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

// Structure for storing profit-weight pairs
struct Pair {
    int profit;
    int weight;
    Pair(int p, int w) : profit(p), weight(w) {}
};

// Function to merge and purge states (keeping only non-dominated pairs)
vector<Pair> MergePurge(vector<Pair> &A, vector<Pair> &B, int m) {
    vector<Pair> merged;
    int i = 0, j = 0;

    // Merge two sorted lists
    while (i < (int)A.size() && j < (int)B.size()) {
        if (A[i].weight < B[j].weight) merged.push_back(A[i++]);
        else if (A[i].weight > B[j].weight) merged.push_back(B[j++]);
        else {
            // If equal weight, take the better profit
            merged.push_back((A[i].profit > B[j].profit) ? A[i++] : B[j++]);
        }
    }
    while (i < (int)A.size()) merged.push_back(A[i++]);
    while (j < (int)B.size()) merged.push_back(B[j++]);

    // Purge dominated pairs
    vector<Pair> purged;
    int bestProfit = -1;
    for (auto &p : merged) {
        if (p.weight <= m && p.profit > bestProfit) {
            purged.push_back(p);
            bestProfit = p.profit;
        }
    }
    return purged;
}

// Dynamic programming using sets
void KnapsackDP(vector<int> &p, vector<int> &w, int n, int m) {
    vector<vector<Pair>> S(n + 1);

    // Initialization
    S[0].push_back(Pair(0, 0));

    // Build states
    for (int i = 1; i <= n; i++) {
        vector<Pair> temp;
        for (auto &prev : S[i - 1]) {
            // Case 1: Don't take item i
            temp.push_back(prev);

            // Case 2: Take item i
            int newProfit = prev.profit + p[i - 1];
            int newWeight = prev.weight + w[i - 1];
            if (newWeight <= m)
                temp.push_back(Pair(newProfit, newWeight));
        }
        S[i] = MergePurge(S[i - 1], temp, m);
    }

    // Get best solution
    Pair best = S[n].back();
    cout << "Maximum profit = " << best.profit << endl;
    cout << "Weight = " << best.weight << endl;
```

```
}

int main() {
    int n, m;
    cout << "Enter number of items: ";
    cin >> n;
    cout << "Enter knapsack capacity: ";
    cin >> m;

    vector<int> profit(n), weight(n);
    cout << "Enter profits: ";
    for (int i = 0; i < n; i++) cin >> profit[i];
    cout << "Enter weights: ";
    for (int i = 0; i < n; i++) cin >> weight[i];

    KnapsackDP(profit, weight, n, m);
    return 0;
}

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

// Structure for storing profit-weight pairs
struct Pair {
    int profit;
    int weight;
    Pair(int p, int w) : profit(p), weight(w) {}
};

// Function to merge and purge states (keeping only non-dominated pairs)
vector<Pair> MergePurge(vector<Pair> &A, vector<Pair> &B, int m) {
    vector<Pair> merged;
    int i = 0, j = 0;

    // Merge two sorted lists
    while (i < (int)A.size() && j < (int)B.size()) {
        if (A[i].weight < B[j].weight) merged.push_back(A[i++]);
        else if (A[i].weight > B[j].weight) merged.push_back(B[j++]);
        else {
            // If equal weight, take the better profit
            merged.push_back((A[i].profit > B[j].profit) ? A[i++] : B[j++]);
        }
    }
    while (i < (int)A.size()) merged.push_back(A[i++]);
    while (j < (int)B.size()) merged.push_back(B[j++]);

    // Purge dominated pairs
    vector<Pair> purged;
    int bestProfit = -1;
    for (auto &p : merged) {
        if (p.weight <= m && p.profit > bestProfit) {
            purged.push_back(p);
            bestProfit = p.profit;
        }
    }
    return purged;
}

// Dynamic programming using sets
void KnapsackDP(vector<int> &p, vector<int> &w, int n, int m) {
    vector<vector<Pair>> S(n + 1);

    // Initialization
    S[0].push_back(Pair(0, 0));
    // Build states
    for (int i = 1; i <= n; i++) {
        vector<Pair> temp;
        for (auto &prev : S[i - 1]) {
            // Case 1: Don't take item i
            temp.push_back(prev);

            // Case 2: Take item i
            int newProfit = prev.profit + p[i - 1];
            int newWeight = prev.weight + w[i - 1];
            if (newWeight <= m)
                temp.push_back(Pair(newProfit, newWeight));
        }
        S[i] = MergePurge(S[i - 1], temp, m);
    }

    // Get best solution
    Pair best = S[n].back();
    cout << "Maximum profit = " << best.profit << endl;
    cout << "Weight = " << best.weight << endl;
}

int main() {
    int n, m;
    cout << "Enter number of items: ";
    cin >> n;
    cout << "Enter knapsack capacity: ";
    cin >> m;

    vector<int> profit(n), weight(n);
    cout << "Enter profits: ";
    for (int i = 0; i < n; i++) cin >> profit[i];
    cout << "Enter weights: ";
    for (int i = 0; i < n; i++) cin >> weight[i];

    KnapsackDP(profit, weight, n, m);
    return 0;
}



*/
```