```
C[n + 1, n] ← 0
for d ← 1 to n − 1 do //diagonal count
    for i ← 1 to n − d do
        j ← i + d
        minval ← ∞
        for k ← i to j do
            if C[i, k − 1] + C[k + 1, j] < minval
                minval ← C[i, k − 1] + C[k + 1, j]; kmin ← k
        R[i, j] ← kmin
        sum ← P[i]; for s ← i + 1 to j do sum ← sum + P[s]
        C[i, j] ← minval + sum
return C[1, n], R
```

The algorithm's space efficiency is clearly quadratic; the time efficiency of this version of the algorithm is cubic (why?). A more careful analysis shows that entries in the root table are always nondecreasing along each row and column. This limits values for $R[i, j]$ to the range $R[i, j-1], \ldots, R[i+1, j]$ and makes it possible to reduce the running time of the algorithm to $\Theta(n^2)$.

Exercises 8.3

1. Finish the computations started in the section's example of constructing an optimal binary search tree.
2. **a.** Why is the time efficiency of algorithm *OptimalBST* cubic?
   b. Why is the space efficiency of algorithm *OptimalBST* quadratic?
3. Write a pseudocode for a linear-time algorithm that generates the optimal binary search tree from the root table.
4. Devise a way to compute the sums $\sum_{s=i}^{j} p_s$, which are used in the dynamic programming algorithm for constructing an optimal binary search tree, in constant time (per sum).
5. True or false: The root of an optimal binary search tree always contains the key with the highest search probability?
6. How would you construct an optimal binary search tree for a set of n keys if all the keys are equally likely to be searched for? What will be the average number of comparisons in a successful search in such a tree if $n = 2^k$?
7. **a.** Show that the number of distinct binary search trees $b(n)$ that can be constructed for a set of n orderable keys satisfies the recurrence relation

$$b(n) = \sum_{k=0}^{n-1} b(k)b(n-1-k) \quad \text{for } n > 0, \quad b(0) = 1.$$

   b. It is known that the solution to this recurrence is given by the Catalan numbers. Verify this assertion for $n = 1, 2, \ldots, 5$.
   c. Find the order of growth of $b(n)$. What implication does the answer to this question have for the exhaustive-search algorithm for constructing an optimal binary search tree?
8. Design a $\Theta(n^2)$ algorithm for finding an optimal binary search tree.
9. Generalize the optimal binary search algorithm by taking into account unsuccessful searches.
10. ***Matrix chain multiplication*** Consider the problem of minimizing the total number of multiplications made in computing the product of n matrices

$$A_1 \cdot A_2 \cdot \ldots \cdot A_n$$

whose dimensions are d_0 by d_1, d_1 by $d_2, \ldots, d_{n-1}$ by d_n, respectively. Assume that all intermediate products of two matrices are computed by the brute-force (definition-based) algorithm.
   a. Give an example of three matrices for which the number of multiplications in $(A_1 \cdot A_2) \cdot A_3$ and $A_1 \cdot (A_2 \cdot A_3)$ differ at least by a factor 1000.
   b. How many different ways are there to compute the chained product of n matrices?
   c. Design a dynamic programming algorithm for finding an optimal order of multiplying n matrices.

8.4 The Knapsack Problem and Memory Functions

We start this section with designing the 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. (This problem was introduced in Section 3.4, where we discussed solving it by an exhaustive-search algorithm.) We assume here that all the weights and the knapsack's capacity are positive integers; the item values do not have to be integers.

To design a dynamic programming algorithm, we need to derive a recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solutions to its smaller subinstances. 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 $V[i, j]$ 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 ith item and those that do. Note the following:

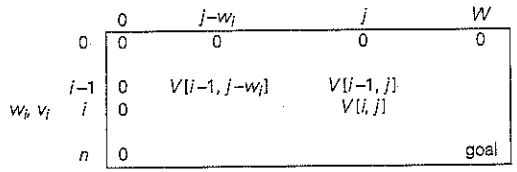

FIGURE 8.12 Table for solving the knapsack problem by dynamic programming

1. Among the subsets that do not include the ith item, the value of an optimal subset is, by definition, $V[i-1, j]$.
2. Among the subsets that do include the ith item (hence, $j - w_i \geq 0$), an optimal subset is made up of this item and an optimal subset of the first $i-1$ items that fit into the knapsack of capacity $j - w_i$. The value of such an optimal subset is $v_i + V[i-1, j-w_i]$.

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 ith 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:

$$V[i, j] = \begin{cases} \max\{V[i-1, j],\ v_i + V[i-1, j-w_i]\} & \text{if } j - w_i \geq 0 \\ V[i-1, j] & \text{if } j - w_i < 0. \end{cases} \quad \textbf{(8.12)}$$

It is convenient to define the initial conditions as follows:

$$V[0, j] = 0 \text{ for } j \geq 0 \text{ and } V[i, 0] = 0 \text{ for } i \geq 0. \quad \textbf{(8.13)}$$

Our goal is to find $V[n, W]$, 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 8.12 illustrates the values involved in equations (8.12) and (8.13). For $i, j > 0$, to compute the entry in the ith row and the jth column, $V[i, j]$, we compute the maximum of the entry in the previous row and the same column and the sum of v_i 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.

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 $W = 5$

		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 8.13 Example of solving an instance of the knapsack problem by the dynamic programming algorithm

The dynamic programming table, filled by applying formulas (8.12) and (8.13), is shown in Figure 8.13.

Thus, the maximal value is $V[4, 5] = \$37$. We can find the composition of an optimal subset by tracing back the computations of this entry in the table. Since $V[4, 5] \neq V[3, 5]$, item 4 was included in an optimal solution along with an optimal subset for filling $5 - 2 = 3$ remaining units of the knapsack capacity. The latter is represented by element $V[3, 3]$. Since $V[3, 3] = V[2, 3]$, item 3 is not a part of an optimal subset. Since $V[2, 3] \neq V[1, 3]$, item 2 is a part of an optimal selection, which leaves element $V[1, 3-1]$ to specify its remaining composition. Similarly, since $V[1, 2] \neq V[0, 2]$, item 1 is the final part of the optimal solution {item 1, item 2, item 4}. ■

The time efficiency and space efficiency of this algorithm are both in $\Theta(nW)$. The time needed to find the composition of an optimal solution is in $O(n + W)$. You are asked to prove these assertions in the exercises.

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 which are necessary and does it only once. Such a method exists; it is based on using ***memory functions*** [Bra96].

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 capacity of the knapsack).

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's 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 V[0..n, 0..W] whose entries are initialized with -1's except for
//row 0 and column 0 initialized with 0's
if V[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]))
    V[i, j] ← value
return V[i, j]
```

EXAMPLE 2 Let us apply the memory function method to the instance considered in Example 1. Figure 8.14 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. ■

In general, we cannot expect more than a constant-factor gain in using the memory function method for the knapsack problem because its time efficiency class is the same as that of the bottom-up algorithm (why?). A more significant improvement can be expected for dynamic programming algorithms in which a computation of one value takes more than constant time. You should also keep in mind that a memory function method may be less space-efficient than a space-efficient version of a bottom-up algorithm.

		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 8.14 Example of solving an instance of the knapsack problem by the memory function algorithm

Exercises 8.4

1. **a.** Apply the bottom-up dynamic programming algorithm to the following instance of the knapsack problem:

item	weight	value
1	3	\$25
2	2	\$20
3	1	\$15
4	4	\$40
5	5	\$50

capacity $W = 6$.

   b. How many different optimal subsets does the instance of part (a) have?

   c. In general, how can we use the table generated by the dynamic programming algorithm to tell whether there is more than one optimal subset for the knapsack problem's instance?

2. **a.** Write a pseudocode of the bottom-up dynamic programming algorithm for the knapsack problem.

   b. Write a pseudocode of the algorithm that finds the composition of an optimal subset from the table generated by the bottom-up dynamic programming algorithm for the knapsack problem.

3. For the bottom-up dynamic programming algorithm for the knapsack problem, prove that

   a. its time efficiency is in $\Theta(nW)$.

   b. its space efficiency is in $\Theta(nW)$.

   c. the time needed to find the composition of an optimal subset from a filled dynamic programming table is in $O(n + W)$.

4. **a.** True or false: A sequence of values in a row of the dynamic programming table for an instance of the knapsack problem is always nondecreasing?