CMSC 451: Lecture 10
Dynamic Programming: Chain Matrix Multiplication

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

$$C = A_1 \cdot A_2 \cdots A_n$$

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 (see Fig. 1). The number of columns of A must equal the number of rows of B. In particular for $1 \le i \le p$ and $1 \le j \le r$, we have

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

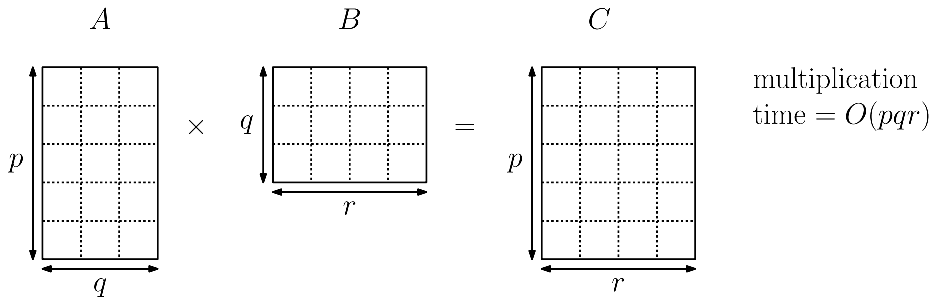

Fig. 1: Matrix Multiplication.

This corresponds to the (hopefully familiar) rule that $C[i,j]$ is the dot product of the ith (horizontal) row of A and the jth (vertical) column of B. Observe that there are pr total entries in C and each takes $O(q)$ time to compute, thus the total time to multiply these two matrices is proportional to the product of the dimensions, pqr.

Note that although any legal way of parenthesizing the matrices 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.

$$\begin{aligned} \text{cost}[((A_1A_2)A_3)] &= (5 \cdot 4 \cdot 6) + (5 \cdot 6 \cdot 2) = 180, \\ \text{cost}[(A_1(A_2A_3))] &= (4 \cdot 6 \cdot 2) + (5 \cdot 4 \cdot 2) = 88. \end{aligned}$$

Even for this small example, considerable savings can be achieved by reordering the evaluation sequence.

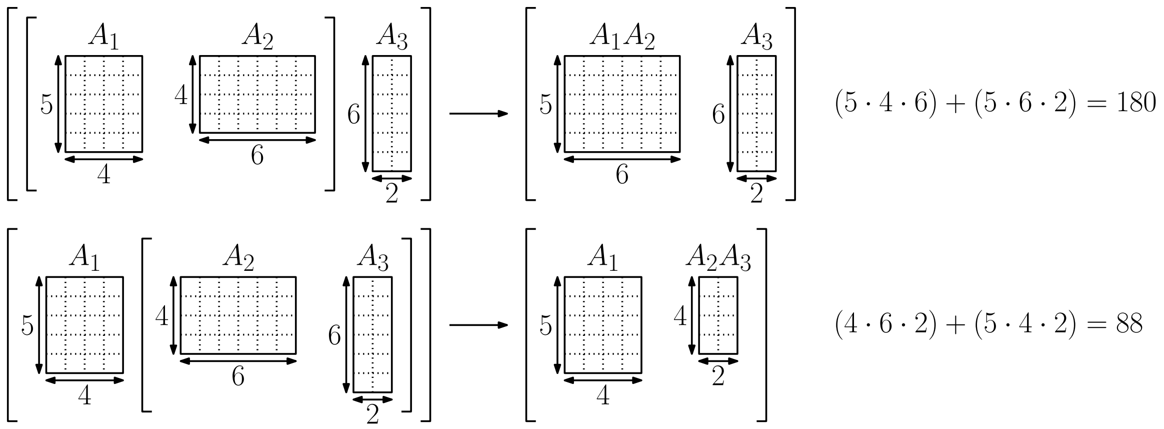
A_1
A_2
A_3
A_1A_2
A_2A_3
$(5 \cdot 4 \cdot 6) + (5 \cdot 6 \cdot 2) = 180$
$(4 \cdot 6 \cdot 2) + (5 \cdot 4 \cdot 2) = 88$

Fig. 2: Order of operations affects total operation count.

Chain Matrix Multiplication Problem: Given a sequence of matrices $A_1, \ldots, A_n$ and dimensions $p_0, \ldots, p_n$ where A_i is of dimension $p_{i-1} \times p_i$, determine the order of multiplication (represented, say, as a binary tree) that minimizes the number of operations.

Important Note: This algorithm *does not* perform the multiplications, it just determines the best order in which to perform the multiplications and the total number of operations. The output can be thought of as a binary tree whose leaves are the matrices. Indeed, there are many tree-related problems that can be solved using DP, and chain-matrix multiplication is a good archetype for these solutions.

Brute-Force Solution: 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 or two matrices, 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 $P(n) = C(n-1)$, where $C(n)$ is the nth Catalan number:

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

By applying the definition of $\binom{a}{b}$ and using Stirling's formula,[1] it can be shown that $C(n)$ is $\Omega(4^n/n^{3/2})$. Since 4^n is exponential and $n^{3/2}$ is just polynomial, the exponential will dominate, implying that function grows very fast. Thus, this will not be practical except for very small n. In summary, brute force is not a reasonable option.

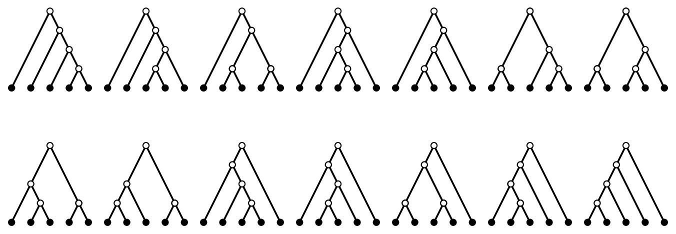
Fig. 3: Different evaluation orders for five matrices.

Dynamic-programming approach: A naive approach to this problem, namely that of trying all valid ways of parenthesizing the expression, will lead to an exponential running time. We will solve it through dynamic programming.

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. As is common to any DP solution, we need to find some way to break the problem into smaller subproblems, and we need to determine a recursive formulation, which represents the optimum solution to each problem in terms of solutions to the subproblems. Let us think of how we can do this.

Since matrices cannot be reordered, it makes sense to think about sequences of matrices. For $1 \leq i \leq j \leq n$, let $A(i,j)$ denote the result of multiplying matrices i through j, that is,

$$A(i,j) \;=\; A_i \cdot A_{i+1} \cdots A_j$$

Our subproblems will then be the minimum number of operations needed to compute $A(i,j)$, which we denote by $M(i,j)$.

Let's explore the properties of this subchain product. It is easy to see that $A(i,j)$ is a $p_{i-1} \times p_j$ matrix. (Think about this for a second to be sure you see why.) Now, in order to determine how to perform this multiplication optimally, we need to make many decisions. What we want to do is to break the problem into problems of a similar structure. In parenthesizing the expression, we can consider the highest level of parenthesization. At this level we are simply multiplying two matrices together. That is,

$$A(i,j) \;=\; (A_i \cdot \ldots \cdot A_k) \cdot (A_{k+1} \cdot \ldots \cdot A_j) \;=\; A(1,k) \cdot A(k+1,n), \qquad \text{for } i \leq k \leq j-1.$$

Thus the problem of determining the optimal sequence involves the following issues:

- What is the best place to split the chain? (what is k?)
- How much does it cost to compute each of $A(i,k)$ and $A(k+1,j)$?

[1]Stirling's formula provides an algebraic approximation to the factorial function. It states that, in the limit for large n, $n! \approx \sqrt{2\pi n}(n/e)^n$. From the perspective of asymptotics, this implies that $n!$ grows at least as fast as $\Omega(n^n)$.

- How much does it cost for the final product $A(i, k \cdot A(k+1, j)$?

For the first issue, will apply the same ("don't be smart") idea as in other DP problems. We'll try all possible splits, compute their costs, and take the best. For the second issue, the principle of optimality (a requirement of DP solutions) applies here. In order to achieve the globally optimal solution, the subproblems of computing $A(i, k)$ and $A(k+1, j)$ should each be solved optimally. (There is no advantage to be gained by solving one subproblem suboptimally, in order to help do better on the other.)

For the final issue, recall that when you multiply a chain of matrices, the size of the result is the number of rows in the first and the number of columns in the last. Thus, $A(i, k)$ is a $p_{i-1} \times p_k$ matrix, and $A(k+1, j)$ is a $p_k \times p_j$ matrix. It follows that the time to multiply them is $p_{i-1}p_kp_j$. We now have everything we need to give the recursive formulation.

Recursive formulation: For $1 \leq i \leq j \leq n$, recall that $M(i, j)$ denotes the minimum cost (number of operations) needed to compute the product $A(i, j) = A_iA_{i+1} \ldots A_j$. The desired total cost of multiplying all the matrices is that of computing the entire chain $A(1, n)$, which is given by $M(1, n)$. The optimum cost can be described by the following recursive formulation.

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, $M(i, i) = 0$.

General case: If $i < j$, then we are asking about the product $A(i, j)$. This can be split into two groups $A(i, k)$ times $A(k+1, j)$, by considering each k, $i \leq k < j$ (see Fig. 4) and taking the best.

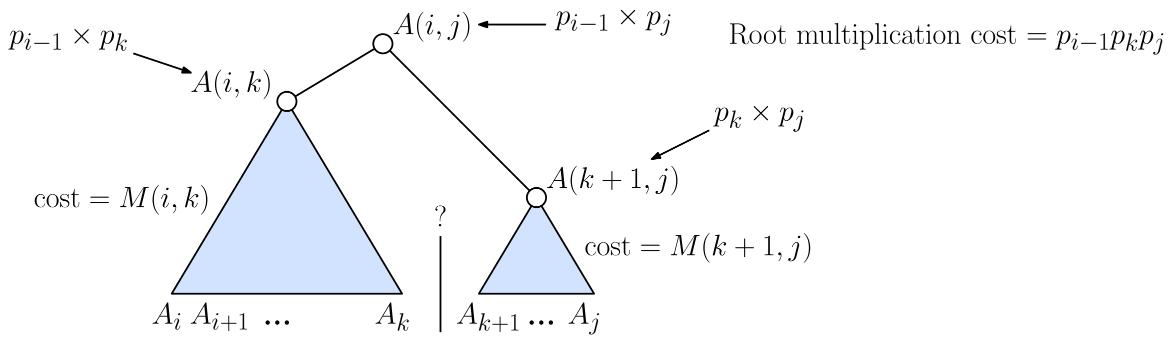

Fig. 4: Dynamic programming recursive formulation.

As observed above, the cost of computing $A(i, k)$ is given recursively by $M(i, k)$, and the cost of computing $A(k+1, j)$ is given by $M(k+1, j)$. The time needed for the final product is $p_{i-1}p_kp_j$. Thus, we have the following recursive rule:

$$M(i, j) \;=\; \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \leq k \leq j-1} \big(M(i, k) + M(k+1, j) + p_{i-1}p_kp_j\big) & \text{if } i < j. \end{cases}$$

Memoized Implementation: As with other DP problems, there are two natural implementations of the recursive rule that will lead to an efficient algorithm. One is memoization and bottom-up. Let's first present the memoized version. Recall that this involves designing a recursive function which saves results once computed. To do this, we create a table $M[1..n, 1..n]$, where $M[i, j]$ will store the function value $M(i, j)$. Initially, all entries are set to -1, which indicates that they are undefined. Eventually, we are interested in $M[1, n]$ as the final count of the number of operations to multiply all the matrices. In addition, we store a parallel table of "hooks" to allow us to reconstruct the optimal sequence. For each entry $M[i, j]$, it stores the splitting index k that led to the minimum cost. Later we'll see how it is used.

Memoized Chain Matrix Multiplication

```
memo-cmm(i, j) {                                    // memoized chain matrix mult
    if (M[i, j] == -1) {                            // undefined?
        minCost = INFINITY
        for (k = i to j - 1) {                      // get costs for all splits
            cost = memo-cmm(i, k) + memo-cmm(k+1, j) + p[i-1]*p[k]*p[j]
            if (cost < minCost) {                   // found a new optimum?
                minCost = cost                      // ...save it
                H[i, j] = k                         // ...save the split index
            }
        }
        M[i, j] = minCost                           // save optimum cost
    }
    return M[i, j]                                  // return table entry
}
```

An example is shown in Fig. 5. We have turned the matrix on its side to better illustrate its relationship to the binary tree.

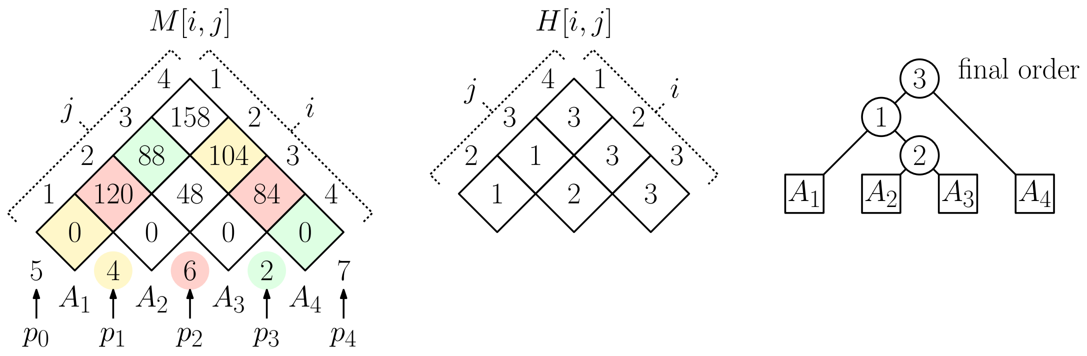

Fig. 5: Chain matrix multiplication for the $A_1 \cdot A_2 \cdots A_4$, where $\langle p_0, \ldots, p_4 \rangle = \langle 5, 4, 6, 2, 7 \rangle$.

For example, when computing $M[1, 4]$ in the above example, we take the minimum of the following three options:

$$
\begin{aligned}
(k = 1)\ M[1,1] + M[2,4] + p_0 \cdot p_1 \cdot p_4 &= 0 + 104 + 140 = 244 \\
(k = 2)\ M[1,2] + M[3,4] + p_0 \cdot p_2 \cdot p_4 &= 120 + 84 + 210 = 414 \\
(k = 3)\ M[1,3] + M[4,4] + p_0 \cdot p_3 \cdot p_4 &= 88 + 0 + 70 = 158.
\end{aligned}
$$

Clearly, the best choice is 158, so $M[1, 4] = 158$ and (since this happened when $k = 3$) $H[1, 4] = 3$.

The running time of the procedure is $O(n^3)$. There are $O(n^2)$ table entries to be filled. For each table entry, we need to iterate through the $j - i$ possible splitting points. Since $1 \leq i \leq j \leq n$, $j - i \leq n - 1$, and so it takes $O(n)$ time in the worst case to compute each table entry. (A more careful analysis shows that the total number of operations grows roughly as $n^3/6$.)

Extracting the final Sequence: Extracting the actual multiplication sequence is a fairly easy extension. Recall that when we compute $M[i, j]$, we store the optimal split index k in $H[i, j]$. 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, $H[i, j]$ tells us what multiplication to perform *last*.

The multiplication algorithm is presented in the code block below. The initial call is do-mult$(1, n)$. An example is given in Fig. 5. (It's a good idea to trace through this example to be sure you understand it.)

Extracting Optimum Sequence

```
do-mult(i, j) {                                  // multiply the matrices
    if (i == j)                                  // basis case
        return A[i]
    else {
        k = H[i,j]
        X = do-mult(i, k)                        // X = A[i] * ... * A[k]
        Y = do-mult(k+1, j)                      // Y = A[k+1] * ... * A[j]
        return X * Y                             // multiply matrices X and Y
    }
}
```

Bottom-up implementation: The bottom-up process fills the array $M[1..n, 1..n]$ by a purely iterative process. This is a bit tricky, however!

You might think that we can just fill the table row-by-row, and column-by-column (as we did with the longest common subsequence problem). However, this simple approach will not work here. To see why, suppose that we are computing the values in row 3. When computing $M[3, 5]$, we would need to access both $M[3, 4]$ and $M[4, 5]$. But $M[4, 5]$ is in row 4, which has not yet been computed!

The trick is to compute the matrix *diagonal-by-diagonal*, working out from the middle of the array. In particular, we organize our computation according to the number of matrices in the subsequence. For example, $M[3, 5]$ represents a chain of $5 - 3 + 1 = 3$ matrices, whereas $M[3, 4]$ and $M[4, 5]$ each represent chains of only two matrices. We first solve the problem for chains of length 1 (which is trivial), then chains of length 2, and so on, until we come to $M[1, n]$, which is the total chain of length n.

To implement this, for $1 \leq i \leq j \leq n$, let $\ell = j - i + 1$ denote the length of the subchain being multiplied. How shall we set up the loops to do this? The case $\ell = 1$ (that is, entries of the form $M[i, i]$) is trivial, since there is only one matrix, and nothing needs to be multiplied, so we have $M[i, i] = 0$. Otherwise, our outer loop runs from $\ell = 2, \ldots, n$. If a subchain of length ℓ starts at position i, then $j = i + \ell - 1$. How high should i go (so we don't index out of bounds)? Since we want $j \leq n$, we have

$$i + \ell - 1 \ \leq\ n, \qquad \text{or equivalently} \qquad i \ \leq\ n - \ell + 1.$$

So our inner loop will be based on i running from 1 up to $n-\ell+1$. The procedure is presented in the code block below. (Also, see Fig. 5 for an example.) We will skip the H array, but it can easily be added here.

Bottom-Up Chain Matrix Multiplication

```
bottom-up-cmm() {                                   // bottom-up chain matrix mult
    for (i = 1 to n) M[i, i] = 0                    // initialize
    for (L = 2 to n) {                              // L = length of subchain
        for (i = 1 to n - L + 1) {                  // generate all chains of length L
            j = i + L - 1
            minCost = INFINITY
            for (k = i to j - 1)                    // check all splits
                minCost = min(minCost, M[i, k] + M[k+1, j] + p[i-1]*p[k]*p[j])
            M[i, j] = minCost                       // save optimum cost
        }
    }
    return M[1, n]                                  // return the total cost
}
```

Summary: We have presented an $O(n^3)$ time for the problem of determining the best way to multiply n matrices (of various dimensionalities) together. Who cares about this problem? Frankly, I know of no compelling applications. However, there are numerous applications of DP which involve a tree-like partition of the solutions space, and this is an easy example of the general structure.

An interesting application is that of computing the optimum binary search tree for a set keys, where the keys have distinct probabilities of being accessed. This problem is discussed in this Wikipedia article. (The algorithm that we presented corresponds to the naive implementaiton of Knuth's algorithm, which is mentioned in the article.)