Chapter 12

Optimization II: Dynamic Programming

In the last chapter, we saw that greedy algorithms are efficient solutions to certain optimization problems. However, there are optimization problems for which no greedy algorithm exists. In this chapter, we will examine a more general technique, known as dynamic programming, for solving optimization problems.

Dynamic programming is a technique of implementing a top-down solution using bottom-up computation. We have already seen several examples of how top-down solutions can be implemented bottom-up. Dynamic programming extends this idea by saving the results of many subproblems in order to solve the desired problem. As a result, dynamic programming algorithms tend to be more costly, in terms of both time and space, than greedy algorithms. On the other hand, they are often much more efficient than straightforward recursive implementations of the top-down solution. Thus, when greedy algorithms are not possible, dynamic programming algorithms are often the most appropriate.

12.1 Making Change

Suppose we wish to produce a specific value $n \in \mathbb{N}$ from a given set of coin denominations $d_1 < d_2 < \cdots < d_k$, each of which is a positive integer. Our goal is to achieve a value of exactly $n$ using a minimum number of coins. To ensure that it is always possible to achieve a value of exactly $n$, we assume that $d_1 = 1$ and that we have as many coins in each denomination as we need.
An obvious greedy strategy is to choose at each step the largest coin that does not cause the total to exceed \( n \). For some sets of coin denominations, this strategy will result in the minimum number of coins for any \( n \). However, suppose \( n = 30, d_1 = 1, d_2 = 10, \) and \( d_3 = 25 \). The greedy strategy first takes 25. At this point, the only denomination that does not cause the total to exceed \( n \) is 1. The greedy strategy therefore gives a total of six coins: one 25 and five 1s. This solution is not optimal, however, as we can produce 30 with three 10s.

Let us consider a more direct top-down solution. If \( k = 1 \), then \( d_k = 1 \), so the only solution contains \( n \) coins. Otherwise, if \( d_k > n \), we can reduce the size of the problem by removing \( d_k \) from the set of denominations, and the solution to the resulting problem is the solution to the original problem. Finally, suppose \( d_k \leq n \). There are now two possibilities: the optimal solution either contains \( d_k \) or it does not. In what follows, we consider these two cases separately.

Let us first consider the case in which the optimal solution does not contain \( d_k \). In this case, we do not change the optimal solution if we remove \( d_k \) from the set of denominations. We therefore have reduced the original problem to a smaller problem instance.

Now suppose the optimal solution contains \( d_k \). Suppose we remove one \( d_k \) coin from this optimal solution. What remains is an optimal solution to the instance with the same set of denominations and a target value of \( n - d_k \). Now working in the other direction, if we have the optimal solution to the smaller instance, we can obtain an optimal solution to the original instance by adding a \( d_k \) coin. Again, we have reduced the original problem to a smaller problem instance.

To summarize, when \( d_k \leq n \), the optimal solution can be obtained from the optimal solution to one of two smaller problem instances. We have no way of knowing in advance which of these smaller instances is the right one; however, if we obtain both of them, we can compare the two resulting candidate solutions. The one with fewer coins is the optimal solution. In fact, if we could quickly determine which of these smaller instances would yield fewer coins, we could use this test as the selection criterion for a greedy algorithm. Therefore, let us focus for now on the more difficult aspect of this problem — that of determining the minimum number of coins in an optimal solution.

Based on the above discussion, the following recurrence gives the minimum number of coins needed to obtain a value of \( n \) from the denominations...
CHAPTER 12. OPTIMIZATION II: DYNAMIC PROGRAMMING

\[ d_1, \ldots, d_k: \]

\[ C(n, k) = \begin{cases} 
  n & \text{if } k = 1 \\
  C(n, k - 1) & \text{if } k > 1, \ d_k > n \\
  \min(C(n, k - 1), C(n - d_k, k) + 1) & \text{if } k > 1, \ n \geq d_k.
\end{cases} \quad (12.1) \]

This recurrence gives us a recursive algorithm for computing \( C(n, k) \). However, the direct recursive implementation of this recurrence is inefficient.

In order to see this, let us consider the special case in which \( d_i = i \) for \( 1 \leq i \leq k \) and \( n \geq k^2 \). Then for \( k > 1 \), the computation of \( C(n, k) \) requires the computation of \( C(n, k - 1) \) and \( C(n - k, k) \). The computation of \( C(n - k, k) \) then requires the computation of \( C(n - k, k - 1) \). Furthermore,

\[
n - k \geq k^2 - k \\
= k(k - 1) \\
\geq (k - 1)^2
\]

for \( k \geq 1 \). Thus, when \( n \geq k^2 \), the computation of \( C(n, k) \) requires the computation of two values, \( C(n_1, k - 1) \) and \( C(n_2, k - 1) \), where \( n_1 \geq (k - 1)^2 \) and \( n_2 \geq (k - 1)^2 \). It is then easily shown by induction on \( k \) that \( C(n, k) \) requires the computation of \( 2^{k-1} \) values \( C(n_i, k) \), where \( n_i \geq 1 \) for \( 1 \leq i \leq 2^{k-1} \). In such cases, the running time is exponential in \( k \).

A closer look at the above argument reveals that a large amount of redundant computation is taking place. For example, the subproblem \( C(n - 2k + 2, k - 2) \) must be computed twice:

1. \( C(n, k) \) requires the computation of \( C(n, k - 1) \), which requires \( C(n - k + 1, k - 1) \), which requires \( C(n - 2k + 2, k - 1) \), which requires \( C(n - 2k + 2, k - 2) \); and

2. \( C(n, k) \) also requires \( C(n - k, k) \), which requires \( C(n - k, k - 1) \), which requires \( C(n - k, k - 2) \), which requires \( C(n - 2k + 2, k - 2) \).

Applying this reasoning again to both computations of \( C(n - 2k + 2, k - 2) \), we can see that \( C(n - 4k + 8, k - 4) \) must be computed four times. More generally, for even \( i < k \), \( C(n - ik + i^2/2, k - i) \) must be computed \( 2^{i/2} \) times for \( n \geq k^2 \).

In order to avoid the redundant computation that leads to exponential running time, we can compute all values \( C(i, j) \) for \( 0 \leq i \leq n \), \( 1 \leq j \leq k \), saving them in an array. If we compute recurrence (12.1) bottom-up, rather than top-down, we will have all of the values we need in order to compute...
each \( C(i, j) \) in constant time. All \((n + 1)k\) of these values can therefore be computed in \( \Theta(nk) \) time. Once all of these values have been computed, then the optimal collection of coins can be constructed in a greedy fashion, as suggested above. The algorithm is shown in Figure 12.1. This algorithm is easily seen to use \( \Theta(nk) \) time and space.

A characteristic of this problem that is essential in order for the dynamic programming approach to work is that it is possible to decompose a large problem instance into smaller problem instances in a way that optimal solutions to the smaller instances can be used to produce an optimal solution to the larger instance. This is, of course, one of the main principles of the top-down approach. However, this characteristic may be stated succinctly for optimization problems: For any optimal solution, any portion of that solution is itself an optimal solution to a smaller instance. This principle is known as the principle of optimality. It applies to the change-making problem because any sub-collection of an optimal collection of coins is itself an optimal collection for the value it yields; otherwise, we could replace the sub-collection with a smaller sub-collection yielding the same value, and obtain a better solution to the original instance.

The principle of optimality usually applies to optimization problems, but not always in a convenient way. For example, consider the problem of finding a longest simple path in a graph from a given vertex \( u \) to a given vertex \( v \). If we take a portion of the longest path, say from \( x \) to \( y \), this subpath is not necessarily the longest simple path from \( x \) to \( y \) in the original graph. However, it is guaranteed to be the longest simple path from \( x \) to \( y \) in the subgraph consisting of only those vertices on that subpath and all edges between them in the original graph. Thus, a subproblem consists of a start vertex, a final vertex, and a subset of the vertices. Because a graph with \( n \) vertices has \( 2^n \) subsets of vertices, there are an exponential number of subproblems to solve. Thus, in order for dynamic programming to be an effective design technique, the principle of optimality must apply in a way that yields relatively few subproblems.

One characteristic that often leads to relatively few subproblems, while at the same time causing direct recursive implementations to be quite expensive, is that the top-down solution results in overlapping subproblems. As we have already discussed, the top-down solution for the change-making problem can result in two subproblems which have a subproblem in common. This overlap results in redundant computation in the direct recursive implementation. On the other hand, it reduces the total number of subproblems, so that the dynamic programming approach is more efficient.
Figure 12.1 Algorithm for computing the minimum number of coins needed
to achieve a given value

Precondition: $d[1..k]$ is an array of Ints such that $1 = d[1] < d[2] < \cdots < d[k]$, and $n$ is a Nat.
Postcondition: Returns an array $A[1..k]$ such that $A[i]$ gives the number
of coins of denomination $d[i]$ in a minimum-sized collection of coins with
value $n$.

```plaintext
CHANGE(d[1..k], n)
    C ← new Array[0..n, 1..k]; A ← new Array[1..k]
    for i ← 0 to n
        C[i, 1] ← i
    for i ← 0 to n
        for j ← 2 to k
            if i < d[j]
                C[i, j] ← C[i, j − 1]
            else
                C[i, j] ← Min(C[i, j − 1], C[i − d[j], j] + 1)
    for j ← 1 to k
        A[j] ← 0
    i ← n; j ← k
    // Invariant: $\sum_{l=1}^{k} A[l]d[l] = n - i$, and there is an optimal solution
    // that includes all of the coins in $A[1..k]$, but no additional coins from
    // $d[j+1..k]$. 
    while j > 1
        if i < d[j] or C[i, j − 1] < C[i − d[j], j] + 1
            j ← j − 1
        else
    A[1] ← i
    return A[1..k]
```
12.2 Chained Matrix Multiplication

Recall that the product $AB$, where $A$ is a $k \times m$ matrix and $B$ is an $m \times n$ matrix, is the $k \times n$ matrix $C$ such that

$$C_{ij} = \sum_{l=1}^{m} A_{il}B_{lj} \quad \text{for } 1 \leq i \leq k, 1 \leq j \leq n.$$ 

If we were to compute the matrix product by directly computing each of the $kn$ sums, we would perform a total of $kmn$ scalar multiplications.

Now suppose we wish to compute the product,

$$M_1M_2 \cdots M_n,$$

where $M_i$ is a $d_{i-1} \times d_i$ matrix for $1 \leq i \leq n$. Because matrix multiplication is associative, we have some choice over the order in which the multiplications are performed. For example, to compute $M_1M_2M_3$, we may either

- first compute $M_1M_2$, then multiply on the right by $M_3$; or
- first compute $M_2M_3$, then multiply on the left by $M_1$.

In other words, we may compute either $(M_1M_2)M_3$ or $M_1(M_2M_3)$.

Now suppose $d_0 = 2$, $d_1 = 3$, $d_2 = 4$, and $d_3 = 1$. Then the three matrices are dimensioned as follows:

- $M_1$: $2 \times 3$;
- $M_2$: $3 \times 4$; and
- $M_3$: $4 \times 1$.

If we compute $(M_1M_2)M_3$, we first multiply a $2 \times 3$ matrix by a $3 \times 4$ matrix, then we multiply the resulting $2 \times 4$ matrix by a $4 \times 1$ matrix. The total number of scalar multiplications is

$$2 \cdot 3 \cdot 4 + 2 \cdot 4 \cdot 1 = 32.$$ 

On the other hand, if we compute $M_1(M_2M_3)$, we first multiply a $3 \times 4$ matrix by a $4 \times 1$ matrix, then we multiply the resulting $3 \times 1$ matrix by a $2 \times 3$ matrix. The total number of scalar multiplications is

$$3 \cdot 4 \cdot 1 + 2 \cdot 3 \cdot 1 = 18.$$
Thus, the way in which the matrices are parenthesized can affect the number of scalar multiplications performed in computing the matrix product. This fact motivates an optimization problem: Given a sequence of positive integer dimensions $d_0, \ldots, d_n$, determine the minimum number of scalar multiplications needed to compute the product $M_1 \cdots M_n$, assuming $M_i$ is a $d_{i-1} \times d_i$ matrix for $1 \leq i \leq n$, and that the number of scalar multiplications required to multiply two matrices is as described above.

Various greedy strategies might be applied to this problem, but none can guarantee an optimal solution. Let us therefore look for a direct top-down solution to the problem of finding the minimum number of scalar multiplications for a product $M_i \cdots M_j$. Let us focus on finding the last matrix multiplication. This multiplication will involve the products $M_i \cdots M_k$ and $M_k+1 \cdots M_j$ for some $k$, $1 \leq k < n$. The sizes of these two matrices are $d_{i-1} \times d_k$ and $d_k \times d_j$. Therefore, once these two matrices are computed, an additional $d_{i-1}d_kd_j$ scalar multiplications must be performed. The principle of optimality clearly holds for this problem, as a better way of computing either sub-product results in fewer total scalar multiplications. Therefore, the following recurrence gives the minimum number of scalar multiplications needed to compute $M_i \cdots M_j$:

$$m(i, j) = \begin{cases} 
0 & \text{if } i = j \\
\min_{i \leq k < j} (m(i, k) + m(k + 1, j) + d_{i-1}d_kd_j) & \text{if } i < j
\end{cases} \quad (12.2)$$

In order to compute $m(i, j)$, $2(j - i)$ subproblems need to be solved. It is easily seen that there is a great deal of overlap between these subproblems. Therefore, dynamic programming is appropriate for computing $m(i, j)$. We need a matrix $m[1..n, 1..n]$. In order to compute $m[i, j]$, we need to use values in row $i$ to the left of column $j$ and values in column $j$ below row $i$. It therefore makes sense to compute $m$ by rows from bottom to top, and left to right within each row. The algorithm is given in Figure 12.2. It is easily seen to run in $\Theta(n^3)$ time and to use $\Theta(n^2)$ space.

### 12.3 All-Pairs Shortest Paths

In Section 11.3, we discussed the single-source shortest paths problem for directed graphs. In this section, we generalize the problem to all pairs of vertices; i.e., we wish to find, for each pair of vertices $u$ and $v$, a shortest path from $u$ to $v$. An obvious solution is to apply Dijkstra’s algorithm $n$ times, each time using a different vertex as the source. This would result
ChainedMatrixMult\([d[0..n]]\)

\[
\begin{align*}
&\text{ChainedMatrixMult}(d[0..n]) \\
&m \leftarrow \text{new} \ \text{Array}[1..n, 1..n] \\
&\text{for } i \leftarrow n \text{ to } 1 \text{ by } -1 \\
&\quad m[i, i] \leftarrow 0 \\
&\text{for } j \leftarrow i + 1 \text{ to } n \\
&\quad m[i, j] \leftarrow \infty \\
&\text{for } k \leftarrow i \text{ to } j - 1 \\
&\quad m[i, j] \leftarrow \text{Min}(m[i, j], m[i, k] + m[k + 1, j] + d[i - 1]d[k]d[j]) \\
&\text{return } m[1, n]
\end{align*}
\]

in an algorithm with running time in \(\Theta(n^3)\). Although the algorithm we present in this section is no faster asymptotically, it serves as a good example of how certain space optimizations can sometimes be made for dynamic programming algorithms. It also serves as an illustration of how dynamic programming can be applied to problems that are not optimization problems in the strictest sense of the word.

Let \(G = (V, E)\) be a directed graph, and let \(\text{len} : V^2 \rightarrow N \cup \{\infty\}\) be a function giving the length of each edge, so that

- \(\text{len}(u, u) = 0\) for \(u \in V\); and
- \(\text{len}(u, v) = \infty\) iff \(u \neq v\) and \((u, v) \notin E\), for \((u, v) \in V^2\).

We wish to find, for each ordered pair \((u, v) \in V^2\), the length of the shortest path from \(u\) to \(v\); if there is no such path, we define the length to be \(\infty\). Note that we have simplified the problem so that instead of finding the actual paths, we will only be finding their lengths.

This optimization problem is somewhat nonstandard in that the objective function is not a numeric-valued function. Instead, its range can be thought of as a matrix of values. However, the optimum is well-defined, as it occurs when all values are simultaneously minimized, and this is always possible.
Let $p$ be a shortest path from $i$ to $j$, and consider any vertex $k$ other than $i$ or $j$. Then either $k$ is in $p$ or it isn’t. If $k$ is not in $p$, then $p$ remains the shortest path from $i$ to $j$ if we remove $k$ from the graph. Otherwise, we can break $p$ into a path from $i$ to $k$ and a path from $k$ to $j$. Clearly, each of these paths are shortest paths between their endpoints. Thus, if we can find the shortest path from $i$ to $k$ and the shortest path from $k$ to $j$, we can determine the shortest path from $i$ to $j$.

A shortcoming to this approach is that we haven’t actually reduced the size of the problem, as the shortest paths from $i$ to $k$ and $k$ to $j$ are with respect to the original graph. One way to avoid this shortcoming is to generalize the problem so that a set of possible intermediate vertices is given as additional input. The problem is then to find, for each ordered pair $(i, j)$ of vertices, the length of the shortest path from $i$ to $j$ with all vertices other than $i$ and $j$ on this path belong to the given set. If the given set is $V$, then the result is the solution to the all-pairs shortest paths problem.

In order to keep the number of subproblems from being too large, we can restrict the sets we allow as input. Specifically, our additional input can be a natural number $k$, which denotes the set of all natural numbers strictly less than $k$.

Let $L_k(i, j)$ denote the length of the shortest path from $i$ to $j$ with intermediate vertices strictly less than $k$, where $0 \leq i < n$, $0 \leq j < n$, and $0 \leq k \leq n$. Using the above reasoning, we have the following recurrence for $L_k(i, j)$:

$$L_k(i, j) = \begin{cases} 
\text{len}(i, j) & \text{if } k = 0 \\
\min(L_{k-1}(i, j), L_{k-1}(i, k-1) + L_{k-1}(k-1, j)) & \text{if } k > 0.
\end{cases}$$

(12.3)

We can then implement a dynamic programming algorithm to compute all $L_k(i, j)$ using a 3-dimensional array. However, we can save a great deal of space by making some observations. Note that in order to compute an entry $L_k(i, j)$, for $k > 0$, we only use entries $L_{k-1}(i, j)$, $L_{k-1}(i, k-1)$, and $L_{k-1}(k-1, j)$. We claim that $L_{k-1}(i, k-1) = L_k(i, k-1)$ and that $L_{k-1}(k-1, j) = L_k(k-1, j)$. To see this, note that

$$L_k(i, k-1) = \min(L_{k-1}(i, k-1), L_{k-1}(i, k-1) + L_{k-1}(k-1, k-1))$$

$$= L_{k-1}(i, k-1)$$

and

$$L_k(k-1, j) = \min(L_{k-1}(k-1, j), L_{k-1}(k-1, k-1) + L_{k-1}(k-1, j))$$

$$= L_{k-1}(k-1, j).$$
Figure 12.3 Floyd’s algorithm for all-pairs shortest paths

Precondition: \( G \) refers to a Graph in which the data associated with each edge is a Nat giving its length.

Postcondition: Returns an array \( L[0..n-1,0..n-1] \) such that \( L[i,j] \) is the length of the shortest path from \( i \) to \( j \) in \( G \).

```plaintext
FLOYD(G)
    \( n \leftarrow G.Size() \);
    \( L \leftarrow \text{new Array}[0..n-1,0..n-1] \)
    for \( i \leftarrow 0 \) to \( n-1 \)
        for \( j \leftarrow 0 \) to \( n-1 \)
            if \( i = j \)
                \( L[i,j] \leftarrow 0 \)
            else
                \( d \leftarrow G.Get(i,j) \)
                if \( d = \text{nil} \)
                    \( L[i,j] \leftarrow \infty \)
                else
                    \( L[i,j] \leftarrow d \)
    for \( k \leftarrow 1 \) to \( n \)
        for \( i \leftarrow 0 \) to \( n-1 \)
            for \( j \leftarrow 0 \) to \( n-1 \)
                \( L[i,j] \leftarrow \text{Min}(L[i,j], L[i,k-1]+L[k-1,j]) \)
    return \( L[0..n-1,0..n-1] \)
```

As a result, we can use a 2-dimensional array \( L[0..n-1,0..n-1] \) to represent \( L_{k-1} \). We can then transform this array into \( L_k \) by updating each value in turn. The algorithm, known as Floyd’s algorithm, is shown in Figure 12.3. We assume that the length of each edge is given by its key. It is easily seen that, regardless of whether \( G \) is implemented as a MatrixGraph or a ListGraph, the algorithm runs in \( \Theta(n^3) \) time and uses only a constant amount of space other than what is required for input and output.

12.4 The Knapsack Problem

In Exercise 11.14 (page 390), we introduced the fractional knapsack problem. One part of this exercise was to show that the greedy algorithm for the
fractional knapsack problem does not extend to the so-called 0-1 knapsack problem — the variation in which the items cannot be broken. Specifically, in this variation we are given a set of $n$ items, each having a positive weight $w_i \in \mathbb{N}$ and a positive value $v_i \in \mathbb{N}$, and a weight bound $W \in \mathbb{N}$. We wish to find a subset $S \subseteq \{1, \ldots, n\}$ that maximizes

$$\sum_{i \in S} v_i$$

subject to the constraint that

$$\sum_{i \in S} w_i \leq W.$$

To solve this problem, first note that either item $n$ is in an optimal solution, or it isn’t. If it is, then we can obtain an optimal solution by solving the problem in which item $n$ has been removed and the weight bound has been decreased by $w_n$. Otherwise, we can obtain an optimal solution by solving the problem in which item $n$ has been removed. We therefore have the following recurrence giving the optimal value $V_i(j)$ that can be obtained from the first $i$ items with a weight bound of $j$, where $0 \leq i \leq n$ and $0 \leq j \leq W$:

$$V_i(j) = \begin{cases} 
0 & \text{if } i = 0 \\
V_{i-1}(j) & \text{if } i > 0, \ j < w_i \\
\max(V_{i-1}(j), V_{i-1}(j - w_i) + v_i) & \text{otherwise.}
\end{cases} \quad (12.4)$$

The optimal value is then given by $V_n(W)$.

It is not hard to see that the optimal value can be computed in $\Theta(nW)$ time and space using dynamic programming — the details are left as an exercise. However, suppose $W$ is much larger than the values of the items. In this case, another approach might be more appropriate. Let

$$V = \sum_{i=1}^{n} v_i.$$

Let us then compute the minimum weight required to achieve each possible value $v \leq V$. The largest value $v$ yielding a minimum weight no larger than $W$ is then our optimal value.

Taking this approach, we observe that item $n$ is either in the set of items for which value $v$ can be achieved with minimum weight, or it isn’t. If it
is, then the minimum weight can be computed by removing item $n$ and finding the minimum weight needed to achieve a value of $v - v_n$. Otherwise, the minimum weight can be computed by removing item $n$. The following recurrence therefore gives the minimum weight $W_i(j)$ needed to achieve a value of exactly $j$ from the first $i$ items, for $0 \leq i \leq n$, $0 \leq j \leq V$:

$$W_i(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\infty & \text{if } i = 0, j > 0 \\
W_{i-1}(j) & \text{if } i > 0, 0 < j < v_i \\
\min(W_{i-1}(j), W_{i-1}(j - v_i) + w_i) & \text{otherwise.}
\end{cases}$$  \hspace{1cm} (12.5)

The optimal value is then the maximum $j \leq V$ such that $W_n(j) \leq W$.

It is not hard to see that a dynamic programming algorithm based on this recurrence can find the optimal value in $\Theta(nV)$ time and space. Again, we leave the details as an exercise. Note that we could potentially improve the algorithm further if we could first find a better upper bound than $V$ for the optimal value. This would allow us to reduce the number of columns we need to compute in our array. We will explore this idea further in Chapter 17.

### 12.5 Summary

Dynamic programming algorithms provide more power for solving optimization problems than do greedy algorithms. Efficient dynamic programming algorithms can be found when the following conditions apply:

- The principle of optimality can be applied to decompose the problem into subinstances of the same problem.

- There is significant overlap between the subinstances.

- The total number of subinstances, including those obtained by recursively decomposing subinstances, is relatively small.

Although dynamic programming algorithms proceed bottom-up, the first step in formulating a dynamic programming algorithm is to formulate a top-down solution. This top-down solution usually takes the form of a recurrence for computing the optimal value of the objective function. The top-down solution is then implemented bottom-up, storing all of the solutions to the subproblems. In some cases, we optimize the space usage by discarding some of these solutions.
Because dynamic programming algorithms typically solve subinstances that are not used in the optimal solution, they tend to be less efficient than greedy algorithms. Hence, greedy algorithms are preferred when they exist. However, for many problems, there are no greedy algorithms that guarantee optimal solutions. In such cases, dynamic programming algorithms may be the most efficient.

Although the examples in this chapter have all been optimization problems, it is not hard to see that dynamic programming can be applied to other problems as well. Any computation that can be expressed as a recurrence can be computed bottom-up, yielding a dynamic programming solution. We explore some examples in the exercises.

12.6 Exercises

Exercise 12.1 Prove by induction on \( n + k \) that \( C(n, k) \), as defined in recurrence (12.1), gives the minimum number of coins needed to give a value of exactly \( n \) if the denominations are \( d_1 < d_2 < \cdots < d_k \) and \( d_1 = 1 \).

Exercise 12.2 Prove that \textsc{Change}, shown in Figure 12.1, meets its specification. You do not need to focus on the first half of the algorithm; i.e., you can assume that \( C(i, j) \), as defined in recurrence (12.1), is assigned to \( C[i, j] \). Furthermore, you may use the result of Exercise 12.1 in your proof.

* Exercise 12.3 As we have seen, the greedy algorithm suggested in Section 12.1 works for some sets of coin denominations but not for others.

a. Prove that for denominations \( d_1 < d_2 < \cdots < d_k \), where \( k > 1 \), if the greedy algorithm fails for some value, then it must fail for some value \( n < d_k + d_{k-1} \).

b. Devise an efficient dynamic programming algorithm which takes as input a set of denominations and returns \texttt{true} if the greedy algorithm always works for this set, or returns \texttt{false} otherwise. You may assume that the denominations are given in increasing order. Your algorithm should use \( O(Mk) \) time and space, where \( M \) is the largest denomination and \( k \) is the number of denominations.

Exercise 12.4 Prove by induction on \( j - i \) that \( m(i, j) \), as defined in recurrence 12.2, is the minimum number of scalar multiplications needed to compute a product \( M_i \cdots M_j \), where \( M_k \) is a \( d_{k-1} \times d_k \) matrix for \( i \leq k \leq j \).
CHAPTER 12. OPTIMIZATION II: DYNAMIC PROGRAMMING  405

Exercise 12.5  Prove by induction on \( k \) that \( L_k(i, j) \), as defined in recurrence 12.3, gives the length of the shortest path from \( i \) to \( j \) in which all intermediate vertices are strictly less than \( k \).

Exercise 12.6

a. Modify Floyd’s algorithm (Figure 12.3) so that it returns an array \( S[0..n-1,0..n-1] \) such that for \( i \neq j \), \( S[i, j] \) gives the vertex \( k \) such that \((i, k)\) is the first edge in a shortest path from \( i \) to \( j \). If there is no path from \( i \) to \( j \), or if \( i = j \), then \( S[i, j] \) should be \(-1\).

b. Give an algorithm that takes the array \( S[0..n-1,0..n-1] \) defined above, along with \( i \) and \( j \) such that \( 0 \leq i < n \) and \( 0 \leq j < n \), and prints the vertices along a shortest path from \( i \) to \( j \). The first vertex printed should be \( i \), followed by the vertices in order along the path, until the last vertex \( j \) is printed. If \( i = j \), only \( i \) should be printed. If there is no path from \( i \) to \( j \), a message to that effect should be printed. Your algorithm should run in \( O(n) \) time.

Exercise 12.7  Give an algorithm for the 0-1 knapsack problem that runs in \( O(nW) \) time and space, where \( n \) is the number of items and \( W \) is the weight bound. Your algorithm should use dynamic programming to compute recurrence (12.4) for \( 0 \leq i \leq n \) and \( 0 \leq j \leq W \), then use these values to guide a greedy algorithm for selecting the items to put into the knapsack. Your algorithm should return an array \( selected[1..n] \) of booleans such that \( selected[i] \) is \( \text{true} \) iff item \( i \) is in the optimal packing.

Exercise 12.8  Repeat Exercise 12.7 using recurrence (12.5) instead of (12.4). Your algorithm should use \( \Theta(nV) \) time and space, where \( n \) is the number of items and \( V \) is the total value of all the items.

Exercise 12.9  Let \( A[1..n] \) be an array of integers. An increasing subsequence of \( A \) is a sequence of indices \( \langle i_1, \ldots, i_k \rangle \) such that \( i_j < i_{j+1} \) and \( A[i_j] < A[i_{j+1}] \) for \( 1 \leq j < k \). (Note that the indices in the subsequence are not necessarily contiguous.) A longest increasing subsequence of \( A \) is an increasing subsequence of \( A \) with maximum length.

a. Give a recurrence for \( L(i) \), the length of the longest increasing subsequence of \( A[1..i] \) that ends with \( i \), where \( 1 \leq i \leq n \).

b. Give a dynamic programming algorithm that prints the indices of a longest increasing subsequence of \( A[1..n] \). Your algorithm should operate in \( O(n^2) \) time.
Exercise 12.10 Let $A[1..m]$ and $B[1..n]$ be two arrays. An array $C[1..k]$ is a common subsequence of $A$ and $B$ if there are two sequences of indices $\langle i_1, \ldots, i_k \rangle$ and $\langle j_1, \ldots, j_k \rangle$ such that

- $i_1 < i_2 < \cdots < i_k$;
- $j_1 < j_2 < \cdots < j_k$; and

A longest common subsequence of $A$ and $B$ is a common subsequence of $A$ and $B$ with maximum size.

a. Give a recurrence for $L(i, j)$, the length of the longest common subsequence of $A[1..i]$ and $B[1..j]$.

b. Give a dynamic programming algorithm that returns the longest common subsequence of $A[1..m]$ and $B[1..n]$. Your algorithm should operate in $O(mn)$ time.

Exercise 12.11 A palindrome is a string that reads the same from right to left as it does from left to right (“abcba”, for example). Give a dynamic programming algorithm that takes a string $s$ as input, and returns a longest palindrome contained as a substring within $s$. Your algorithm should operate in $O(n^2)$ time, where $n$ is the length of $s$. You may use the results of Exercise 4.13 (page 143) in analyzing your algorithm. [Hint: For each pair of indices $i \leq j$, determine whether the substring from $i$ to $j$ is a palindrome.]

* Exercise 12.12 Suppose we have two $k$-dimensional boxes $A$ and $B$ whose $k$ dimensions are $a_1, \ldots, a_k$ and $b_1, \ldots, b_k$, respectively. We say that $A$ fits inside of $B$ if there is a permutation $a_{i_1}, \ldots, a_{i_k}$ of the dimensions of $A$ such that $a_{i_j} < b_j$ for $1 \leq j \leq k$. Design a dynamic programming algorithm that takes as input a positive integer $k$ and the dimensions of $n$ $k$-dimensional boxes, and returns the maximum size of any subset of the boxes that can be ordered such that each box (except the last) in the ordering fits inside of the next. Your algorithm should run in $O(\max(n^2k, nk \log k))$ time in the worst case. Note that your algorithm doesn’t need to return a subset or an ordering — only the size of the subset.

Exercise 12.13 Let $G = (V, E)$ be a directed graph. The transitive closure of $G$ is a directed graph $G' = (V, E')$, where $E'$ is the set of all $(u, v) \in V^2$
such that \( u \neq v \) and there is a path from \( u \) to \( v \) in \( G \). Give an \( O(n^3) \) dynamic programming algorithm to produce a MatrixGraph that is the transitive closure of a given MatrixGraph.

**Exercise 12.14** A convex polygon is a polygon whose interior angles are all less than 180 degrees. For example, in Figure 12.4, polygon a is convex, but polygon b is not. A triangulation of a convex polygon is a set of non-intersecting diagonals that partition the polygon into triangles, as shown in Figure 12.4 c. Give a dynamic programming algorithm that takes as input a convex polygon and produces a triangulation that minimizes the sum of the lengths of the diagonals, where the length of an edge \((x_1, y_1), (x_2, y_2)\) is given by

\[
\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.
\]

You may assume that the polygon is represented as a sequence of points in the Cartesian plane \( \langle p_1, p_2, \ldots, p_n \rangle \) such that the edges of the polygon are \((p_1, p_2), (p_2, p_3), \ldots, (p_{n-1}, p_n), \) and \((p_n, p_1)\). You may further assume that \( n \geq 3 \). Your algorithm should run in \( O(n^3) \) time.

* **Exercise 12.15** A chain is a rooted tree with exactly one leaf. We are given a chain representing a sequence of \( n \) pipelined processes. Each node \( i \) in the chain represents a process and has a positive execution time \( e_i \in \mathbb{N} \). Each edge \((i, j)\) has a positive communication cost \( c_{ij} \in \mathbb{N} \). For edge \((i, j)\), if processes \( i \) and \( j \) are executed on separate processors, the time needed to send data from process \( i \) to process \( j \) is \( c_{ij} \); if the processes are executed on the same processor, this time is 0. We wish to assign processes to processors such that each processor has total weight no more than a given value \( B \in \mathbb{N} \). The weight of a processor is given by the sum of the execution times of the processes assigned to that processor, plus the sum of the communication
costs of edges between tasks on that processor and tasks on other processors (see Figure 12.5). The communication cost of an assignment is the sum of the communication costs of edges that connect nodes assigned to different processors.

Give a dynamic programming algorithm that finds the minimum communication cost of any assignment of processes to processors such that each processor has weight no more than $B$. Note that we place no restriction on the number of processors used. Your algorithm should run in $O(n^2)$ time. Prove that your algorithm is correct.

**Exercise 12.16** Given two strings $x$ and $y$, we define the *edit distance* from $x$ to $y$ as the minimum number of operations required to transform $x$ into $y$, where the operations are chosen from the following:

- insert a character;
- delete a character; or
CHAPTER 12. OPTIMIZATION II: DYNAMIC PROGRAMMING

• change a character.

** a. Prove that there is an optimal sequence of operations transforming \( x \) to \( y \) in which the edits proceed from left to right.

b. Using the above result, give a dynamic programming algorithm that takes as input \( x \) and \( y \) and returns the edit distance from \( x \) to \( y \). Your algorithm should run in \( O(mn) \) time, where \( m \) is the length of \( x \) and \( n \) is the length of \( y \). Prove that your algorithm is correct.

Exercise 12.17 Suppose we wish to store \( n \) keys, \( k_1 < k_2 < \cdots < k_n \), in a binary search tree (see Chapter 6). We define the cost of a look-up of key \( k_i \) as \( d_i + 1 \), where \( d_i \) is the depth of \( k_i \) in the tree. Thus, the cost of a look-up is simply the number of nodes examined. Suppose further that we are given, for \( 1 \leq i \leq n \), the probability \( p_i \in \mathbb{R} > 0 \) that any given look-up is for key \( k_i \). We assume that

\[
\sum_{i=1}^{n} p_i = 1.
\]

We say that a binary search tree containing these keys is optimal if the expected cost of a look-up in this tree is minimum over the set of all binary search trees containing these keys.

a. Let us extend the definition of the cost of a look-up to pertain to a specific subtree, so that the cost with respect to subtree \( T \) is the number of nodes in \( T \) examined during that look-up. For \( i \leq j \), let \( S_{ij} \) be the set of all binary search trees with keys \( k_1, \ldots, k_n \) such that there is a subtree containing exactly the keys \( k_i, \ldots, k_j \). Let \( C_{ij} \) denote the minimum over \( S_{ij} \) of the expected cost of a look-up with respect to the subtree containing keys \( k_i, \ldots, k_j \). Prove that

\[
C_{ij} = \begin{cases} 
p_i & \text{if } i = j \\
\min_{i \leq k \leq j} (C_{i,k-1} + C_{k+1,j}) + \sum_{k=i}^{j} p_k & \text{if } i < j
\end{cases}
\]

* b. Give a dynamic programming algorithm that takes as input \( p_1, \ldots, p_n \) and returns the expected cost of a look-up for an optimal binary search tree whose keys \( k_1 < k_2 < \cdots < k_n \) have the given probabilities. (Note that we don’t need the values of the keys in order to compute this value.) Your algorithm should run in \( O(n^3) \) time and \( O(n^2) \) space.
** c. Suppose \( r_{ij} \) is the root of an optimal binary search containing the keys \( k_i, \ldots, k_j \), where \( i \leq j \). Prove that \( r_{i,j-1} \leq r_{ij} \leq r_{i+1,j} \) for \( 1 \leq i < j \leq n \).

* d. Using the above result, improve your algorithm to run in \( O(n^2) \) time.

** Exercise 12.18** Give a dynamic programming algorithm that takes as input two natural numbers \( k \leq n \) and returns the probability that flipping a fair coin \( n \) times yields at least \( k \) heads. Your algorithm should run in \( O(n) \) time. Prove that your algorithm is correct.

* Exercise 12.19** Give a dynamic programming algorithm that takes as input a natural number \( n \) and returns the number of different orderings of \( n \) elements using < and/or =. For example, for \( n = 3 \), there are 13 orderings:

\[
\begin{align*}
&x < y < z \quad x < z < y \quad y < x < z \quad y < z < x \\
&z < x < y \quad z < y < x \quad x = y < z \quad z < x = y \\
&x = z < y \quad y < x = z \quad y = z < x \quad x < y = z \\
&x = y = z
\end{align*}
\]

Your algorithm should run in \( O(n^2) \) time and use \( O(n) \) space. Prove that your algorithm is correct.

* Exercise 12.20** Suppose we have a mathematical structure containing three elements, \( a, b, \) and \( c \), and a multiplication operation given by the following table:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>a</td>
<td>c</td>
<td>b</td>
</tr>
</tbody>
</table>

Note that this multiplication operation is neither commutative nor associative. Give a dynamic programming algorithm that takes as input a string over \( a, b, \) and \( c \), and returns a boolean indicating whether it is possible to parenthesize the string so that the result is \( a \). (For example, if we parenthesize \( abca \) as \( (a(bc))a \), we get a result of \( a \).) Your algorithm should run in \( O(n^3) \) time, where \( n \) is the length of the input string. Prove that your algorithm is correct.

* Exercise 12.21** Suppose we are given an array \( L[1..n] \) of positive integers representing the lengths of successive words in a paragraph. We wish to format the paragraph so that each line contains no more than \( m \) characters,
including a single blank character between adjacent words on the same line. Furthermore, we wish to minimize a “sloppiness” criterion. Specifically, we wish to minimize the following objective function:

\[ \sum_{i=1}^{k-1} f(m - c_i), \]

where \( k \) is the total number of lines used, \( f : \mathbb{N} \to \mathbb{N} \) is some nondecreasing function, and \( c_i \) is the number of characters (including blanks between adjacent words) on line \( i \). Give an efficient dynamic programming algorithm for computing the optimal arrangement. Your algorithm should run in \( O(n^2) \) time and use \( O(n) \) space. [\textbf{Hint:} Reduce this problem to the problem for which the measure of sloppiness includes the last line — i.e., the optimization function is as above, but with \( k - 1 \) replaced by \( k \).]

* Exercise 12.22 We are given a set of \( n \) points \((x_i, y_i)\), where each \( x_i \) and \( y_i \) is a real number and all the \( x_i \)'s are distinct. A bitonic tour of these points is a cycle that begins at the rightmost point, proceeds strictly to the left to the leftmost point, then proceeds strictly to the right to return to the rightmost point; furthermore, this cycle contains every point exactly once (see Figure 12.6). We wish to find the bitonic tour having minimum Euclidean length; i.e., the distance between two points \((x_1, y_1)\) and \((x_2, y_2)\) is given by

\[ \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}. \]

Give an efficient dynamic programming algorithm for finding a minimum-length bitonic tour. Your algorithm should use \( O(n^2) \) time and space. [\textbf{Hint:} Reduce this problem to that of finding a minimum-length bitonic path that includes all the points exactly once, but does not return to the starting point.]

* Exercise 12.23 Suppose we were to modify the scheduling problem of Section 11.1 so that each job also has a natural number \emph{execution time}, which must be fulfilled without interruption. Thus, a schedule including job \( i \) must have job \( i \) scheduled for \( e_i \) contiguous time units lying between time 0 and time \( d_i \), where \( e_i \) is the execution time and \( d_i \) is the deadline of job \( i \). Give an efficient dynamic programming algorithm to generate a schedule with maximum value. Your algorithm should use \( O(n(m + \lg n)) \) time and \( O(mn) \) space, where \( n \) is the number of jobs and \( m \) is the maximum deadline.
12.7 Chapter Notes

The mathematical foundation for dynamic programming was given by Bellman [10]. The Change algorithm in Figure 12.1 is due to Wright [115]. The ChainedMatrixMult algorithm in Figure 12.2 is due to Godbole [54]. Floyd’s algorithm (Figure 12.3) is due to Floyd [38], but is based on a theorem due to Warshall [110] for computing the transitive closure of a boolean matrix. Because a boolean matrix can be viewed as an adjacency matrix for a directed graph, this is the same as finding the transitive closure of a directed graph (Exercise 12.13).

The algorithm suggested by Exercise 12.3 is due to Kozen and Zaks [81]. Exercise 12.10 is solved by Chvatal, Klarner, and Knuth [22]. Wagner and Fischer [109] solved Exercise 12.16 and provided an alternative solution to Exercise 12.10. Exercise 12.17 is solved by Gilbert and Moore [53] and Knuth [79], but a more elegant solution is given by Yao [117]. Exercises 12.19 and 12.20 are from Brassard and Bratley [18].