**Unit 3 Dynammic Programming**

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems.

In divide-and-conquer algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming is

applicable when the subproblems are not independent, that is, when subproblems share subsubproblems.

In divide-and-conquer algorithm does repeatedly solving the common subsubproblems. A dynamic-programming algorithm solves every subsubproblem just once and then saves its answer in a table.

Dynamic programming is typically applied to optimization problems.

The development of a dynamic-programming algorithm can be broken into a sequence of four

steps.

1.Characterize the structure of an optimal solution.

2.Recursively define the value of an optimal solution.

3.Compute the value of an optimal solution in a bottom-up fashion.

4.Construct an optimal solution from computed information.

**Matrix-chain multiplication**

In matrix-chain multiplication, We are given a sequence (chain) A 1 , A 2 , ..., A n of n matrices to be multiplied, and we wish to compute the product.

A product of matrices is fully parenthesized if it is either a

single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. Matrix multiplication is associative, and so all parenthesizations yield the same product.

For example, if the chain of matrices is A 1 , A 2 , A 3 , A 4 , the product A 1 A 2 A 3 A 4 can be fully parenthesized in five distinct ways:

(A 1 (A 2 (A 3 A 4 ))) , (A 1 ((A 2 A 3 ) A 4 )) , ((A 1 A 2 ) (A 3 A 4 )) ,

((A 1 (A 2 A 3 )) A 4 ) , (((A 1 A 2 ) A 3 ) A 4 ).

The way we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product. Consider first the cost of multiplying two matrices.

The matrix-chain multiplication problem can be stated as follows: given a chain A 1 , A 2 , ..., A n of n matrices, where for i = 1, 2, ..., n, matrix A i has dimension p i-1 × p i , fully parenthesize the product A 1 A 2 A n in a way that minimizes the number of scalar multiplications.

**Note that in the matrix-chain multiplication problem, we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices that has the lowest cost.**

**Step 1: The structure of an optimal parenthesization**

For the matrix-chain multiplication problem, we can perform this step as

follows. For convenience, let us adopt the notation A i j , where i ≤ j, for the matrix that results from evaluating the product A i A i+1 A j . Observe that if the problem is nontrivial, i.e., i < j, then any parenthesization of the product A i A i+1 A j must split the product between A k and A k+1 for some integer k in the range i ≤ k < j.

**Step 2: A recursive solution**

For the matrix-chain multiplication problem, we pick as our subproblems the

problems of determining the minimum cost of a parenthesization of A i A i+1 A j for 1 ≤ i ≤ j ≤ n.

Let m[i, j] be the minimum number of scalar multiplications needed to compute the matrix A i j ; for the full problem, the cost of a cheapest way to compute A 1 n would thus be m[1, n].



The m[i, j] values give the costs of optimal solutions to subproblems. To help us keep track of how to construct an optimal solution, let us define s[i, j] to be a value of k at which we can split the product A i A i+1 A j to obtain an optimal parenthesization. That is, s[i, j] equals a value k such that

m[i, j] = m[i, k] + m[k + 1, j] + p i-1 p k p j .

**Step 3: Computing the optimal costs**

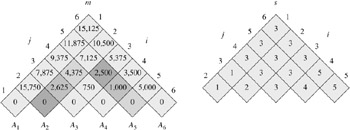
MATRIX-CHAIN-ORDER for n = 6 and the following matrix dimensions:

matrix dimension are

A 1 30 × 35, A 2 35 × 15 , A 3 15 × 5, A 4 5 × 10, A 5 10 × 20,A 6

20 × 25.

The tables are rotated so that the main diagonal runs horizontally. Only the main diagonal and



The minimum number of scalar multiplications to multiply the 6 matrices is m[1, 6] = 15,125.

MATRIX-CHAIN-ORDER(p)

1 n ← length[p] – 1

2 for i ← 1 to n

3do m[i, i] ← 0

4 for l ← 2 to n

5do for i ← 1 to n - l + 1

6do j ← i + l - 1

7m[i, j] ← ∞

8for k ← i to j - 1

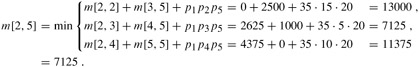
9do q ← m[i, k] + m[k + 1, j] + p i-1 p k p j

10if q < m[i, j]

11then m[i, j] ← q

12s[i, j] ← k

13 return m and s



**Step 4: Constructing an optimal solution**

The following recursive procedure prints an optimal parenthesization of A i , A i+1 , ..., A j , given the s table computed by MATRIX- CHAIN-ORDER and the indices i and j. The initial call PRINT-OPTIMAL-PARENS(s, 1, n) prints an optimal parenthesization of A 1 , A 2 , ..., A n .

PRINT-OPTIMAL-PARENS(s, i, j)

1 if i = j

2 then print "A" i

3 else print "("

4 PRINT-OPTIMAL-PARENS(s, i, s[i, j])

5PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)

6print ")"

In the example of Figure 15.3, the call PRINT-OPTIMAL-PARENS(s, 1, 6) prints the

parenthesization ((A 1 (A 2 A 3 )) ((A 4 A 5 )A 6 )).

**Example**

Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is 5, 10, 3, 12, 5, 50, 6 .

**Longest common subsequence**

In biological applications, we often want to compare the DNA of two (or more) different organisms. A strand of DNA consists of a string of molecules called bases, where the possible bases are adenine, guanine, cytosine, and thymine. Representing each of these bases by their initial letters, a strand of DNA can be expressed as a string over the finite set {A, C, G, T}.

For example, the DNA of one organism may be

S 1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA, while the DNA of another organism

may be S 2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA. One goal of comparing two

strands of DNA is to determine how "similar" the two strands are, as some measure of how closely related the two organisms are.

In the longest-common-subsequence problem, we are given two sequences X = x 1 , x 2 , ...,x m and Y = y 1 , y 2 , ..., y n and wish to find a maximum-length common subsequence of X and Y .

**Step 1: Characterizing a longest common subsequence**

Let X = x 1 , x 2 , ..., x m any LCS of X and Y. and Y = y 1 , y 2 , ..., y n

be sequences, and let Z = z 1 , z 2 , ..., z k be

1. If x m = y n , then z k = x m = y n and Z k-1 is an LCS of X m-1 and Y n-1 .

2. If x m ≠ y n , then z k ≠ x m implies that Z is an LCS of X m-1 and Y.

3. If x m ≠ y n , then z k ≠ y n implies that Z is an LCS of X and Y n-1 .

**Step 2: A recursive solution**

when finding an LCS of X = x 1 , x 2 , ..., x m and Y = y 1 , y 2 , ..., y n . If x m = y n , we must find an LCS of X m-1 and Y n-1 . Appending x m = y n to this LCS yields an LCS of X and Y. If x m ≠ y n , then we must solve two subproblems: finding an LCS of X m-1 and Y and finding an LCS of X and Y n-1 . Whichever of these two LCS's is longer is an LCS of X and Y.

**Step 3.Computing the length of an LCS**

Procedure LCS-LENGTH takes two sequences X = x 1 , x 2 , ..., x m and Y = y 1 , y 2 , ..., y n as inputs. It stores the c[i, j] values in a table c[0 m, 0 n] whose entries are computed in row-major order. It also maintains the table b[1 m, 1 n] to simplify construction of an optimal solution.

The procedure returns the b and c tables; c[m, n] contains the length of an LCS of X and Y.

LCS-LENGTH(X, Y)

1 m ← length[X]

2 n ← length[Y]

3 for i ← 1 to m

4do c[i, 0] ←

5 for j ← 0 to n

6do c[0, j] ←

7 for i ← 1 to m

8do for j ← 1 to n

9do if x i = y j

10 then c[i, j] ← c[i - 1, j - 1] + 1

11 b[i, j] ← " "

12 else if c[i - 1, j] ≥ c[i, j - 1]

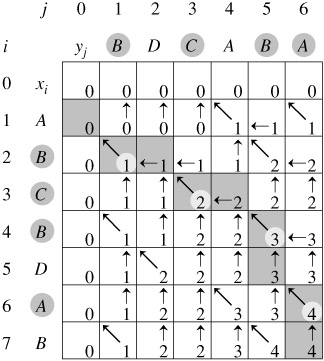
13 b[i, j] ← "↑"

else c[i, j] ← c[i, j - 1]

b [i, j] ← ←

14 return c and b

LCS-LENGTH on the sequences X = {A, B, C, B,D, A, B }and Y = {B, D, C, A, B, A }. The running time of the procedure is O(mn), since each table entry takes O(1) time to compute.



the length of an LCS B, C, B, A of X and Y.