Introduction to

3rd Edition

# The Design and Analysis of Algorithms

8

# Dynamic Programming

動態規劃（Dynamic Programming）

一、基本概念

動態規劃過程是：每次決策依賴於當前狀態，又隨即引起狀態的轉移。一個決策序列就是在變化的狀態中產生出來的，所以，這種多階段最優化決策解決問題的過程就稱為動態規劃。

二、基本思想與策略

基本思想與分治法類似，也是將待求解的問題分解為若干個子問題（階段），按順序求解子階段，前一子問題的解，為後一子問題的求解提供了有用的資訊。在求解任一子問題時，列出各種可能的區域性解，通過決策保留那些有可能達到最優的區域性解，丟棄其他區域性解。依次解決各子問題，最後一個子問題就是初始問題的解。

由於動態規劃解決的問題多數有重疊子問題這個特點，為減少重複計算，對每一個子問題只解一次，將其不同階段的不同狀態儲存在一個二維陣列中。

與分治法最大的差別是：適合於用動態規劃法求解的問題，經分解後得到的子問題往往不是互相獨立的（即下一個子階段的求解是建立在上一個子階段的解的基礎上，進行進一步的求解）。

三、適用的情況

能採用動態規劃求解的問題的一般要具有3個性質：

(1) 最優化原理：如果問題的最優解所包含的子問題的解也是最優的，就稱該問題具有最優子結構，即滿足最優化原理。

(2) 無後效性：即某階段狀態一旦確定，就不受這個狀態以後決策的影響。也就是說，某狀態以後的過程不會影響以前的狀態，只與當前狀態有關。

(3) 有重疊子問題：即子問題之間是不獨立的，一個子問題在下一階段決策中可能被多次使用到。（該性質並不是動態規劃適用的必要條件，但是如果沒有這條性質，動態規劃演算法同其他演算法相比就不具備優勢）

四、求解的基本步驟

動態規劃所處理的問題是一個多階段決策問題，一般由初始狀態開始，通過對中間階段決策的選擇，達到結束狀態。這些決策形成了一個決策序列，同時確定了完成整個過程的一條活動路線(通常是求最優的活動路線)。如圖所示。動態規劃的設計都有著一定的模式，一般要經歷以下幾個步驟。

初始狀態→|決策１|→|決策２|→…→|決策ｎ|→結束狀態

圖1 動態規劃決策過程示意圖

**Dynamic programming** is an algorithm design technique with a rather interesting history. It was invented by a prominent U.S. mathematician, Richard Bellman, in the 1950s as a general method for optimizing multistage decision processes. Thus, the word "programming" in the name of this technique stands for **"planning"** and does not refer to computer programming. After proving its worth as an important tool of applied mathematics, dynamic programming has eventually come to be considered, at least in computer science circles, as a general algorithm design technique that does not have to be limited to special types of optimization problems. It is from this point of view that we will consider this technique here.

This technique can be illustrated by revisiting the Fibonacci numbers discussed in Section 2.5. (If you have not read that section, you will be able to follow the discussion anyway. But it is a beautiful topic, so if you feel a temptation to read it, do succumb to it.) The Fibonacci numbers are the elements of the sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots,$$

which can be defined by the simple recurrence

$$F(n) = F(n-1) + F(n-2) \text{ for } n > 1 \qquad (8.1)$$

and two initial conditions

$$F(0) = 0, \qquad F(1) = 1. \qquad (8.2)$$

**ALGORITHM** $F(n)$

**//Computes the $n$th Fibonacci number recursively by using its definition**

**//Input: A nonnegative integer $n$**

**//Output: The $n$th Fibonacci number**

**if $n \leq 1$ return $n$**

**else return $F(n-1) + F(n-2)$**

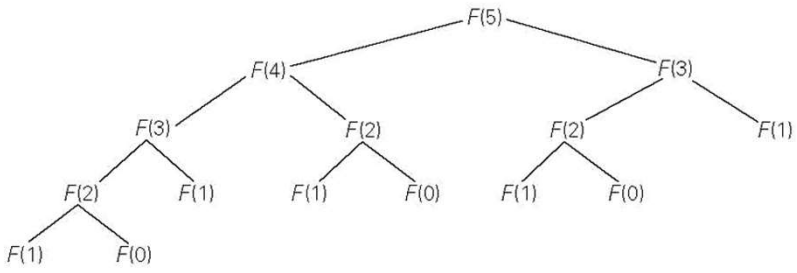


**FIGURE 2.6** Tree of recursive calls for computing the 5th Fibonacci number by the definition-based algorithm.

**ALGORITHM** *Fib*(*n*)

//Computes the *n*th Fibonacci number iteratively by using its definition

//Input: A nonnegative integer *n*

//Output: The *n*th Fibonacci number

$F[0] \leftarrow 0$; $F[1] \leftarrow 1$

for $i \leftarrow 2$ to $n$ do

$F[i] \leftarrow F[i-1] + F[i-2]$

return $F[n]$

Since a majority of dynamic programming applications deal with optimization problems, we also need to mention a general principle that underlines such applications. Richard Bellman called it the *principle of optimality*. In terms somewhat different from its original formulation, it says that an optimal solution to any instance of an optimization problem is composed of optimal solutions to its subinstances. The principle of optimality holds much more often than not. (To give a rather rare example, it fails for finding the longest simple path in a graph.) Although its applicability to a particular problem needs to be checked, of course, such a check is usually not a principal difficulty in developing a dynamic programming algorithm.

## 8.1 Three Basic Examples

**EXAMPLE 1** *Coin-row problem* There is a row of $n$ coins whose values are some positive integers $c_1, c_2, \ldots, c_n$, not necessarily distinct. The goal is to pick up the maximum amount of money subject to the constraint that no two coins adjacent in the initial row can be picked up.

Let $F(n)$ be the maximum amount that can be picked up from the row of $n$ coins. To derive a recurrence for $F(n)$, we partition all the allowed coin selections into two groups: those that include the last coin and those without it. The largest amount we can get from the first group is equal to $c_n + F(n-2)$—the value of the $n$th coin plus the maximum amount we can pick up from the first $n-2$ coins.

The maximum amount we can get from the second group is equal to $F(n-1)$ by the definition of $F(n)$. Thus, we have the following recurrence subject to the obvious initial conditions:

$$F(n) = \max\{c_n + F(n-1), F(n-1)\} \quad \text{for } n > 1, \qquad F(0) = 0, \quad F(1) = c_1. \tag{8.3}$$

We can compute $F(n)$ by filling the one-row table left to right in the manner similar to the way it was done for the $n$th Fibonacci number by Algorithm *Fib*$(n)$ in Section 2.5.

**ALGORITHM** *CoinRow*(*C*[1..*n*])

```
//Applies formula (8.3) bottom up to find the maximum amount of money
//that can be picked up from a coin row without picking two adjacent coins
//Input: Array C[1..n] of positive integers indicating the coin values
//Output: The maximum amount of money that can be picked up
F[0]←0; F[1]←C[1]
for i ←2 to n do
    F[i]←max(C[i]+ F[i − 2], F[i − 1])
return F[n]
```

The application of the algorithm to the coin row of denominations 5, 1, 2, 10, 6, 2 is shown in Figure 8.1. It yields the maximum amount of 17. It is worth pointing out that, in fact, we also solved the problem for the first $i$ coins in the row given for every $1 \le i \le 6$. For example, for $i = 3$, the maximum amount is $F(3) = 7$.

$F[0] = 0,\ F[1] = c_1 = 5$

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $C$ | | 5 | 1 | 2 | 10 | 6 | 2 |
| $F$ | 0 | 5 | | | | | |

$F[2] = \max\{1 + 0, 5\} = 5$

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $C$ | | 5 | 1 | 2 | 10 | 6 | 2 |
| $F$ | 0 | 5 | 5 | | | | |

$F[3] = \max\{2 + 5, 5\} = 7$

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $C$ | | 5 | 1 | 2 | 10 | 6 | 2 |
| $F$ | 0 | 5 | 5 | 7 | | | |

$F[4] = \max\{10 + 5, 7\} = 15$

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $C$ | | 5 | 1 | 2 | 10 | 6 | 2 |
| $F$ | 0 | 5 | 5 | 7 | 15 | | |

$F[5] = \max\{6 + 7, 15\} = 15$

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $C$ | | 5 | 1 | 2 | 10 | 6 | 2 |
| $F$ | 0 | 5 | 5 | 7 | 15 | 15 | |

$F[6] = \max\{2 + 15, 15\} = 17$

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $C$ | | 5 | 1 | 2 | 10 | 6 | 2 |
| $F$ | 0 | 5 | 5 | 7 | 15 | 15 | **17** |

**FIGURE 8.1** Solving the coin-row problem by dynamic programming for the coin row 5, 1, 2, 10, 6, 2.

**To find the coins with the maximum total value found, we need to backtrace the computations to see which of the two possibilities—$c_n + F(n - 2)$ or $F(n - 1)$—produced the maxima in formula (8.3). In the last application of the formula, it was the sum $c_6 + F(4)$, which means that the coin $c_6 = 2$ is a part of an optimal solution. Moving to computing $F(4)$, the maximum was produced by the sum $c_4 + F(2)$, which means that the coin $c_4 = 10$ is a part of an optimal solution as well. Finally, the maximum in computing $F(2)$ was produced by $F(1)$, implying that the coin $c_2$ is not the part of an optimal solution and the coin $c_1 = 5$ is.**

Thus, the optimal solution is $\{c_1, c_4, c_6\}$. To avoid repeating the same computations during the backtracing, the information about which of the two terms in (8.3) was larger can be recorded in an extra array when the values of $F$ are computed.

Using the *CoinRow* to find $F(n)$, the largest amount of money that can be picked up, as well as the coins composing an optimal set, clearly takes $\Theta(n)$ time and $\Theta(n)$ space. This is by far superior to the alternatives: the straightforward topdown application of recurrence (8.3) and solving the problem by exhaustive search (Problem 3 in this section's exercises).

**EXAMPLE 2** *Change-making problem* Consider the general instance of the following well-known problem. Give change for amount $n$ using the minimum number of coins of denominations $d_1 < d_2 < \ldots < d_m$. For the coin denominations used in the United States, as for those used in most if not all other countries, there is a very simple and efficient algorithm discussed in the next chapter. Here, we consider a dynamic programming algorithm for the general case, assuming availability of unlimited quantities of coins for each of the $m$ denominations $d_1 < d_2 < \ldots < d_m$ where $d_1 = 1$.

Let $F(n)$ be the minimum number of coins whose values add up to $n$; it is convenient to define $F(0) = 0$. The amount $n$ can only be obtained by adding one coin of denomination $d_j$ to the amount $n - d_j$ for $j = 1, 2, \ldots, m$ such that $n \geq d_j$ .Therefore, we can consider all such denominations and select the one Minimizing $F(n - d_j) + 1$. Since 1 is a constant, we can, of course, find the smallest $F(n - d_j)$ first and then add 1 to it. Hence, we have the following recurrence for $F(n)$:

$$F(n) = \min_{j:n \geq d_j} \{F(n - d_j)\} + 1 \quad \text{for } n > 0, \qquad F(0) = 0. \tag{8.4}$$

We can compute $F(n)$ by filling a one-row table left to right in the manner similar to the way it was done above for the coin-row problem, but computing a table entry here requires finding the minimum of up to $m$ numbers.

ALGORITHM *ChangeMaking*($D[1..m]$, $n$)

//Applies dynamic programming to find the minimum number of coins

//of denominations $d_1 < d_2 < \ldots < d_m$ where $d_1 = 1$ that add up to a

//given amount $n$

//Input: Positive integer $n$ and array $D[1..m]$ of increasing positive

// integers indicating the coin denominations where $D[1] = 1$

//Output: The minimum number of coins that add up to $n$

$F[0] \leftarrow 0$

```
for i ←1 to n do
    temp←∞; j ←1
    while j ≤ m and i ≥ D[j ] do
        temp ←min(F [i − D[j ]], temp)
        j ←j + 1
    F[i]←temp + 1
return F[n]
```

The application of the algorithm to amount $n = 6$ and denominations 1, 3, 4 is shown in Figure 8.2. The answer it yields is two coins. The time and space efficiencies of the algorithm are obviously $O(nm)$ and $\Theta(n)$, respectively.

$F[0] = 0$

| $n$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $F$ | 0 | | | | | | |

$F[1] = \min\{F[1-1]\} + 1 = 1$

| $n$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $F$ | 0 | 1 | | | | | |

$F[2] = \min\{F[2-1]\} + 1 = 2$

| $n$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $F$ | 0 | 1 | 2 | | | | |

$F[3] = \min\{F[3-1], F[3-3]\} + 1 = 1$

| $n$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $F$ | 0 | 1 | 2 | 1 | | | |

$F[4] = \min\{F[4-1], F[4-3], F[4-4]\} + 1 = 1$

| $n$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $F$ | 0 | 1 | 2 | 1 | 1 | | |

$F[5] = \min\{F[5-1], F[5-3], F[5-4]\} + 1 = 2$

| $n$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $F$ | 0 | 1 | 2 | 1 | 1 | 2 | |

$F[6] = \min\{F[6-1], F[6-3], F[6-4]\} + 1 = 2$

| $n$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $F$ | 0 | 1 | 2 | 1 | 1 | 2 | **2** |

**FIGURE 8.2** Application of Algorithm *MinCoinChange* to amount $n = 6$ and coin denominations 1, 3, and 4.

To find the coins of an optimal solution, we need to backtrace the computations to see which of the denominations produced the minima in formula (8.4). For the instance considered, the last application of the formula (for $n = 6$), the minimum was produced by $d_2 = 3$. The second minimum (for $n = 6 - 3$) was Also produced for a coin of that denomination. Thus, the minimum-coin set for $n = 6$ is two 3's.

**EXAMPLE 3** *Coin-collecting problem* Several coins are placed in cells of an $n \times m$ board, no more than one coin per cell. A robot, located in the upper left cell of the board, needs to collect as many of the coins as possible and bring them to the bottom right cell. On each step, the robot can move either one cell to the right or one cell down from its current location. When the robot visits a cell with a coin, it always picks up that coin. Design an algorithm to find the maximum number of coins the robot can collect and a path it needs to follow to do this.

Let $F(i, j)$ be the largest number of coins the robot can collect and bring to the cell $(i, j)$ in the $i$th row and $j$th column of the board. It can reach this cell either from the adjacent cell $(i - 1, j)$ above it or from the adjacent cell $(i, j - 1)$ to the left of it. The largest numbers of coins that can be brought to these cells are $F(i - 1, j)$ and $F(i, j - 1)$, respectively. Of course, there are no adjacent cells above the cells in the first row, and there are no adjacent cells to the left of the cells in the first column. For those cells, we assume that $F(i - 1, j)$ and $F(i, j - 1)$ are equal to 0 for their nonexistent neighbors. Therefore, the largest number of coins the robot can bring to cell $(i, j)$ is the maximum of these two numbers plus one possible coin at cell $(i, j)$ itself. In other words, we have the following formula for $F(i, j)$:

$$F(i, j) = \max\{F(i-1, j), F(i, j-1)\} + c_{ij} \quad \text{for } 1 \le i \le n,\ 1 \le j \le m$$
$$F(0, j) = 0 \text{ for } 1 \le j \le m \quad \text{and} \quad F(i, 0) = 0 \text{ for } 1 \le i \le n, \qquad (8.5)$$

where $c_{ij} = 1$ if there is a coin in cell $(i, j)$, and $c_{ij} = 0$ otherwise.

Using these formulas, we can fill in the $n \times m$ table of $F(i, j)$ values either row by row or column by column, as is typical for dynamic programming algorithms involving two-dimensional tables.

**ALGORITHM** *RobotCoinCollection*(*C*[1..*n*, 1..*m*])

//Applies dynamic programming to compute the largest number of
//coins a robot can collect on an $n \times m$ board by starting at (1, 1)
//and moving right and down from upper left to down right corner
//Input: Matrix *C*[1..*n*, 1..*m*] whose elements are equal to 1 and 0
//for cells with and without a coin, respectively
//Output: Largest number of coins the robot can bring to cell (*n*, *m*)

*F*[1, 1]←*C*[1, 1]; for *j* ←2 to *m* do *F*[1, *j*]←*F*[1, *j* − 1]+ *C*[1, *j*]
for *i* ←2 to *n* do
    *F*[*i*, 1]←*F*[*i* − 1, 1]+ *C*[*i*, 1]
    for *j* ←2 to *m* do
        *F*[*i*, *j* ]←max(*F* [*i* − 1, *j*], *F*[*i*, *j* − 1]) + *C*[*i*, *j* ]
return *F*[*n*, *m*]

The algorithm is illustrated in Figure 8.3b for the coin setup in Figure 8.3a. Since computing the value of *F*(*i*, *j*) by formula (8.5) for each cell of the table takes constant time, the time efficiency of the algorithm is $\Theta(nm)$. Its space efficiency is, obviously, also $\Theta(nm)$.

Tracing the computations backward makes it possible to get an optimal path:if $F(i-1, j) > F(i, j-1)$, an optimal path to cell $(i, j)$ must come down from the adjacent cell above it; if $F(i-1, j) < F(i, j-1)$, an optimal path to cell $(i, j)$ must come from the adjacent cell on the left; and if $F(i-1, j) = F(i, j-1)$, it can reach cell $(i, j)$ from either direction. This yields two optimal paths for the instance in Figure 8.3a, which are shown in Figure 8.3c. If ties are ignored, one optimal path can be obtained in $\Theta(n+m)$ time.

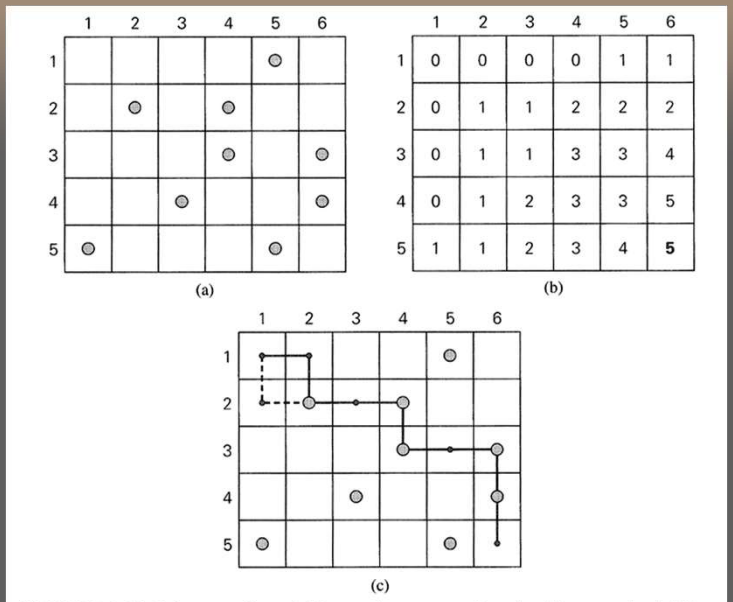


**FIGURE 8.3** (a) Coins to collect. (b) Dynamic programming algorithm results. (c) Two paths to collect 5 coins, the maximum number of coins possible.