Figure 8.3 illustrates the instance given earlier, where we have to pay 8 units with coins worth 1, 4 and 6 units. For example, $c[3,8]$ is obtained in this case as the smaller of $c[2,8] = 2$ and $1 + c[3, 8 - d_3] = 1 + c[3,2] = 3$. The entries elsewhere in the table are obtained similarly. The answer to this particular instance is that we can pay 8 units using only two coins. In fact the table gives us the solution to our problem for all the instances involving a payment of 8 units or less.

Amount:	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3	1	2	3	4	2
$d_3 = 6$	0	1	2	3	1	2	1	2	2

Figure 8.3. Making change using dynamic programming

Here is a more formal version of the algorithm.

```
function coins(N)
    {Gives the minimum number of coins needed to make
     change for N units. Array d[1..n] specifies the coinage:
     in the example there are coins for 1, 4 and 6 units.}
    array d[1..n] = [1,4,6]
    array c[1..n, 0..N]
    for i ← 1 to n do c[i,0] ← 0
    for i ← 1 to n do
        for j ← 1 to N do
            c[i,j] ← if i = 1 and j < d[i] then +∞
                     else if i = 1 then 1 + c[1, j - d[1]]
                     else if j < d[i] then c[i - 1, j]
                     else min(c[i - 1, j], 1 + c[i, j - d[i]])
    return c[n,N]
```

If an unlimited supply of coins with a value of 1 unit is available, then we can always find a solution to our problem. If this is not the case, there may be values of N for which no solution is possible. This happens for instance if all the coins represent an even number of units, and we are required to pay an odd number of units. In such instances the algorithm returns the artificial result $+\infty$. Problem 8.9 invites the reader to modify the algorithm to handle a situation where the supply of coins of a particular denomination is limited.

Although the algorithm appears only to say how many coins are required to make change for a given amount, it is easy once the table c is constructed to discover exactly which coins are needed. Suppose we are to pay an amount j using coins of denominations $1, 2, \ldots, i$. Then the value of $c[i,j]$ says how many coins are needed. If $c[i,j] = c[i-1,j]$, no coins of denomination i are necessary, and we move up to $c[i-1,j]$ to see what to do next; if $c[i,j] = 1 + c[i, j - d_i]$, then we hand over one coin of denomination i, worth d_i, and move left to $c[i, j - d_i]$ to see what to do next. If $c[i-1,j]$ and $1 + c[i, j - d_i]$ are both equal to $c[i,j]$, we may choose either course of action. Continuing in this way, we eventually arrive back at $c[0,0]$, and now there remains nothing to pay. This stage of the algorithm is essentially a greedy algorithm that bases its decisions on the information in the table, and never has to backtrack.

Analysis of the algorithm is straightforward. To see how many coins are needed to make change for N units when n different denominations are available, the algorithm has to fill up an $n \times (N+1)$ array, so the execution time is in $\Theta(nN)$. To see which coins should be used, the search back from $c[n,N]$ to $c[0,0]$ makes $n-1$ steps to the row above (corresponding to not using a coin of the current denomination) and $c[n,N]$ steps to the left (corresponding to handing over a coin). Since each of these steps can be made in constant time, the total time required is in $\Theta(n + c[n,N])$.

8.3 The principle of optimality

The solution to the problem of making change obtained by dynamic programming seems straightforward, and does not appear to hide any deep theoretical considerations. However it is important to realize that it relies on a useful principle called the *principle of optimality*, which in many settings appears so natural that it is invoked almost without thinking. This principle states that in an optimal sequence of decisions or choices, each subsequence must also be optimal. In our example, we took it for granted, when calculating $c[i,j]$ as the lesser of $c[i-1,j]$ and $1 + c[i, j - d_i]$, that if $c[i,j]$ is the optimal way of making change for j units using coins of denominations 1 to i, then $c[i-1,j]$ and $c[i, j - d_i]$ must also give the *optimal* solutions to the instances they represent. In other words, although the only value in the table that really interests us is $c[n,N]$, we took it for granted that all the other entries in the table must also represent optimal choices: and rightly so, for in this problem the principle of optimality applies.

Although this principle may appear obvious, it does not apply to every problem we might encounter. When the principle of optimality does *not* apply, it will probably not be possible to attack the problem in question using dynamic programming. This is the case, for instance, when a problem concerns the optimal use of limited resources. Here the optimal solution to an instance may not be obtained by combining the optimal solutions to two or more subinstances, if the resources used in these subsolutions add up to more than the total resources available.

For example, if the shortest route from Montreal to Toronto passes through Kingston, then that part of the journey from Montreal to Kingston must also follow the shortest possible route, as must the part of the journey from Kingston to Toronto. Thus the principle of optimality applies. However if the fastest way to drive from Montreal to Toronto takes us through Kingston, it does not necessarily follow that it is best to drive as fast as possible from Montreal to Kingston, and then to drive as fast as possible from Kingston to Toronto. If we use too much petrol on the first half of the trip, we may have to fill up somewhere on the second half, losing more time than we gained by driving hard. The sub-trips from Montreal to Kingston, and from Kingston to Toronto, are not independent, since they share a resource, so choosing an optimal solution for one sub-trip may prevent our using an optimal solution for the other. In this situation, the principle of optimality does not apply.