Matrix–Chain Multiplication

Problem Statement

- Given a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices to be multiplied, and we wish to compute the product
- $A_1 A_2 \ldots A_n$
- Parenthesize it to resolve all ambiguities in how the matrices are multiplied together
- Matrix multiplication is associative, and so all parenthesizations yield the same product

Problem Statement

- 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
- For example, if the chain of matrices is $\langle A_1, A_2, A_3, A_4 \rangle$, then we can fully parenthesize the product $A_1A_2A_3A_4$ in five distinct ways:

$$
\begin{aligned}
&(A_1(A_2(A_3A_4))) ,\\
&(A_1((A_2A_3)A_4)) ,\\
&((A_1A_2)(A_3A_4)) ,\\
&((A_1(A_2A_3))A_4) ,\\
&(((A_1A_2)A_3)A_4) .
\end{aligned}
$$

Problem Statement

```
MATRIX-MULTIPLY(A, B)
1  if A.columns ≠ B.rows
2       error "incompatible dimensions"
3  else let C be a new A.rows × B.columns matrix
4       for i = 1 to A.rows
5            for j = 1 to B.columns
6                 c_ij = 0
7                 for k = 1 to A.columns
8                      c_ij = c_ij + a_ik · b_kj
9       return C
```

Problem Statement

- $A_1A_2...A_3$
- 10 X 100, 100 X 5, and 5 X 50
- $((A_1A_2)A_3)$
- $(A_1(A_2A_3))$
- Computing the product according to the first parenthesization is 10 times faster

Problem Statement

- Given a chain $<A_1, A_2, \ldots, A_n>$ of n matrices, where for i=1,2,...,n matrix Ai has dimension pi−1 x pi , fully parenthesize the product $A_1 A_2 \ldots A_n$ in a way that minimizes the number of scalar multiplications
- We are not actually multiplying matrices
- determine an order for multiplying matrices that has the lowest cost

Problem Statement

- time invested in determining this optimal order is more than paid for by the time saved later on when actually performing the matrix multiplications
- Such as performing only 7500 scalar multiplications instead of 75,000

Counting the number of parenthesizations

- When $n = 1$, only one way to fully parenthesize the matrix product
- When $n \geq 2$, a fully parenthesized matrix product is product of two fully parenthesized matrix subproducts, and the split between the two subproducts may occur between the k^{th} and $(k + 1)^{st}$ matrices for any $k = 1,2,\ldots,n-1$

Counting the number of parenthesizations

- Thus we obtain the recurrence

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

- recurrence is similar to sequence of Catalan numbers which grows as $\Omega(4^n / n^{3/2})$
- number of solutions is thus exponential in n
- brute–force method of exhaustive search makes for a poor strategy

Applying dynamic programming

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.
4. Construct an optimal solution from computed information.

Characterize structure of an optimal solution

- To parenthesize the product $A_iA_{i+1}...A_j$, we split the product between A_k and A_{k+1} for some integer k in the range $i \leq k < j$
- Cost of parenthesizing this way =

  cost of computing the matrix $A_{i..k}$ +

  cost of computing $A_{k+1..j}$ +

  cost of multiplying them together

Characterize structure of an optimal solution

- then optimally parenthsize $A_iA_{i+1}...A_k$, and also $A_{k+1}A_{k+2}...A_j$
- ensure that when we search for the correct place to split the product, we have considered all possible places, so that we are sure of having examined the optimal one

Step 2: A recursive solution

- m[i, j] – minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$
- If i = j , the problem is trivial; the chain consists of just one matrix $A_{i..i} = A_i$ no scalar multiplications are necessary to compute the product
- m[i, i] = 0 for i = 1, 2,...,n

Step 2: A recursive solution

- Compute m[i, j] when $i < j$
- We take advantage of structure of an optimal solution from step 1
- Let us assume that to optimally parenthesize, we split product $A_iA_{i+1}...A_j$ between A_k and A_{k+1}, where $i \leq k < j$
- $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1}p_kp_j$
- We do not know value of k so recursive defition becomes

Step 2: A recursive solution

$$m[i,j] = \begin{cases} 0 & \text{if } i = j\ , \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j\ . \end{cases}$$

- But complete information required is not avaiable in m
- we define s[i,j] to be a value of k at which we split the product $A_iA_{i+1}...A_j$ in an optimal parenthesization

Step 3: Computing the optimal costs

- We could easily write a recursive algorithm based on recurrence to compute the minimum cost m[1,n]
- this recursive algorithm takes exponential time, which is no better than the brute–force method of checking each way of parenthesizing the product

Step 3: Computing the optimal costs

- We have relatively few distinct subproblems:
- One subproblem for each choice of i and j satisfying $1 \leqslant i \leqslant j \leqslant n$,

$$\binom{n}{2} + n = \Theta(n^2)$$

- A recursive algorithm may encounter each subproblem many times in different branches of its recursion tree
- Property of overlapping subproblems is the second hallmark of when dynamic programming applies (the first hallmark being optimal substructure)

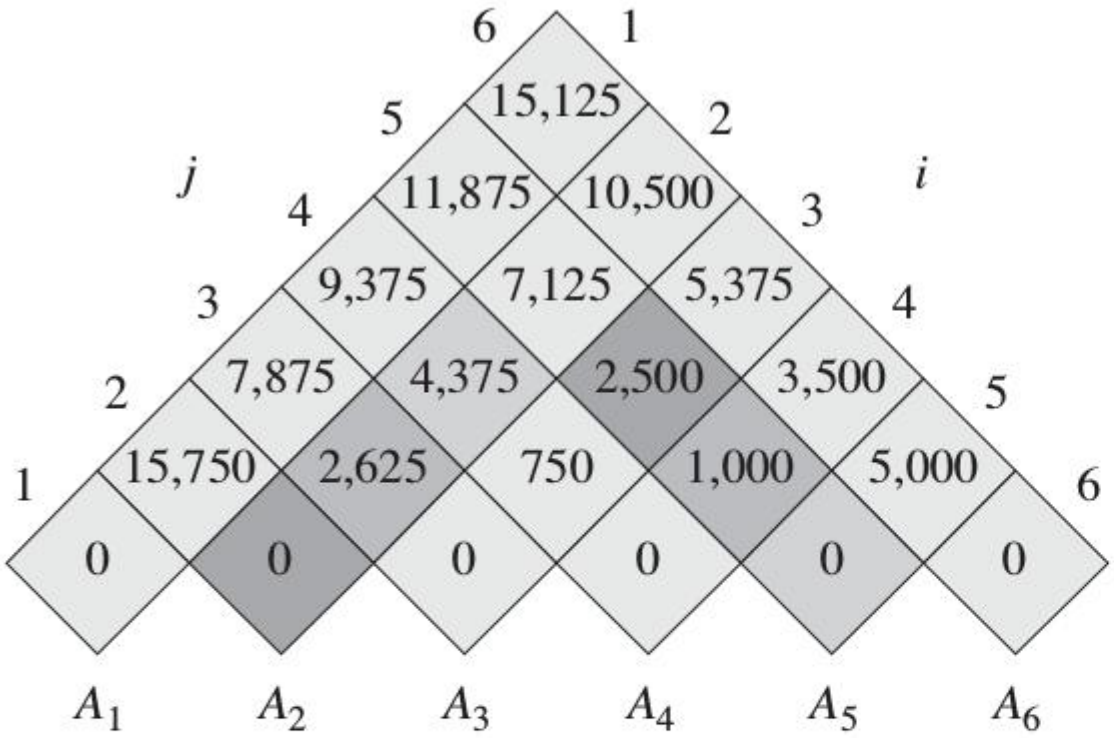
Bottom–up DP

- A_i has dimensions p_{i-1} x p_i for i = 1, 2,...,n sequence p = <p_0, p_1,...,p_n>, where p.length = n + 1 auxiliary table m[1..n; 1..n] for storing the m[i, j] costs and another auxiliary table s[1..n–1, 2..n] that records which index of k achieved the optimal cost in computing m[i,j]
- We shall use the table s to construct an optimal solution

Bottom–up DP

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                    // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

Bottom–up DP



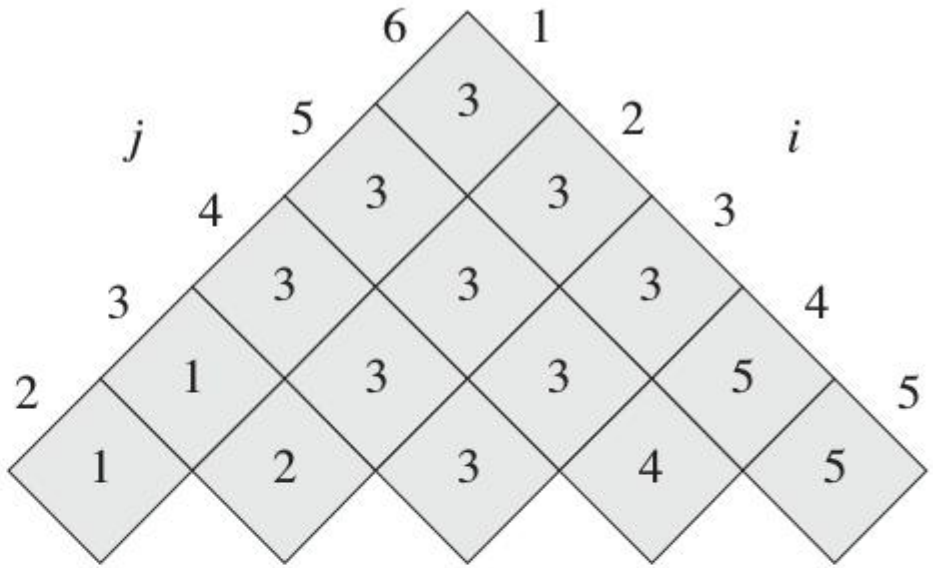


Figure 15.5 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

Bottom–up DP

$$
\begin{aligned}
m[2,5] &= \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 &= 0 + 2500 + 35 \cdot 15 \cdot 20 &= 13{,}000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 &= 2625 + 1000 + 35 \cdot 5 \cdot 20 &= 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 &= 4375 + 0 + 35 \cdot 10 \cdot 20 &= 11{,}375 \end{cases} \\
&= 7125\,.
\end{aligned}
$$

Bottom–up DP

- A simple inspection of the nested loop structure of MATRIX–CHAIN–ORDER yields a running time of $O(n^3)$ for the algorithm.
- The loops are nested three deep, and each loop index (l, i, and k) takes on at most n–1 values
- requires $\theta(n^2)$ space to store the m and s tables
- more efficient than the exponential–time method of enumerating all possible parenthesizations and checking each one

Step 4: Constructing an optimal solution

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2      print "A"_i
3  else print "("
4      PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5      PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6      print ")"
```

- the call PRINT–OPTIMAL–PARENS (s, 1, 6) prints the parenthesization $((A_1(A_2A_3))((A_4\ A_5)\ A_6))$