The Matrix Chain Problem

Given a chain of matrices to multiply, determine the how the matrices should be parenthesized to minimize the number of single element multiplications involved.

First off, it should be noted that matrix multiplication is associative, but not commutative. But since it is associative, we always have:

((AB)(CD)) = (A(B(CD)))

or equality for any such grouping as long as the matrices in the product appear in the same order.

It may appear on the surface that the amount of work done won't change if you change the parenthesization of the expression, but we can prove that is not the case with the following example:

Let A be a 2x10 matrix
Let B be a 10x50 matrix
Let C be a 50x20 matrix

Note that any matrix multiplication between a matrix with dimensions ixj and another with dimensions jxk will perform ixjxk element multiplications creating an answer that is a matrix with dimensions ixk. Also note that the second dimension in the first matrix and the first dimension in the second matrix must be equal in order to allow matrix multiplication.

Consider computing A(BC):

# multiplications for (BC) = 10x50x20 = 10000, creating a 10x20 answer matrix

# multiplications for A(BC) = 2x10x20 = 400,

Total multiplications = 10000 + 400 = 10400.

Consider computing (AB)C:

# multiplications for (AB) = 2x10x50 = 1000, creating a 2x50 answer matrix

# multiplications for (AB)C = 2x50x20 = 2000,

Total multiplications = 1000 + 2000 = 3000, a significant difference.

Thus, the goal of the problem is given a chain of matrices to multiply, determine the fewest number of multiplications necessary to compute the product. We will formally define the problem below:

Let $A = A_0 \bullet A_1 \bullet \ldots A_{n-1}$

Let $N_{i,j}$ denote the minimal number of multiplications necessary to find the product $A_i \bullet A_{i+1} \bullet \ldots A_j$. And let $d_i x d_{i+1}$ denote the dimensions of matrix A_i.

We must attempt to determine the minimal number of multiplications necessary($N_{0,n-1}$) to find A, assuming that we simply do each single matrix multiplication in the standard method.

The key to solving this problem is noticing the sub-problem optimality condition:

If a particular parenthesization of the whole product is optimal, then any sub-parenthesization in that product is optimal as well. Consider the following illustration:

Assume that we are calculating ABCDEF and that the following parenthesization is optimal:

(A (B ((CD) (EF))))

Then it is necessarily the case that

(B ((CD) (EF)))

is the optimal parenthesization of BCDEF.

Why is this?

Because if it wasn't, and say (((BC) (DE)) F) was better, then it would also follow that

(A (((BC) (DE)) F)) was better than

(A (B ((CD) (EF)))), contradicting its optimality.

This line of reasoning is nearly identical to the reasoning we used when deriving Floyd-Warshall's algorithm.

Now, we must make one more KEY observation before we design our algorithm:

Our final multiplication will ALWAYS be of the form

$(A_0 \bullet A_1 \bullet \ldots A_k) \bullet (A_{k+1} \bullet A_{k+2} \bullet \ldots A_{n-1})$

In essence, there is exactly one value of k for which we should "split" our work into two separate cases so that we get an optimal result. Here is a list of the cases to choose from:

$(A_0) \bullet (A_1 \bullet A_{k+2} \bullet \ldots A_{n-1})$
$(A_0 \bullet A_1) \bullet (A_2 \bullet A_{k+2} \bullet \ldots A_{n-1})$
$(A_0 \bullet A_1 \bullet A_2) \bullet (A_3 \bullet A_{k+2} \bullet \ldots A_{n-1})$
...
$(A_0 \bullet A_1 \bullet \ldots A_{n-3}) \bullet (A_{n-2} \bullet A_{n-1})$
$(A_0 \bullet A_1 \bullet \ldots A_{n-2}) \bullet (A_{n-1})$

Basically, count the number of multiplications in each of these choices and pick the minimum. One other point to notice is that you have to account for the minimum number of multiplications in each of the two products.

Consider the case multiplying these 4 matrices:

A: 2x4
B: 4x2
C: 2x3
D: 3x1

1. (A)(BCD) - This is a 2x4 multiplied by a 4x1,
so 2x4x1 = 8 multiplications, plus whatever work it will take to multiply (BCD).

2. (AB)(CD) - This is a 2x2 multiplied by a 2x1,
so 2x2x1 = 4 multiplications, plus whatever work it will take to multiply (AB) and (CD).

3. (ABC)(D) - This is a 2x3 multiplied by a 3x1, so 2x3x1 = 6 multiplications, plus whatever work it will take to multiply (ABC).

Thus, we can state the following recursive formula:

$N_{i,j}$ = min value of $N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}$, over all valid values of k.

One way we can think about turning this recursive formula into a dynamic programming solution is by deciding which sub-problems are necessary to solve first. Clearly it's necessary to solve the smaller problems before the larger ones. In particular, we need to know $N_{i,i+1}$, the number of multiplications to multiply any adjacent pair of matrices before we move onto larger tasks. Similarly, the next task we want to solve is finding all the values of the form $N_{i,i+2}$, then $N_{i,i+3}$, etc.

Iterative Algorithm

```
1) Initialize N[i][i] = 0, and all other entries in N to ∞.
2) for i=1 to n-1 do the following
        2i) for j=0 to n-1-i do
                2ii) for k=j to j+i-1
                        2iii) if (N[j][j+i-1] >
                                N[j][k]+N[k+1][j+i-1]+$d_j d_{k+1} d_{i+j}$)

                            N[j][j+i-1]=
                                N[j][k]+N[k+1][j+i-1]+$d_j d_{k+1} d_{i+j}$
```

Here is the example we worked through in class:

Matrix	Dimensions
A	2x4
B	4x2
C	2x3
D	3x1
E	1x4

	A	B	C	D	E
A	0	16	28	22	30
B		0	24	14	30
C			0	6	14
D				0	12
E					0

First we determine the number of multiplications necessary for 2 matrices:

AxB uses 2x4x2 = 16 multiplications
BxC uses 4x2x3 = 24 multiplications
CxD uses 2x3x1 = 6 multiplications
DxE uses 3x1x4 = 12 multiplications

Now, let's determine the number of multiplications necessary for 3 matrices

(AxB)xC uses 16 + 0 + 2x2x3 = 28 multiplications
Ax(BxC) uses 0 + 24 + 2x4x3 = 48 multiplications, so 28 is min.

(BxC)xD uses 24 + 0 + 4x3x1 = 36 multiplications
Bx(CxD) uses 0 + 6 + 4x2x1 = 14 multiplications, is 14 is min.

(CxD)xE uses 6 + 0 + 2x1x4 = 14 multiplications
Cx(DxE) uses 0 + 12 + 2x3x4 = 36, so 14 is min.

Four matrices next:

Ax(BxCxD) uses 0 + 14 + 2x4x1 = 22 multiplications
(AxB)x(CxD) uses 16 + 6 + 2x2x1 = 26 multiplications
(AxBxC)xD uses 28 + 0 + 2x3x1 = 34 multiplications, 22 is min.

Bx(CxDxE) uses 0 + 14 + 4x2x4 = 46 multiplications
(BxC)x(DxE) uses 24 + 12 + 4x3x4 = 84 multiplications
(BxCxD)xE uses 14 + 0 + 4x1x4 = 30 multiplications, 30 is min.

For the answer:

Ax(BxCxDxE) uses 0 + 30 + 2x4x4 = 62 multiplications
(AxB)x(CxDxE) uses 16 + 14 + 2x2x4 = 46 multiplications
(AxBxC)x(DxE) uses 28 + 12 + 2x3x4 = 64 multiplications
(AxBxCxD)xE uses 22 + 0 + 2x1x4 = 30 multiplications

Answer = 30 multiplications

Recursive Algorithm with Memoization

When coding, the recursive algorithm memoized tends to be easier for most people. The recursive algorithm takes in two indexes, start and end, representing the consecutive matrices to multiply for that recursive case. The algorithm is simply to try each split point and take the best one. Assume that the method sketch shown below has access to all the dimensions of each matrix and that memo is the memoization array. In this code, let d[i][0] store the # of rows in matrix i and d[i][1] store the number of columns in matrix i:

```
int solveRec(int start, int end) {

        if (start == end) return 0;
        if (memo[start][end] != -1) return memo[start][end];

        int res = solveRec(start, end-1) + d[start][0]*d[end][0]*d[end][0];

        for (int i=start; i<end; i++) {
                int tmp = solveRec(start, i) + solveRec(i+1, end) +
                                d[start][0]*d[i][1]*d[end][1];
                res = Math.min(res, tmp);
        }

        return memo[start][end] = res;
}
```