// Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy

```
// USING DYNAMIC PROGRAMMING
// TC  O(n*w)
// SC  O(n*w)
#include <bits/stdc++.h>
using namespace std;

// Recursive function with memoization to solve 0-1 Knapsack problem
int solve(int w, vector<int> &wt, vector<int> &val, int ind, vector<vector<int>> &dp) {
    if (ind < 0)  // Base case: no items left to consider
        return 0;

    // Check if result already computed for current state
    if (dp[ind][w] != -1)
        return dp[ind][w];

    // Case 1: Do not include current item
    int notInclude = solve(w, wt, val, ind - 1, dp);

    // Case 2: Include current item (if it fits within remaining weight)
    int include = 0;
    if (wt[ind] <= w) {
        include = val[ind] + solve(w - wt[ind], wt, val, ind - 1, dp);
    }

    // Store and return the maximum value of including or not including the item
    return dp[ind][w] = max(notInclude, include);
}
```

```
// Wrapper function to initialize dp array and start the recursive solution
int knapsack(int w, vector<int> &wt, vector<int> &val, int n) {
    vector<vector<int>> dp(n, vector<int>(w + 1, -1));  // DP table initialized to -1
    return solve(w, wt, val, n - 1, dp);  // Start with the last item
}

int main() {
    int w = 50;  // Knapsack capacity
    vector<int> wt = {10, 20, 30};  // Weights of items
    vector<int> val = {60, 100, 120};  // Values of items
    int n = wt.size();

    cout << "Maximum Profit: " << knapsack(w, wt, val, n) << endl;
    return 0;
}
```