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 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 vertex are $s$, $2$, $3$, and $4$. If we choose $2$ or $3$, we cannot go to any other vertex. If we choose $4$, we cannot go to $3$. If we choose $5$, we can go to any other vertex.

We need to decide on vertices $x_1$, $x_2$, $x_3$, and $x_4$. The choices for the first vertex are $s$, $x_2$, $x_3$, and $x_4$. If we choose $x_2$, we can go to $x_3$. If we choose $x_3$, we cannot go to $x_4$. If we choose $x_4$, we cannot go to $x_3$.

The length of the constructed path is the sum of the weights of the edges. The shortest path is the one with the smallest length.

Example 15.2 [0/1 Knapsack Problem] Consider the 0/1 knapsack problem of Section 13.4. We want to make decisions on the values of $x_1$, $x_2$, $x_3$, and $x_4$. The choices for $x_1$ are $0$ and $1$. If we choose $0$, we cannot go to $1$. If we choose $1$, we can go to $2$.

The available knapsack capacity for the remaining objects is $c - w_i$. Let $c = 1$, the available knapsack capacity is $c - w_i$. If we set $x_1 = 1$, the available knapsack capacity is $c - w_i$. If we set $x_1 = 0$, the available knapsack capacity is $c - w_i$.

The weight of the knapsack is the sum of the weights of the objects. The maximum weight is $w_i$.

The maximum profit is the maximum profit over all possible sequences of decisions. The maximum profit is the maximum profit over all possible sequences of decisions.

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 $20$. 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 go to New York. The cheapest way to go from Atlanta to New York is a direct flight with cost $10$. Using this direct flight results in a total Los Angeles-to-New York cost of $20$. However, the cheapest routing is Los Angeles—Atlanta—Chicago—New York with a cost of $140$, which involves using a suboptimal decision sequence for the go from Atlanta to New York problem (Atlanta—Chicago—New York).

If instead we encode the problem state as a triple $(s, o, d)$, where $s$ 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 $i$, $i+1$, ..., $n$. From Example 15.2, it follows that

$$f(i, y) = \begin{cases} \rho_i & \text{if } y \geq w_i \\ 0 & \text{if } 0 \leq y < w_i \end{cases}$$

and

$$f(i, y) = \max \{ f(i+1, y), f(i+1, y-w_i) + \rho_i \} \text{ if } y \geq w_i$$
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 \texttt{Int}. The invocation \( F(1,c) \) returns the value of \( f(1,c) \).

```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 \( t(n) \leq 2t(n-1) + b \) for \( n \geq 1 \). 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 \( F(2,10) \). Its left and right children, respectively, denote the invocations \( F(2,9) \) 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 \( f(3,8), f(4,6), f(4,5), f(5,6), f(5,7), f(5,8), f(5,3), f(5,2), \) 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 \( f(i,y) \) values that have already been computed. The elements of this list are triples of the form \((i,x,f(i,y))\). Before making an invocation \( F(i+1,y) \), we can check whether the list \( L \) contains a tuple of the form \((i,x,y)\); if so, the invocation is made.
Iterative Solution with Integer Weights

We can devise a fairly simple iterative algorithm (Program 15.2) to solve for $f(i, c)$ 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 $f[i][y]$ 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 $O(nc)$ and that of Traceback is $O(n^2)$.

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 $O(n^22^n)$. 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 $f(i, y)$.

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 $f(0, 0), (1, 0), (1, 0), (2, 0), (2, 1), (2, 1), (2, 2), (2, 2)$, and $(2, 2)$. To compute $f(1, 10)$, we use recurrence 15.2 which yields $f(1, 10) = \max(f(1, 10), f(2, 10) + p_1)$. From $P(2)$, we get $f(2, 10) = 11$, and $f(2, 8) = 9$ (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 $f(1, 10) = f(2, 6) + p_1, x_3 = 1$. Since $f(2, 6) = f(3, 6 - w_2) + p_2, x_2 = 1, x_3 = x_4 = 0$ because $f(3, 4) = f(4, 4) = f(5, 4)$. Finally, since $f(5, 4) = 0$, $x_5 = 1$.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 $x_1, \ldots, x_n$. Let $(a, b)$ and $(c, d)$ be pairs that correspond to two different 0/1 assignments to $x_1, \ldots, x_n$. 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 $w_n \leq c, P(a) = [(0,0), (w_n, p_n)]$. 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 $(x, i)$ is a pair of $Q$ if $w_i \leq x \leq c$ 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 $Q(4)$ is eliminated because it is dominated by the pair $Q(4)$. 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 $x_1, \ldots, x_n$, no $P(i)$ has more than $2^{n-1} + 1$ pairs. When computing $P(i)$, $Q$ may be computed in $\Theta(1 P(i+1))$ time. The time needed to merge $P(i+1)$ and $Q$ is also $\Theta(1 P(i+1))$. So all the $P(i)$s may be computed in $\Theta(n \sum_{i=1}^{n} P(i+1)) = O(2^n)$ time. When the weights are integer, $1 P(i+1)$ is a constant. In this case the complexity becomes $O(n \max(\|n\|, 2^n))$. 
