Contoh-contoh persoalan lain yang diselesaikan dengan Dynamic Programming

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 nth 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:

$$F(n) = \max\{c_n + F(n-2),\ F(n-1)\} \quad \text{for } n > 1,$$
$$F(0) = 0, \qquad F(1) = c_1. \tag{8.3}$$

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 nth 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

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

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.

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 $F(3) = 7$.

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 top-

down application of recurrence (8.3) and solving the problem by exhaustive search (Problem 3 in this section's exercises). ■

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 ith row and jth 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 $F(i, j)$:

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

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 × 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.

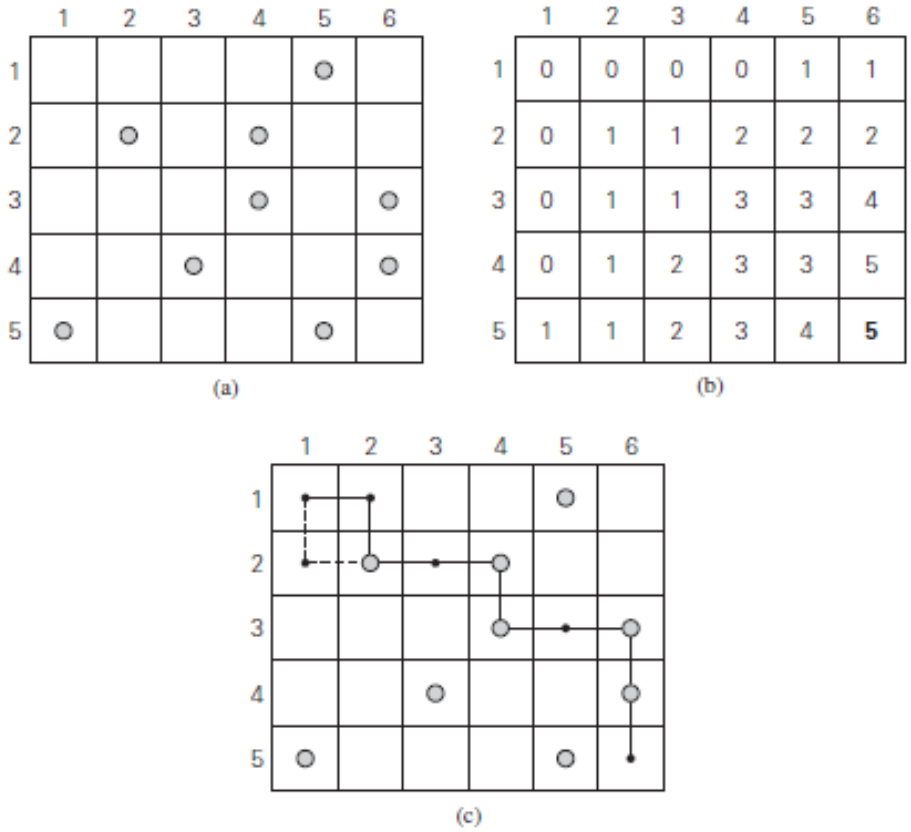

FIGURE 8.3 (a) Coins to collect. (b) Dynamic programming algorithm results. (c) Two paths to collect 5 coins, the maximum number of coins possible.

■

5.6 STRING EDITING

We are given two strings $X = x_1, x_2, \ldots, x_n$ and $Y = y_1, y_2, \ldots, y_m$, where x_i, $1 \leq i \leq n$, and y_j, $1 \leq j \leq m$, are members of a finite set of symbols known as the *alphabet*. We want to transform X into Y using a sequence of *edit operations* on X. The permissible edit operations are insert, delete, and change (a symbol of X into another), and there is a cost associated with performing each. The cost of a sequence of operations is the sum of the costs of the individual operations in the sequence. The problem of string editing is to identify a minimum-cost sequence of edit operations that will transform X into Y.

Let $D(x_i)$ be the cost of deleting the symbol x_i from X, $I(y_j)$ be the cost of inserting the symbol y_j into X, and $C(x_i, y_j)$ be the cost of changing the symbol x_i of X into y_j.

Example 5.19 Consider the sequences $X = x_1, x_2, x_3, x_4, x_5 = a, a, b, a, b$ and $Y = y_1, y_2, y_3, y_4 = b, a, b, b$. Let the cost associated with each insertion and deletion be 1 (for any symbol). Also let the cost of changing any symbol to any other symbol be 2. One possible way of transforming X into Y is delete each $x_i, 1 \leq i \leq 5$, and insert each $y_j, 1 \leq j \leq 4$. The total cost of this edit sequence is 9. Another possible edit sequence is delete x_1 and x_2 and insert y_4 at the end of string X. The total cost is only 3. □

A solution to the string editing problem consists of a sequence of decisions, one for each edit operation. Let $\mathcal{E}$ be a minimum-cost edit sequence for transforming X into Y. The first operation, O, in $\mathcal{E}$ is delete, insert, or change. If $\mathcal{E}' = \mathcal{E} - \{O\}$ and X' is the result of applying O on X, then $\mathcal{E}'$ should be a minimum-cost edit sequence that transforms X' into Y. Thus the principle of optimality holds for this problem. A dynamic programming solution for this problem can be obtained as follows. Define $cost(i, j)$ to be the minimum cost of any edit sequence for transforming $x_1, x_2, \ldots, x_i$ into $y_1, y_2, \ldots, y_j$ (for $0 \leq i \leq n$ and $0 \leq j \leq m$). Compute $cost(i, j)$ for each i and j. Then $cost(n, m)$ is the cost of an optimal edit sequence.

For $i = j = 0$, $cost(i, j) = 0$, since the two sequences are identical (and empty). Also, if $j = 0$ and $i > 0$, we can transform X into Y by a sequence of

deletes. Thus, $cost(i,0) = cost(i-1,0) + D(x_i)$. Similarly, if $i = 0$ and $j > 0$, we get $cost(0,j) = cost(0,j-1) + I(y_j)$. If $i \neq 0$ and $j \neq 0$, $x_1, x_2, \ldots, x_i$ can be transformed into $y_1, y_2, \ldots, y_j$ in one of three ways:

1. Transform $x_1, x_2, \ldots, x_{i-1}$ into $y_1, y_2, \ldots, y_j$ using a minimum-cost edit sequence and then delete x_i. The corresponding cost is $cost(i-1,j) + D(x_i)$.

2. Transform $x_1, x_2, \ldots, x_{i-1}$ into $y_1, y_2, \ldots, y_{j-1}$ using a minimum-cost edit sequence and then change the symbol x_i to y_j. The associated cost is $cost(i-1,j-1) + C(x_i, y_j)$.

3. Transform $x_1, x_2, \ldots, x_i$ into $y_1, y_2, \ldots, y_{j-1}$ using a minimum-cost edit sequence and then insert y_j. This corresponds to a cost of $cost(i, j-1) + I(y_j)$.

The minimum cost of any edit sequence that transforms $x_1, x_2, \ldots, x_i$ into $y_1, y_2, \ldots, y_j$ (for $i > 0$ and $j > 0$) is the minimum of the above three costs, according to the principle of optimality. Therefore, we arrive at the following recurrence equation for $cost(i,j)$:

$$cost(i,j) = \begin{cases} 0 & i = j = 0 \\ cost(i-1,0) + D(x_i) & j = 0,\ i > 0 \\ cost(0,j-1) + I(y_j) & i = 0,\ j > 0 \\ cost'(i,j) & i > 0,\ j > 0 \end{cases} \tag{5.13}$$

$$\text{where } cost'(i,j) = \min \{ \begin{array}{l} cost(i-1,j) + D(x_i), \\ cost(i-1,j-1) + C(x_i, y_j), \\ cost(i,j-1) + I(y_j) \ \} \end{array}$$

We have to compute $cost(i,j)$ for all possibles values of i and j ($0 \leq i \leq n$ and $0 \leq j \leq m$). There are $(n+1)(m+1)$ such values. These values can be computed in the form of a table, M, where each row of M corresponds to a particular value of i and each column of M corresponds to a specific value of j. $M(i,j)$ stores the value $cost(i,j)$. The zeroth row can be computed first since it corresponds to performing a series of insertions. Likewise the zeroth column can also be computed. After this, one could compute the entries of M in row-major order, starting from the first row. Rows should be processed in the order $1, 2, \ldots, n$. Entries in any row are computed in increasing order of column number.

The entries of M can also be computed in column-major order, starting from the first column. Looking at Equation 5.13, we see that each entry of M takes only $O(1)$ time to compute. Therefore the whole algorithm takes $O(mn)$ time. The value $cost(n,m)$ is the final answer we are interested in. Having computed all the entries of M, a minimum edit sequence can be

obtained by a simple backward trace from $cost(n, m)$. This backward trace is enabled by recording which of the three options for $i > 0, j > 0$ yielded the minimum cost for each i and j.

Example 5.20 Consider the string editing problem of Example 5.19. $X = a, a, b, a, b$ and $Y = b, a, b, b$. Each insertion and deletion has a unit cost and a change costs 2 units. For the cases $i = 0, j > 1$, and $j = 0, i > 1$, $cost(i, j)$ can be computed first (Figure 5.18). Let us compute the rest of the entries in row-major order. The next entry to be computed is $cost(1, 1)$.

$$\begin{aligned} cost(1,1) &= \min \{cost(0,1) + D(x_1), cost(0,0) + C(x_1, y_1), cost(1,0) + I(y_1)\} \\ &= \min \{2, 2, 2\} = 2 \end{aligned}$$

Next is computed $cost(1, 2)$.

$$\begin{aligned} cost(1,2) &= \min \{cost(0,2) + D(x_1), cost(0,1) + C(x_1, y_2), cost(1,1) + I(y_2)\} \\ &= \min \{3, 1, 3\} = 1 \end{aligned}$$

The rest of the entries are computed similarly. Figure 5.18 displays the whole table. The value $cost(5, 4) = 3$. One possible minimum-cost edit sequence is delete x_1, delete x_2, and insert y_4. Another possible minimum cost edit sequence is change x_1 to y_2 and delete x_4. □

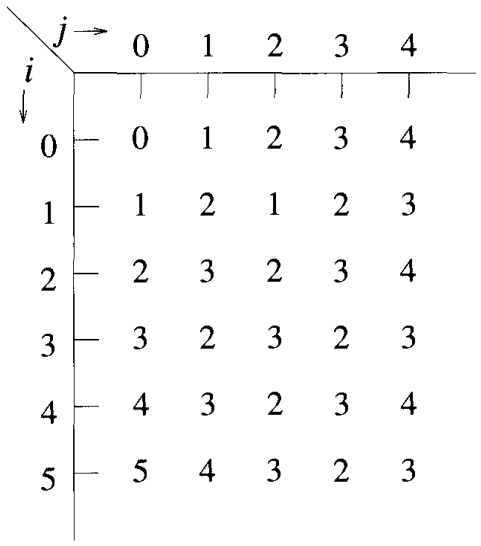

Figure 5.18 Cost table for Example 5.20

5.8 RELIABILITY DESIGN

In this section we look at an example of how to use dynamic programming to solve a problem with a multiplicative optimization function. The problem is to design a system that is composed of several devices connected in series (Figure 5.19). Let r_i be the reliability of device D_i (that is, r_i is the probability that device i will function properly). Then, the reliability of the entire system is Πr_i. Even if the individual devices are very reliable (the r_i's are very close to one), the reliability of the system may not be very good. For example, if $n = 10$ and $r_i = .99$, $1 \leq i \leq 10$, then $\Pi r_i = .904$. Hence, it is desirable to duplicate devices. Multiple copies of the same device type are connected in parallel (Figure 5.20) through the use of switching circuits. The switching circuits determine which devices in any given group are functioning properly. They then make use of one such device at each stage.

If stage i contains m_i copies of device D_i, then the probability that all m_i have a malfunction is $(1 - r_i)^{m_i}$. Hence the reliability of stage i becomes

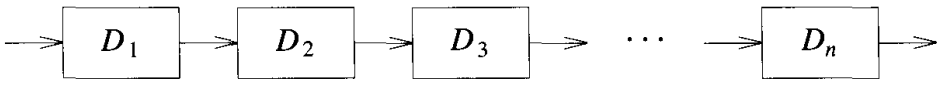

Figure 5.19 n devices D_i, $1 \leq i \leq n$, connected in series

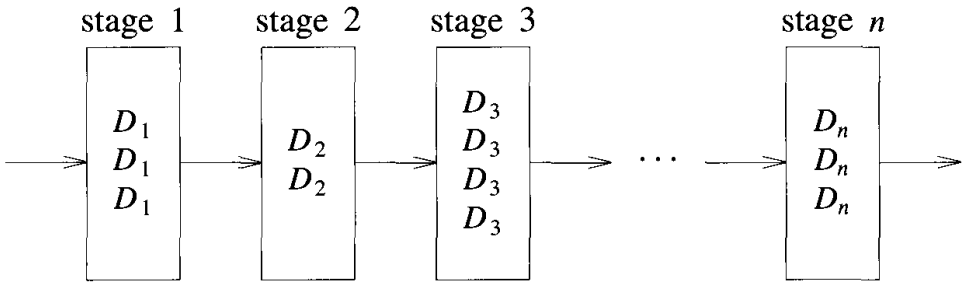

Figure 5.20 Multiple devices connected in parallel in each stage

$1-(1-r_i)^{m_i}$. Thus, if $r_i = .99$ and $m_i = 2$, the stage reliability becomes .9999. In any practical situation, the stage reliability is a little less than $1-(1-r_i)^{m_i}$ because the switching circuits themselves are not fully reliable. Also, failures of copies of the same device may not be fully independent (e.g., if failure is due to design defect). Let us assume that the reliability of stage i is given by a function $\phi_i(m_i)$, $1 \leq n$. (It is quite conceivable that $\phi_i(m_i)$ may decrease after a certain value of m_i.) The reliability of the system of stages is $\Pi_{1\leq i\leq n}\phi_i(m_i)$.

Our problem is to use device duplication to maximize reliability. This maximization is to be carried out under a cost constraint. Let c_i be the cost of each unit of device i and let c be the maximum allowable cost of the system being designed. We wish to solve the following maximization problem:

$$\text{maximize } \Pi_{1\leq i\leq n}\ \phi_i(m_i)$$

$$\text{subject to } \sum_{1\leq i\leq n} c_i m_i \leq c \tag{5.17}$$

$$m_i \geq 1 \text{ and integer, } 1 \leq i \leq n$$

A dynamic programming solution can be obtained in a manner similar to that used for the knapsack problem. Since, we can assume each $c_i > 0$, each m_i must be in the range $1 \leq m_i \leq u_i$, where

$$u_i = \left\lfloor (c + c_i - \sum_1^n c_j)/c_i \right\rfloor$$

The upper bound u_i follows from the observation that $m_j \geq 1$. An optimal solution $m_1, m_2, \ldots, m_n$ is the result of a sequence of decisions, one decision for each m_i. Let $f_i(x)$ represent the maximum value of $\Pi_{1 \leq j \leq i} \phi(m_j)$ subject to the constraints $\sum_{1 \leq j \leq i} c_j m_j \leq x$ and $1 \leq m_j \leq u_j$, $1 \leq j \leq i$. Then, the value of an optimal solution is $f_n(c)$. The last decision made requires one to choose m_n from $\{1, 2, 3, \ldots, u_n\}$. Once a value for m_n has been chosen, the remaining decisions must be such as to use the remaining funds $c - c_n m_n$ in an optimal way. The principal of optimality holds and

$$f_n(c) = \max_{1 \leq m_n \leq u_n} \{\phi_n(m_n) f_{n-1}(c - c_n m_n)\} \tag{5.18}$$

For any $f_i(x)$, $i \geq 1$, this equation generalizes to

$$f_i(x) = \max_{1 \leq m_i \leq u_i} \{\phi_i(m_i) f_{i-1}(x - c_i m_i)\} \tag{5.19}$$

Clearly, $f_0(x) = 1$ for all x, $0 \leq x \leq c$. Hence, (5.19) can be solved using an approach similar to that used for the knapsack problem. Let S^i consist of tuples of the form (f, x), where $f = f_i(x)$. There is at most one tuple for each different x that results from a sequence of decisions on $m_1, m_2, \ldots, m_n$. The dominance rule (f_1, x_1) dominates (f_2, x_2) iff $f_1 \geq f_2$ and $x_1 \leq x_2$ holds for this problem too. Hence, dominated tuples can be discarded from S^i.

Example 5.25 We are to design a three stage system with device types D_1, D_2, and D_3. The costs are \$30, \$15, and \$20 respectively. The cost of the system is to be no more than \$105. The reliability of each device type is .9, .8 and .5 respectively. We assume that if stage i has m_i devices of type i in parallel, then $\phi_i(m_i) = 1 - (1 - r_i)^{m_i}$. In terms of the notation used earlier, $c_1 = 30$, $c_2 = 15$, $c_3 = 20$, $c = 105$, $r_1 = .9$, $r_2 = .8$, $r_3 = .5, u_1 = 2, u_2 = 3$, and $u_3 = 3$.

We use S^i to represent the set of all undominated tuples (f, x) that may result from the various decision sequences for $m_1, m_2, \ldots, m_i$. Hence, $f(x) = f_i(x)$. Beginning with $S^0 = \{(1, 0)\}$, we can obtain each S^i from S^{i-1} by trying out all possible values for m_i and combining the resulting tuples together. Using S_j^i to represent all tuples obtainable from S^{i-1} by choosing $m_i = j$, we obtain $S_1^1 = \{(.9,\ 30)\}$ and $S_2^1 = \{(.9,\ 30), (.99, 60)\}$. The set

$S_1^2 = \{(.72, 45), (.792, 75)\}$; $S_2^2 = \{(.864, 60)\}$. Note that the tuple (.9504, 90) which comes from (.99, 60) has been eliminated from S_2^2 as this leaves only \$10. This is not enough to allow $m_3 = 1$. The set $S_3^2 = \{(.8928, 75)\}$. Combining, we get $S^2 = \{(.72, 45), (.864, 60), (.8928, 75)\}$ as the tuple (.792, 75) is dominated by (.864, 60). The set $S_1^3 = \{(.36, 65), (.432, 80), (.4464, 95)\}$, $S_2^3 = \{(.54, 85), (.648, 100)\}$, and $S_3^3 = \{(.63, 105)\}$. Combining, we get $S^3 = \{(.36, 65), (.432, 80), (.54, 85), (.648, 100)\}$.

The best design has a reliability of .648 and a cost of 100. Tracing back through the S^i's, we determine that $m_1 = 1, m_2 = 2$, and $m_3 = 2$. □

As in the case of the knapsack problem, a complete dynamic programming algorithm for the reliability problem will use heuristics to reduce the size of the S^i's. There is no need to retain any tuple (f, x) in S^i with x value greater that $c - \sum_{i \leq j \leq n} c_j$ as such a tuple will not leave adequate funds to complete the system. In addition, we can devise a simple heuristic to determine the best reliability obtainable by completing a tuple (f, x) in S^i. If this is less than a heuristically determined lower bound on the optimal system reliability, then (f, x) can be eliminated from S^i.