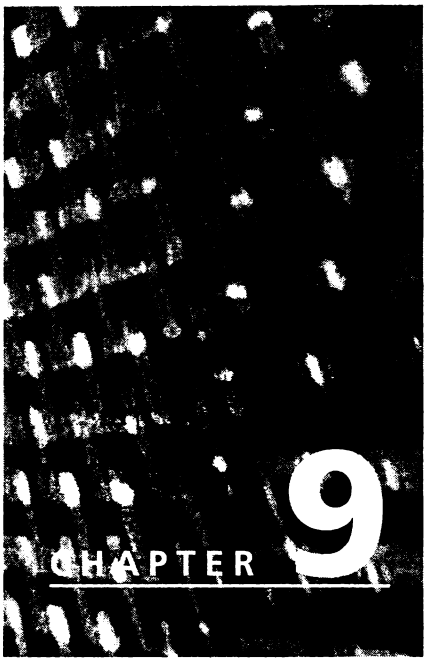


DYNAMIC PROGRAMMING

Dynamic programming is a design strategy that involves dynamically constructing a solution S to a given problem using solutions $S_1, S_2, \ldots, S_m$ to smaller (or simpler) instances of the problem. The solution S_i to a given smaller problem instance is itself built from the solutions to even smaller (or simpler) problem instances, and so forth. We start with the known solutions to the smallest (simplest) problem instances and build from there in a bottom-up fashion. To be able to reconstruct S from $S_1, S_2, \ldots, S_m$, we usually require some additional information. We let *Combine* denote the function that combines $S_1, S_2, \ldots, S_m$, using the additional information to obtain S, so that

$$S = Combine(S_1, S_2, \ldots, S_m).$$

Dynamic programming is similar to divide-and-conquer in the sense that it is based on a recursive division of a problem instance into smaller or simpler problem instances. However, whereas divide-and-conquer algorithms often use a top-down resolution method, dynamic programming algorithms invariably proceed by solving all the simplest problem instances before combining them into more complicated problem instances in a bottom-up fashion. Further,

unlike many instances of divide-and-conquer, dynamic programming algorithms typically do not recalculate the solution to a given problem instance. Dynamic programming algorithms for optimization problems also can avoid generating suboptimal problem instances when the *Principle of Optimality* holds, thereby leading to increased efficiency.

9.1 Optimization Problems and the Principle of Optimality

The method of dynamic programming is most effective in solving optimization problems when the Principle of Optimality holds. Consider the set of all *feasible* solutions to an optimization problem; that is, all the solutions satisfying the constraints of the problem. An *optimal* solution S is a solution that optimizes (minimizes or maximizes) the objective function. If we wish to obtain an optimal solution S to the given problem instance, then we must optimize (minimize or maximize) over *all* solutions $S_1, S_2, \ldots, S_m$ such that $S = Combine(S_1, S_2, \ldots, S_m)$. For many problems, it is computationally infeasible to examine all feasible solutions because exponentially many possibilities exist. Fortunately, we can drastically reduce the number of problem instances that we need to consider if the Principle of Optimality holds.

DEFINITION 9.1.1 Given an optimization problem and an associated function *Combine*, the *Principle of Optimality* holds if the following is always true: If $S = Combine(S_1, S_2, \ldots, S_m)$ and S is an *optimal* solution to the problem instance, then $S_1, S_2, \ldots, S_m$, are *optimal* solutions to their associated problem instances.

The efficiency of dynamic programming solutions based on a recurrence relation expressing the principle of optimality results from (1) the bottom-up resolution of the recurrence, thereby eliminating redundant recalculations, and (2) eliminating suboptimal solutions to subproblems as we build up optimal solutions to larger problems; that is, we use only optimal solution "building blocks" in constructing our optimal solution.

We first illustrate the Principle of Optimality for the problem of finding a parenthesization of a matrix product of matrices $M_0, \ldots, M_{n-1}$ that minimizes the total number of (scalar) multiplications over all possible parenthesizations. If $(M_0 \cdots M_k)(M_{k+1} \cdots M_{n-1})$ is the "first-cut" set of parentheses (and the last product performed), then the matrix products $M_0 \cdots M_k$ and $M_{k+1} \cdots M_{n-1}$ must both be parenthesized in such a way as to minimize the number of multiplications required to carry out the respective products. As a second example, consider the

problem of finding optimal binary search trees for a set of distinct keys. Recall that a binary search tree T for keys $K_0 < \cdots < K_{n-1}$ is a binary tree on n nodes, each containing a key such that the following property is satisfied: Given any node v in the tree, each key in the left subtree rooted at v is no larger than the key in v, and each key in the right subtree rooted at v is no smaller than the key in v (see Figure 4.17). If K_i is the key in the root, then the left subtree L of the root contains $K_0, \ldots, K_{i-1}$, and the right subtree R of the root contains $K_{i+1}, \ldots, K_{n-1}$. Given a binary search tree T for keys $K_0, \ldots, K_{n-1}$, let K_i denote the key associated with the root of T, and let L and R denote the left and right subtrees (of the root) of T, respectively. Again, it follows that L (solution S_1) is a binary search tree for keys $K_0, \ldots, K_{i-1}$, and R (solution S_2) is a binary search tree for keys $K_{i+1}, \ldots, K_{n-1}$. Given L and R, the function *Combine*(L,R) merely reconstructs the tree T using K_i as the root. In the next section, we show that the Principle of Optimality holds for this problem by showing that if T is an optimal binary search tree, then so are L and R.

9.2 Optimal Parenthesization for Computing a Chained Matrix Product

Our first example of dynamic programming is an algorithm for the problem of parenthesizing a chained matrix product so as to minimize the number of multiplications performed when computing the product. When solving this problem, we will assume the straightforward method of matrix multiplication. If A and B are matrices of dimensions $p \times q$ and $q \times r$, then the matrix product AB involves pqr multiplications. Given a sequence (or chain) of matrices $M_0, M_1, \ldots, M_{n-1}$, consider the product $M_0M_1 \cdots M_{n-1}$, where the matrix M_i has dimension $d_i \times d_{i+1}$, $i = 0, \ldots, n$, for a suitable sequence of positive integers $d_0, d_1, \ldots, d_n$. Because a matrix product is an associative operation, we can evaluate the chained product in one of many ways, depending on how we choose to parenthesize the expression. It turns out that the manner in which the expression is parenthesized can make a major difference in the total number of multiplications performed when computing the chained product. In this section, we consider the problem of finding an *optimal parenthesization*—that is, a parenthesization that minimizes the total number of multiplications performed using ordinary matrix products.

We illustrate the problem with an example that commonly occurs in multivariate calculus. Suppose A and B are $n \times n$ matrices, X is an $n \times 1$ column vector, and we wish to evaluate ABX. The product ABX can be parenthesized in two ways, $(AB)X$ and $A(BX)$, resulting in $n^3 + n^2$ and $2n^2$ multiplications, respectively. Thus, the two ways of parenthesizing make a rather dramatic difference in the number of multiplications performed; that is, order $\Theta(n^3)$ versus order $\Theta(n^2)$.

The following is a formal, recursive definition of a fully parenthesized chained matrix product and its associated first cut.

DEFINITION 9.2.1 Given the sequence of matrices $M_0, M_1, \ldots, M_{n-1}$, P is a *fully parenthesized* matrix product of $M_0, M_1, \ldots, M_{n-1}$ (which, for convenience, we simply call a *parenthesization of* $M_0 M_1 \cdots M_{n-1}$) if P satisfies

$$P = M_0, \quad n = 1,$$
$$P = (P_1 P_2), \quad n > 1,$$

where for some k, P_1 and P_2 are parenthesizations of the matrix products $M_0 M_1 \cdots M_k$ and $M_{k+1} M_{k+2} \cdots M_{n-1}$, respectively. We call P_1 and P_2 the *left* and *right* parenthesizations of P, respectively. We call the index k the *first-cut index of* P.

The table in Figure 9.1 shows all the parenthesizations of the matrices M_0, M_1, M_2, and M_3 having dimensions 20×10, 10×50, 50×5, and 5×30, respectively, with the optimal parenthesizations highlighted.

There is one-to-one correspondence between parenthesizations of $M_0 M_1 \cdots M_{n-1}$ and 2-trees having n leaf nodes. Given a parenthesization P of $M_0 M_1 \cdots M_{n-1}$, if $n = 1$, its associated 2-tree $T(P)$ consists of a single node corresponding to the matrix M_0; otherwise, $T(P)$ has left subtree $T(P_1)$ and right subtree $T(P_2)$, where P_1 and P_2 are the left and right parenthesizations of P. The 2-tree $T(P)$ is the *expression tree* for P (see Figure 9.2).

FIGURE 9.1
Number of multiplications performed for each full parenthesization shown for matrices M_0, M_1, M_2, and M_3 having dimensions 20×10, 10×50, 50×5, and 5×30, respectively. The optimal parenthesizations are shaded.

no. mult.	no. mult.	no. mult.	no. mult.
M_0	$(M_0 M_1)$ 10000	$(M_0(M_1 M_2))$ 3500	$(M_0(M_1(M_2 M_3)))$ 28500
		$((M_0 M_1) M_2)$ 15000	$(M_0((M_1 M_2) M_3))$ 10000
			$((M_0 M_1)(M_2 M_3))$ 47500
			$((M_0(M_1 M_2)) M_3)$ 6500
			$(((M_0 M_1) M_2) M_3)$ 18000

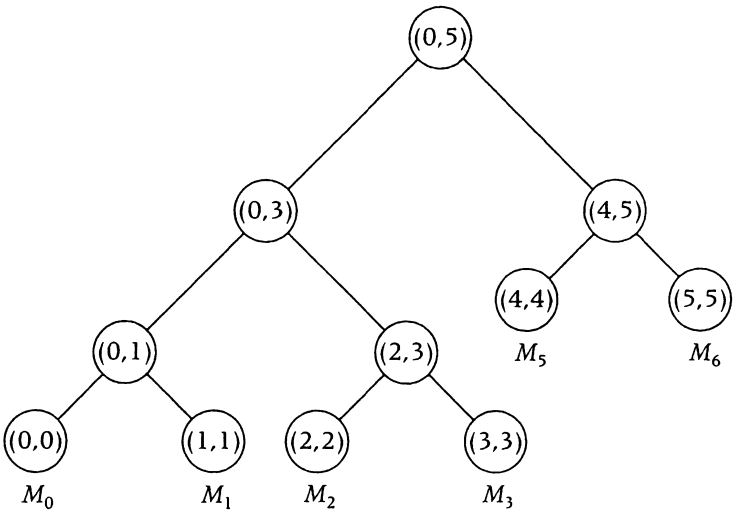


FIGURE 9.2

Associated expression 2-tree for parenthesization $(((M_0M_1)(M_2M_3))(M_4M_5))$. The label (i, j) inside each node indicates that the matrix product associated with the node involves matrices $M_i, M_{i+1}, \ldots, M_j$.

Thus, the number of parenthesizations p_n equals the number t_n of 2-trees having n leaf nodes, so that by Exercise 4.14 we have

$$p_n = \frac{1}{n}\binom{2n-2}{n-1} \geq \frac{4^{n-1}}{2n^2 - n} \in \Omega\left(\frac{4^n}{n^2}\right). \qquad \textbf{(9.2.1)}$$

Hence, a brute-force algorithm that examines all possible parenthesizations is computationally infeasible.

We are led to consider a dynamic programming solution to our problem by noting that the Principle of Optimality holds for optimal parenthesizing. Indeed, if we consider any optimal parenthesization P for $M_0M_1 \cdots M_{n-1}$, clearly both the left and right parenthesizations P_1 and P_2 of P must be optimal for P to be optimal.

For $0 \leq i \leq j \leq n-1$, let m_{ij} denote the number of multiplications performed using an optimal parenthesization of $M_iM_{i+1} \cdots M_j$. By the Principle of Optimality, we have the following recurrence for m_{ij} based on making an optimal choice for the first-cut index:

$$m_{ij} = \min_k\{m_{ik} + m_{k+1,j} + d_id_{k+1}d_{j+1} : 0 \leq i \leq k < j \leq n-1\} \qquad \textbf{(9.2.2)}$$
$$\textbf{init. cond. } m_{ii} = 0, i = 0, \ldots, n-1.$$

The value $m_{0,n-1}$ corresponds to the minimum number of multiplications performed when computing $M_0M_1 \cdots M_{n-1}$. We could base a divide-and-conquer algorithm *ParenthesizeRec* directly on a top-down implementation of the recurrence relation (9.2.2). Unfortunately, a great many recalculations are performed by *ParenthesizeRec*, and it ends up doing $\Omega(3^n)$ multiplications

to compute the minimum number $m_{0,n-1}$ corresponding to an optimal parenthesization.

A straightforward dynamic programming algorithm proceeds by computing the values m_{ij}, $0 \le i \le j \le n-1$, in a bottom-up fashion using (9.2.2), thereby avoiding recalculations. Note that the values m_{ij}, $0 \le i \le j \le n-1$, occupy the upper-right triangular portion of an $n \times n$ table. Our bottom-up resolution proceeds throughout the upper-right triangular portion diagonal by diagonal, starting from the bottom diagonal consisting of the elements $m_{ii} = 0$, $i = 0, \ldots, n-1$. The q^{th} diagonal consists of the elements $m_{i,i+q}$, $q = 0, \ldots, n-1$. Figure 9.3 illustrates the computation of m_{ij} for the example given in Figure 9.1. When computing m_{ij}, we also generate a table c_{ij} of indices k, where the minimum in (9.2.2) occurs; that is, c_{ij} is where the first cut in $M_i M_{i+1} \cdots M_j$ is made in an optimal parenthesization. The values c_{ij} can then be used to actually compute the matrix product according to the optimal parenthesization.

The following procedure, *OptimalParenthesization,* accepts as input the *dimension sequence* $d[0{:}n]$, where matrix M_i has dimension $d_i \times d_{i+1}$, $i = 0, \ldots, n-1$. Procedure *OptimalParenthesization* outputs the matrix $m[0{:}n-1, 0{:}n-1]$, where $m[i,j] = m_{ij}$, $0 \le i \le j \le n-1$, is defined by recurrence (9.2.2). *OptimalParenthesization* also outputs the matrix *FirstCut*$[0{:}n-1, 0{:}n-1]$, where *FirstCut*$[i,j] = c_{ij}$, $0 \le i \le j \le n-1$, which is the first-cut index in an optimal parenthesization for $M_i \cdots M_j$.

FIGURE 9.3

Table showing values m_{ij}, $0 \le i \le j \le 3$, computed diagonal by diagonal from $q = 0$ to $q = 3$ using the bottom-up resolution of (9.2.2) for matrices M_0, M_1, M_2, and M_3 having dimensions 20×10, 10×50, 50×5, and 5×30, respectively. The values of c_{ij} are shown underneath each m_{ij}, $0 \le i \le j \le 3$.

i \ j	0	1	2	3	
0	0	10000	3500	6500	
		0	0	2	$q = 3$
1		0	2500	4000	
			1	2	$q = 2$
2			0	7500	
				2	$q = 1$
3				0	
					$q = 0$

```
procedure OptimalParenthesization(d[0:n], m[0:n − 1, 0:n − 1], FirstCut[0:n − 1,
          0:n − 1])
Input:    d[0:n] (dimension sequence for matrices M_0, M_1, ..., M_{n−1})
Output:   m[0:n − 1, 0:n − 1]  (m[i, j] = number of multiplications performed in an
                                optimal parenthesization for computing M_i ··· M_j,
                                0 ≤ i ≤ j ≤ n − 1)
          FirstCut[0:n − 1, 0:n − 1]  (index of first cut in optimal parenthesization of
                                M_i ··· M_j, 0 ≤ i ≤ j ≤ n − 1)
   for i ← 0 to n − 1 do      // initialize M[i, i] to zero
       m[i, i] ← 0
   endfor
   for diag ← 1 to n − 1 do
      for i ← 0 to n − 1 − diag do
         j ← i + diag          // compute m_ij according to (9.2.2)
         Min ← m[i + 1, j] + d[i]*d[i + 1]*d[j + 1]
         TempCut ← i
         for k ← i + 1 to j − 1 do
            Temp ← m[i, k] + m[k + 1, j] + d[i]*d[k + 1]*d[j + 1]
            if Temp < Min then
               Min ← Temp
               TempCut ← k
            endif
         endfor
         m[i, j] ← Min
         FirstCut[i, j] ← TempCut
      endfor
   endfor
end OptimalParenthesization
```

A simple loop counting shows that the complexity of *OptimalParenthesization* is in $\Theta(n^3)$.

It is now straightforward to write pseudocode for a recursive function *ChainMatrixProd* for computing the chained matrix product $M_0 \cdots M_{n-1}$ using an optimal parenthesization. We assume that the matrices $M_0, \ldots, M_{n-1}$ and the matrix $FirstCut[0{:}n-1, 0{:}n-1]$ are global variables to the procedure *ChainMatrixProd*. The chained matrix product $M_0 \cdots M_{n-1}$ is computed by initially invoking the function *ChainMatrixProd* with $i = 0$ and $j = n - 1$. *ChainMatrixProd* invokes a function *MatrixProd*, which computes the matrix product of two input matrices.

```
function ChainMatrixProd(i, j) recursive
Input:   i, j (indices delimiting matrix chain M_i, ..., M_j)
             M_0, ..., M_n-1 (global matrices)
             FirstCut[0:n - 1, 0:n - 1] (global matrix computed by
             OptimalParenthesization)
Output:  M_i ··· M_j (matrix chain)
    if j > i then
         X ← ChainMatrixProd(i, FirstCut[i, j])
         Y ← ChainMatrixProd(FirstCut[i, j] + 1, j)
         return(MatrixProd(X, Y))
    else
         return(M_i)
    endif
end ChainMatrixProd
```

For the example given in Figure 9.1, invoking *ChainMatrixProd* with M_0, M_1, M_2, M_3 computes the chained matrix product $M_0M_1M_2M_3$ according to the parenthesization $((M_0(M_1M_2))M_3)$.

9.3 Optimal Binary Search Trees

We now use dynamic programming and the Principle of Optimality to generate an algorithm for the problem of finding optimal binary search trees. Given a search tree T and a search element X, the following recursive strategy finds any occurrence of a key X. First, X is compared to the key K associated with the root. If X is found there, we are done. If X is not found, and if X is less than K, then we search the left subtree; otherwise, we search the right subtree.

Consider, for example, the binary search tree given in Figure 9.4 involving the four keys "Ann," "Joe," "Pat," and "Ray". The internal nodes correspond to the successful searches X = "Ann," X = "Joe," X = "Pat," X= "Ray," and the leaf nodes correspond to the unsuccessful searches $X <$ "Ann," "Ann" $< X <$ "Joe," "Joe" $< X <$ "Pat," "Pat" $< X <$ "Ray," "Ray" $< X$. Suppose, for example that X = "Ann." Then *SearchBinSrchTree* makes three comparisons, first comparing X to "Pat," then comparing X to "Joe," and finally comparing X to "Ann." Now suppose that X = "Pete." Then *SearchBinSrchTree* makes two comparisons, first comparing X to "Pat" and then comparing X to "Ray." *SearchBinSrchTree* implicitly branches to the left child of the node containing the key "Ray"—that is, to the leaf (implicit node) corresponding to the interval "Pat" $< X <$ "Ray." Let p_0, p_1, p_2, p_3 be the probability that X = "Ann," X = "Joe," X = "Pat," X = "Ray," respectively, and let q_0, q_1, q_2, q_3, q_4, denote the probability that $X <$ "Ann," "Ann" $< X <$ "Joe," "Joe" $< X <$ "Pat," "Pat" $< X <$ "Ray," "Ray" $< X$, respectively.

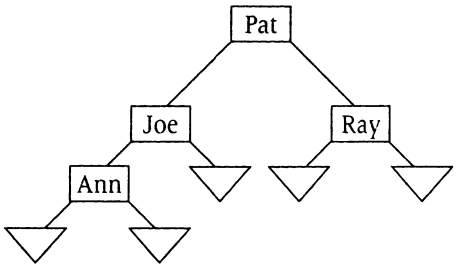


FIGURE 9.4

Search tree with leaf nodes drawn representing unsuccessful searches.

Then, the average number of comparisons made by *SearchBinSrchTree* for the tree T of Figure 9.4 is given by

$$3p_0 + 2p_1 + p_2 + 2p_3 + 3q_0 + 3q_1 + 2q_2 + 2q_3 + 2q_4.$$

Now consider a general binary search tree T whose internal nodes correspond to a fixed set of n keys $K_0, K_1, \ldots, K_{n-1}$ with associated probabilities $\mathbf{p} = (p_0, p_1, \ldots, p_{n-1})$, and whose $n + 1$ leaf (external) nodes correspond to the $n + 1$ intervals I_0: $X < K_0$, I_1: $K_0 < X < K_1$, $\ldots$, I_{n-1}: $K_{n-2} < X < K_{n-1}$, I_n: $X > K_{n-1}$ with associated probabilities $\mathbf{q} = (q_0, q_1, \ldots, q_n)$. (When implementing T, the leaf nodes need not actually be included. However, when discussing the average behavior of *SearchBinSrchTree*, it is useful to include them.) We now derive a formula for the average number of comparisons $A(T, n, \mathbf{p}, \mathbf{q})$ made by *SearchBinSrchTree*. Let d_i denote the depth of the internal node corresponding to K_i, $i = 0, \ldots, n - 1$. Similarly, let e_i denote the depth of the leaf node corresponding to the interval I_i, $i = 0, 1, \ldots, n$. If $X = K_i$, then *SearchBinSrchTree* traverses the path from the root to the internal node corresponding to K_i. Thus, it terminates after performing $d_i + 1$ comparisons. On the other hand, if X lies in I_i, then *SearchBinSrchTree* traverses the path from the root to the leaf node corresponding to I_i and terminates after performing e_i comparisons. Thus, we have

$$A(T, n, \mathbf{p}, \mathbf{q}) = \sum_{i=0}^{n-1} p_i(d_i + 1) + \sum_{i=0}^{n} q_i e_i. \qquad \textbf{(9.3.1)}$$

We now consider the problem of determining an *optimal* binary search tree T, optimal in the sense that T minimizes $A(T, n, \mathbf{p}, \mathbf{q})$ over all binary search trees T. This problem is solved by a complete tree in the case where all the p_i's are equal and all the q_i's are equal. Here we use dynamic programming to solve the problem for general probabilities p_i and q_i. In fact, we solve the slightly more general problem, where we relax the condition that $p_0, \ldots, p_{n-1}$ and $q_0, \ldots, q_n$ are probabilities by allowing them to be arbitrary nonnegative real numbers. One

could regard these numbers as frequencies, as we did when discussing Huffman codes in Chapter 7. That is, we solve the problem

$$\underset{T}{\text{minimize}}\ A(T, n, \mathbf{p}, \mathbf{q}) \tag{9.3.2}$$

over all binary search trees T of size n, where $p_0, \ldots, p_{n-1}$ and $q_0, \ldots, q_n$ are given fixed nonnegative real numbers. For convenience, we sometimes refer to $A(T, n, \mathbf{p}, \mathbf{q})$ as the *cost* of T. We define $\sigma(\mathbf{p}, \mathbf{q})$ by

$$\sigma(\mathbf{p}, \mathbf{q}) = \sum_{i=0}^{n-1} p_i + \sum_{i=0}^{n} q_i. \tag{9.3.3}$$

Note that we have removed the probability constraint that $\sigma(\mathbf{p}, \mathbf{q}) = 1$.

As with chained matrix products, we could obtain an optimal search tree by enumerating all binary search trees on the given identifiers and choosing the one with minimum $A(T, n, \mathbf{p}, \mathbf{q})$. However, the number of different binary search trees on n identifiers is the same as the number of binary trees on n nodes, which is given by the n^{th} Catalan number

$$b_n + \frac{1}{n+1}\binom{2n}{n} \in \Omega\left(\frac{4^n}{n^2}\right).$$

Thus, a brute-force algorithm for determining an optimal binary search tree using simple enumeration is computationally infeasible. Fortunately, the Principle of Optimality holds for the optimal binary search tree problem, so we look for a solution using dynamic programming.

Let K_i denote the key associated with the root of T, and let L and R denote the left and right subtrees (of the root) of T, respectively. As we remarked earlier, L is a binary search tree for the keys $K_0, \ldots, K_{i-1}$, and R is a binary search tree for the keys $K_{i+1}, \ldots, K_{n-1}$. For convenience, let $A(T) = A(T, n, \mathbf{p}, \mathbf{q})$, $A(L) = A(L, i, p_0, \ldots, p_{i-1}, q_0, \ldots, q_i)$, and $A(R) = A(R, n - i - 1, p_{i+1}, \ldots, p_{n-1}, q_{i+1}, \ldots, q_n)$. Clearly, each node of T that is different from the root corresponds to exactly one node in either L or R. Further, if N is a node in T corresponding to a node N' in L, then the depth of N in T is exactly one greater then the depth of N' in L. A similar result holds if N corresponds to a node in R. Thus, it follows immediately from Formula (9.3.1) that

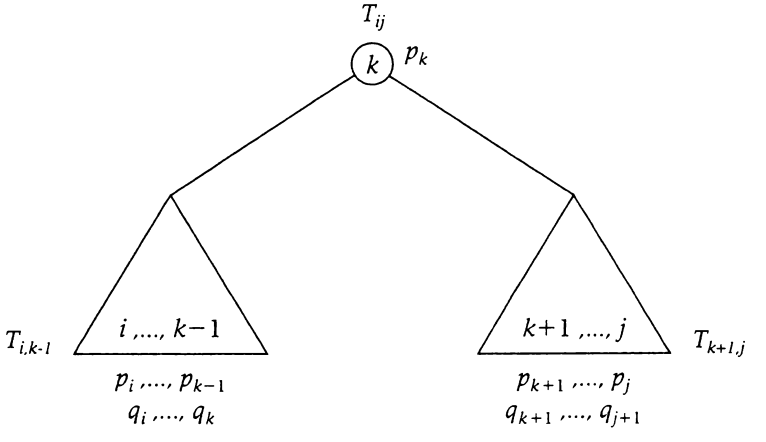
$$A(T) = A(L) + A(R) + \sigma(\mathbf{p}, \mathbf{q}). \tag{9.3.4}$$

We now employ recurrence relation (9.3.4) to show that the Principle of Optimality holds for the problem of finding an optimal search tree. Suppose that T is an optimal search tree; that is, T minimizes $A(T)$. We must show that L and R are also optimal search trees. Suppose there exists a binary search tree L' with $i - 1$ nodes involving the keys $K_0, \ldots, K_{i-1}$ such that $A(L') < A(L)$. Clearly, the tree T' obtained from T by replacing L with L' is a binary search tree. Further, it follows from Formula (9.3.4) that $A(T') < A(T)$, contradicting the assumption that T is an optimal binary search tree. Hence, L is an optimal binary search tree. By symmetry, R is also an optimal binary search tree, which establishes that the Principle of Optimality holds for the optimal binary search tree problem.

Because the Principle of Optimality holds, when constructing an optimal search tree T, we need only consider binary search trees L and R, both of which are optimal. This observation, together with recurrence relation (9.3.4), is the basis of the following dynamic programming algorithm for constructing an optimal binary search tree. For $i,j \in \{0, \ldots, n-1\}$, we let T_{ij} denote an *optimal* search tree involving the consecutive keys $K_i, K_{i+1}, \ldots, K_j$, where T_{ij} is the null tree if $i > j$. Thus, if K_k is the root key, then the left subtree L is $T_{i,k-1}$, and the right subtree R is $T_{k+1,j}$ (see Figure 9.5). Moreover, $T = T_{0,n-1}$ is an optimal search tree involving all n keys. For convenience, we define $\sigma(i,j) = \sum_{k=i}^{j} p_k + \sum_{k=i}^{j+1} q_k$.

FIGURE 9.5

Principle of Optimality: If T_{ij} is optimal, then L and R must be optimal; that is, $L = T_{i,k-1}$ and $R = T_{k+1,j}$.



We define $A(T_{ij})$ by

$$A(T_{ij}) = A(T_{ij}, j - i + 1, p_i, p_{i+1}, \ldots, p_j, q_i, q_{i+1}, \ldots, q_{j+1}). \quad \textbf{(9.3.5)}$$

Because the keys are sorted in nondecreasing order, it follows from the Principle of Optimality and (9.3.4) that

$$A(T_{ij}) = \min_k\{A(T_{i,k-1}) + A(T_{k+1,j})\} + \sigma(i,j), \quad \textbf{(9.3.6)}$$

where the minimum is taken over all $k \in \{i, i+1, \ldots, j\}$.

Recurrence relation (9.3.6) yields an algorithm for computing an optimal search tree T. The algorithm begins by generating all single-node binary search trees, which are trivially optimal. Namely, $T_{00}, T_{11}, \ldots, T_{n-1,n-1}$. Using (9.3.6), the algorithm can then generate optimal search trees $T_{01}, T_{12}, \ldots, T_{n-2,n-1}$. In general, at the k^{th} stage in the algorithm, recurrence relation (9.3.6) is applied to construct the optimal search trees $T_{0,k-1}, T_{1,k}, \ldots, T_{n-k,n-1}$, using the previously generated optimal search trees as building blocks. Figure 9.6 illustrates the algorithm for a sample instance involving $n = 4$ keys. Note that there are two possible choices for T_{02} in Figure 9.6, each having a minimum cost of 1.1. The tree with the smaller root key was selected.

FIGURE 9.6

Action of algorithm to find an optimal binary search tree using dynamic programing.

i	0	1	2	3	4
p_i	.15	.1	.2	.3	
q_i	.05	.05	0	.05	.1

$T_{00} =$ (0) $T_{11} =$ (1) $T_{22} =$ (2) $T_{33} =$ (3)

$A(T_{00}) = .25$ $A(T_{11}) = .15$ $A(T_{22}) = .25$ $A(T_{33}) = .45$

{ (0)–(1) , (1)–(0) } $\Rightarrow T_{01} =$ (0)–(1) $A(T_{01}) = .5$

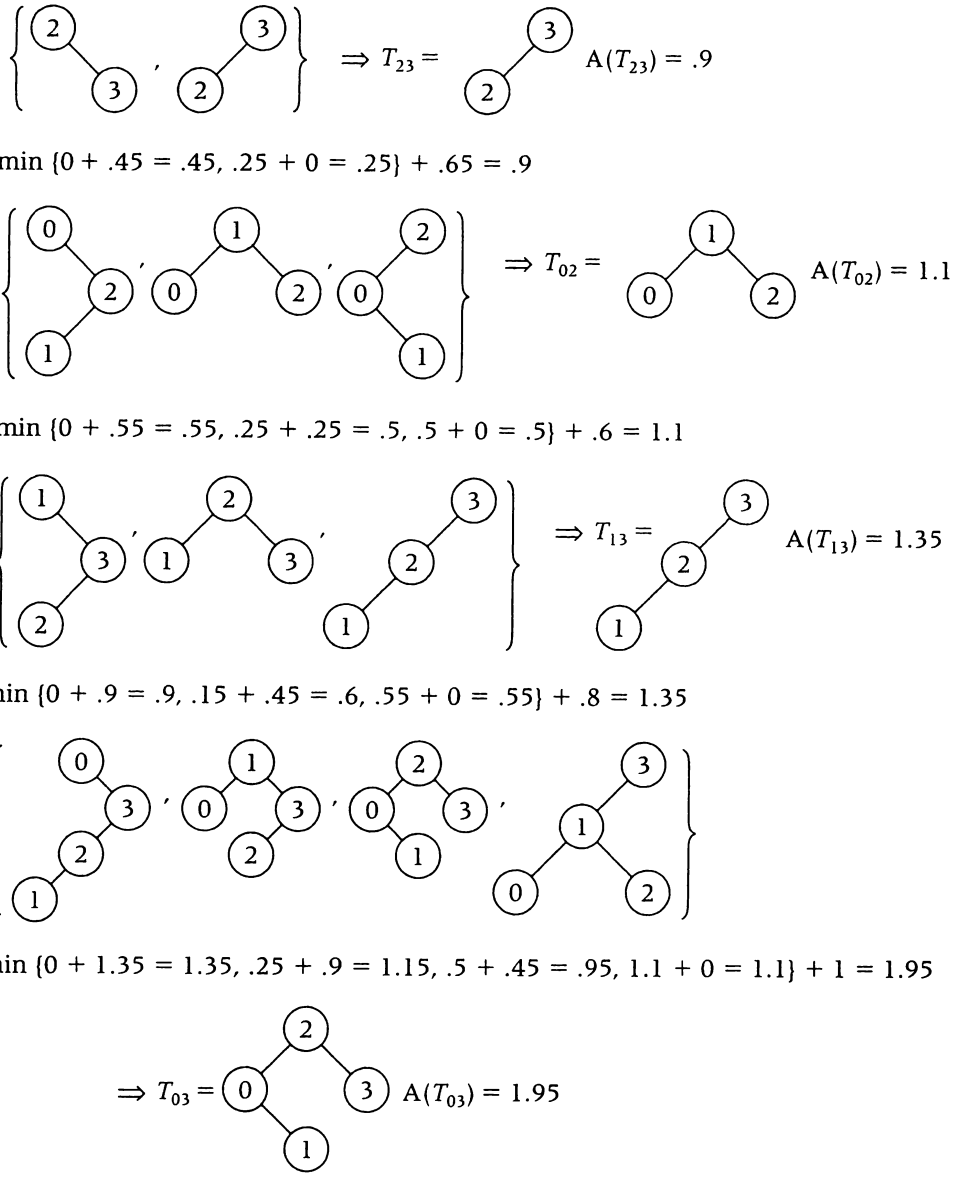
min {0 + .15 = .15, .25 + 0 = .25} + .35 = .5

{ (1)–(2) , (2)–(1) } $\Rightarrow T_{12} =$ (2)–(1) $A(T_{12}) = .55$

min {0 + .25 = .25, .15 + 0 = .15} + .4 = .55

continued

FIGURE 9.6
Continued



The following pseudocode for the algorithm *OptimalSearchTree* implements the preceding strategy. *OptimalSearchTree* computes the root, *Root*[*i*, *j*], of each tree T_{ij}, and the cost, *A*[*i*, *j*], of T_{ij}. An optimal binary search tree *T* for all the keys (namely, $T_{0,n-1}$) can be easily constructed using recursion from the two-dimensional array *Root*[0:*n* − 1, 0:*n* − 1].

```
procedure OptimalSearchTree(P[0:n − 1], Q[0:n], Root[0:n − 1, 0:n − 1], A[0:n − 1,
          0:n − 1])
Input:    P[0:n − 1] (an array of probabilities associated with successful searches)
          Q[0:n] (an array of probabilities associated with unsuccessful searches)
Output:   Root[0:n − 1, 0:n − 1] (Root[i, j] is the key of the root node of T_ij)
          A[0:n − 1, 0:n − 1] (A[i, j] is the cost A(T_ij) of T_ij)
   for i ← 0 to n − 1 do
      Root[i, i] ← i
      Sigma[i, i] ← p[i] + q[i ] + q[i + 1]
      A[i, i] ← Sigma[i, i]
   endfor
   for Pass ← 1 to n − 1 do // Pass is one less than the size of the optimal
                                trees T_ij being constructed in the given pass.
      for i ← 0 to n − 1 − Pass do
         j ← i + Pass
                                //Compute σ(p_i, . . . , p_j, q_i, . . ., q_j+1)
         Sigma[i, j] ← Sigma[i, j − 1] + p[j] + q[j+1]
         Root[i, j] ← i
         Min ← A[i + 1, j]
         for k ← i + 1 to j do
            Sum ← A[i, k − 1] + A[k + 1, j]
            if Sum < Min then
               Min ← Sum
               Root[i, j] ← k
            endif
         endfor
         A[i, j] ← Min + Sigma[i, j]
      endfor
   endfor
end OptimalSearchTree
```

In Figure 9.7, we illustrate the action of *OptimalSearchTree* for the optimal binary search tree just described. For convenience, we change all the probabilities to frequencies, which, in the case we are considering, result by multiplying each probability by 100 to change all the numbers to integers. As we have remarked, working with frequencies instead of probabilities can always be done. In fact, when we are constructing optimal subtrees, it is actually frequencies that we are dealing with instead of probabilities. The optimal subtrees T_{ij} are built starting from the base case T_{ii}, $i = 0, 1, 2, 3$. The figure shows how the tables are built during each pass for $A(T_{ij}) = A[i, j]$,

$Root(T_{ij}) = Root[i, j]$, and $Sigma[i, j] = p_i + \cdots + p_j + q_i + \cdots + q_{j+1} = Sigma[i, j-1] + p_j + q_{j+1}$.

FIGURE 9.7

Building arrays *A*[*i*, *j*], *Root*[*i*, *j*], *Sigma*[*i*, *j*] from inputs *P*[0:3] = (15,10,20,30) and *Q*[0:4] = (5,5,0,5,10) to *OptimalSearchTree*.

i	0	1	2	3	4
p_i	15	10	20	30	
q_i	5	5	0	5	10

A[*i*, *j*]

i \ *j*	0	1	2	3
0	25	*	*	*
1		15	*	*
2			25	*
3				45

Root[*i*, *j*]

i \ *j*	0	1	2	3
0	0	*	*	*
1		1	*	*
2			2	*
3				3

Sigma[*i*, *j*]

i \ *j*	0	1	2	3
0	25	*	*	*
1		15	*	*
2			25	*
3				45

i \ *j*	0	1	2	3
0	25	50	*	*
1		15	55	*
2			25	90
3				45

i \ *j*	0	1	2	3
0	0	0	*	*
1		1	2	*
2			2	3
3				3

i \ *j*	0	1	2	3
0	25	35	*	*
1		15	40	*
2			25	65
3				45

i \ *j*	0	1	2	3
0	25	50	110	*
1		15	55	135
2			25	90
3				45

i \ *j*	0	1	2	3
0	0	0	1	*
1		1	2	3
2			2	3
3				3

i \ *j*	0	1	2	3
0	25	35	60	*
1		15	40	80
2			25	65
3				45

i \ *j*	0	1	2	3
0	25	50	110	195
1		15	55	135
2			25	90
3				45

i \ *j*	0	1	2	3
0	0	0	1	2
1		1	2	3
2			2	3
3				3

i \ *j*	0	1	2	3
0	25	35	60	100
1		15	40	80
2			25	65
3				45

Because *OptimalSearchTree* does the same amount of work for any input $P[0{:}n-1]$ and $Q[0{:}n]$, the best-case, worst-case, and average complexities are equal. Clearly, the number of additions made in computing *Sum* has the same order as the total number of additions made by *OptimalSearchTree*. Therefore, we choose the addition made in computing *Sum* as the basic operation. Since *Pass* varies from 1 to $n - 1$, i varies from 0 to $n - 1 - Pass$, and k varies from $i + 1$ to

$i + Pass$, it follows that the total number of additions made in computing *Sum* is given by

$$\begin{aligned}
&\sum_{t=1}^{n-1} \sum_{i=0}^{n-1-t} \sum_{k=i+1}^{i+t} 1 \\
&= \sum_{t=1}^{n-1} (n-t)t \\
&= n\sum_{t=1}^{n-1} t - \sum_{t=1}^{n-1} t^2 \\
&= n\left[(n-1)\frac{n}{2}\right] - \left[(n-1)n\frac{(2n-1)}{6}\right] \in \Theta(n^3).
\end{aligned}$$

9.4 Longest Common Subsequence

In this section, we consider the problem of determining how close two character strings are to one another. For example, a spell checker might compare a text string created on a word processor with pattern strings from a stored dictionary. If there is no exact match between the text string and any pattern string, then the spell checker offers several alternative pattern strings that are fairly close, in some sense, to the text string. As another example, a forensic scientist might compare two DNA strings to measure how close they match.

We can measure the closeness of two strings in several ways. In Chapter 20, we will consider one important measure called the *edit distance*, which is commonly used by search engines to find approximate matchings for a user-entered text string for which an exact match cannot be found. The edit distance is also used by spell checkers. Roughly speaking, the edit distance between two strings is the minimum number of changes that need to be made (adding, deleting, or changing characters) to transform one string to the other.

In this section, we consider another closeness measure, the longest common subsequence (LCS) contained in a text string and a particular pattern string. Computing either the LCS or the edit distance is an optimization problem that satisfies the Principle of Optimality and can be solved using dynamic programming. However, the solution to the LCS problem is easier to understand because it has a simpler recurrence relation, which we now describe.

Suppose $T = T_0 T_1 \cdots T_{n-1}$ is a text string that we want to compare to a pattern string $P = P_0 P_1 \cdots P_{m-1}$, where we assume that the characters in each string are drawn from some fixed alphabet A. A *subsequence* of T is a string of the form $T_{i_1} T_{i_2} \cdots T_{i_k}$, where $0 \leq i_1 < i_2 < \cdots < i_k \leq n-1$. Note that a *substring* of T is a special case of a subsequence of T in which the subscripts making up the subse-

quence increase by one. For example, consider the pattern string "Cincinnati" and the text string "Cincinatti" (a common misspelling). You can easily check that the longest common subsequence of the pattern string and the text string has length 9 (just one less than the common length of both strings), whereas it takes two changes to transform the text string to the pattern string (so that the edit distance between the two strings is two).

We now describe a dynamic programming algorithm to determine the length of the longest common subsequence of T and P. For simplicity of notation, we assume that the strings are stored in arrays $T[0{:}n-1]$ and $P[0{:}m-1]$, respectively. For integers i and j, we define $LCS[i,j]$ to be the length of the longest common subsequence of the substrings $T[0{:}i-1]$ and $P[0{:}j-1]$ (so that $LCS[n, m]$ is the length of the longest common subsequence of T and P). For convenience, we set $LCS[i,j] = 0$ if $i = 0$ or $j = 0$ (corresponding to empty strings). Note that $LCS[1, 1] = 1$ if $T[0] = P[0]$; otherwise, $LCS[1, 1] = 0$. This initial condition is actually a special case of the following recurrence relation for $LCS[i,j]$:

$$LCS[i,j] = LCS[i-1,j-1] + 1 \quad \text{if } T[i-1] = P[j-1];$$
$$\text{otherwise, } LCS[i,j] = \max\{LCS[i,j-1], LCS[i-1,j]\}. \qquad \textbf{(9.4.1)}$$

To verify recurrence relation (9.4.1), note first that if $T[i-1] \neq P[j-1]$, then a longest common subsequence of $T[0{:}i-1]$ and $P[0{:}j-1]$ might end in $T[i-1]$ or $P[j-1]$, but certainly not both. In other words, if $T[i-1] \neq P[j-1]$, then a longest common subsequence of $T[0{:}i-1]$ and $P[0{:}j-1]$ must be drawn from either the pair $T[0{:}i-2]$ and $P[0{:}j-1]$ or from the pair $T[0{:}i-1]$ and $P[0{:}j-2]$. Moreover, such a longest common subsequence must be a longest common subsequence of the pair of substrings from which it is drawn (that is, the principle of optimality holds). This verifies that

$$LCS[i,j] = \max\{LCS[i,j-1], LCS[i-1,j]\} \text{ if } T[i-1] \neq P[j-1]. \textbf{(9.4.2)}$$

On the other hand, if $T[i-1] = P[j-1] = C$, then a longest common subsequence must end either at $T[i-1]$ in $T[0{:}i-1]$ or at $P[j-1]$ in $P[0, j-1]$, or both,; otherwise, by adding the common value C to a given subsequence, we would increase the length of the subsequence by 1. Also, if the last term of a longest common subsequence ends at an index $k < i-1$ in $T[0{:}i-1]$ (so that $T[k] = C$), then clearly we achieve an equivalent longest common subsequence by swapping $T[i-1]$ for $T[k]$ in the subsequence. By a similar argument involving $P[0{:}j-1]$, when $T[i-1] = P[j-1]$, we can assume without loss of generality that a longest common subsequence in $T[0{:}i-1]$ and $P[0{:}j-1]$ ends at $T[i-1]$ and $P[j-1]$. However, then removing these end points from the subsequence clearly

must result in a longest common subsequence in $T[0{:}i-2]$ and $P[0{:}j-2]$, respectively (that is, the principle of optimality again holds). It follows that

$$LCS[i,j] = LCS[i-1,j-1] + 1 \text{ if } T[i-1] = P[j-1], \quad \textbf{(9.4.3)}$$

which, together with Formula (9.4.2), completes the verification of Formula (9.4.1).

The following algorithm is the straightforward row-by-row computation of the array $LCS[0{:}n, 0{:}m]$ based on the recurrence (9.4.1).

```
procedure LongestCommonSubseq(T[0:n − 1], P[0:m − 1], LCS[0:n, 0:m])
Input:  T[0:n − 1], P[0:m − 1] (strings)
Output: LCS[0:n, 0:m] (array such that LCS[i, j] is length of the longest common
          subsequence of T[0:i − 1] and P[0:j − 1] )
   for i ← 0 to n do      // initialize for boundary conditions
      LCS[i, 0] ← 0
   endfor
   for j ← 0 to m do      // initialize for boundary conditions
      LCS[0, j] ← 0
   endfor
   for i ← 1 to n do      // compute the row index by i of LCS[0:n, 0:m]
      for j ← 1 to m do   // compute LCS[i, j] using (9.4.1)
         if T[i − 1] = P[j − 1] then
            LCS[i, j] ← LCS[i − 1, j − 1] + 1
         else
            LCS[i, j] ← max(LCS[i, j − 1], LCS[i − 1, j])
         endif
      endfor
   endfor
end LongestCommonSubseq
```

Using the comparison of text characters as our basic operation, we see that *LongestCommonSubseq* has complexity in $O(nm)$, which is a rather dramatic improvement over the exponential complexity $\Theta(2^n m)$ brute-force algorithm that would examine each of the 2^n subsequences of $T[0{:}n-1]$ and determine the longest subsequence that also occurs in $P[0{:}m-1]$.

Figure 9.8 shows the array $LCS[0{:}8, 0{:}11]$ output by *LongestCommonSubseq* for $T[0{:}7]$ = "usbeeune" and $P[0{:}10]$ = "subsequence".

Note that *LongestCommonSubseq* determines the length of the longest common subsequence of $T[0{:}n-1]$ and $P[0{:}m-1]$ but does not output the actual subsequence itself. In the previous problem of finding the optimal paranthesization of a chained matrix product, in addition to knowing the minimum number of multi-

plications required, it was also important to determine the actual paranthesization that did the job. Thus, we needed to compute the array $FirstCut[0{:}n-1, 0{:}n-1]$ to be able to construct the optimal paranthesization. Similarly, in the optimal binary search tree problem, in addition to knowing the average search complexity of the optimal search tree, it was important to determine the optimal search tree itself. Thus, we kept track of the key $Root[i, j]$ in the root of the optimal binary search tree containing the keys $K_i < \cdots < K_j$. However, in the LCS problem, knowing the actual common subsequence is not as important as knowing its length. For example, in a process like spell checking, the subsequence is not as important as its length. Typically, to correct a misspelled word in the text, the spell checker displays a list of pattern strings that share subsequences exceeding a threshold length (depending on the length of the strings), as opposed to exhibiting common subsequences. Nevertheless, it is interesting that a longest common subsequence can be determined just from the array $LCS[0{:}n-1, 0{:}m-1]$ (and $T[0{:}n-1]$ and $P[0{:}m-1]$) without the need to maintain any additional information.

One way we can generate a longest common subsequence is to start at the bottom-right corner (n, m) of the array LCS and work our way backward through the array to build the subsequence in reverse order. The moves are dictated by looking at how we get the value assigned to a given position when we used (9.4.1) to build the array LCS. More precisely, if we are currently at position (i, j) in LCS, and $T[i-1] = P[j-1]$, then this common value is appended to the beginning of the string already generated (starting with the null string), and we move to position $(i-1, j-1)$ in LCS. On the other hand, if $T[i-1] \neq P[j-1]$, then we move to position $(i-1, j)$ or $(i, j-1)$, depending on whether $LCS[i-1, j]$ is greater than $LCS[i, j-1]$. When $LCS[i-1, j]$ is equal to $LCS[i, j-1]$, either move can be made. In the latter case, the two different choices might not only generate different longest common subsequences but also yield different longest common strings corresponding to these subsequences. For example, Figure 9.8a

FIGURE 9.8(a)

The matrix $LCS[0{:}8, 0{:}11]$ for the strings $T[0{:}7]$ = "usbeeune" and $P[0{:}10]$ = "subsequence", with the path in LCS generating the longest common string "sbeune" using the move-left rule.

			s	u	b	s	e	q	u	e	n	c	e
	LCS	0	1	2	3	4	5	6	7	8	9	10	11
	0	0	0	0	0	0	0	0	0	0	0	0	0
u	1	0	0	1	1	1	1	1	1	1	1	1	1
s	2	0	1	1	1	2	2	2	2	2	2	2	2
b	3	0	1	1	2	2	2	2	2	2	2	2	2
e	4	0	1	1	2	2	3	3	3	3	3	3	3
e	5	0	1	1	2	2	3	3	3	4	4	4	4
u	6	0	1	2	2	2	3	3	4	4	4	4	4
n	7	0	1	2	2	2	3	3	4	4	5	5	5
e	8	0	1	2	2	2	3	3	4	5	5	5	6

			s	u	b	s	e	q	u	e	n	c	e
	LCS	0	1	2	3	4	5	6	7	8	9	10	11
	0	0	0	0	0	0	0	0	0	0	0	0	0
u	1	0	0	1	1	1	1	1	1	1	1	1	1
s	2	0	1	1	1	2	2	2	2	2	2	2	2
b	3	0	1	1	2	2	2	2	2	2	2	2	2
e	4	0	1	1	2	2	3	3	3	3	3	3	3
e	5	0	1	1	2	2	3	3	3	4	4	4	4
u	6	0	1	2	2	2	3	3	4	4	4	4	4
n	7	0	1	2	2	2	3	3	4	4	5	5	5
e	8	0	1	2	2	2	3	3	4	5	5	5	6

FIGURE 9.8(b)

The path in the matrix *LCS* generating the longest common string "useene" using the move-up rule.

shows the path generated using the move-left rule, which requires us to move to position $(i-1, j)$ when $T[i-1] \neq P[j-1]$, whereas Figure 9.8b shows the path resulting from always moving up to position $(i, j-1)$. The darker shaded positions (i, j) in these paths correspond to where $T[i-1] = P[j-1]$. These two paths yield the longest common strings "sbeune" and "useune", respectively. When generating a path in *LCS*, we obtain a longest common subsequence when we reach a position where $LCS[i, j] = 0$.

9.5 Closing Remarks

In subsequent chapters, we will use dynamic programming to solve a number of important problems. For example, in Chapter 12, we will discuss Floyd's dynamic programming solution to the all-pairs shortest-path problem in weighted directed graphs. In Chapter 20 we will use dynamic programming to solve the edit distance version of the approximate string matching problem. Dynamic programming, because it is based on a bottom-up resolution of recurrence relations, is usually amenable to straightforward level-by-level parallelization. However, this straightforward parallelization usually does not result in optimal speedup, and more clever parallel algorithms, sometimes based on finding recurrences better suited to parallelization, must be sought. We will see such an example in Chapter 16 for computing shortest paths.

References and Suggestions for Further Reading

Bellman, R. E. *Dynamic Programming*. Princeton, NJ: Princeton University Press, 1957. The first systematic study of dynamic programming.

Two other classic references on dynamic programming are:

Bellman, R. E., and S. E. Dreyfus. *Applied Dynamic Programming*. Princeton, NJ: Princeton University Press, 1962.

Nemhauser, G. *Introduction to Dynamic Programming*. New York: Wiley, 1966.

EXERCISES

Section 9.1 Optimization Problems and the Principle of Optimality

9.1 Suppose the matrix $C[0{:}n-1, 0{:}n-1]$ contains the cost of $C[i, j]$ of flying directly from airport i to airport j. Consider the problem of finding the cheapest flight from i to j where we may fly to as many intermediate airports as desired. Verify that the Principle of Optimality holds for the minimum-cost flight. Derive a recurrence relation based on the Principle of Optimality.

9.2 Does the Principle of Optimality hold for the costliest trips (no revisiting of airports, please)? Discuss.

9.3 Does the Principle of Optimality hold for coin changing? Discuss with various interpretations of the *Combine* function.

Section 9.2 Optimal Parenthesization for Computing a Chained Matrix Product

9.4 Given the matrix product $M_0M_1 \cdots M_{n-1}$ and a 2-tree T with n leaves, show that there is a unique parenthesization P such that $T = T(P)$.

9.5 Give pseudocode for *ParenthesizeRec* and analyze its complexity.

9.6 Show that the complexity of *OptimalParenthesization* is in $\Theta(n^3)$.

9.7 Using *OptimalParenthesization*, find an optimal parenthesization for the chained product of five matrices with dimensions 6×7, 7×8, 8×3, 3×10, and 10×6.

9.8 Write a program implementing *OptimalParenthesization* and run it for some sample inputs.

Section 9.3 Optimal Binary Search Trees

9.9 Use dynamic programming to find an optimal binary search tree for the following probabilities, where we assume that the search key is in the search tree; that is, $q_i = 0$, $i = 0, \ldots, n$:

keys	i	0	1	2	3
probabilities	p_i	.4	.3	.2	.1

9.10 Use dynamic programming to find an optimal search tree for the following probabilities:

i	0	1	2	3	4	5
p_i	.2	.1	.2	.05	.05	
q_i	.05	0	.25	0	.1	0

9.11 a. Design and analyze a recursive algorithm that computes an optimal binary search tree T for all the keys from the two-dimensional array *Root*[0:$n-1$, 0:$n-1$] generated by *OptimalSearchTree*.

b. Show the action of your algorithm from part (a) for the instance given in Exercise 9.10.

9.12 a. Give a set of probabilities $p_0, \ldots, p_{n-1}$ (assume a successful search so that $q_0 = q_1 = \cdots = q_n = 0$), such that a completely right-skewed search tree T (the left child of every node is **null**) is an optimal search tree with respect to these probabilities.

b. More generally, prove the following induction on n: If T is *any* given binary search tree with n nodes, then there exists a set of probabilities $p_0, \ldots, p_{n-1}$ such that T is the unique optimal binary search tree with respect to these probabilities.

Section 9.4 Longest Common Subsequence

9.13 Show the array *LCS*[0:9, 0:10] that is built by *LongestCommonSubseq* for *T*[0:8] = "alligator" and *P*[0:9] = "algorithms".

9.14 By following various paths in the array *LCS*[0:8, 0:11] given in Figure 9.8, find all the longest common subsequences of the strings "usbeeune" and "subsequence".

9.15 Design and analyze an algorithm that generates all longest common subsequences given the input array *LCS*[0:$n-1$, 0:$m-1$].

9.16 Write a program that implements *LongestCommonSubseq*, and run it for some sample inputs.

Additional Problems

9.17 Consider a sequence of n distinct integers. Design and analyze a dynamic programming algorithm to find the length of the longest increasing subsequence. For example, consider the sequence:

45 23 9 3 99 108 76 12 77 16 18 4

A longest increasing subsequence is 3 12 16 18, having length 4.

9.18 The 0/1 knapsack problem is NP-hard when the input is measured in binary. However, when the input is measured in unary (see the discussion at the end of Section 7.3 of Chapter 7), dynamic programming can be used to find a polynomial-complexity solution. Design and analyze a dynamic programming solution to the 0/1 knapsack problem, with positive integer capacity and weights, which is quadratic in $C + n$, where C is the capacity and n is the number of objects. *Hint:* Let $V[i, j]$ denote the maximum value that can be placed in a knapsack of capacity j using objects drawn from $\{b_0, \ldots, b_{i-1}\}$. Use the principle of optimality to find a recurrence relation for $V[i, j]$.

9.19 Design and analyze a dynamic programming solution to the coin-changing problem under similar assumptions to that in the previous exercise.

9.20 Given n integers, the partition problem is to find a bipartition of the integers into two subsets having the same sum or determine that no such bipartition exists. Design and analyze a dynamic programming algorithm for solving the partition problem.