15.1 THE METHOD

In dynamic programming, as in the greedy method, we view the solution to a problem as the result of a sequence of decisions. In the greedy method we make irrevokable decisions one at a time using a greedy criterion. However, in dynamic programming, we examine the decision sequence to see whether an optimal decision sequence contains optimal decision subsequences.

Example 15.1 [*Shortest Path*] Consider the digraph of Figure 12.2. We wish to find a shortest path from the source vertex $s = 1$ to the destination vertex $d = 5$. We need to make decisions on the intermediate vertices. The choices for the first decision are 2, 3, and 4. That is, from vertex 1 we may move to any one of these vertices. Suppose we decide to move to vertex 3. Now we need to decide on how to get from 3 to 5. If we go from 3 to 5 in a suboptimal way, then the 1-to-5 path constructed cannot be optimal, even under the restriction that from vertex 1 we must go to vertex 3. For example, if we use the suboptimal path 3, 2, 5 with length 9, the constructed 1-to-5 path 1, 3, 2, 5 has length 11. Replacing the suboptimal path 3, 2, 5 with an optimal one 3, 4, 5 results in the path 1, 3, 4, 5 of length 9.

So for this shortest-path problem, suppose that our first decision gets us to some vertex v. Although we do not know how to make this first decision, we do know that the remaining decisions must be optimal for the problem of going from v to d. ■

Example 15.2 [*0/1 Knapsack Problem*] Consider the 0/1 knapsack problem of Section 13.4. We need to make decisions on the values of $x_1, \cdots, x_n$. Suppose we are deciding the values of the x_is in the order $1, 2, \cdots, n$. If we set $x_1 = 0$, then the available knapsack capacity for the remaining objects (i.e., objects 2, 3, $\cdots, n$) is c. If we set $x_1 = 1$, the available knapsack capacity is $c - w_1$. Let $r \in \{c, c - w_1\}$ denote the remaining knapsack capacity.

Following the first decision, we are left with the problem of filling a knapsack with capacity r. The available objects (i.e., 2 through n) and the available capacity r define the *problem state* following the first decision. Regardless of whether x_1 is 0 or 1, $[x_2, \cdots, x_n]$ must be an optimal solution for the problem state following the first decision. If not, there is a solution $[y_2, \cdots, y_n]$ that provides greater profit for the problem state following the first decision. So $[x_1, y_2, \ldots, y_n]$ is a better solution for the initial problem.

Suppose that $n = 3$, $w = [100, 14, 10]$, $p = [20, 18, 15]$, and $c = 116$. If we set $x_1 = 1$, then following this decision, the available knapsack capacity is 16. $[x_2, x_3] = [0, 1]$ is a feasible solution to the two-object problem that remains. It returns a profit of 15. However, it is not an optimal solution to the remaining two-object problem, as $[x_2, x_3] = [1, 0]$ is feasible and returns a greater profit of 18. So $x = [1, 0, 1]$ can be improved to $[1, 1, 0]$. If we set $x_1 = 0$, the available capacity for the two-object instance that remains is 116. If the subsequence $[x_2, x_3]$ is not an optimal solution for this remaining instance, then $[x_1, x_2, x_3]$ cannot be optimal for the initial instance. ■

Example 15.3 [*Airfares*] A certain airline has the following airfare structure: From Atlanta to New York or Chicago, or from Los Angeles to Atlanta, the fare is \$100; from Chicago to New York, it is \$20; and for passengers connecting through Atlanta, the Atlanta to Chicago segment is only \$20. A routing from Los Angeles to New York involves decisions on the intermediate airports. If problem states are encoded as (origin, destination) pairs, then following a decision to go from Los Angeles to Atlanta, the problem state is, we are at Atlanta and need to get to New York. The cheapest way to go from Atlanta to New York is a direct flight with cost \$100. Using this direct flight results in a total Los Angeles-to-New York cost of \$200. However, the cheapest routing is Los Angeles--Atlanta--Chicago--New York with a cost of \$140, which involves using a suboptimal decision subsequence for the go from Atlanta to New York problem (Atlanta--Chicago--New York).

If instead we encode the problem state as a triple (*tag*, *origin*, *destination*) where *tag* is zero for connecting flights and one for all others, then once we reach Atlanta, the state becomes (0, Atlanta, New York) for which the optimal routing is through Chicago. ■

When optimal decision sequences contain optimal decision subsequences, we can establish recurrence equations, called **dynamic-programming recurrence equations**, that enable us to solve the problem in an efficient way.

Example 15.4 [*0/1 Knapsack*] In Example 15.2, we saw that for the 0/1 knapsack problem, optimal decision sequences were composed of optimal subsequences. Let $f(i,y)$ denote the value of an optimal solution to the knapsack instance with remaining capacity y and remaining objects $i, i+1, \cdots, n$. From Example 15.2, it follows that

$$f(n,y) = \begin{cases} p_n & \text{if } y \geq w_n \\ 0 & 0 \leq y < w_n \end{cases} \tag{15.1}$$

and

$$f.(i,y) = \begin{cases} \max\{f(i+1,y), f(i+1,y-w_i) + p_i\} & \text{if } y \geq w_i \\ f(i+1,y) & 0 \leq y < w_i \end{cases} \tag{15.2}$$

By making use of the observation that optimal decision sequences are made up of optimal subsequences, we have obtained a recurrence for f. $f(1,c)$ is the value of the optimal solution to the knapsack problem we started with. Recurrence 15.2 may be used to determine $f(1,c)$ either recursively or iteratively. In the iterative approach, we start with $f(n,*)$, as given by Equation 15.1 and then obtain $f(i,*)$ in the order $i = n-1, n-2, \cdots, 2$ using recurrence 15.2. Finally, $f(1,c)$ is computed using 15.2.

For the instance of Example 15.2, we see that $f(3,y) = 0$ if $0 \le y < 10$, and 15 if $y \ge 10$. Using recurrence (15.2), we obtain $f(2,y) = 0$ if $0 \le y < 10$, 15 if $10 \le y < 14$; 18 if $14 \le y < 24$; and 33 if $y \ge 24$. The optimal solution has value $f(1,116) = \max\{f(2,116), f(2, 116 - w_1) + p_1\} = \max\{f(2,116), f(2,16) + 20\}$ $= \max\{33, 38\} = 38$.

To obtain the values of the x_is, we proceed as follows: If $f(1,c) = f(2,c)$, then we may set $x_1 = 0$ because we can utilize the c units of capacity getting a return of $f(1,c)$ from objects $2, \cdots, n$. In case $f(1,c) \neq f(2,c)$, then we must set $x_1 = 1$. Next we need to find an optimal solution that uses the remaining capacity $c - w_1$. This solution has value $f(2,c-w_1)$. Proceeding in this way, we may determine the value of all the x_is.

For our sample instance, we see that $f(2,116) = 33 \neq f(1,116)$. Therefore, $x_1 = 1$ and we need to find x_2 and x_3 so as to obtain a return of $38 - p_1 = 18$ and use a capacity of at most $116 - w_1 = 16$. Note that $f(2,16) = 18$. Since $f(3,16)$ $= 14 \neq f(2,16)$, $x_2 = 1$; the remaining capacity is $16 - w_2 = 2$. Since $f(3,2) = 0$, we set $x_3 = 0$. ■

When dynamic programming is used, we first set up a recurrence for the value of the optimal solution by making use of the **principle of optimality**, which states that no matter what the first decision, the remaining decisions must be optimal with respect to the state that results from this first decision. Since the principle of optimality may not hold for some formulations of some problems, it is necessary to verify that it does hold for the problem being solved. *Dynamic programming cannot be applied when this principle does not hold.* After we solve the recurrence equation for the value of the optimal solution, we perform a **traceback** step in which the solution itself is constructed.

It is very tempting to write a simple recursive program to solve the dynamic-programming recurrence. *However, as we shall see in subsequent sections, unless care is taken to avoid recomputing previously computed values, the recursive program will have prohibitive complexity.* When the recursive program is designed to avoid this recomputation, the complexity is drastically reduced. The dynamic-programming recurrence may also be solved by iterative code that naturally avoids recomputation of already computed values. Although this iterative code has the same time complexity as the "careful" recursive code, the former has the advantage of not requiring additional space for the recursion stack. As a result, the iterative code generally runs faster than the careful recursive code.

15.2 APPLICATIONS

15.2.1 0/1 Knapsack Problem

Recursive Solution

The dynamic-programming recurrence equations for the 0/1 knapsack problem were developed in Example 15.4. A natural way to solve a recurrence such as 15.2 for the value $f(1,c)$ of an optimal knapsack packing is by a recursive program such as Program 15.1. This code assumes that `p`, `w`, and `n` are global and that `p` is of type `int`. The invocation `F(1,c)` returns the value of $f(1,c)$.

```
int F(int i, int y)
{// Return f(i,y).
   if (i == n) return (y < w[n]) ? 0 : p[n];
   if (y < w[i]) return F(i+1,y);
   return max(F(i+1,y), F(i+1,y-w[i]) + p[i]);
}
```

Program 15.1 Recursive function for knapsack problem

Let $t(n)$ be the time this code takes to solve an instance with n objects. We see that $t(1) = a$ and $t(n) \le 2t(n-1) + b$ for $n > 1$. Here a and b are constants. This recurrence solves to $t(n) = O(2^n)$.

Example 15.5 Consider the case $n = 5$, $p = [6, 3, 5, 4, 6]$, $w = [2, 2, 6, 5, 4]$, and $c = 10$. To determine $f(1,10)$, function `F` is invoked as `F(1,10)`. The recursive calls made are shown by the tree of Figure 15.1. Each node has been labeled by the value of y. Nodes on level j have $i = j$. So the root denotes the invocation `F(1,10)`. Its left and right children, respectively, denote the invocations `F(2,10)` and `F(2,8)`. In all, 28 invocations are made. Notice that several invocations redo the work of previous invocations. For example, $f(3,8)$ is computed twice, as are $f(4,8)$, $f(4,6)$, $f(4,2)$, $f(5,8)$, $f(5,6)$, $f(5,3)$, $f(5,2)$, and $f(5,1)$. If we save the results of previous invocations, we can reduce the number of invocations to 19 because we eliminate the shaded nodes of Figure 15.1. ■

As observed in Example 15.5, Program 15.1 is doing more work than necessary. To avoid computing the same $f(i,y)$ value more than once, we may keep a list L of $f(i,y)$s that have already been computed. The elements of this list are triples of the form $(i,y,f(i,y))$. Before making an invocation `F(i,y)`, we see whether the list L contains a triple of the form $(i,y,*)$ where $*$ denotes a wildcard. If so, $f(i,y)$ is retrieved from the list. If not, the invocation is made

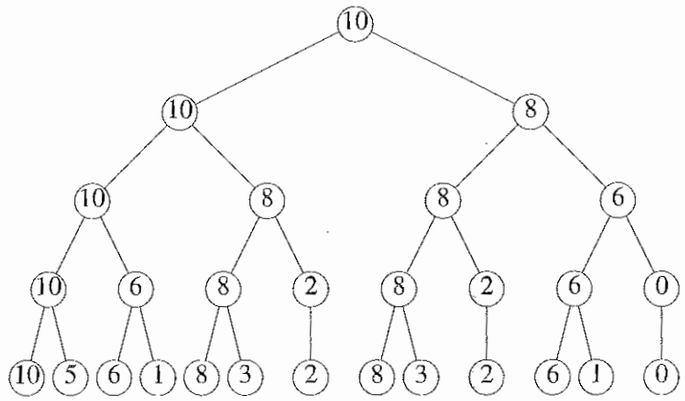

Figure 15.1 Tree of recursive calls

and then the triple $(i,y,f(i,y))$ added to L. L may be stored as a hash table (see Section 7.4) or as a binary search tree (see Chapter 11).

Iterative Solution with Integer Weights

We can devise a fairly simple iterative algorithm (Program 15.2) to solve for $f(1,c)$ when the weights are integer. This algorithm is based on the strategy outlined in Example 15.4, and it computes each $f(i,y)$ exactly once. Program 15.2 uses a two-dimensional array `f[][]` to store the values of the function f. The code for the traceback needed to determine the x_i values that result in the optimal filling appears in Program 15.2.

The complexity of function `Knapsack` is $\Theta(nc)$ and that of `Traceback` is $\Theta(n)$.

Tuple Method *(Optional)*

There are two drawbacks to the code of Program 15.2. First, it requires that the weights be integer. Second, it is slower than Program 15.1 when the knapsack capacity is large. In particular, if $c > 2^n$, its complexity is $\Omega(n\,2^n)$. We can overcome both of these shortcomings by using a tuple approach in which for each i, $f(i,y)$ is stored as an ordered list $P(i)$ of pairs $(y,f(i,y))$ that correspond to the y values at which the function f changes. The pairs in each $P(i)$ are in increasing order of y. Also, since $f(i,y)$ is a nondecreasing function of y, the pairs are also in increasing order of $f(i,y)$.


```
template<class T>
void Knapsack(T p[], int w[], int c, int n, T** f)
{// Compute f[i][y] for all i and y.

   // initialize f[n][]
   for (int y = 0; y < w[n]; y++)
      f[n][y] = 0;
   for (int y = w[n]; y <= c; y++)
      f[n][y] = p[n];

   // compute remaining f's
   for (int i = n - 1; i > 1; i--) {
      for (int y = 0; y < w[i]; y++)
         f[i][y] = f[i+1][y];
      for (int y = w[i]; y <= c; y++)
         f[i][y] = max(f[i+1][y],
                       f[i+1][y-w[i]] + p[i]);
   }
   f[1][c] = f[2][c];
   if (c >= w[1])
      f[1][c] = max(f[1][c], f[2][c-w[1]] + p[1]);
}

template<class T>
void Traceback(T **f, int w[], int c, int n, int x[])
{// Compute x for optimal filling.
   for (int i = 1; i < n; i++)
      if (f[i][c] == f[i+1][c]) x[i] = 0;
      else {x[i] = 1;
            c -= w[i];}
   x[n] = (f[n][c]) ? 1 : 0;
}
```

Program 15.2 Iterative computation of `f` and `x`

Example 15.6 For the knapsack instance of Example 15.5, the f function is given in Figure 15.2. When $i = 5$, the function f is completely specified by the pairs $P(5) = [(0,0), (4,6)]$. The pairs $P(i)$ for $i = 4$, 3, and 2 are $[(0,0), (4,6), (9,10), (10,11)]$, $[(0,0), (4,6), (9,10), (10,11)]$, and $[(0,0), (2,3), (4,6), (6,9), (9,10), (10,11)]$.

To compute $f(1,10)$, we use recurrence 15.2 which yields $f(1,10) = \max\{f(2,10), f(2,8) + p_1\}$. From $P(2)$, we get $f(2,10) = 11$, and $f(2,8) = 9$ ($f(2,8) = 9$ comes from the pair (6,9)). Therefore, $f(1,10) = \max\{11, 15\} = 15$.

To determine the x_i values, we begin with x_1. Since $f(1,10) = f(2,6) + p_1$, $x_5 = 1$. Since $f(2,6) = f(3,6-w_2) + p_2 = f(3,4) + p_2$, $x_2 = 1$. $x_3 = x_4 = 0$ because $f(3,4) = f(4,4) = f(5,4)$. Finally, since $f(5,4) \neq 0$, $x_5 = 1$. ■

	y										
i	0	1	2	3	4	5	6	7	8	9	10
5	0	0	0	0	6	6	6	6	6	6	6
4	0	0	0	0	6	6	6	6	6	10	10
3	0	0	0	0	6	6	6	6	6	10	11
2	0	0	3	3	6	6	9	9	9	10	11

Figure 15.2 f function for Example 15.6

If we examine the pairs in each $P(i)$, we see that each pair $(y, f(i,y))$ corresponds to a different combination of 0/1 assignments to the variables $x_i, \cdots, x_n$. Let (a,b) and (c,d) be pairs that correspond to two different 0/1 assignments to $x_i, \cdots, x_n$. If $a \geq c$ and $b < d$, then (a,b) is dominated by (b,c). Dominated assignments do not contribute pairs to $P(i)$. If two or more assignments result in the same pair, only one is in $P(i)$.

Under the assumption that $w_n \leq c$, $P(n) = [(0,0), (w_n,p_n)]$. These two pairs correspond to x_n equal to zero and one, respectively. For each i, $P(i)$ may be obtained from $P(i+1)$. First, compute the ordered set of pairs Q such that (s,t) is a pair of Q iff $w_i \leq s \leq c$ and $(s-w_i, t-p_i)$ is a pair of $P(i+1)$. Now Q has the pairs with $x_i = 1$ and $P(i+1)$ has those with $x_i = 0$. Next, merge Q and $P(i+1)$ eliminating dominated as well as duplicate pairs to get $P(i)$.

Example 15.7 Consider the data of Example 15.6. $P(5) = [(0,0), (4,6)]$, so $Q = [(5,4), (9,10)]$. When merging $P(5)$ and Q to create $P(4)$, the pair (5,4) is eliminated because it is dominated by the pair (4,6). As a result, $P(4) = [(0,0), (4,6), (9,10)]$. To compute $P(3)$, we first obtain $Q = [(6,5), (10,11)]$ from $P(4)$. Next, merging with $P(4)$ yields $P(3) = [(0,0), (4,6), (9,10), (10,11)]$. Finally, to get $P(2)$, $Q = [(2,3), (6,9)]$ is computed from $P(3)$. Merging $P(3)$ and Q yields $P(2) = [(0,0), (2,3), (4,6), (6,9), (9,10), (10,11)]$. ■

Since the pairs in each $P(i)$ represent different 0/1 assignments to $x_i, \cdots, x_n$, no $P(i)$ has more than 2^{n-i+1} pairs. When computing $P(i)$, Q may be computed in $\Theta(|P(i+1)|)$ time. The time needed to merge $P(i+1)$ and Q is also $\Theta(|P(i+1)|)$. So all the $P(i)$s may be computed in $\Theta(\sum_{i=2}^{n} |P(i+1)|) = O(2^n)$ time. When the weights are integer, $|P(i)| \leq c + 1$. In this case the complexity becomes $O(\min\{nc, 2^n\})$.