12.2 Dynamic Programming

In this section, we discuss the ***dynamic programming*** algorithm-design technique. This technique is similar to the divide-and-conquer technique (Section 11.1.1), in that it can be applied to a wide variety of different problems. There are few algorithmic techniques that can take problems that seem to require exponential time and produce polynomial-time algorithms to solve them. Dynamic programming is one such technique. In addition, the algorithms that result from applications of the dynamic programming technique are usually quite simple—often needing little more than a few lines of code to describe some nested loops for filling in a table.

12.2.1 Matrix Chain-Product

Rather than starting out with an explanation of the general components of the dynamic programming technique, we begin by giving a classic, concrete example. Suppose we are given a collection of n two-dimensional arrays (matrices) for which we wish to compute the product

$$A = A_0 \cdot A_1 \cdot A_2 \cdots A_{n-1},$$

where A_i is a $d_i \times d_{i+1}$ matrix, for $i = 0, 1, 2, \ldots, n-1$. In the standard matrix multiplication algorithm (which is the one we use), to multiply a $d \times e$-matrix B times an $e \times f$-matrix C, we compute the product, A, as

$$A[i][j] = \sum_{k=0}^{e-1} B[i][k] \cdot C[k][j].$$

This definition implies that matrix multiplication is associative, that is, it implies that $B \cdot (C \cdot D) = (B \cdot C) \cdot D$. Thus, we can parenthesize the expression for A any way we wish and we still end up with the same answer. We do not necessarily perform the same number of primitive (that is, scalar) multiplications in each parenthesization, however, as is illustrated in the following example.

Example 12.2: *Let B be a 2×10-matrix, let C be a 10×50-matrix, and let D be a 50×20-matrix. Computing $B \cdot (C \cdot D)$ requires $2 \cdot 10 \cdot 20 + 10 \cdot 50 \cdot 20 = 10,400$ multiplications, whereas computing $(B \cdot C) \cdot D$ requires $2 \cdot 10 \cdot 50 + 2 \cdot 50 \cdot 20 = 3000$ multiplications.*

The ***matrix chain-product*** problem is to determine the parenthesization of the expression defining the product A that minimizes the total number of scalar multiplications performed. As the example above illustrates, the differences between different solutions can be dramatic, so finding a good solution can result in significant speedups.

Defining Subproblems

Of course, one way to solve the matrix chain-product problem is to simply enumerate all the possible ways of parenthesizing the expression for A and determine the number of multiplications performed by each one. Unfortunately, the set of all different parenthesizations of the expression for A is equal in number to the set of all different binary trees that have n external nodes. This number is exponential in n. Thus, this straightforward ("brute force") algorithm runs in exponential time, for there are an exponential number of ways to parenthesize an associative arithmetic expression.

We can improve the performance achieved by the brute-force algorithm significantly, however, by making a few observations about the nature of the matrix chain-product problem. The first observation is that the problem can be split into ***subproblems***. In this case, we can define a number of different subproblems, each of which computes the best parenthesization for some subexpression $A_i \cdot A_{i+1} \cdots A_j$. As a concise notation, we use $N_{i,j}$ to denote the minimum number of multiplications needed to compute this subexpression. Thus, the original matrix chain-product problem can be characterized as that of computing the value of $N_{0,n-1}$. This observation is important, but we need one more in order to apply the dynamic programming technique.

Characterizing Optimal Solutions

The other important observation we can make about the matrix chain-product problem is that it is possible to characterize an optimal solution to a particular subproblem in terms of optimal solutions to its subproblems. We call this property the ***subproblem optimality*** condition.

In the case of the matrix chain-product problem, we observe that, no matter how we parenthesize a subexpression, there has to be some final matrix multiplication that we perform. That is, a full parenthesization of a subexpression $A_i \cdot A_{i+1} \cdots A_j$ has to be of the form $(A_i \cdots A_k) \cdot (A_{k+1} \cdots A_j)$, for some $k \in \{i, i+1, \ldots, j-1\}$. Moreover, for whichever k is the correct one, the products $(A_i \cdots A_k)$ and $(A_{k+1} \cdots A_j)$ must also be solved optimally. If this were not so, then there would be a global optimal that had one of these subproblems solved suboptimally. But this is impossible, since we could then reduce the total number of multiplications by replacing the current subproblem solution by an optimal solution for the subproblem. This observation implies a way of explicitly defining the optimization problem for $N_{i,j}$ in terms of other optimal subproblem solutions. Namely, we can compute $N_{i,j}$ by considering each place k where we could put the final multiplication and taking the minimum over all such choices.

Designing a Dynamic Programming Algorithm

We can therefore characterize the optimal subproblem solution, $N_{i,j}$, as

$$N_{i,j} = \min_{i \le k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\},$$

where $N_{i,i} = 0$, since no work is needed for a single matrix. That is, $N_{i,j}$ is the minimum, taken over all possible places to perform the final multiplication, of the number of multiplications needed to compute each subexpression plus the number of multiplications needed to perform the final matrix multiplication.

Notice that there is a ***sharing of subproblems*** going on that prevents us from dividing the problem into completely independent subproblems (as we would need to do to apply the divide-and-conquer technique). We can, nevertheless, use the equation for $N_{i,j}$ to derive an efficient algorithm by computing $N_{i,j}$ values in a bottom-up fashion, and storing intermediate solutions in a table of $N_{i,j}$ values. We can begin simply enough by assigning $N_{i,i} = 0$ for $i = 0, 1, \ldots, n-1$. We can then apply the general equation for $N_{i,j}$ to compute $N_{i,i+1}$ values, since they depend only on $N_{i,i}$ and $N_{i+1,i+1}$ values that are available. Given the $N_{i,i+1}$ values, we can then compute the $N_{i,i+2}$ values, and so on. Therefore, we can build $N_{i,j}$ values up from previously computed values until we can finally compute the value of $N_{0,n-1}$, which is the number that we are searching for. The details of this ***dynamic programming*** solution are given in Code Fragment 12.1.

Algorithm MatrixChain$(d_0, \ldots, d_n)$:
Input: Sequence $d_0, \ldots, d_n$ of integers
Output: For $i, j = 0, \ldots, n-1$, the minimum number of multiplications $N_{i,j}$ needed to compute the product $A_i \cdot A_{i+1} \cdots A_j$, where A_k is a $d_k \times d_{k+1}$ matrix

```
for i ← 0 to n − 1 do
    N_{i,i} ← 0
for b ← 1 to n − 1 do
    for i ← 0 to n − b − 1 do
        j ← i + b
        N_{i,j} ← +∞
        for k ← i to j − 1 do
            N_{i,j} ← min{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}}.
```

Code Fragment 12.1: Dynamic programming algorithm for the matrix chain-product problem.

Thus, we can compute $N_{0,n-1}$ with an algorithm that consists primarily of three nested for-loops. The outside loop is executed n times. The loop inside is executed at most n times. And the inner-most loop is also executed at most n times. Therefore, the total running time of this algorithm is $O(n^3)$.

12.2.2 DNA and Text Sequence Alignment

A common text processing problem, which arises in genetics and software engineering, is to test the similarity between two text strings. In a genetics application, the two strings could correspond to two strands of DNA, that we want to compare. Likewise, in a software engineering application, the two strings could come from two versions of source code for the same program. We might want to compare the two versions to determine what changes have been made from one version to the next. Indeed, determining the similarity between two strings is so common that the Unix and Linux operating systems have a built-in program, `diff`, for comparing text files.

Given a string $X = x_0x_1x_2\cdots x_{n-1}$, a ***subsequence*** of X is any string that is of the form $x_{i_1}x_{i_2}\cdots x_{i_k}$, where $i_j < i_{j+1}$; that is, it is a sequence of characters that are not necessarily contiguous but are nevertheless taken in order from X. For example, the string *AAAG* is a subsequence of the string *CGATAATTGAGA*.

The DNA and text similarity problem we address here is the ***longest common subsequence*** (LCS) problem. In this problem, we are given two character strings, $X = x_0x_1x_2\cdots x_{n-1}$ and $Y = y_0y_1y_2\cdots y_{m-1}$, over some alphabet (such as the alphabet $\{A,C,G,T\}$ common in computational genetics) and are asked to find a longest string S that is a subsequence of both X and Y. One way to solve the longest common subsequence problem is to enumerate all subsequences of X and take the largest one that is also a subsequence of Y. Since each character of X is either in or not in a subsequence, there are potentially 2^n different subsequences of X, each of which requires $O(m)$ time to determine whether it is a subsequence of Y. Thus, this brute-force approach yields an exponential-time algorithm that runs in $O(2^n m)$ time, which is very inefficient. Fortunately, the LCS problem is efficiently solvable using ***dynamic programming***.

The Components of a Dynamic Programming Solution

As mentioned above, the dynamic programming technique is used primarily for ***optimization*** problems, where we wish to find the "best" way of doing something. We can apply the dynamic programming technique in such situations if the problem has certain properties.

Simple Subproblems: There has to be some way of repeatedly breaking the global-optimization problem into subproblems. Moreover, there should be a simple way of defining subproblems with just a few indices, like i, j, k, and so on.

Subproblem Optimization: An optimal solution to the global problem must be a composition of optimal subproblem solutions.

Subproblem Overlap: Optimal solutions to unrelated subproblems can contain subproblems in common.

Applying Dynamic Programming to the LCS Problem

Recall that in the LCS problem, we are given two character strings, X and Y, of length n and m, respectively, and are asked to find a longest string S that is a subsequence of both X and Y. Since X and Y are character strings, we have a natural set of indices with which to define subproblems—indices into the strings X and Y. Let us define a subproblem, therefore, as that of computing the value $L[i,j]$, which we will use to denote the length of a longest string that is a subsequence of both $X[0..i] = x_0x_1x_2\ldots x_i$ and $Y[0..j] = y_0y_1y_2\ldots y_j$. This definition allows us to rewrite $L[i,j]$ in terms of optimal subproblem solutions. This definition depends on which of two cases we are in. (See Figure 12.1.)

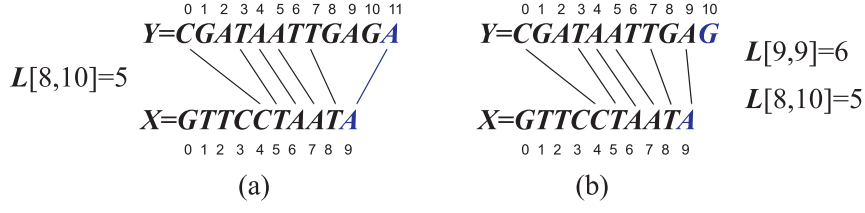

Figure 12.1: The two cases in the longest common subsequence algorithm: (a) $x_i = y_j$; (b) $x_i \neq y_j$. Note that the algorithm stores only the $L[i,j]$ values, not the matches.

- $x_i = y_j$. In this case, we have a match between the last character of $X[0..i]$ and the last character of $Y[0..j]$. We claim that this character belongs to a longest common subsequence of $X[0..i]$ and $Y[0..j]$. To justify this claim, let us suppose it is not true. There has to be some longest common subsequence $x_{i_1}x_{i_2}\ldots x_{i_k} = y_{j_1}y_{j_2}\ldots y_{j_k}$. If $x_{i_k} = x_i$ or $y_{j_k} = y_j$, then we get the same sequence by setting $i_k = i$ and $j_k = j$. Alternately, if $x_{j_k} \neq x_i$, then we can get an even longer common subsequence by adding x_i to the end. Thus, a longest common subsequence of $X[0..i]$ and $Y[0..j]$ ends with x_i. Therefore, we set

$$L[i,j] = L[i-1,j-1]+1 \quad \text{if } x_i = y_j.$$

- $x_i \neq y_j$. In this case, we cannot have a common subsequence that includes both x_i and y_j. That is, we can have a common subsequence end with x_i or one that ends with y_j (or possibly neither), but certainly not both. Therefore, we set

$$L[i,j] = \max\{L[i-1,j], L[i,j-1]\} \quad \text{if } x_i \neq y_j.$$

In order to make both of these equations make sense in the boundary cases when $i = 0$ or $j = 0$, we assign $L[i,-1] = 0$ for $i = -1,0,1,\ldots,n-1$ and $L[-1,j] = 0$ for $j = -1,0,1,\ldots,m-1$.

The LCS Algorithm

The definition of $L[i,j]$ satisfies subproblem optimization, since we cannot have a longest common subsequence without also having longest common subsequences for the subproblems. Also, it uses subproblem overlap, because a subproblem solution $L[i,j]$ can be used in several other problems (namely, the problems $L[i+1,j]$, $L[i,j+1]$, and $L[i+1,j+1]$). Turning this definition of $L[i,j]$ into an algorithm is actually quite straightforward. We initialize an $(n+1)\times(m+1)$ array, L, for the boundary cases when $i=0$ or $j=0$. Namely, we initialize $L[i,-1]=0$ for $i=-1,0,1,\ldots,n-1$ and $L[-1,j]=0$ for $j=-1,0,1,\ldots,m-1$. Then, we iteratively build up values in L until we have $L[n-1,m-1]$, the length of a longest common subsequence of X and Y. We give a pseudo-code description of this algorithm in Code Fragment 12.2.

Algorithm LCS(X,Y):
Input: Strings X and Y with n and m elements, respectively
Output: For $i=0,\ldots,n-1$, $j=0,\ldots,m-1$, the length $L[i,j]$ of a longest string that is a subsequence of both the string $X[0..i]=x_0x_1x_2\cdots x_i$ and the string $Y[0..j]=y_0y_1y_2\cdots y_j$

```
for i ← −1 to n − 1 do
  L[i, −1] ← 0
for j ← 0 to m − 1 do
  L[−1, j] ← 0
for i ← 0 to n − 1 do
  for j ← 0 to m − 1 do
    if x_i = y_j then
      L[i, j] ← L[i − 1, j − 1] + 1
    else
      L[i, j] ← max{L[i − 1, j], L[i, j − 1]}
return array L
```

Code Fragment 12.2: Dynamic programming algorithm for the LCS problem.

The running time of the algorithm of Code Fragment 12.2 is easy to analyze, because it is dominated by two nested **for** loops, with the outer one iterating n times and the inner one iterating m times. Since the if-statement and assignment inside the loop each requires $O(1)$ primitive operations, this algorithm runs in $O(nm)$ time. Thus, the dynamic programming technique can be applied to the longest common subsequence problem to improve significantly over the exponential-time brute-force solution to the LCS problem.

Algorithm LCS (Code Fragment 12.2) computes the length of the longest common subsequence (stored in $L[n-1,m-1]$), but not the subsequence itself. As shown in the following proposition, a simple postprocessing step can extract the longest common subsequence from the array L returned by the algorithm.

Proposition 12.3: *Given a string X of n characters and a string Y of m characters, we can find the longest common subsequence of X and Y in $O(nm)$ time.*

Justification: Algorithm LCS computes $L[n-1,m-1]$, the ***length*** of a longest common subsequence, in $O(nm)$ time. Given the table of $L[i,j]$ values, constructing a longest common subsequence is straightforward. One method is to start from $L[n,m]$ and work back through the table, reconstructing a longest common subsequence from back to front. At any position $L[i,j]$, we can determine whether $x_i = y_j$. If this is true, then we can take x_i as the next character of the subsequence (noting that x_i is ***before*** the previous character we found, if any), moving next to $L[i-1,j-1]$. If $x_i \neq y_j$, then we can move to the larger of $L[i,j-1]$ and $L[i-1,j]$. (See Figure 12.2.) We stop when we reach a boundary cell (with $i=-1$ or $j=-1$). This method constructs a longest common subsequence in $O(n+m)$ additional time. ■

L	-1	0	1	2	3	4	5	6	7	8	9	10	11
-1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1
1	0	0	1	1	2	2	2	2	2	2	2	2	2
2	0	0	1	1	2	2	2	3	3	3	3	3	3
3	0	1	1	1	2	2	2	3	3	3	3	3	3
4	0	**1**	1	1	2	2	2	3	3	3	3	3	3
5	0	1	1	1	**2**	2	2	3	4	4	4	4	4
6	0	1	1	2	2	**3**	3	3	4	4	5	5	5
7	0	1	1	2	2	3	**4**	4	4	4	5	5	6
8	0	1	1	2	3	3	4	5	**5**	5	5	5	6
9	0	1	1	2	3	4	4	5	5	5	6	6	**6**

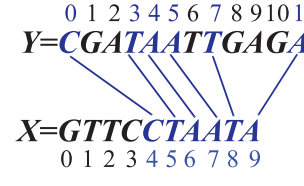

Figure 12.2: The algorithm for constructing a longest common subsequence from the array L.