8.2 Coin Changing Revisited

In Section 7.1, we developed a greedy algorithm for the coin-changing problem in which the goal was to make change for an amount A using the fewest number of coins, where the available denominations were

$$denom[1] > denom[2] > \cdots > denom[n] = 1.$$

We saw that whether the greedy algorithm produced the fewest number of coins depended on which denominations of coins were available. In this section, we develop a dynamic programming algorithm for the coin-changing problem that produces the fewest number of coins no matter which denominations are available.

To break the given problem into subproblems, we vary the amount and restrict the denominations available. More precisely, we consider the problem of computing the minimum number of coins for an amount j, $0 \le j \le A$, where the available denominations are

$$denom[i] > denom[i+1] > \cdots > denom[n] = 1,$$

$1 \le i \le n$. We call this the *i, j-problem*. The original problem is $i = 1$ and $j = A$. We let $C[i][j]$ denote the solution to the i, j-problem; that is, $C[i][j]$ is the minimum number of coins to make change for the amount j, using coins i through n (see Figure 8.2.1). (We later address the problem of determining which coins achieve the minimum.)

								j						
		0	1	2	3	4	5	6	7	8	9	10	11	12
	1	0	1	2	3	4	5	1	2	3	4	1	2	2
i	2	0	1	2	3	4	5	1	2	3	4	5	6	2
	3	0	1	2	3	4	5	6	7	8	9	10	11	12

Figure 8.2.1 The array C for the denominations

$$denom[1] = 10, \quad denom[2] = 6, \quad denom[3] = 1,$$

and amounts up to 12. The index i specifies that coins i through 3 are available, and j is the amount. When $i = 3$, only the coin of denomination 1 is available. Thus, in the last row, it takes j coins to make change for the amount j. When $i = 2$, the coins 6 and 1 are available. For example, the minimum number of coins to make change for the amount 8 is three—one coin of denomination 6 and two coins of denomination 1. When $i = 1$, all of the coins are available. For example, the minimum number of coins to make change for the amount 11 is two—one coin of denomination 10 and one coin of denomination 1.

To solve the i, j-problem, $i < n$, we must decide whether to use a coin of denomination $denom[i]$. If we do *not* use a coin of denomination $denom[i]$, in order to achieve the amount j, we must solve the $(i+1), j$ problem. Since this is a subproblem (i.e., a smaller problem in the sense that fewer coins are available), we can design our algorithm so that this subproblem is already solved. Thus, given that we do not use coin i, the minimum number of coins to make change for the amount j is $C[i+1][j]$.

On the other hand, if we use a coin of denomination $\mathit{denom}[i]$, we must complete the solution by making change for the amount $j - \mathit{denom}[i]$ using coins of denominations

$$\mathit{denom}[i] > \mathit{denom}[i+1] > \cdots > \mathit{denom}[n] = 1.$$

If we use, say, k coins for the amount $j - \mathit{denom}[i]$, our solution to the problem of making change for the amount j uses $1 + k$ coins (since we already used one coin of denomination $\mathit{denom}[i]$). To minimize $1 + k$, we must choose k as small as possible. In other words, we must use the minimum number of coins to solve the subproblem of making change for the amount $j - \mathit{denom}[i]$ using coins i through n. (This is an example of the *optimal substructure property*, which we discuss thoroughly later in this section.) Therefore, we must solve the $i, (j - \mathit{denom}[i])$-problem. Since this is also a subproblem (i.e., a smaller problem in the sense the amount is smaller than the original amount), we can also design our algorithm so that this subproblem is already solved. Thus, given that we do use coin i, the minimum number of coins to make change for the amount j is $1 + C[i][j - \mathit{denom}[i]]$.

Now, either we use coin i or we don't! Thus, the solution to the i, j problem is

$$C[i][j] = \begin{cases} C[i+1][j] & \text{if } \mathit{denom}[i] > j \\ \min\{C[i+1][j], 1 + C[i][j - \mathit{denom}[i]]\} & \text{if } \mathit{denom}[i] \le j. \end{cases} \tag{8.2.1}$$

Contrast the dynamic programming algorithm outlined in the preceding paragraphs with the greedy algorithm in Section 7.1. When the greedy algorithm considers using a denomination d for an amount j, if $d \le j$, it uses it—without regard to the consequences of solving the smaller problem of making change for the amount $d - j$. For example, if the available denominations are $10, 6, 1$ and the amount is 12, the greedy algorithm chooses one 10—without regard to the fact that the resulting problem of making change for the amount 2 requires two coins. The greedy algorithm thus chooses one 10 and two 1's. On the other hand, the dynamic programming algorithm considers using denomination d and not using it and picks the better alternative. In this sense, dynamic programming can be considered an enhancement of the greedy technique. Again, if the available denominations are $10, 6, 1$ and the amount is 12, the dynamic programming algorithm *considers* choosing a 10. Since this leaves the problem of making change for the amount 2, *if* the dynamic programming algorithm chooses a 10, three coins will be required to make change for the amount 12: one 10 and two 1's. The dynamic programming algorithm also considers not choosing a 10. In this case, the algorithm will make change for the amount 12 using only the denominations

1 and 6, and the optimal choice is to use two 6's. Since choosing two 6's results in fewer coins than using a 10, the dynamic programming algorithm chooses two 6's.

Our dynamic programming algorithm begins by computing $C[n][j]$ for $j = 0$ to the specified amount A. Since only the coin of denomination 1 is available,

$$C[n][j] = j \tag{8.2.2}$$

for $j = 0$ to A. After computing $C[i+1][j]$ for all j, it computes $C[i][j]$ in the order $j = 0$ to A using equation (8.2.1).

Algorithm 8.2.1 Coin Changing Using Dynamic Programming, Version 1. This dynamic programming algorithm computes the minimum number of coins to make change for a given amount. The input is an array *denom* that specifies the denominations of the coins,

$$denom[1] > denom[2] > \cdots > denom[n] = 1,$$

and an amount A. The output is an array C whose value, $C[i][j]$, is the minimum number of coins to make change for the amount j, using coins i through n, $1 \le i \le n$, $0 \le j \le A$.

```
 Input Parameters:   $denom, A$
Output Parameter:    $C$

$dynamic\_coin\_change1(denom, A, C)$ {
   $n = denom.last$
   for $j = 0$ to $A$
      $C[n][j] = j$
   for $i = n - 1$ downto 1
      for $j = 0$ to $A$
         if $(denom[i] > j \,||\, C[i+1][j] < 1 + C[i][j - denom[i]])$
            $C[i][j] = C[i+1][j]$
         else
            $C[i][j] = 1 + C[i][j - denom[i]]$
}
```

In Algorithm 8.2.1, the first for loop runs in time $\Theta(A)$ and the nested for loops run in time $\Theta(nA)$. Therefore, the run time of Algorithm 8.2.1 is $\Theta(nA)$.

As written, Algorithm 8.2.1 determines the minimum number of coins but does not tell us which coins to use to achieve the minimum. We can determine which coins to use by adding a statement to Algorithm 8.2.1 that records whether coin i is used to make change for the amount j. (We can determine which coins to use without an auxiliary array by examining the array C; see Exercise 7. Here we prefer to show how to use an auxiliary array since this method has wider applicability.)

Algorithm 8.2.2 Coin Changing Using Dynamic Programming, Version 2. This dynamic programming algorithm computes the minimum number of

coins to make change for a given amount and tracks which coins are used. The input is an array *denom* that specifies the denominations of the coins,

$$denom[1] > denom[2] > \cdots > denom[n] = 1,$$

and an amount A. The output consists of arrays C and *used*. The value, $C[i][j]$, is the minimum number of coins to make change for the amount j, using coins i through n. The value, $used[i][j]$, is true or false to signify whether coin i appears in the smallest set of coins computed by Algorithm 8.2.1 for the amount j using only coins i through n. The values of i and j satisfy $1 \le i \le n$ and $0 \le j \le A$.

```
  Input Parameters:   denom, A
Output Parameters:    C, used

dynamic_coin_change2(denom, A, C, used) {
   n = denom.last
   for j = 0 to A {
      C[n][j] = j
      used[n][j] = true
   }
   for i = n - 1 downto 1
      for j = 0 to A
         if (denom[i] > j || C[i + 1][j] < 1 + C[i][j - denom[i]]) {
            C[i][j] = C[i + 1][j]
            used[i][j] = false
         }
         else {
            C[i][j] = 1 + C[i][j - denom[i]]
            used[i][j] = true
         }
}
```

Example 8.2.3. The *used* array computed by Algorithm 8.2.2 for the denominations 10, 6, 1 and amount $A = 12$ is shown in Figure 8.2.2. □

We can now write an algorithm to output a minimum size set of coins chosen from among coins i through n for an amount j. The algorithm inputs the index i, the amount j, the array *denom* of Algorithm 8.2.2, and the array *used* computed by Algorithm 8.2.2.

Algorithm 8.2.4 Computing a Minimum Size Set of Coins for a Given Amount. This algorithm outputs a minimum size set of coins to make change for an amount j using any of coins i through n with denominations specified by Algorithm 8.2.2. The algorithm inputs the index i, the amount j, the array *denom* of Algorithm 8.2.2, and the array *used* computed by Algorithm 8.2.2.

```
  Input Parameters:   i, j, denom, used
Output Parameters:    None
```

	j													
		0	1	2	3	4	5	6	7	8	9	10	11	12
	1	F	F	F	F	F	F	F	F	F	F	T	T	F
i	2	F	F	F	F	F	F	T	T	T	T	T	T	T
	3	T	T	T	T	T	T	T	T	T	T	T	T	T

Figure 8.2.2 The *used* array computed by Algorithm 8.2.2 for the denominations $10, 6, 1$ and amount $A = 12$. $used[i][j]$ is true (T) or false (F) to signify whether coin i appears in the smallest set of coins computed by Algorithm 8.2.1 for the amount j using only coins i through n. For example, $used[1][11]$ is true because the minimum set of coins $\{10, 1\}$ for the amount 11 uses a coin of denomination 10. By contrast, $used[1][12]$ is false because the minimum set of coins $\{6, 6\}$ for the amount 12 does not use a coin of denomination 10.

```
optimal_coins_set(i, j, denom, used) {
  if (j == 0)
    return
  if (used[i][j]) {
    println("Use a coin of denomination " + denom[i])
    optimal_coins_set(i, j - denom[i], denom, used)
  }
  else
    optimal_coins_set(i + 1, j, denom, used)
}
```

Algorithm 8.2.4 terminates correctly. Each time the algorithm is called with $j > 0$, either i is incremented by 1 or j is decremented at least by 1. If j becomes 0 before $i = n$, the algorithm terminates correctly. If $i = n$ and $j > 0$, $used[i][j]$ is true so j continually decrements by 1 until it is 0, and the algorithm also terminates correctly.

When Algorithm 8.2.4 is called as

$$optimal\_coins\_set(1, A, denom, used)$$

j can be decremented at most A times and i can be incremented at most $n - 1$ times. Thus, the run time of Algorithm 8.2.4 with $i = 1$ and $j = A$, which outputs a minimum size set of coins chosen from among all available denominations to make change for the amount A, is $O(n + A)$.

Constructing a Dynamic Programming Algorithm

To construct a dynamic programming algorithm, the first step is to break the given problem into subproblems using *parameters* to characterize the subproblems. The solution to the original problem will be built from these subproblems. The parameters control the size of the subproblems, and small problem sizes *must* be included. The dynamic programming algorithm will begin by solving small subproblems and end with a solution to the original problem. For example, to compute the nth Fibonacci number f_n (see Section

8.1), f_n is given in terms of smaller subproblems f_{n-1} and f_{n-2}. The parameter is the subscript. In this section, to compute the minimum number of coins, we broke the problem of making change for an amount A using the fewest number of coins, where the available denominations are

$$denom[1] > denom[2] > \cdots > denom[n] = 1$$

into subproblems of making change for an amount j using the fewest number of coins, where the available denominations are

$$denom[i] > denom[2] > \cdots > denom[n] = 1.$$

The parameter i characterizes the number of coins available, and the parameter j varies the amount.

After defining the subproblems, we define the desired quantity to be computed in terms of the parameters. To compute the Fibonacci sequence, the quantity to compute was, in effect, already defined—namely f_n. For the coin-changing problem, we defined $C[i][j]$ to be the minimum number of coins for the i, j-problem.

The next step is to obtain initial conditions and a recurrence relation for the desired quantity. To compute the nth Fibonacci number, we used the recurrence relation

$$f_n = f_{n-1} + f_{n-2}$$

valid for $n \geq 3$, and initial conditions

$$f_1 = f_2 = 1.$$

To compute the minimum number of coins, $C[i][j]$, we used the recurrence relation (8.2.1) and initial conditions (8.2.2).

A dynamic programming algorithm computes the values of the sequence defined by the recurrence relation and initial conditions. It does so bottom up; that is, it first uses the initial conditions to compute the trivial cases. It then uses the recurrence relation to compute the next easiest cases, then the next easiest cases, and so on, until it computes the solution to the original problem.

To compute the nth Fibonacci number, the dynamic programming algorithm first computed f_1 and f_2. The algorithm then computed f_3, then f_4, and so on, until it computed the solution f_n to the original problem.

To compute the minimum number of coins, the dynamic programming algorithm first computed $C[n][j]$ for all j using the initial conditions (one denomination available). The algorithm then computed the next easiest cases $C[n-1][j]$ for all j using the recurrence relation (two denominations available). It then computed the next easiest cases $C[n-2][j]$ for all j (three denominations available), and so on, until it computed the solution $C[1][A]$ to the original problem (all denominations available).

Dynamic programming is most often used to solve an optimization problem. An *optimization problem* is a problem that asks for the largest or smallest value meeting some specified criteria. To compute the instance that gives an optimal solution, the algorithm may track the indexes that lead to optimal solutions of subproblems. For example, to compute the minimum number of coins and to construct a set of coins that gives this minimum value, the dynamic programming algorithm tracked the coins that gave the minimum values (see Algorithm 8.2.2).

The Optimal Substructure Property

The **optimal substructure property** is

If S is an optimal solution to a problem, then the components of S are optimal solutions to subproblems.

In order for a dynamic programming algorithm to solve an optimization problem correctly, the optimal substructure property *must* hold.

Example 8.2.5. The optimal substructure property holds for the coin-changing problem. Given available denominations of coins, if S is a minimum size set of coins for an amount A and we remove one coin of denomination d from S, then S, with this coin removed, is a minimum size set of coins for the amount $A - d$. □

The optimal substructure property does *not* say that if S_1 and S_2 are optimal solutions to subproblems, then combining S_1 and S_2 gives an optimal solution to the original problem. This is the *converse* of the optimal substructure property.

Example 8.2.6. Suppose that the available denominations are $10, 6, 1$ for the coin-changing problem. One coin of denomination 1 and one coin of denomination 6 give an optimal solution for the amount 7. Five coins of denomination 1 give an optimal solution for the amount 5. Combining these solutions, which results in six coins of denomination 1 and one coin of denomination 6, does *not* yield an optimal solution for the amount 12.

Thus, for the coin-changing problem, if S_1 and S_2 are optimal solutions to subproblems, combining S_1 and S_2 does not necessarily give an optimal solution to the original problem; the converse of the optimal substructure property does *not* hold for the coin-changing problem. As we observed earlier, the optimal substructure property *does* hold for this problem. □

We close with an example of a problem for which the optimal substructure property does *not* hold.

Example 8.2.7. We show that the optimal substructure property does *not* hold for the *longest simple path problem*: Given a connected, weighted graph G and vertices v and w in G, find a longest simple path in G from v to w (a *simple path* is a path with no repeated vertices). Assume that all of the weights in G are positive.

By inspection, a longest simple path in the graph of Figure 8.2.3 from vertex 1 to vertex 4 is $1, 3, 4$. *If* the optimal substructure property holds for the longest simple path problem, then $1, 3$ is a longest simple path from 1 to

3. But 1, 3 is *not* a longest simple path from 1 to 4 because 1, 2, 4, 3 is longer. Therefore, the optimal substructure property does not hold for the longest simple path problem.

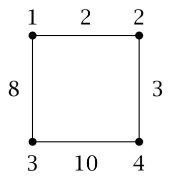

Figure 8.2.3 A graph that shows that the optimal substructure property does not hold for the longest simple path problem. A longest simple path from vertex 1 to vertex 4 is 1, 3, 4, but 1, 3 is *not* a longest simple path from vertex 1 to vertex 3.

It can be shown that the longest simple path problem is NP-complete and is, therefore, unlikely to have a polynomial-time algorithm. □

Exercises

In Exercises 1–3, show the array C computed by Algorithm 8.2.1 for the given denominations and the amount $A = 12$.

1S. 10, 5, 1 **2.** 10, 3, 1 **3.** 10, 7, 1

In Exercises 4–6, show the array used computed by Algorithm 8.2.2 for the given denominations and the amount $A = 12$.

4S. 10, 5, 1 **5.** 10, 4, 1 **6.** 10, 3, 1

7S. Write an algorithm whose input is an index $i \le n$, an amount $j \le A$, the array *denom*, and the array C, and whose output is a minimal set of coins to make change for the amount j. Algorithm 8.2.1 describes i, n, A, *denom*, and C. What is the worst-case time of your algorithm?

8. Write a version of Algorithm 8.2.1 in which the output is a *one*-dimensional array C' in which $C'[j]$ is equal to the minimum number of coins for the amount j using all available coins (i.e., $C'[j] = C[1][j]$ for all j, where C is the output of Algorithm 8.2.1). Your algorithm must use only $O(n)$ storage cells. What is the worst-case time of your algorithm?

9. Show that the optimal substructure problem holds for the shortest-path problem: Given a connected, weighted graph G and vertices v and w in G, find a shortest path in G from v to w. Assume that all of the weights in G are positive.

10S. Show that the converse of the optimal substructure property does not hold for the shortest-path problem.

11. Does the converse of the optimal substructure property hold for the longest simple path problem?