Chapter 8

Dynamic Programming

In the previous chapter we saw that it is often possible to divide an instance into subinstances, to solve the subinstances (perhaps by dividing them further), and then to combine the solutions of the subinstances so as to solve the original instance. It sometimes happens that the natural way of dividing an instance suggested by the structure of the problem leads us to consider several overlapping subinstances. If we solve each of these independently, they will in turn create a host of identical subinstances. If we pay no attention to this duplication, we are likely to end up with an inefficient algorithm; if, on the other hand, we take advantage of the duplication and arrange to solve each subinstance only once, saving the solution for later use, then a more efficient algorithm will result. The underlying idea of dynamic programming is thus quite simple: avoid calculating the same thing twice, usually by keeping a table of known results that fills up as subinstances are solved.

Divide-and-conquer is a *top-down* method. When a problem is solved by divide-and-conquer, we immediately attack the complete instance, which we then divide into smaller and smaller subinstances as the algorithm progresses. Dynamic programming on the other hand is a *bottom-up* technique. We usually start with the smallest, and hence the simplest, subinstances. By combining their solutions, we obtain the answers to subinstances of increasing size, until finally we arrive at the solution of the original instance.

We begin the chapter with two simple examples of dynamic programming that illustrate the general technique in an uncomplicated setting. The following sections pick up the problems of making change, which we met in Section 6.1, and of filling a knapsack, encountered in Section 6.5.

8.1 Two simple examples

8.1.1 Calculating the binomial coefficient

Consider the problem of calculating the binomial coefficient

$$\binom{n}{k} = \begin{cases} 1 & \text{if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n \\ 0 & \text{otherwise.} \end{cases}$$

Suppose $0 \le k \le n$. If we calculate $\binom{n}{k}$ directly by

```
function C(n, k)
    if k = 0 or k = n then return 1
    else return C(n − 1, k − 1)+C(n − 1, k)
```

many of the values $C(i, j)$, $i < n$, $j < k$, are calculated over and over. For example, the algorithm calculates $C(5,3)$ as the sum of $C(4,2)$ and $C(4,3)$. Both these intermediate results require us to calculate $C(3,2)$. Similarly the value of $C(2,2)$ is used several times. Since the final result is obtained by adding up a number of 1s, the execution time of this algorithm is sure to be in $\Omega\left(\binom{n}{k}\right)$. We met a similar phenomenon before in the algorithm *Fibrec* for calculating the Fibonacci sequence; see Section 2.7.5.

If, on the other hand, we use a table of intermediate results—this is of course *Pascal's triangle*—we obtain a more efficient algorithm; see Figure 8.1. The table should be filled line by line. In fact, it is not even necessary to store the entire table: it suffices to keep a vector of length k, representing the current line, and to update this vector from left to right. Thus to calculate $\binom{n}{k}$ the algorithm takes a time in $\Theta(nk)$ and space in $\Theta(k)$, if we assume that addition is an elementary operation.

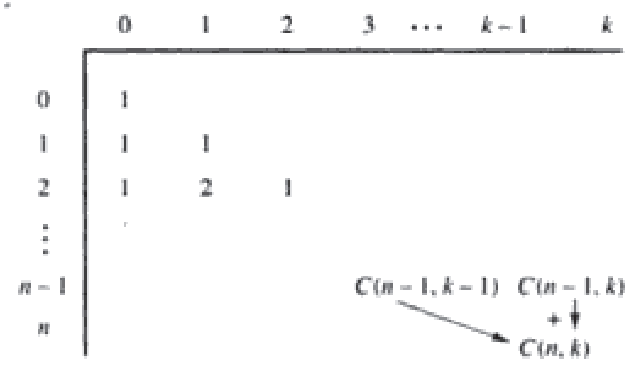

Figure 8.1. Pascal's triangle

8.1.2 The World Series

Imagine a competition in which two teams A and B play not more than $2n - 1$ games, the winner being the first team to achieve n victories. We assume that there are no tied games, that the results of each match are independent, and that for any given match there is a constant probability p that team A will be the winner, and hence a constant probability $q = 1 - p$ that team B will win.

Let $P(i, j)$ be the probability that team A will win the series given that they still need i more victories to achieve this, whereas team B still need j more victories if they are to win. For example, before the first game of the series the probability that team A will be the overall winner is $P(n, n)$: both teams still need n victories to win the series. If team A require 0 more victories, then in fact they have already won the series, and so $P(0, i) = 1$, $1 \le i \le n$. Similarly if team B require 0 more victories, then they have already won the series, and the probability that team A will be the overall winners is zero: so $P(i, 0) = 0$, $1 \le i \le n$. Since there cannot be a situation where both teams have won all the matches they need, $P(0, 0)$ is meaningless. Finally, since team A win any given match with probability p and lose it with probability q,

$$P(i, j) = pP(i-1, j) + qP(i, j-1), \quad i \ge 1, \quad j \ge 1.$$

Thus we can compute $P(i, j)$ as follows.

```
function $P(i, j)$
    if $i = 0$ then return 1
    else if $j = 0$ then return 0
    else return $pP(i-1, j) + qP(i, j-1)$
```

Let $T(k)$ be the time needed in the worst case to calculate $P(i, j)$, where $k = i + j$. With this method, we see that

$$\begin{aligned} T(1) &= c \\ T(k) &\le 2T(k-1) + d, \quad k > 1 \end{aligned}$$

where c and d are constants. Rewriting $T(k-1)$ in terms of $T(k-2)$, and so on, we find

$$\begin{aligned} T(k) &\le 4T(k-2) + 2d + d, \quad k > 2 \\ &\vdots \\ &\le 2^{k-1}T(1) + (2^{k-2} + 2^{k-3} + \cdots + 2 + 1)d \\ &= 2^{k-1}c + (2^{k-1} - 1)d \\ &= 2^k(c/2 + d/2) - d. \end{aligned}$$

$T(k)$ is therefore in $O(2^k)$, which is $O(4^n)$ if $i = j = n$. In fact, if we look at the way the recursive calls are generated, we find the pattern shown in Figure 8.2, which is identical to that obtained in the naive calculation of the binomial coefficient. To see this, imagine that any call $P(m, n)$ in the figure is replaced by $C(m + n, n)$.

Thus $P(i,j)$ is replaced by $C(i+j,j)$, $P(i-1,j)$ by $C(i+j-1,j)$, and $P(i,j-1)$ by $C(i+j-1,j-1)$. Now the pattern of calls shown by the arrows corresponds to the calculation

$$C(i+j,j) = C(i+j-1,j) + C(i+j-1,j-1)$$

of a binomial coefficient. The total number of recursive calls is therefore exactly $2\binom{i+j}{j} - 2$; see Problem 8.1. To calculate the probability $P(n,n)$ that team A will win given that the series has not yet started, the required time is thus in $\Omega\left(\binom{2n}{n}\right)$.

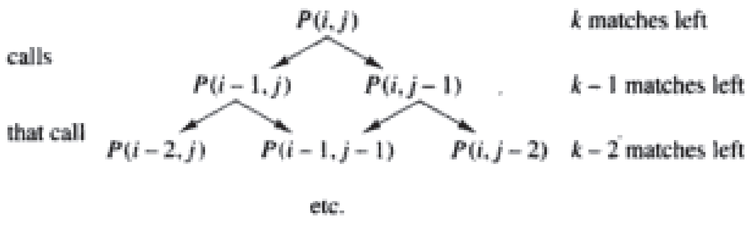

Figure 8.2. **Recursive calls made by a call on $P(i,j)$**

Problem 1.42 asks the reader to show that $\binom{2n}{n} \geq 4^n/(2n+1)$. Combining these results, we see that the time required to calculate $P(n,n)$ is in $O(4^n)$ and in $\Omega(4^n/n)$. The method is therefore not practical for large values of n. (Although sporting competitions with $n > 4$ are the exception, this problem does have other applications!)

To speed up the algorithm, we proceed more or less as with Pascal's triangle: we declare an array of the appropriate size and then fill in the entries. This time, however, instead of filling the array line by line, we work diagonal by diagonal. Here is the algorithm to calculate $P(n,n)$.

```
function series(n, p)
    array P[0..n, 0..n]
    q ← 1 − p
    {Fill from top left to main diagonal}
    for s ← 1 to n do
        P[0, s] ← 1;  P[s, 0] ← 0
        for k ← 1 to s − 1 do
            P[k, s − k] ← pP[k − 1, s − k] + qP[k, s − k − 1]
    {Fill from below main diagonal to bottom right}
    for s ← 1 to n do
        for k ← 0 to n − s do
            P[s + k, n − k] ← pP[s + k − 1, n − k] + qP[s + k, n − k − 1]
    return P[n, n]
```

Since the algorithm has to fill an $n \times n$ array, and since a constant time is required to calculate each entry, its execution time is in $\Theta(n^2)$. As with Pascal's triangle, it is easy to implement this algorithm so that storage space in $\Theta(n)$ is sufficient.

.2 Making change (2)

Recall that the problem is to devise an algorithm for paying a given amount to a customer using the smallest possible number of coins. In Section 6.1 we described a greedy algorithm for this problem. Unfortunately, although the greedy algorithm is very efficient, it works only in a limited number of instances. With certain systems of coinage, or when coins of a particular denomination are missing or in short supply, the algorithm may either find a suboptimal answer, or not find an answer at all.

For example, suppose we live where there are coins for 1, 4 and 6 units. If we have to make change for 8 units, the greedy algorithm will propose doing so using one 6-unit coin and two 1-unit coins, for a total of three coins. However it is clearly possible to do better than this: we can give the customer his change using just two 4-unit coins. Although the greedy algorithm does not find this solution, it is easily obtained using dynamic programming.

As in the previous section, the crux of the method is to set up a table containing useful intermediate results that are then combined into the solution of the instance under consideration. Suppose the currency we are using has available coins of n different denominations. Let a coin of denomination i, $1 \le i \le n$, have value d_i units. We suppose, as is usual, that each $d_i > 0$. For the time being we shall also suppose that we have an unlimited supply of coins of each denomination. Finally suppose we have to give the customer coins worth N units, using as few coins as possible.

To solve this problem by dynamic programming, we set up a table $c[1..n, 0..N]$, with one row for each available denomination and one column for each amount from 0 units to N units. In this table $c[i,j]$ will be the minimum number of coins required to pay an amount of j units, $0 \le j \le N$, using only coins of denominations 1 to i, $1 \le i \le n$. The solution to the instance is therefore given by $c[n,N]$ if all we want to know is how many coins are needed. To fill in the table, note first that $c[i,0]$ is zero for every value of i. After this initialization, the table can be filled either row by row from left to right, or column by column from top to bottom. To pay an amount j using coins of denominations 1 to i, we have in general two choices. First, we may choose not to use any coins of denomination i, even though this is now permitted, in which case $c[i,j] = c[i-1,j]$. Alternatively, we may choose to use at least one coin of denomination i. In this case, once we have handed over the first coin of this denomination, there remains to be paid an amount of $j - d_i$ units. To pay this takes $c[i, j-d_i]$ coins, so $c[i,j] = 1 + c[i, j-d_i]$. Since we want to minimize the number of coins used, we choose whichever alternative is the better. In general therefore

$$c[i,j] = \min(c[i-1,j], 1 + c[i, j-d_i]).$$

When $i = 1$ one of the elements to be compared falls outside the table. The same is true when $j < d_i$. It is convenient to think of such elements as having the value $+\infty$. If $i = 1$ and $j < d_1$, then both elements to be compared fall outside the table. In this case we set $c[i,j]$ to $+\infty$ to indicate that it is impossible to pay an amount j using only coins of type 1.

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 $i \leftarrow 1$ to $n$ do $c[i,0] \leftarrow 0$
    for $i \leftarrow 1$ to $n$ do
        for $j \leftarrow 1$ to $N$ do
            $c[i,j] \leftarrow$ if $i = 1$ and $j < d[i]$ then $+\infty$
                      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