7. KNAPSACK PROBLEM AND MEMORYFUNCTIONS

Designing a dynamic programming algorithm for the knapsack problem:

Given n items of known weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.

Assume that all the weights and the knapsack capacity are positive integers; the item values do not have to be integers.

0 / 1 knapsack problem means; the chosen item should be either null or whole.

Recurrence relation that expresses a solution to an instance of the knapsack problem

Let us consider an instance defined by the first *i* items, $1 \le i \le n$, with weights $w_1, \ldots, w_i$, values $v_1, \ldots, v_i$, and knapsack capacity j, $1 \le j \le W$. Let *F(i, j)* be the value of an optimal solution to this instance, i.e., the value of the most valuable subset of the first *i* items that fit into the knapsack of capacity *j*. We can divide all the subsets of the first *i* items that fit the knapsack of capacity *j* into two categories: those that do not include the *i*th item and those that do. Note the following:

1. Among the subsets that do not include the *i*th item, the value of an optimal subset is, by definition, $F(i-1, j)$.
2. Among the subsets that do include the *i*th item (hence, $j - w_i \ge 0$), an optimal subset is made up of this item and an optimal subset of the first $i-1$ items that fits into the knapsack of capacity $j - w_i$. The value of such an optimal subset is $v_i + F(i-1, j-w_i)$.

Thus, the value of an optimal solution among all feasible subsets of the first *I* items is the maximum of these two values. Of course, if the *i*th item does not fit into the knapsack, the value of an optimal subset selected from the first *i* items is the same as the value of an optimal subset selected from the first $i-1$ items. These observations lead to the following recurrence:

$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \ge 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

It is convenient to define the initial conditions as follows:

$F(0, j) = 0$ for $j \geq 0$ and $F(i, 0) = 0$ for $i \geq 0$.

Our goal is to find $F(n, W)$, the maximal value of a subset of the n given items that fit into the knapsack of capacity W, and an optimal subset itself.

*For F(i, j),*compute the maximum of the entry in the previous row and the same column and the sum of *vi* and the entry in the previous row and $w_i$columns to the left. The table can be filled either row by row or column by column.

ALGORITHM DPKnapsack(*w*[*1..n*], *v*[*1..n*], *W*)

var *V*[*0..n,0..W*], *P*[*1..n,1..W*]*:* int

for *j := 0* to *W***do**

V[*0,j*] *:=0*

for *i := 0* to *n***do**

V[*i,0*] *:=0*

for *i := 1* to *n***do**

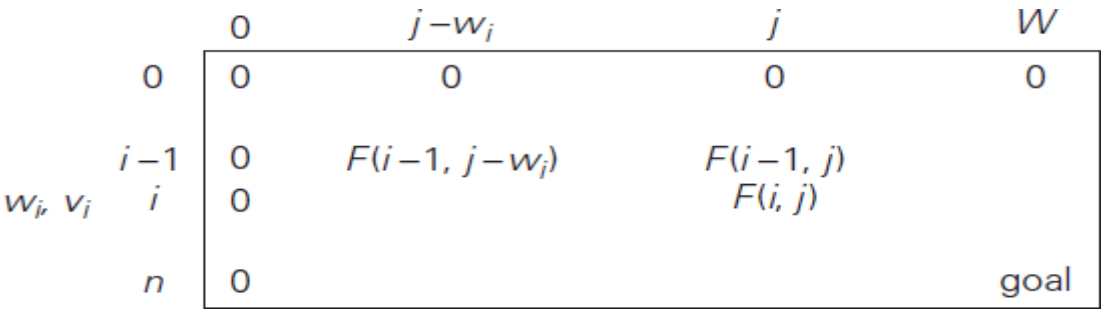
for *j := 1* to *W* **do**

if $w[i] \leq j$ and $v[i] + V[i-1,j-w[i]] > V[i-1,j]$ then
$V[i,j] := v[i] + V[i-1,j-w[i]]$; $P[i,j] := j-w[i]$
else
$V[i,j] := V[i-1,j]$; $P[i,j] := j$

return $V[n,W]$ and the optimal subset by back tracing

Note: Running timeandspace: O(nW).

Table3.1 for solving the knapsack problem by dynamic programming.



EXAMPLE 1 Let us consider the instance given by the following data:

Table 3.2 An instance of the knapsack problem:

item	**weight**	**value**	capacity
1	2	$12	
2	1	$10	$W = 5$
3	3	$20	
4	2	$15	

The maximal value is $F(4, 5) = \$37$. We can find the composition of an optimal subset by **back tracing** (Back tracing finds the actual optimal subset, i.e. solution), the computations of this entry in the table. Since $F(4, 5) > F(3, 5)$, item 4 has to be included in an optimal solution along with an optimal subset for filling $5 - 2 = 3$ remaining units of the knapsack capacity. The value of the latter is $F(3, 3)$. Since $F(3, 3) = F(2, 3)$, item 3 need not be in an optimal subset. Since $F(2, 3) > F(1, 3)$, item 2 is a part of an optimal selection, which leaves element $F(1, 3 - 1)$ to specify its remaining composition. Similarly, since $F(1, 2) > F(0, 2)$, item 1 is the final part of the optimal solution {item 1, item 2, item 4}.

Table 3.3 Solving an instance of the knapsack problem by the dynamic programming algorithm.

	Capacity j					
	i	0	1	2	3	4	5

	0	0	0	0	0	0	0
$w1 = 2, v1 = 12$	1	0	0	12	12	12	12
$w2 = 1, v2 = 10$	2	0	10	12	22	22	22
$w3 = 3, v3 = 20$	3	0	10	12	22	30	32
$w4 = 2, v4 = 15$	4	0	10	15	25	30	**37**

MEMORY FUNCTION:

The direct top-down approach to finding a solution to such a recurrence leads to an algorithm that solves common sub problems more than once and hence is very inefficient.

The bottom up fills a table with solutions to all smaller sub problems, but each of them is solved only once. An unsatisfying aspect of this approach is that solutions to some of these smaller sub problems are often not necessary for getting a solution to the problem given.

Since this drawback is not present in the top-down approach, it is natural to try to combine the strengths of the top-down and bottom-up approaches. The goal is to get a method that solves only sub problems that are necessary and does so only once. Such a method exists; it is based on using **memory functions.**

This method solves a given problem in the top-down manner but, in addition, maintains a table of the kind that would have been used by a bottom-up dynamic programming algorithm.

Initially, all the table's entries are initialized with a special "null" symbol to indicate that they have not yet been calculated. Thereafter, whenever a new value needs to be calculated, the method checks the corresponding entry in the table first: if this entry is not "null," it is simply retrieved from the table; otherwise, it is computed by the recursive call whose result is then recorded in the table.

The following algorithm implements this idea for the knapsack problem. After initializing the table, the recursive function needs to be called with i = n (the number of

items) and j = W (the knapsack capacity).

ALGORITHM MFKnapsack(i, j)

```
//Implements the memory function method for the knapsack problem
//Input: A nonnegative integer i indicating the number of the first items being considered
//        and a nonnegative integer j indicating the knapsack capacity
//Output: The value of an optimal feasible subset of the first i items
//Note: Uses as global variables input arrays Weights [1..n], Values[1..n],
//        and table F[0..n, 0..W ] whose entries are initialized with −1's except for
//        row 0 and column 0 initialized with0's
if F[i, j ]< 0
    if j <Weights[i]
        value←MFKnapsack(i − 1, j)
    else
    value←max(MFKnapsack(i − 1, j),
        Values[i]+ MFKnapsack(i − 1, j −Weights[i]))
    F[i, j ]←value
return F[i, j ]
```

EXAMPLE 2 Let us apply the memory function method to the instance considered in Example 1.

	Capacity j						
	I	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w1 = 2, v1 = 12$	1	0	0	12	12	12	12
$w2 = 1, v2 = 10$	2	0	-	12	22	-	22
$w3 = 3, v3 = 20$	3	0	-	-	22	-	32

$w4 = 2, v4 = 15$	4	0	-	-	-	-	37

Only 11 out of 20 nontrivial values (i.e., not those in row 0 or in column 0) have been computed. Just one nontrivial entry, V (1, 2), is retrieved rather than being recomputed. For larger instances, the proportion of such entries can be significantly larger.