For a second example, consider the problem of finding not the shortest, but the longest simple route between two cities, using a given set of roads. A *simple* route is one that never visits the same spot twice, so this condition rules out infinite routes round and round a loop. If we know that the longest simple route from Montreal to Toronto passes through Kingston, it does *not* follow that it can be obtained by taking the longest simple route from Montreal to Kingston, and then the longest simple route from Kingston to Toronto. It is too much to expect that when these two simple routes are spliced together, the resulting route will also be simple. Once again, the principle of optimality does not apply.

Nevertheless, the principle of optimality applies more often than not. When it does, it can be restated as follows: the optimal solution to any nontrivial instance of a problem is a combination of optimal solutions to *some* of its subinstances. The difficulty in turning this principle into an algorithm is that it is not usually obvious which subinstances are relevant to the instance under consideration. Coming back to the example of finding the shortest route, how can we tell whether the subinstance consisting of finding the shortest route from Montreal to Ottawa is relevant when we want the shortest route from Montreal to Toronto? This difficulty prevents our using an approach similar to divide-and-conquer starting from the original instance and recursively finding optimal solutions to the relevant subinstances, and only to these. Instead, dynamic programming efficiently solves every subinstance to figure out which ones are in fact relevant; only then are these combined into an optimal solution to the original instance.

8.4 The knapsack problem (2)

As in Section 6.5, we are given a number of objects and a knapsack. This time, however, we suppose that the objects may *not* be broken into smaller pieces, so we may decide either to take an object or to leave it behind, but we may not take a fraction of an object. For $i = 1, 2, \ldots, n$, suppose that object i has a positive weight w_i and a positive value v_i. The knapsack can carry a weight not exceeding W. Our aim is again to fill the knapsack in a way that maximizes the value of the included objects, while respecting the capacity constraint. Let x_i be 0 if we elect not to take object i, or 1 if we include object i. In symbols the new problem may be stated as:

$$\text{maximize } \sum_{i=1}^{n} x_i v_i \qquad \text{subject to } \sum_{i=1}^{n} x_i w_i \leq W$$

where $v_i > 0$, $w_i > 0$ and $x_i \in \{0,1\}$ for $1 \leq i \leq n$. Here the conditions on v_i and w_i are constraints on the instance; those on x_i are constraints on the solution. Since the problem closely resembles the one in Section 6.5, it is natural to enquire first whether a slightly modified version of the greedy algorithm we used before will still work. Suppose then that we adapt the algorithm in the obvious way, so that it looks at the objects in order of decreasing value per unit weight. If the knapsack is not full, the algorithm should select a *complete* object if possible before going on to the next.

Unfortunately the greedy algorithm turns out not to work when x_i is required to be 0 or 1. For example, suppose we have three objects available, the first of which

weighs 6 units and has a value of 8, while the other two weigh 5 units each and have a value of 5 each. If the knapsack can carry 10 units, then the optimal load includes the two lighter objects for a total value of 10. The greedy algorithm, on the other hand, would begin by choosing the object that weighs 6 units, since this is the one with the greatest value per unit weight. However if objects cannot be broken the algorithm will be unable to use the remaining capacity in the knapsack. The load it produces therefore consists of just one object with a value of only 8.

To solve the problem by dynamic programming, we set up a table $V[1..n, 0..W]$, with one row for each available object, and one column for each weight from 0 to W. In the table, $V[i, j]$ will be the maximum value of the objects we can transport if the weight limit is j, $0 \leq j \leq W$, and if we only include objects numbered from 1 to i, $1 \leq i \leq n$. The solution of the instance can therefore be found in $V[n, W]$.

The parallel with the problem of making change is close. As there, the principle of optimality applies. We may fill in the table either row by row or column by column. In the general situation, $V[i, j]$ is the larger (since we are trying to maximize value) of $V[i-1, j]$ and $V[i-1, j-w_i]+v_i$. The first of these choices corresponds to not adding object i to the load. The second corresponds to choosing object i, which has for effect to increase the value of the load by v_i and to reduce the capacity available by w_i. Thus we fill in the entries in the table using the general rule

$$V[i, j] = \max(V[i-1, j], V[i-1, j-w_i]+v_i).$$

For the out-of-bounds entries we define $V[0, j]$ to be 0 when $j \geq 0$, and we define $V[i, j]$ to be $-\infty$ for all i when $j < 0$. The formal statement of the algorithm, which closely resembles the function *coins* of the previous section, is left as an exercise for the reader; see Problem 8.11.

Figure 8.4 gives an example of the operation of the algorithm. In the figure there are five objects, whose weights are respectively 1, 2, 5, 6 and 7 units, and whose values are 1, 6, 18, 22 and 28. Their values per unit weight are thus 1.00, 3.00, 3.60, 3.67 and 4.00. If we can carry a maximum of 11 units of weight, then the table shows that we can compose a load whose value is 40.

Weight limit:	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1, v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2, v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5, v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6, v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5 = 7, v_5 = 28$	0	1	6	7	7	18	22	28	29	34	35	40

Figure 8.4. The knapsack using dynamic programming

Just as for the problem of making change, the table V allows us to recover not only the value of the optimal load we can carry, but also its composition. In our example, we begin by looking at $V[5, 11]$. Since $V[5, 11] = V[4, 11]$ but $V[5, 11] \neq V[4, 11 - w_5]+v_5$, an optimal load cannot include object 5. Next $V[4, 11] \neq V[3, 11]$