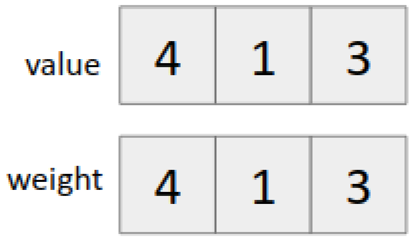
0-1 Knapsack

Problem

Given an array of items with their {weight, value}, and a knapsack (bori) with weight W. Find the maximum value of items that can be stolen and put into a knapsack.

Note: We either have to pick full item or no item, we cannot take partial items.

Example:



value	4	1	3
weight	4	1	3

W = 50 kg

Possible combinations that we can steal:

{15, 30}	{40}
V = 60 + 100 = 160	V = 150

It is important to note that we cannot apply greedy technique here as items are indivisible.

Way of thinking:

We iterate from left to right in the items array. For each item we have 2 choices

1. Take it ➢ Remaining capacity of the knapsack decreases.
2. Don’t take it ➢ Capacity of knapsack remains the same.

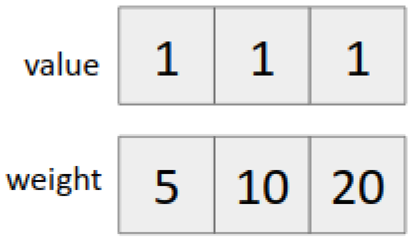
Let f(n, W) = denotes the maximum value of items that we can pick till item n and current capacity of knapsack W.

Therefore f(n, W) = max(*f(n-1, W), f(n-1, W-weight[n]) + value[n]*)

Since we can represent it as a recurrence relation, it follows optimal substructure property.

Let us check if it follows the overlapping subproblem property also?

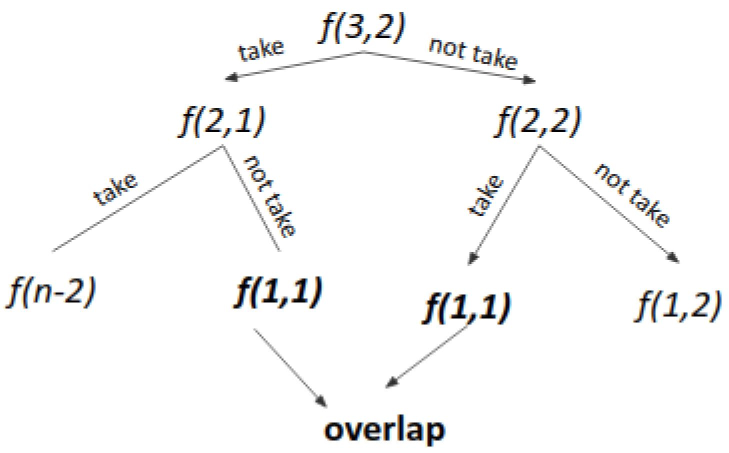
Example



value	1	1	1
weight	5	10	20

W = 2kg

Recursion tree for the above example



Since *f(1,1)* repeats ➢ It follows overlapping subproblem property also.

Hence it can be solved using dynamic programming.

Brute force Approach

1. Make all subsequences and pick the one with maximum value, which fits the constraints of knapsack.

Time Complexity: O(2^n)

where n is the number of items

Optimal Solution (Using dynamic programming)

Approach 1 (Memoization)

1. Write the recursive solution.
2. Memoize it.

Approach 2 (Tabulation)

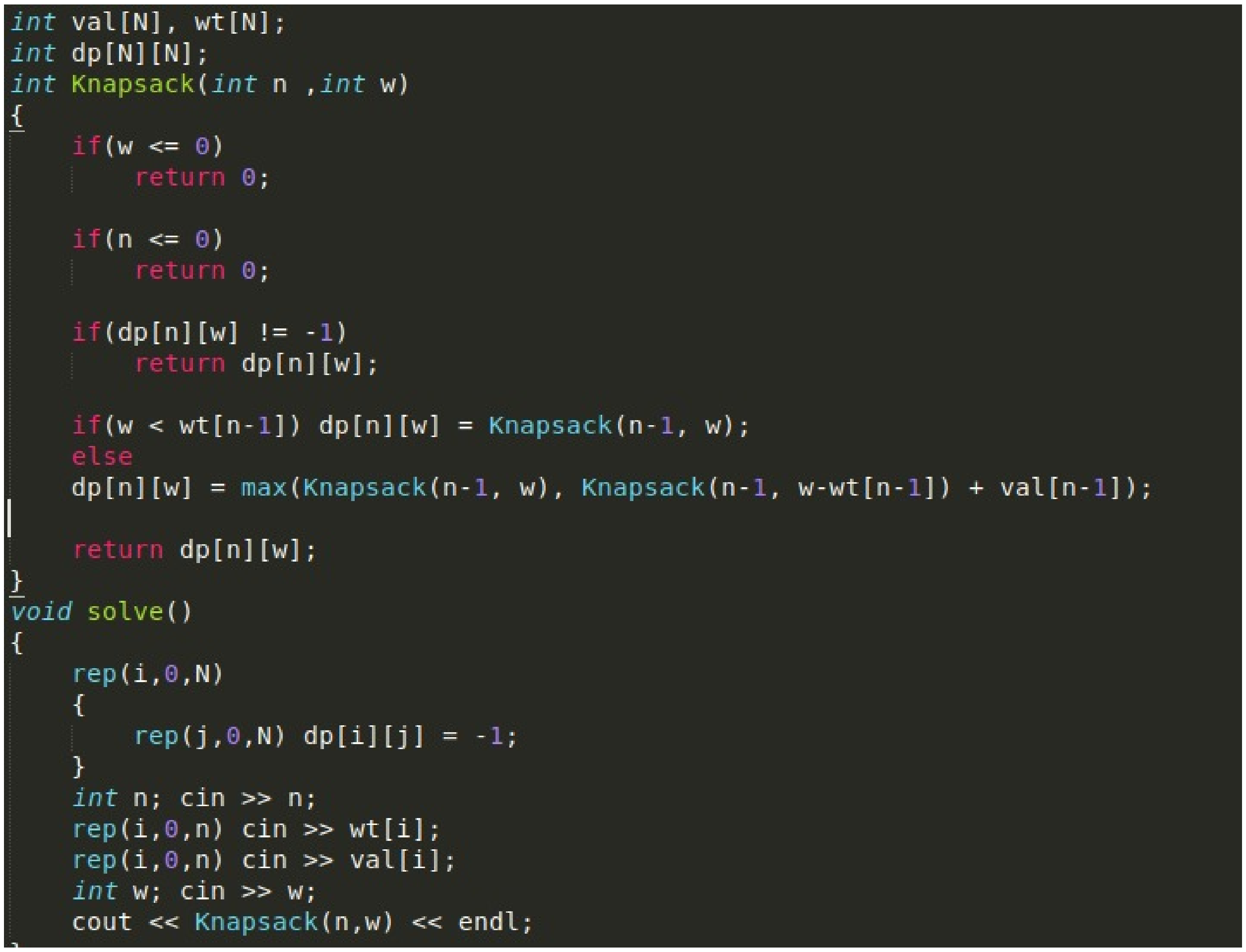
1. For each item, compute the answer for every weight from 0 to W.
2. Use recurrence of taking and not taking

   dp[n][w] = max(dp[n-1][w], dp[n-1][w-wt[n]] + val[n])

   provided w-wt[n] is non negative
3. Output the answer (*dp[n][w]*)

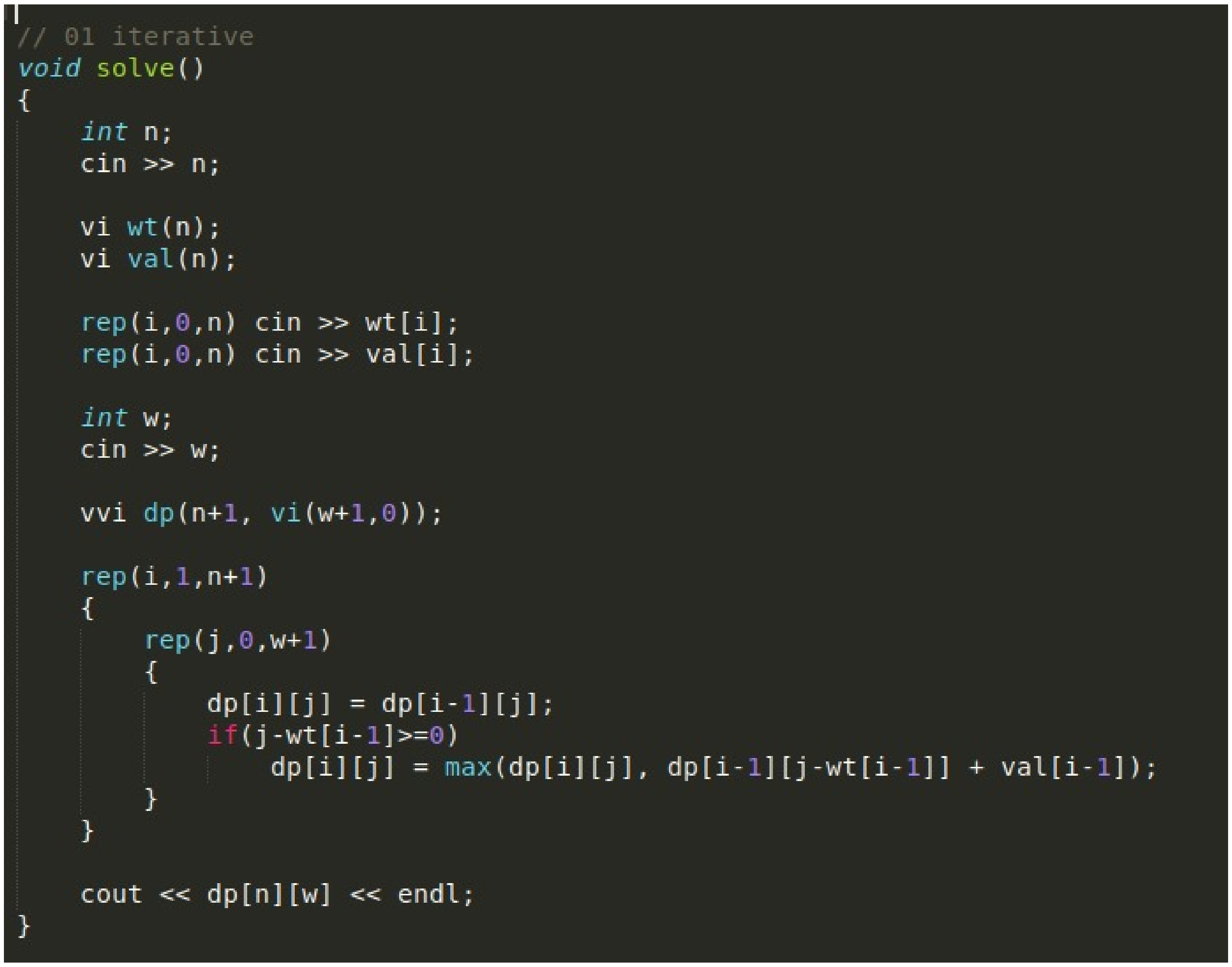
Time Complexity: 0(n*W)

Code (Memoization)



```cpp
int val[N], wt[N];
int dp[N][N];
int Knapsack(int n ,int w)
{
    if(w <= 0)
        return 0;

    if(n <= 0)
        return 0;

    if(dp[n][w] != -1)
        return dp[n][w];

    if(w < wt[n-1]) dp[n][w] = Knapsack(n-1, w);
    else
    dp[n][w] = max(Knapsack(n-1, w), Knapsack(n-1, w-wt[n-1]) + val[n-1]);

    return dp[n][w];
}
void solve()
{
    rep(i,0,N)
    {
        rep(j,0,N) dp[i][j] = -1;
    }
    int n; cin >> n;
    rep(i,0,n) cin >> wt[i];
    rep(i,0,n) cin >> val[i];
    int w; cin >> w;
    cout << Knapsack(n,w) << endl;
```

Code (Iterative)



```
// 01 iterative
void solve()
{
    int n;
    cin >> n;

    vi wt(n);
    vi val(n);

    rep(i,0,n) cin >> wt[i];
    rep(i,0,n) cin >> val[i];

    int w;
    cin >> w;

    vvi dp(n+1, vi(w+1,0));

    rep(i,1,n+1)
    {
        rep(j,0,w+1)
        {
            dp[i][j] = dp[i-1][j];
            if(j-wt[i-1]>=0)
                dp[i][j] = max(dp[i][j], dp[i-1][j-wt[i-1]] + val[i-1]);
        }
    }

    cout << dp[n][w] << endl;
}
```