    for (int mask = 0; mask < N; mask++){
        int i = __builtin_popcount(mask);
        for (int j = 0; j < n; j++){
            if(!(mask & (1 << j)) && adj[i][j]){
                dp[mask | (1 << j)] += dp[mask];
            }
        }
    }
    cout << "Number of perfect matchings: " << dp[N - 1] << "\n";
}

// -------------------------------------------------------------------
// 13. Partition into Two Subsets with Minimum Difference
// -------------------------------------------------------------------
/*
   Problem:
     Given an array, partition it into two subsets so that the difference of their sums is minimized.

   Explanation:
     Enumerate all subsets to determine one subset sum and use total sum to compute difference.

   (No DP recurrence; brute-force enumeration)
*/
void solve_partition_min_difference() {
    cout << "\n----- 13. Partition into Two Subsets (Min Difference) -----\n";
    int n;
    cout << "Enter n (number of elements): ";
    cin >> n;
    vector<int> arr(n);
    int total = 0;
    cout << "Enter the elements:\n";
    for (int i = 0; i < n; i++){
        cin >> arr[i];
        total += arr[i];
    }
    int N = 1 << n;
    int best = INT_MAX;
    for (int mask = 0; mask < N; mask++){
        int sum = 0;
        for (int i = 0; i < n; i++){
            if(mask & (1 << i))
                sum += arr[i];
        }
        best = min(best, (int)abs(total - 2 * sum));
    }
    cout << "Minimum difference between two subsets: " << best << "\n";
}

// ----------------------------------------------------------------------
// 14. Team Formation (Divide into Two Teams Minimizing Difference)
// ----------------------------------------------------------------------
/*
   Problem:
     Given an even number of players with skill levels, split them into two teams (each with n/2 players)
     such that the difference in total skills is minimized.

   Explanation:
     Enumerate over all bitmasks with exactly n/2 bits set.

   (No DP recurrence; brute-force enumeration)
*/
void solve_team_formation() {
    cout << "\n----- 14. Team Formation (Equal Teams) -----\n";
    int n;
    cout << "Enter even n (number of players): ";
    cin >> n;
    if(n % 2 != 0) {
        cout << "n must be even.\n";
        return;
    }
    vector<int> skill(n);
    cout << "Enter skill values:\n";
    for (int i = 0; i < n; i++) cin >> skill[i];

    int N = 1 << n;
    int half = n / 2;
    int best = INT_MAX;
    for (int mask = 0; mask < N; mask++){
        if(__builtin_popcount(mask) == half){
            int sum1 = 0;
            for (int i = 0; i < n; i++){
                if(mask & (1 << i))
```

```cpp
                    sum1 += skill[i];
            }
            int sum2 = accumulate(skill.begin(), skill.end(), 0LL) - sum1;
            best = min(best, (int)abs(sum1 - sum2));
        }
    }
    cout << "Minimum skill difference between two teams: " << best << "\n";
}

// --------------------------------------------------------------------
// 15. Count Subset Sum Ways (Alternate Counting)
// --------------------------------------------------------------------
/*
   Problem:
     Count the number of ways to choose a subset from an array that sums to a given target S.

   Explanation:
     This is similar to example 3 but explicitly counts the ways.

   (No DP recurrence; brute-force bitmask enumeration)
*/
void solve_count_subset_sum_ways() {
    cout << "\n----- 15. Count Subset Sum Ways -----\n";
    int n, S;
    cout << "Enter n (number of elements) and target sum S: ";
    cin >> n >> S;
    vector<int> arr(n);
    cout << "Enter the elements:\n";
    for (int i = 0; i < n; i++) cin >> arr[i];

    int N = 1 << n;
    int ways = 0;
    for (int mask = 0; mask < N; mask++){
        int sum = 0;
        for (int i = 0; i < n; i++){
            if(mask & (1 << i))
                sum += arr[i];
        }
        if(sum == S) ways++;
    }
    cout << "Number of ways to achieve sum " << S << ": " << ways << "\n";
}

// --------------------------------------------------------------------
// 16. Longest Hamiltonian Path (Maximizing Weight) [General Graph]
// --------------------------------------------------------------------
/*
   Problem:
     Given a weighted complete graph with n vertices, find the maximum total weight
     path that visits every vertex exactly once (does not need to return to the start).

   DP Equation:
     Let dp[mask][v] be the maximum weight path ending at vertex v covering vertices in mask.
     Transition:
         dp[mask | (1<<u)][u] = max(dp[mask | (1<<u)][u], dp[mask][v] + weight[v][u])
*/
void solve_longest_path_bitmask() {
    cout << "\n----- 16. Longest Hamiltonian Path -----\n";
    int n;
    cout << "Enter number of vertices: ";
    cin >> n;
    vector<vector<int>> weight(n, vector<int>(n));
    cout << "Enter the weight matrix:\n";
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            cin >> weight[i][j];

    int N = 1 << n;
    vector<vector<int>> dp(N, vector<int>(n, 0));
    // Initialize: single vertex path has weight 0.
    for (int i = 0; i < n; i++)
        dp[1 << i][i] = 0;

    int ans = 0;
    for (int mask = 0; mask < N; mask++){
        for (int u = 0; u < n; u++){
            if(mask & (1 << u)){
                for (int v = 0; v < n; v++){
                    if(!(mask & (1 << v))){
                        dp[mask | (1 << v)][v] = max(dp[mask | (1 << v)][v], dp[mask][u] + weight[u][v]);
                        ans = max(ans, dp[mask | (1 << v)][v]);
                    }
                }
            }
        }
    }
    cout << "Maximum weight of a Hamiltonian path: " << ans << "\n";
}

// ----------------------------------------------------------------------
// 17. Count Independent Sets in a Graph
// ----------------------------------------------------------------------
/*
   Problem:
     Given an undirected graph with n vertices (n small), count the total number of independent sets.

   Explanation:
     Enumerate all subsets and count those that are independent.

   (No DP recurrence; brute-force enumeration)
*/
void solve_count_independent_sets() {
    cout << "\n----- 17. Count Independent Sets -----\n";
    int n, m;
    cout << "Enter number of vertices and edges: ";
    cin >> n >> m;
    vector<vector<bool>> adj(n, vector<bool>(n, false));
    cout << "Enter " << m << " edges (u v) (0-indexed):\n";
    for (int i = 0; i < m; i++){
        int u, v;
        cin >> u >> v;
        adj[u][v] = adj[v][u] = true;
    }
    int N = 1 << n;
    int count = 0;
    for (int mask = 0; mask < N; mask++){
        bool independent = true;
        for (int i = 0; i < n && independent; i++){
            if(mask & (1 << i)){
                for (int j = i+1; j < n; j++){
                    if(mask & (1 << j)){
                        if(adj[i][j]){
                            independent = false;
                            break;
                        }
                    }
                }
            }
        }
        if(independent) count++;
    }
    cout << "Total number of independent sets: " << count << "\n";
}

// --------------------------------------------------------------------
// 18. Minimum Dominating Set (Graph)
// --------------------------------------------------------------------
/*
   Problem:
     A dominating set of a graph is a set of vertices such that every vertex is either in the set
     or adjacent to a vertex in the set. Find the size of the minimum dominating set.

   Explanation:
     Enumerate all subsets; for each, check if it is a dominating set.

   (No DP recurrence; brute-force enumeration)
*/
void solve_min_dominating_set() {
    cout << "\n----- 18. Minimum Dominating Set -----\n";
    int n, m;
    cout << "Enter number of vertices and edges: ";
    cin >> n >> m;
    vector<vector<bool>> adj(n, vector<bool>(n, false));
    // Each vertex dominates itself.
    for (int i = 0; i < n; i++) adj[i][i] = true;
    cout << "Enter " << m << " edges (u v) (0-indexed):\n";
    for (int i = 0; i < m; i++){
        int u, v;
        cin >> u >> v;
        adj[u][v] = adj[v][u] = true;
    }
    int N = 1 << n;
    int ans = n;
    for (int mask = 0; mask < N; mask++){
        vector<bool> dominated(n, false);
        for (int i = 0; i < n; i++){
            if(mask & (1 << i)){
                for (int j = 0; j < n; j++){
                    if(adj[i][j])
                        dominated[j] = true;
                }
            }
        }
        bool valid = true;
        for (int i = 0; i < n; i++){
            if(!dominated[i]) { valid = false; break; }
        }
        if(valid)
            ans = min(ans, (int)__builtin_popcount(mask));
    }
    cout << "Minimum dominating set size: " << ans << "\n";
}

// ---------------------------------------------------------------------
// 19. Task Ordering with Prerequisites
// ---------------------------------------------------------------------
/*
   Problem:
     Given n tasks with prerequisites (each task i has a bitmask pre[i] that indicates
     which tasks must be completed before i), count the number of valid orderings.

   DP Equation:
     Let dp[mask] be the number of valid orderings for tasks in mask.
     Transition:
         For each task i not in mask, if (mask & pre[i] == pre[i]), then:
         dp[mask | (1 << i)] += dp[mask]
*/
void solve_task_ordering() {
    cout << "\n----- 19. Task Ordering with Prerequisites -----\n";
    int n;
    cout << "Enter number of tasks: ";
    cin >> n;
    vector<int> pre(n, 0);
    cout << "For each task i (0-indexed), enter a bitmask (as integer) representing prerequisites:\n";
    cout << "(For example, if task 2 requires tasks 0 and 1, enter 3 (binary 11))\n";
    for (int i = 0; i < n; i++){
        cout << "Prerequisites for task " << i << ": ";
        cin >> pre[i];
    }
    int N = 1 << n;
    vector<int> dp(N, 0);
    dp[0] = 1;
    for (int mask = 0; mask < N; mask++){
        for (int i = 0; i < n; i++){
            if(!(mask & (1 << i)) && ((mask & pre[i]) == pre[i])){
                dp[mask | (1 << i)] += dp[mask];
            }
        }
    }
    cout << "Total number of valid orderings: " << dp[N - 1] << "\n";
}

// --------------------------------------------------------------------
// 20. Maximum XOR Subset (Bitmask Enumeration)
// --------------------------------------------------------------------
/*
   Problem:
     Given an array of integers, find the maximum XOR value obtainable from any subset.

   Explanation:
     Enumerate all subsets and compute the XOR value.

   (No DP recurrence; brute-force enumeration)
*/
void solve_max_xor_subset() {
    cout << "\n----- 20. Maximum XOR Subset -----\n";
    int n;
    cout << "Enter n (number of elements): ";
    cin >> n;
    vector<int> arr(n);
    cout << "Enter the elements:\n";
    for (int i = 0; i < n; i++) cin >> arr[i];

    int N = 1 << n;
    int maxXor = 0;
    for (int mask = 0; mask < N; mask++){
        int curXor = 0;
        for (int i = 0; i < n; i++){
            if(mask & (1 << i))
                curXor ^= arr[i];
        }
        maxXor = max(maxXor, curXor);
    }
    cout << "Maximum XOR value from any subset: " << maxXor << "\n";
}

// --------------------------------------------------------------------
// Main Menu
// --------------------------------------------------------------------
int32_t main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);

    while(true) {
        cout << "\n========================================================\n";
        cout << "           Bitmask DP Examples - Menu\n";
        cout << "========================================================\n";
        cout << " 1.  Enumerate All Subsets\n";
        cout << " 2.  Sum of All Subsets\n";
        cout << " 3.  Count Subsets with Given Sum\n";
        cout << " 4.  Maximum Sum Subset\n";
        cout << " 5.  Assignment Problem (Min Cost Matching)\n";
        cout << " 6.  Traveling Salesman Problem (TSP)\n";
        cout << " 7.  Counting Hamiltonian Paths in a DAG\n";
        cout << " 8.  Maximum Independent Set (Graph)\n";
        cout << " 9.  Maximum Clique (Graph)\n";
        cout << "10.  Minimum Vertex Cover (Graph)\n";
        cout << "11.  Set Cover Problem\n";
        cout << "12.  Count Perfect Matchings in Bipartite Graph\n";
        cout << "13.  Partition into Two Subsets (Min Difference)\n";
        cout << "14.  Team Formation (Equal Teams)\n";
        cout << "15.  Count Subset Sum Ways\n";
        cout << "16.  Longest Hamiltonian Path (Max Weight)\n";
        cout << "17.  Count Independent Sets (Graph)\n";
        cout << "18.  Minimum Dominating Set (Graph)\n";
        cout << "19.  Task Ordering with Prerequisites\n";
        cout << "20.  Maximum XOR Subset\n";
        cout << "21.  Run All Examples\n";
        cout << "0.  Exit\n";
        cout << "Enter your choice: ";

        int choice;
        cin >> choice;
        if(choice == 0) break;

        switch(choice) {
            case 1:  solve_enumerate_subsets(); break;
            case 2:  solve_sum_of_subsets(); break;
            case 3:  solve_count_subsets_with_sum(); break;
            case 4:  solve_max_sum_subset(); break;
            case 5:  solve_assignment_problem(); break;
            case 6:  solve_tsp(); break;
            case 7:  solve_count_hamiltonian_paths(); break;
            case 8:  solve_max_independent_set(); break;
            case 9:  solve_max_clique(); break;
            case 10: solve_min_vertex_cover(); break;
            case 11: solve_set_cover(); break;
            case 12: solve_count_perfect_matchings(); break;
            case 13: solve_partition_min_difference(); break;
            case 14: solve_team_formation(); break;
            case 15: solve_count_subset_sum_ways(); break;
            case 16: solve_longest_path_bitmask(); break;
            case 17: solve_count_independent_sets(); break;
            case 18: solve_min_dominating_set(); break;
            case 19: solve_task_ordering(); break;
            case 20: solve_max_xor_subset(); break;
            case 21:
```

```cpp
                solve_enumerate_subsets();
                solve_sum_of_subsets();
                solve_count_subsets_with_sum();
                solve_max_sum_subset();
                solve_assignment_problem();
                solve_tsp();
                solve_count_hamiltonian_paths();
                solve_max_independent_set();
                solve_max_clique();
                solve_min_vertex_cover();
                solve_set_cover();
                solve_count_perfect_matchings();
                solve_partition_min_difference();
                solve_team_formation();
                solve_count_subset_sum_ways();
                solve_longest_path_bitmask();
                solve_count_independent_sets();
                solve_min_dominating_set();
                solve_task_ordering();
                solve_max_xor_subset();
                break;
            default: cout << "Invalid choice.\n";
        }
    }

    return 0;
}

```