0-1 Knapsack Problem using Dynamic Programming Approach

Problem Scenario

A thief is robbing a store and can carry a maximal weight of ***W*** into his knapsack. There are **n** items available in the store and weight of i^{th} item is w_i and its profit is p_i. What items should the thief take?

In this context, the items should be selected in such a way that the thief will carry those items (maximum weight of knapsack = ***W***) for which he will gain maximum profit. Hence, the objective of the thief is to maximize the profit.

In 0 1 knapsack problem, items can't be broken into smaller pieces i.e. items are indivisible, hence the thief can either take the item or not.

Dynamic-Programming Approach

Let ***i*** be the highest-numbered item in an optimal solution **S** for **W** dollars. Then $S' = S - \{i\}$ is an optimal solution for $W - w_i$ dollars and the value to the solution ***S*** is V_i plus the value of the sub-problem.

We can express this fact in the following formula: define **c[i, w]** to be the solution for items **1,2, … , i** and the maximum weight **w**.

The algorithm takes the following inputs

- The maximum weight **W**
- The number of items **n**
- The two sequences $v = \langle v_1, v_2, \dots, v_n \rangle$ and $w = \langle w_1, w_2, \dots, w_n \rangle$

```
Dynamic-0-1-knapsack (v, w, n, W)      → Array (wts. of items)
                        Array (Profits) of items
for w = 0 to W do
   c[0, w] = 0
for i = 1 to n do
   c[i, 0] = 0
   for w = 1 to W do
      if wᵢ ≤ w then
         if c[i-1,w] < vᵢ + c[i-1, w-wᵢ] then
            c[i, w] = vᵢ + c[i-1, w-wᵢ]
         else c[i, w] = c[i-1, w]
      else
         c[i, w] = c[i-1, w]
```

The set of items to take can be deduced from the table, starting at **c[n, w]** and tracing backwards where the optimal values came from.

If *c[i, w] = c[i-1, w]*, then item ***i*** is not part of the solution, and we continue tracing with **c[i-1, w]**. Otherwise, item ***i*** is part of the solution, and we continue tracing with **c[i-1, W-w]**.

Analysis

This algorithm takes θ(*n*, *w*) times as table *c* has (*n* + 1).(*w* + 1) entries, where each entry requires θ(1) time to compute.