8.2 The Knapsack Problem and Memory Functions

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.

Let us consider an instance defined by the first i items $1 \le i \le n$, and knapsack capacity j, $1 \le j \le W$. Let $F(i, j)$ be the value of an optimal solution to this instance. We can divide all the subsets of the first i items that fit the capacity j into two categories: those that do not include the ith item and those that do. Note the following:

1. Among the subsets that do not include the ith item, the value of an optimal subset is, by definition, $F(i-1, j)$.
2. Among the subsets that do include the ith 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:

$$F(i,j) = \begin{cases} \max\{F(i-1,j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0, \\ F(i-1,j) & \text{if } j - w_i < 0. \end{cases} \quad \textbf{(8.6)}$$

It is convenient to define the initial conditions as follows:

$$F(0,j) = 0 \text{ for } j \geq 0 \quad \text{and} \quad F(i,0) = 0 \text{ for } i \geq 0. \quad \textbf{(8.7)}$$

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.

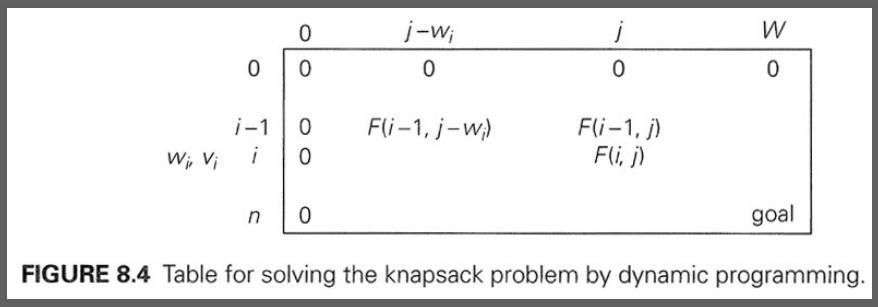

FIGURE 8.4 Table for solving the knapsack problem by dynamic programming.

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

item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

capacity $W = 5$.

The dynamic programming table, filled by applying formulas (8.6) and (8.7), is shown in Figure 8.5.

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3, v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2, v_4 = 15$	4	0	10	15	25	30	**37**

FIGURE 8.5 Example of solving an instance of the knapsack problem by the dynamic programming algorithm.

Thus, the maximal value is $F(4, 5) = \$37$. We can find the composition of an optimal subset by backtracing 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}.

Memory Functions

As we discussed at the beginning of this chapter and illustrated in subsequent sections, dynamic programming deals with problems whose solutions satisfy a recurrence relation with overlapping subproblems. The direct top-down approach to finding a solution to such a recurrence leads to an algorithm that solves common subproblems more than once and hence is very inefficient (typically, exponential or worse). The classic dynamic programming approach, on the other hand, works bottom up: it fills a table with solutions to *all* smaller subproblems, but each of them is solved only once.

An unsatisfying aspect of this approach is that solutions to some of these smaller subproblems 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 subproblems 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.

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 with 0'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. The table in Figure 8.6 gives the results. 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.

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	—	12	22	—	22
$w_3 = 3,\ v_3 = 20$	3	0	—	—	22	—	32
$w_4 = 2,\ v_4 = 15$	4	0	—	—	—	—	**37**

FIGURE 8.6 Example of solving an instance of the knapsack problem by the memory function algorithm.