Lecture 26: Chain Matrix Multiplication

(Thursday, April 30, 1998)

Read: Section 16.1 of CLR.

Chain Matrix Multiplication: This problem involves the question of determining the optimal sequence for performing a series of operations. This general class of problem is important in compiler design for code optimization and in databases for query optimization. We will study the problem in a very restricted instance, where the dynamic programming issues are easiest to see.

Suppose that we wish to multiply a series of matrices

$$A_1 A_2 \ldots A_n$$

Matrix multiplication is an associative but not a commutative operation. This means that we are free to parenthesize the above multiplication however we like, but we are not free to rearrange the order of the matrices. Also recall that when two (nonsquare) matrices are being multiplied, there are restrictions on the dimensions. A $p \times q$ matrix has p rows and q columns. You can multiply a $p \times q$ matrix A times a $q \times r$ matrix B, and the result will be a $p \times r$ matrix C. (The number of columns of A must equal the number of rows of B.) In particular for $1 \leq i \leq p$ and $1 \leq j \leq r$,

$$C[i,j] = \sum_{k=1}^{q} A[i,k]B[k,j].$$

Observe that there are pr total entries in C and each takes $O(q)$ time to compute, thus the total time (e.g. number of multiplications) to multiply these two matrices is $p \cdot q \cdot r$.

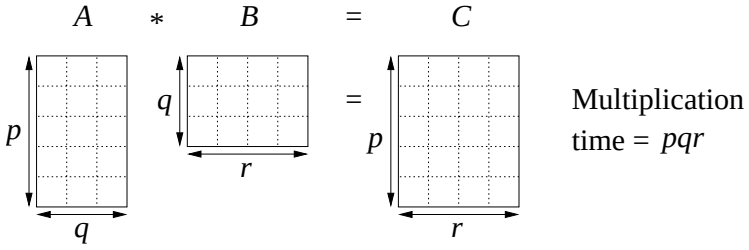

Figure 33: Matrix Multiplication.

Note that although any legal parenthesization will lead to a valid result, not all involve the same number of operations. Consider the case of 3 matrices: A_1 be 5×4, A_2 be 4×6 and A_3 be 6×2.

$$\begin{aligned} mult[((A_1 A_2)A_3)] &= (5 \cdot 4 \cdot 6) + (5 \cdot 6 \cdot 2) = 180, \\ mult[(A_1(A_2 A_3))] &= (4 \cdot 6 \cdot 2) + (5 \cdot 4 \cdot 2) = 88. \end{aligned}$$

Even for this small example, considerable savings can be achieved by reordering the evaluation sequence. The Chain Matrix Multiplication problem is: Given a sequence of matrices $A_1, A_2, \ldots, A_n$ and dimensions $p_0, p_1, \ldots, p_n$ where A_i is of dimension $p_{i-1} \times p_i$, determine the multiplication sequence that minimizes the number of operations.

Important Note: This algorithm does not perform the multiplications, it just figures out the best order in which to perform the multiplications.

Naive Algorithm: We could write a procedure which tries all possible parenthesizations. Unfortunately, the number of ways of parenthesizing an expression is very large. If you have just one item, then there is only one way to parenthesize. If you have n items, then there are $n-1$ places where you could break the list with the outermost pair of parentheses, namely just after the 1st item, just after the 2nd item, etc., and just after the $(n-1)$st item. When we split just after the kth item, we create two sublists to be parenthesized, one with k items, and the other with $n-k$ items. Then we could consider all the ways of parenthesizing these. Since these are independent choices, if there are L ways to parenthesize the left sublist and R ways to parenthesize the right sublist, then the total is $L \cdot R$. This suggests the following recurrence for $P(n)$, the number of different ways of parenthesizing n items:

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2. \end{cases}$$

This is related to a famous function in combinatorics called the *Catalan numbers* (which in turn is related to the number of different binary trees on n nodes). In particular $P(n) = C(n-1)$ and

$$C(n) = \frac{1}{n+1}\binom{2n}{n}.$$

Applying Stirling's formula, we find that $C(n) \in \Omega(4^n/n^{3/2})$. Since 4^n is exponential and $n^{3/2}$ is just polynomial, the exponential will dominate, and this grows very fast. Thus, this will not be practical except for very small n.

Dynamic Programming Solution: This problem, like other dynamic programming problems involves determining a structure (in this case, a parenthesization). We want to break the problem into subproblems, whose solutions can be combined to solve the global problem.

For convenience we can write $A_{i..j}$ to be the product of matrices i through j. It is easy to see that $A_{i..j}$ is a $p_{i-1} \times p_j$ matrix. In parenthesizing the expression, we can consider the highest level of parenthesization. At this level we are simply multiplying two matrices together. That is, for any k, $1 \leq k \leq n-1$,

$$A_{1..n} = A_{1..k}A_{k+1..n}.$$

Thus the problem of determining the optimal sequence of multiplications is broken up into 2 questions: how do we decide where to split the chain (what is k?) and how do we parenthesize the subchains $A_{1..k}$ and $A_{k+1..n}$? The subchain problems can be solved by recursively applying the same scheme. The former problem can be solved by just considering all possible values of k. Notice that this problem satisfies the principle of optimality, because if we want to find the optimal sequence for multiplying $A_{1..n}$ we must use the optimal sequences for $A_{1..k}$ and $A_{k+1..n}$. In other words, the subproblems must be solved optimally for the global problem to be solved optimally.

We will store the solutions to the subproblems in a table, and build the table in a bottom-up manner. For $1 \leq i \leq j \leq n$, let $m[i,j]$ denote the minimum number of multiplications needed to compute $A_{i..j}$. The optimum cost can be described by the following recursive definition. As a basis observe that if $i = j$ then the sequence contains only one matrix, and so the cost is 0. (There is nothing to multiply.) Thus, $m[i,i] = 0$. If $i < j$, then we are asking about the product $A_{i..j}$. This can be split by considering each k, $i \leq k < j$, as $A_{i..k}$ times $A_{k+1..j}$.

The optimum time to compute $A_{i..k}$ is $m[i,k]$, and the optimum time to compute $A_{k+1..j}$ is $m[k+1,j]$. We may assume that these values have been computed previously and stored in our array. Since $A_{i..k}$ is a $p_{i-1} \times p_k$ matrix, and $A_{k+1..j}$ is a $p_k \times p_j$ matrix, the time to multiply them is $p_{i-1} \cdot p_k \cdot p_j$. This suggests the following recursive rule for computing $m[i,j]$.

$$\begin{aligned} m[i,i] &= 0 \\ m[i,j] &= \min_{i \leq k < j}(m[i,k] + m[k+1,j] + p_{i-1}p_kp_j) \qquad \text{for } i < j. \end{aligned}$$

It is not hard to convert this rule into a procedure, which is given below. The only tricky part is arranging the order in which to compute the values. In the process of computing $m[i, j]$ we will need to access values $m[i, k]$ and $m[k+1, j]$ for k lying between i and j. This suggests that we should organize things our computation according to the number of matrices in the subchain. Let $L = j - i + 1$ denote the length of the subchain being multiplied. The subchains of length 1 ($m[i, i]$) are trivial. Then we build up by computing the subchains of lengths $2, 3, \ldots, n$. The final answer is $m[1, n]$. We need to be a little careful in setting up the loops. If a subchain of length L starts at position i, then $j = i + L - 1$. Since we want $j \leq n$, this means that $i + L - 1 \leq n$, or in other words, $i \leq n - L + 1$. So our loop for i runs from 1 to $n - L + 1$ (to keep j in bounds).

Chain Matrix Multiplication

```
Matrix-Chain(array p[1..n], int n) {
    array s[1..n-1,2..n]
    for i = 1 to n do m[i,i] = 0                    // initialize
    for L = 2 to n do {                             // L = length of subchain
        for i = 1 to n-L+1 do {
            j = i + L - 1
            m[i,j] = INFINITY
            for k = i to j-1 do {
                q = m[i, k] + m[k+1, j] + p[i-1]*p[k]*p[j]
                if (q < m[i, j]) { m[i,j] = q;  s[i,j] = k }
            }
        }
    }
    return m[1,n] and s
}
```

The array $s[i, j]$ will be explained later. It is used to extract the actual sequence. The running time of the procedure is $\Theta(n^3)$. We'll leave this as an exercise in solving sums, but the key is that there are three nested loops, and each can iterate at most n times.

Extracting the final Sequence: To extract the actual sequence is a fairly easy extension. The basic idea is to leave a *split marker* indicating what the best split is, that is, what value of k lead to the minimum value of $m[i, j]$. We can maintain a parallel array $s[i, j]$ in which we will store the value of k providing the optimal split. For example, suppose that $s[i, j] = k$. This tells us that the best way to multiply the subchain $A_{i..j}$ is to first multiply the subchain $A_{i..k}$ and then multiply the subchain $A_{k+1..j}$, and finally multiply these together. Intuitively, $s[i, j]$ tells us what multiplication to perform *last*. Note that we only need to store $s[i, j]$ when we have at least two matrices, that is, if $j > i$.

The actual multiplication algorithm uses the $s[i, j]$ value to determine how to split the current sequence. Assume that the matrices are stored in an array of matrices $A[1..n]$, and that $s[i, j]$ is global to this recursive procedure. The procedure returns a matrix.

Extracting Optimum Sequence

```
Mult(i, j) {
    if (i > j) {
        k = s[i,j]
        X = Mult(i, k)                  // X = A[i]...A[k]
        Y = Mult(k+1, j)                // Y = A[k+1]...A[j]
        return X*Y;                     // multiply matrices X and Y
    }
    else
        return A[i];
}
```

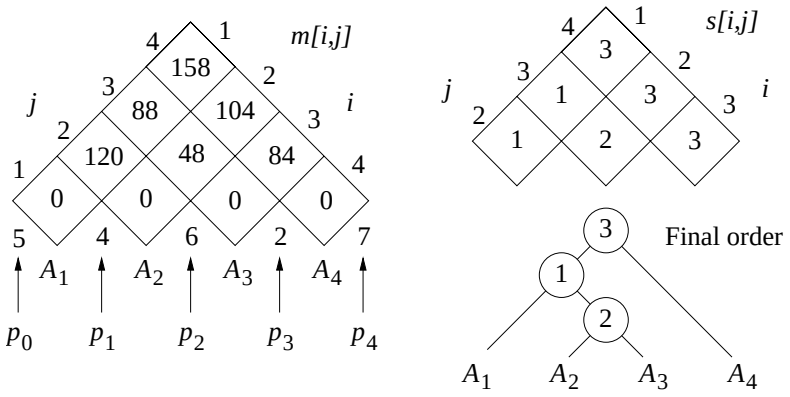

Figure 34: Chain Matrix Multiplication.

In the figure below we show an example. This algorithm is tricky, so it would be a good idea to trace through this example (and the one given in the text). The initial set of dimensions are $\langle 5, 4, 6, 2, 7\rangle$ meaning that we are multiplying A_1 (5×4) times A_2 (4×6) times A_3 (6×2) times A_4 (2×7). The optimal sequence is $((A_1(A_2A_3))A_4)$.

Lecture 27: NP-Completeness: General Introduction

(Tuesday, May 5, 1998)

Read: Chapt 36, up through section 36.4.

Easy and Hard Problems: At this point of the semester hopefully you have learned a few things of what it means for an algorithm to be efficient, and how to design algorithms and determine their efficiency asymptotically. All of this is fine if it helps you discover an acceptably efficient algorithm to solve your problem. The question that often arises in practice is that you have tried every trick in the book, and still your best algorithm is not fast enough. Although your algorithm can solve small problems reasonably efficiently (e.g. $n \leq 20$) the really large applications that you want to solve (e.g. $n \geq 100$) your algorithm does not terminate quickly enough. When you analyze its running time, you realize that it is running in *exponential time*, perhaps $n^{\sqrt{n}}$, or 2^n, or $2^{(2^n)}$, or $n!$, or worse.

Towards the end of the 60's and in the eary 70's there were great strides made in finding efficient solutions to many combinatorial problems. But at the same time there was also a growing list of problems for which there seemed to be no known efficient algorithmic solutions. The best solutions known for these problems required exponential time. People began to wonder whether there was some unknown paradigm that would lead to a solution to these problems, or perhaps some proof that these problems are inherently hard to solve and no algorithmic solutions exist that run under exponential time.

Around this time a remarkable discovery was made. It turns out that many of these "hard" problems were interrelated in the sense that if you could solve any one of them in polynomial time, then you could solve all of them in polynomial time. The next couple of lectures we will discuss some of these problems and introduce the notion of P, NP, and NP-completeness.

Polynomial Time: We need some way to separate the class of efficiently solvable problems from inefficiently solvable problems. We will do this by considering problems that can be solved in polynomial time.