Dynamic Programming

(see Section 15 on Dynamic Programming in Cormen, Leiserson, Rivest, Stein)

We discuss dynamic programming, which is an algorithm design paradigm applicable to a reasonably wide range of computational problems. It is similar to divide-and-conquer: it solves a problem by decomposing it into subproblems, it solves the subproblems recursively, and it derives the overall solution from the solutions to the subproblems. A prime illustration of the divide-and-conquer paradigm is quicksort. A major difference between the two paradigms is that in dynamic programming we store the solution to a subproblem in memory, such that if the same problem comes up again we just need to look up the solution rather than recompute it. Dynamic programming is thus appropriate for problems that are naturally decomposed into overlapping subproblems. Further decomposition leads to the same problem more than once, which would require redundant computations if we did not remember the already computed solution. One such problem is finding an optimal parenthesization for a product of three or more matrices.

Matrix multiplication. The product of two matrices, \( X \cdot Y \), is defined as long as the length of a row in \( X \) is the same as the length of a column in \( Y \). If \( X \) has \( p \) rows and \( q \) columns and \( Y \) has \( r \) rows and \( s \) columns, then \( X \cdot Y \) has \( p \) rows and \( r \) columns. Following the definition of matrix multiplication, we get \( X \cdot Y \) with \( p \cdot q \cdot r \) (elementary) multiplications and \( p \cdot (q-1) \cdot r \) additions. We consider the problem of multiplying a chain of matrices. Specifically, let \( X_i \) be matrices of sizes \( p_{i-1} \) times \( p_i \), for \( 1 \leq i \leq n \). Matrix multiplication is associative, which for a chain of length three means

\[
(X_1 \cdot X_2) \cdot X_3 = X_1 \cdot (X_2 \cdot X_3).
\]

Although the resulting product is the same, the number of elementary operations needed to compute it might be vastly different. As an example, let \( p_0 = 3, p_1 = 2, p_2 = 4, p_3 = 1 \) and count the elementary multiplications needed for the two possible parenthesizations, which are both illustrated in Figure 41.

\[
\begin{align*}
3 \cdot 4 \cdot 4 &= 24 \\
3 \cdot 4 \cdot 1 &= 12 \\
2 \cdot 1 &= 8 \\
3 \cdot 2 \cdot 1 &= 6
\end{align*}
\]

Figure 41: The first parenthesization takes 24 + 12 = 36 and the second takes 8 + 6 = 14 elementary multiplications.

Parenthesizations. To completely parenthesize a chain of \( n \) matrices we need \( n-1 \) pairs of parentheses. Each such parenthesization corresponds to a binary tree with \( n \) leaves and to a well-formed binary string. To map a parenthesization to a binary tree we create an internal node for each pair of matching parentheses. The leaves below this internal node are exactly the matrices enclosed by the pair of parentheses. To map a binary tree to a well-formed string of \( a \)s and \( b \)s we map each internal node to a pair \( a \)b, we write the string of the left subtree between \( a \) and \( b \), and we write the string of the right subtree after \( b \). Observe that this is slightly different from simply rewriting ‘(’ as ‘a’ and ‘)’ as ‘b’. Both constructions are illustrated in Figure 42. The string has equally many \( a \)s and \( b \)s, and each prefix has at least as many \( a \)s as \( b \)s. This is exactly what we mean by a well-formed string. Note that both correspondences are one-to-one, which implies that there are equally many parenthesizations of \( n \) matrices as there are binary trees of \( n \) leaves as there are well-formed strings of \( n-1 \) \( a \)s and \( n-1 \) \( b \)s. We count the number of possible parenthesizations by counting well-formed strings.
Matrix Chain Lemma. There are \( \frac{1}{n} \cdot (2^{n-1}) \) different parenthesizations of \( n \) matrices.

Proof. Set \( m = n - 1 \). We consider the total number of strings of \( m \) as and \( m \) bs, which is \( \binom{2m}{m} \) and includes also strings that are not well-formed. If a string is not well-formed then we exchange as and bs in the smallest prefix that has more bs than as. For example,

\[
[aaabbabb]ababa \rightarrow [bbababaa]ababa,
\]

where the brackets enclose the affected prefix. The new string has \( m + 1 \) as and \( m - 1 \) bs. Most importantly, the operation is one-to-one and exhausts all strings of \( m + 1 \) as and \( m - 1 \) bs. Indeed, every such string has a prefix with more as than bs and we can reverse the operation by taking the smallest such prefix and exchanging as and bs, as before. Thus the number of well-formed strings is

\[
\binom{2m}{m} - \binom{2m}{m-1} = \binom{2m}{m} \left( 1 - \frac{m}{m + 1} \right) = \frac{1}{m + 1} \binom{2m}{m},
\]

which is referred to as the \( m \)-th Catalan number. \( \square \)

To get a rough feeling for the size of a Catalan number, recall \( m! \approx e^{m \ln m - m} \). Hence

\[
\binom{2m}{m} \approx e^{2m \ln 2 - 2(\ln m - m)} = e^{2m \ln 2},
\]

which is \( 4^m \).

Divide-and-conquer algorithm. Define \( m_{ij} \) as the number of multiplications for the best parenthesization of \( X_i \cdot X_{i+1} \cdot \ldots \cdot X_j \). The algorithm assumes a global array \( p[0..n] \) that encodes the sizes of the matrices such that \( X_i \) is a matrix with \( p[i-1] \) rows and \( p[i] \) columns. The input parameters are the indices \( i \) of the first and \( j \) of the last matrix in the chain. We get the minimum number of elementary multiplications needed if we optimally multiply \( X_i \) through \( X_k \), then optimally multiply \( X_{k+1} \) through \( X_j \), and finally multiply the two resulting matrices. We get the optimum by trying all possible values of \( k \).

```java
int M(int i, j)
if i = j then return 0 endif;
min = \( \infty \);
for k = i to j - 1 do
  mult = M(i, k) + M(k + 1, j) + p[i - 1] * p[k] * p[j];
  if mult < min then min = mult endif;
endfor;
return min.
```

To get a handle on the running time, we count the times Function \( M \) is called from parameters \( i \) and \( j \). As illustrated in Figure 43, this is the number of ways to jump from \([1, n]\) to \([i, j]\) using only horizontal hops to the left and vertical hops downwards. Expressing the total number of calls to \( M \) with a recurrence relation, we get

\[
T(n) = 1 + \sum_{k=1}^{n-1} [T(k) + T(n - k)]
\]

\[
= 1 + 2 \sum_{k=1}^{n-1} T(k).
\]

From the entries in the table of Figure 43 we get \( T(1) = 1 \) (the top right corner), \( T(2) = 3 \) (the top right corner plus its two neighbors), \( T(3) = 9 \) (adding the next diagonal), etc., so we may guess that \( T(n) = 3^{n-1} \). Using induction, we can prove that this is indeed the solution to the recurrence relation. In any case, the performance of the divide-and-conquer algorithm is miserable and the reason is that...
it solves the same subproblem many times. It would do better if it remembered the results of the subproblems and looked them up rather than repeating the computations.

**Dynamic programming algorithm.** The dynamic programming algorithm for parenthesizing a chain of matrices uses arrays \( M[1..n, 1..n] \) and \( S[1..n, 1..n] \) to remember optimal solutions to subproblems. \( M[i, j] \) stores the minimum number of elementary multiplications needed for the chain \( X_i \cdot X_{i+1} \cdots X_j \). \( S[i, j] \) stores the index \( k \) of the best partition of the chain into \( X_i \cdots X_k \) and \( X_{k+1} \cdots X_j \). The algorithm works bottom-up, computing the optimal solutions of progressively larger subproblems. The variable \( \ell \) controls the length of the chains considered.

```c
void MATRIXCHAIN(int n)
    for i = 1 to n do M[i, i] = 0 endfor;
    for \ell = 2 to n do
        for i = 1 to n - \ell + 1 do
            j = i + \ell - 1; M[i, j] = \infty;
            for k = i to j - 1 do
                mult = M[i, k] + M[k + 1, j]
                     + p[i - 1] * p[k] * p[j];
                if mult < M[i, j] then
                    M[i, j] = mult; S[i, j] = k
                endif
            endfor
        endfor
    endfor

The main part of the algorithm consists of three nested for-loops, which implies that the running time is \( O(n^3) \). The amount of memory used is \( O(n^2) \). Figure 44 applies the algorithm to a product of four matrices of sizes 4-by-2, 2-by-5, 5-by-1, and 1-by-3. We could save half the amount of memory by embedding the used portion of \( S \) in the unused portion of \( M \). Specifically, we could store \( S[i, j] \) in \( M[j, i] \) for all \( i < j \).

**Recovering the parenthesization.** The number of elementary multiplications needed to evaluate the chain of matrices is given in \( M[1, n] \), and the parenthesization that leads to this number can be recovered from \( S \). We write a function that computes the corresponding binary tree whose leaves are the \( n \) matrices. The input parameters are the indices \( i \) and \( j \) of the first and the last matrix in the currently considered subproblem. Initially, \( i = 1 \) and \( j = n \).

```c
Tree PTREE(int i, j)
    assert i < j;
    \nu = Node(0, NULL, NULL);
    if i = j then \nu \to \nu.val = i
    else \nu \to \ell = PTREE(i, S[i, j]);
         \nu \to r = PTREE(S[i, j] + 1, j)
    endif;
    return \nu.
```

Figure 45 shows the optimal parenthesization and the corresponding binary tree for the above example.

Figure 45: Optimal tree and parenthesization for matrix chain problem in Figure 44.

```
```

Figure 44: A product of four matrices with sized encoded in \( p \). The dynamic programming algorithm stores intermediate results in the arrays \( M \) and \( S \). The optimal parenthesization is \((X_1 \cdot (X_2 \cdot X_3)) \cdot X_4 \) and takes 30 elementary multiplications.