

NATIONAL POLYTECHNIC INSTITUTE
SUPERIOR SCHOOL OF COMPUTER SCIENCES

ALGORITHM ANALYSIS.

Practice 7: Matrix - Chain Multiplication.

Hernandez Martinez Carlos David.
Burciaga Ornelas Rodrigo Andres.

davestring@outlook.com.
andii_burciaga@live.com.

Group: 2cv3.

November 2, 2017

Contents

1	Introduction	2
2	Basic Concepts:	3
2.1	Counting Number of Parenthesizations:	3
2.2	Dynamic Programming:	3
3	Development:	4
3.1	Step 1: The Structure of an Optimal Parenthesization.	4
3.2	Step 2: A Recursive Solution.	5
3.3	Step 3: Computing the Optimal Cost.	6
3.4	Step 4: Constructing an Optimal Solution:	8
4	Results:	9
4.1	Matrix-Chain of Size $n = 6$:	9
4.2	Matrix-Chain of Size $n = 3$:	11
4.3	Matrix-Chain of Size $n = 4$:	13
5	Annexes:	15
6	Conclusion:	16
7	Bibliography References:	17

1 Introduction

Our next example of dynamic programming is an algorithm that solves the problem of matrix-chain multiplication. We are given a sequence (chain) $\langle A_1, A_2, \dots, A_n \rangle$ of n matrices to be multiplied, and we wish to compute the product:

$$A_1 \cdot A_2 \cdot \dots \cdot A_n \tag{1}$$

We can evaluate the expression (1) using the standard algorithm for multiplying pairs of matrices as a subroutine once we have parenthesized it to resolve all ambiguities in how the matrices are multiplied together. Matrix multiplication is associative, and so all parenthesizations yield the same product. 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 $A_1 \cdot A_2 \cdot A_3 \cdot A_4$ in five distinct ways:

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

How we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product. Consider first the cost of multiplying two matrices. As we have known the usual-matrix-product it's $O(n^3)$ while the Strassen's algorithm it's $O(n^{2.81})$. We can multiply two matrices **A** and **B** only if they are compatible: the number of columns of **A** must equal the number of rows of **B**. If **A** is a $p \times q$ matrix and **B** is $q \times r$ matrix, and resulting matrix **C** is a $p \times r$ matrix. The time to compute **C** is dominated by the number of scalar multiplications, which is $p \times q \times r$. In what follows, we shall express costs in terms of the number of scalar multiplications.

We state the **matrix-chain multiplication** problem as follows: given a chain $\langle A_1, A_2, \dots, A_n \rangle$ of n matrices, where for $i = 1, 2, \dots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 \cdot A_2 \cdot \dots \cdot A_n$ in a way that minimizes the number of scalar multiplications. Note that in the **matrix-chain multiplication** problem, we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices that has the lowest cost. Typically, the time invested in determining this optimal order is more than paid for by the time saved later on when actually performing the matrix multiplication.

2 Basic Concepts:

2.1 Counting Number of Parenthesizations:

Before solving the matrix-chain multiplication problem by dynamic programming, let us convince ourselves that exhaustively checking all possible parenthesizations does not yield an efficient algorithm. Denote the number of alternative parenthesizations of a sequence of \mathbf{n} matrices by $\mathbf{P}(\mathbf{n})$. When $n = 1$, we have just one matrix and therefore only one way to fully parenthesize the matrix product. When $n \geq 2$, a fully parenthesized matrix product is the product of two fully parenthesized matrix sub-products, and the split between the two sub-products may occur between the k -th and $(k + 1)$ st matrices for any $k = 1, 2, \dots, n - 1$. Thus, we obtain the recurrence:

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

2.2 Dynamic Programming:

We shall use the dynamic-programming method to determine how to optimally parenthesize a matrix chain. In so doing, we shall follow the four-step sequence:

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution.
- Construct an optimal solution from computed information.

3 Development:

In this section we will implement the **Matrix-Chain Multiplication** algorithm. Also we will explain each method of the **Python** module *matrix_chain_order.py*.

3.1 Step 1: The Structure of an Optimal Parenthesization.

For our first step in the dynamic-programming paradigm, we find the optimal sub-structure and then use it to construct an optimal solution to the problem from optimal solutions to sub-problems. In the matrix-chain multiplication problem, we can perform this step as follows. For convenience, let us adopt the $A_{i\dots j}$, where $i \leq j$, for the matrix that results from evaluating the product $A_i A_{i+1} \dots A_j$. Observe that if the problem is nontrivial, i.e., $i < j$, then to parenthesize the product $A_i A_{i+1} \dots A_j$, we must split the product between A_k and A_{k+1} for some k in the range $i \leq k < j$. That is, for some value of k , we first compute the matrices $A_{i\dots k}$ and $A_{k+1\dots j}$ and then multiply them together to produce the final product $A_{i\dots j}$. The cost of parenthesizing this way is the cost of computing the matrix $A_{i\dots k}$, plus the cost of computing $A_{k+1\dots j}$, plus the cost of multiplying them together.

The optimal substructure of this problem is as follows. Suppose that to optimally parenthesize $A_i A_{i+1} \dots A_j$, we split the product between A_k and A_{k+1} . Then the way we parenthesize the prefix sub-chain $A_i A_{i+1} \dots A_k$ within this optimal parenthesization of $A_i A_{i+1} \dots A_j$ must be an optimal parenthesization of $A_i A_{i+1} \dots A_k$. Why? If there were a less costly way to parenthesize $A_i A_{i+1} \dots A_k$ then we could substitute that parenthesization in the optimal parenthesization $A_i A_{i+1} \dots A_j$ to produce another way to parenthesize $A_i A_{i+1} \dots A_j$ whose cost was lower than the optimum: a contradiction. A similar observation holds for how we parenthesize the sub-chain $A_{k+1} A_{k+2} \dots A_j$ in the optimal parenthesization of $A_i A_{i+1} \dots A_j$: it must be an optimal parenthesization of $A_{k+1} A_{k+2} \dots A_j$.

Now we use our optimal substructure to show that we can construct an optimal solution to the problem from optimal solutions to sub-problems. We have seen that any solution to a nontrivial instance of the **matrix-chain multiplication** problem requires us to split the product, and that any optimal solution contains within it optimal solutions to sub-problem instances. Thus, we can build an optimal solution to an instance of the matrix-chain multiplication problem by splitting the problem into two sub-problems (optimally parenthesizing $A_i A_{i+1} \dots A_k$ and $A_{k+1} A_{k+2} \dots A_j$), finding optimal solutions to sub-problem instances, and then combining these optimal sub-problem solutions. We must 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.

3.2 Step 2: A Recursive Solution.

Next, we define the cost of an optimal solution recursively in terms of the optimal solutions to sub-problems. For the **matrix-chain multiplication** problem, we pick as our sub-problems the problems of determining the minimum cost of parenthesizing $A_i A_{i+1} \dots A_j$ for $1 \leq i \leq j \leq n$. Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i \dots j}$; for the full problem, the lowest-cost way to compute $A_{1 \dots n}$ would thus be $m[1, n]$.

We can define $m[i, j]$ recursively as follows. If $i = j$, the problem is trivial; the chain consists of just one matrix $A_{i \dots i} = A_i$, so that no scalar multiplications are necessary to compute the product. Thus, $m[i, i] = 0$ for $i = 1, 2, \dots, n$. To compute $m[i, j]$ when $i < j$, we take advantage of the structure of an optimal solution from step 1. Let us assume that to optimally parenthesize, we split the product $A_i A_{i+1} \dots A_j$ between A_k and A_{k+1} , where $i \leq k < j$. The $m[i, j]$ equals the minimum cost for computing the sub-products $A_{i \dots k}$ and $A_{k+1 \dots j}$, plus the cost of multiplying these two matrices together. Recalling that each matrix A_i is $p_{i-1} \times p_i$, we see that computing the matrix product A_k and A_{k+1} takes $p_{i-1} \times p_k \times p_j$ scalar multiplications. Thus, we obtain:

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j.$$

This recursive equation assumes that we know the value of k , which we do not. There are only $j - i$ possible values for k , however, namely $k = i, i + 1, \dots, j - 1$. Since the optimal parenthesization must use one of these values for k , we need only check them all to find the best. Thus, our recursive definition for the minimum cost of parenthesizing the product $A_i A_{i+1} \dots A_j$ becomes:

$$m[i, j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{i \leq k < j} \{ m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j \} & \text{if } i < j. \end{cases} \quad (3)$$

The $m[i, j]$ values give the costs of optimal solutions to subproblems, but they do not provide all the information we need to construct an optimal solution. To help us do so, we define $s[i, j]$ to be a value of k at which we split the product $A_i A_{i+1} \dots A_j$ in an optimal parenthesization. That is, $s[i, j]$ equals a value k such that $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$.

3.3 Step 3: Computing the Optimal Cost.

At this point, we could easily write a recursive algorithm based on recurrence (3) to compute the minimum cost $m[1, n]$ for multiplying $A_1A_2\dots A_n$. Observe that we have relatively few distinct sub-problems: one sub-problem for each choice of i and j satisfying $1 \leq i \leq j \leq n$ or $\binom{n}{2} + n = \theta(n^2)$ in all. A recursive algorithm may encounter each sub-problem 2 many times in different branches of its recursion tree. This property of overlapping sub-problems is the second hallmark of when dynamic programming applies (the first hallmark being optimal substructure). Instead of computing the solution to recurrence (3) recursively, we compute the optimal cost by using a tabular, bottom-up approach.

We shall implement the tabular, bottom-up method in the procedure **matrix_chain_order**, which appears below. This procedure assumes that matrix A_i has dimensions $p_{i-1} \times p_i$ for $i = 1, 2, \dots, n$. Its input is a sequence $p = \langle p_0, p_1, \dots, p_n \rangle$ where $p.length = n + 1$. The procedure uses an auxiliary table $m[1 \dots n, 1 \dots n]$ for storing the $m[i, j]$ costs and another auxiliary table $s[1 \dots n - 1, 2 \dots n]$ 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. In order to implement the bottom-up approach, we must determine which entries of the table we refer to when computing $m[i, j]$. Equation (3) show that the cost of $m[i, j]$ of computing a matrix-chain product of $j - i + 1$ matrices depends only on the costs of computing matrix-chain products of fewer than $j - i + 1$ matrices. This is, for $k = i, i + 1, \dots, j - 1$, the matrix $A_{i\dots k}$ is a product of $k - i + 1 < j - i + 1$ matrices and the matrix $A_{k+1\dots j}$ is a product of $j - k < j - i + 1$ matrices. Thus, the algorithm should fill in the table m in a manner that corresponds to solving the parenthesization problem on matrix chains of increasing length. For the subproblem of optimally parenthesizing the chain $A_iA_{i+1}\dots A_j$, we consider the sub-problem size to be the length $j - i + 1$ of the chain.

```
1 def matrix_chain_order ( p ):
2     n = len ( p )
3     # Initialize the matrices of size n x n.
4     m = [ [ "x" for i in range ( 1, n + 1 ) ] for j in range ( 1, n + 1 ) ]
5     s = [ [ "x" for i in range ( 1, n + 1 ) ] for j in range ( 1, n + 1 ) ]
6     # m [ i, i ] = 0 for i = 1, ..., n (minimum costs for chains of length 1).
7     for i in range ( n ):
8         m [ i ] [ i ] = 0
9     # Finds the optimal matrix-chain product.
10    for l in range ( 2, n ):
11        for i in range ( 1, n - l + 1 ):
12            j = i + l - 1
13            m [ i ] [ j ] = int ( 1e100 )
14            for k in range ( i, j ):
15                q = m [ i ] [ k ] + m [ k + 1 ] [ j ] + ( p [ i - 1 ] * p [ k ] * p [ j ] )
16                if ( q < m [ i ] [ j ] ):
17                    m [ i ] [ j ] = q
18                    s [ i ] [ j ] = k
19
20    return m, s
```

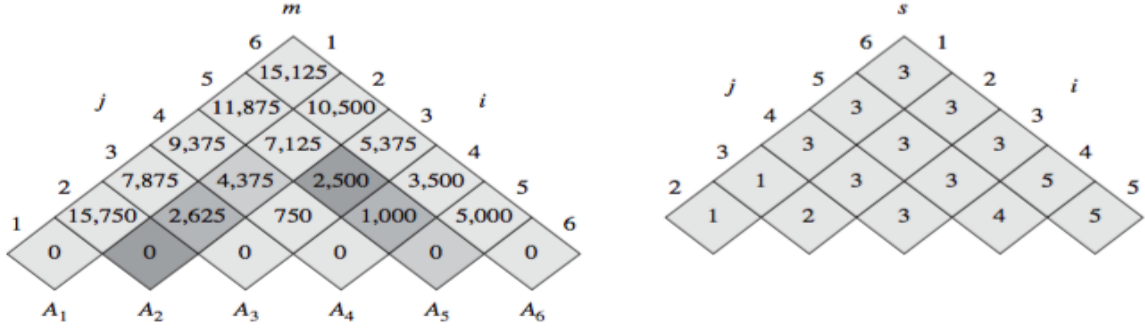


Figure 3.3.0: The m and s tables computed by **matrix_chain_order** for $n = 6$ and the matrix dimensions in table 1. The tables are rotated so that the main diagonal runs horizontally. The m table uses only the main diagonal and upper triangle, and the s table uses only the upper triangle. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1, 6] = 15,125$.

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimensions	30 x 35	35 x 15	15 x 5	5 x 10	10 x 20	20 x 25

Table 1: Matrices dimensions for Figure 3.3.0.

The algorithm first computes $m[i, i] = 0$ for $i = 1, 2, \dots, n$ (the minimum costs for chains of length 1) in lines 7 - 8. It then uses recurrence (3) to compute $m[i, i + 1]$ for $i = 1, 2, \dots, n - 1$ (the minimum costs for chains of length $l = 2$) during the first execution of the **for** loop in lines 10 - 18. The second time through the loop, it computes $m[i, i + 2]$ for $i = 1, 2, \dots, n - 2$ (the minimum costs for chains of length $l = 3$), and so forth. At each step, the $m[i, j]$ cost computed in lines 15 - 18. Depends only on entries $m[i, k]$ and $m[k + 1, j]$ already computed.

Figure 3.3.0 illustrates this procedure on a chain of $n = 6$ matrices. Since we have defined $m[i, j]$ only for $i \leq j$, only the portion of the table m strictly above the main diagonal is used. The figure shows the table rotated to make the main diagonal run horizontally. The matrix chain is listed along the bottom. Using this layout, we can find the minimum cost $m[i, j]$ for multiplying a sub-chain $A_i A_{i+1} \dots A_j$ of matrices at the intersection of lines running northeast from A_i and northwest from A_j . Each horizontal row in the table contains the entries for matrix chains of the same length. **matrix_chain_order** computes the rows from bottom to top and from left to right within each row. It computes each entry $m[i, j]$ using the products $p_{i-1} x p_k x p_j$ for $k = i, i + 1, \dots, j - 1$ and all entries southwest and southeast from $m[i, j]$.

3.4 Step 4: Constructing an Optimal Solution:

Although **matrix_chain_order** determines the optimal number of scalar multiplications needed to compute a matrix-chain product, it does not directly show how to multiply the matrices. The table $s[1 \dots n-1, 2 \dots n]$ gives us the information we need to do so. Each entry $s[i, j]$ records a value of k such that an optimal parenthesization of $A_i A_{i+1} \dots A_j$ splits the product between A_k and A_{k+1} . Thus, we know that the final matrix multiplication in computing $A_{1 \dots n}$ optimally is $A_{1 \dots s[1, n]} A_{s[1, n]+1 \dots n}$. We can determine the earlier matrix multiplications recursively, since $s[1, s[1, n]]$ determines the last matrix multiplication when computing $A_{1 \dots s[1, n]}$ and $s[s[1, n]+1, n]$ determines the last matrix multiplication when computing $A_{s[1, n]+1 \dots n}$. The following recursive procedure prints an optimal parenthesization of $\langle A_i, A_{i+1}, \dots, A_j \rangle$, given the s table computed by **matrix_chain_order** and the indices i and j . The initial call **print_optimal_parens** ($s, 1, n$) prints an optimal parenthesization of $\langle A_1, A_2, \dots, A_n \rangle$.

```
1 def print_optimal_parens ( s, i, j ):
2     global parenthesization
3     if ( i == j ):
4         parenthesization = parenthesization + "A{}".format ( i )
5     else:
6         parenthesization = parenthesization + "("
7         print_optimal_parens ( s, i, s [ i ] [ j ] )
8         print_optimal_parens ( s, s [ i ] [ j ] + 1, j )
9         parenthesization = parenthesization + ")"
```

In the example of Figure 3.3.0 the call **print_optimal_parens** ($s, 1, 6$) prints the parenthesization: $((A_1 (A_2 A_3)) ((A_4 A_5) A_6))$.

4 Results:

All the code shown above doesn't have significance if its operation is not shown. This section will show you the *console* output of the algorithms previously mentioned.

4.1 Matrix-Chain of Size $n = 6$:

The first test consist in calculate the optimal parenthesization for the matrices in Table 2, The output of our program it's captured in Figure 4.1.0.

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimensions	30 x 35	35 x 15	15 x 5	5 x 10	10 x 20	20 x 25

Table 2: Matrix-chain of $n = 6$.

```
MacBook-Pro-de-David:Matrix Chain Order Davestrings$ python3 matrix_chain_order.py

Matrix Chain Order:

Matrix chain: [30, 35, 15, 5, 10, 20, 25]

Table m:
[0, 'x', 'x', 'x', 'x', 'x', 'x']
['x', 0, 15750, 7875, 9375, 11875, 15125]
['x', 'x', 0, 2625, 4375, 7125, 10500]
['x', 'x', 'x', 0, 750, 2500, 5375]
['x', 'x', 'x', 'x', 0, 1000, 3500]
['x', 'x', 'x', 'x', 'x', 0, 5000]
['x', 'x', 'x', 'x', 'x', 'x', 0]

Table s:
['x', 'x', 'x', 'x', 'x', 'x', 'x']
['x', 'x', 1, 1, 3, 3, 3]
['x', 'x', 'x', 2, 3, 3, 3]
['x', 'x', 'x', 'x', 3, 3, 3]
['x', 'x', 'x', 'x', 'x', 4, 5]
['x', 'x', 'x', 'x', 'x', 'x', 5]
['x', 'x', 'x', 'x', 'x', 'x', 'x']

Parent hesi zation: ((A1(A2A3))(A4A5))G
```

Figure 4.1.0: Console output for Table 2 matrices.

As we can see in Figure 4.1.0 we have 2 tables; m and s . As we have explained in section 3, the table s are the pivots for the optimal parenthesization and table m it's the one that contains the minimum cost. This minimum cost it's stored in row 2, column 7 which is 15, 125. Finally in the bottom we can visualize the correct parenthesization.

Also for this test, we don't capture the others parenthesizations as in Figure 4.1.0, but we captured all the costs for all configurations as we can see in Figure 4.1.1, and also we corroborate that the configuration in Figure 4.1.0 it's the optimal.

MacBook-Pro-de-David: Matrix Chain Order All Cost			
All scalar configurations:			
4 4, 5 5 Cost: 5000	1 3, 4 4 Cost: 11375	3 3, 4 4 Cost: 1000	
3 3, 4 5 Cost: 6250	1 4, 5 5 Cost: 24625	1 2, 3 4 Cost: 7125	
3 3, 4 4 Cost: 1000	0 0, 1 5 Cost: 36750	2 2, 3 3 Cost: 750	
3 4, 5 5 Cost: 3500	0 0, 1 1 Cost: 15750	1 1, 2 3 Cost: 6000	
2 2, 3 5 Cost: 5375	4 4, 5 5 Cost: 5000	1 1, 2 2 Cost: 2625	
2 2, 3 3 Cost: 750	3 3, 4 5 Cost: 6250	1 2, 3 3 Cost: 4375	
4 4, 5 5 Cost: 5000	3 3, 4 4 Cost: 1000	1 3, 4 4 Cost: 11375	
2 3, 4 5 Cost: 9500	3 4, 5 5 Cost: 3500	0 0, 1 4 Cost: 28125	
3 3, 4 4 Cost: 1000	2 2, 3 5 Cost: 5375	0 0, 1 1 Cost: 15750	
2 2, 3 4 Cost: 2500	2 2, 3 3 Cost: 750	3 3, 4 4 Cost: 1000	
2 2, 3 3 Cost: 750	4 4, 5 5 Cost: 5000	2 2, 3 4 Cost: 2500	
2 3, 4 4 Cost: 3750	2 3, 4 5 Cost: 9500	2 2, 3 3 Cost: 750	
2 4, 5 5 Cost: 10000	3 3, 4 4 Cost: 1000	2 3, 4 4 Cost: 3750	
1 1, 2 5 Cost: 18500	2 2, 3 4 Cost: 2500	0 1, 2 4 Cost: 27250	
1 1, 2 2 Cost: 2625	2 2, 3 3 Cost: 750	1 1, 2 2 Cost: 2625	
4 4, 5 5 Cost: 5000	2 3, 4 4 Cost: 3750	0 0, 1 2 Cost: 7875	
3 3, 4 5 Cost: 6250	2 4, 5 5 Cost: 10000	0 0, 1 1 Cost: 15750	
3 3, 4 4 Cost: 1000	0 1, 2 5 Cost: 32375	0 1, 2 2 Cost: 18000	
3 4, 5 5 Cost: 3500	1 1, 2 2 Cost: 2625	3 3, 4 4 Cost: 1000	
1 2, 3 5 Cost: 10500	0 0, 1 2 Cost: 7875	0 2, 3 4 Cost: 11875	
2 2, 3 3 Cost: 750	0 0, 1 1 Cost: 15750	2 2, 3 3 Cost: 750	
1 1, 2 3 Cost: 6000	0 1, 2 2 Cost: 18000	1 1, 2 3 Cost: 6000	
1 1, 2 2 Cost: 2625	4 4, 5 5 Cost: 5000	1 1, 2 2 Cost: 2625	
1 2, 3 3 Cost: 4375	3 3, 4 5 Cost: 6250	1 2, 3 3 Cost: 4375	
4 4, 5 5 Cost: 5000	3 3, 4 4 Cost: 1000	0 0, 1 3 Cost: 14875	
1 3, 4 5 Cost: 18125	3 4, 5 5 Cost: 3500	0 0, 1 1 Cost: 15750	
3 3, 4 4 Cost: 1000	0 2, 3 5 Cost: 15125	0 1, 2 3 Cost: 21000	
2 2, 3 4 Cost: 2500	2 2, 3 3 Cost: 750	1 1, 2 2 Cost: 2625	
2 2, 3 3 Cost: 750	1 1, 2 3 Cost: 6000	0 0, 1 2 Cost: 7875	
2 3, 4 4 Cost: 3750	1 1, 2 2 Cost: 2625	0 0, 1 1 Cost: 15750	
1 1, 2 4 Cost: 13000	1 2, 3 3 Cost: 4375	0 1, 2 2 Cost: 18000	
1 1, 2 2 Cost: 2625	0 0, 1 3 Cost: 14875	0 2, 3 3 Cost: 9375	
	0 0, 1 1 Cost: 15750	0 3, 4 4 Cost: 15375	
	2 2, 3 3 Cost: 750	0 4, 5 5 Cost: 26875	
	0 1, 2 3 Cost: 21000		
	1 1, 2 2 Cost: 2625		
	0 0, 1 2 Cost: 7875		
		Minimum costs: 15125	

Figure 4.1.1: All configurations output.

4.2 Matrix-Chain of Size $n = 3$:

The second test consist in calculate the optimal parenthesization for the matrices in Table 3, The output of our program it's captured in Figure 4.2.0.

Matrix	A_1	A_2	A_3
Dimensions	3 x 5	5 x 2	2 x 2

Table 3: Matrix-chain of $n = 3$.

```

MacBook-Pro-de-David: Matrix Chain Order Davestring$ python3 matrix_

Matrix Chain Order:

Matrix chain: [3, 5, 2, 2]

Table m:
[ 0, 'x', 'x', 'x' ]
[ 'x', 0, 30, 42 ]
[ 'x', 'x', 0, 20 ]
[ 'x', 'x', 'x', 0 ]

Table s:
[ 'x', 'x', 'x', 'x' ]
[ 'x', 'x', 1, 2 ]
[ 'x', 'x', 'x', 2 ]
[ 'x', 'x', 'x', 'x' ]

Parenthesization: ((A1A2)A3)

```

Figure 4.2.0: Console output for Table 3 matrices.

As we can see in Figure 4.2.0 we have 2 tables; m and s . As we have explained in section 3, the table s are the pivots for the optimal parenthesization and table m it's the one that contains the minimum cost. This minimum cost it's stored in row 2, column 4 which is 42. Finally in the bottom we can visualize the correct parenthesization.

Also for this test, we don't capture the others parenthesizations as in Figure 4.2.0, but we captured all the costs for all configurations as we can see in Figure 4.2.1, and also we corroborate that the configuration in Figure 4.2.0 it's the optimal.



```
MacBook-Pro-de-David:Matrix Chain Order All Costs Davestring$ ./a.out

All scalar configurations:

1 1, 2 2 Cost: 12
0 0, 1 2 Cost: 42
0 0, 1 1 Cost: 30
0 1, 2 2 Cost: 50

Minimum costs: 42
```

Figure 4.2.1: All configurations output.

In this example it's more evident that in Figure 4.2.1 are displayed all the scalar product for all the configurations. From table 3 lets make the product of the first 2 matrices, the scalar operations are: $3 \times 5 \times 2 = 30$ which resulting matrix it's of size 3×2 . Then, lets multiply this resulting matrix with A_3 which scalar operations are $3 \times 2 \times 2 = 12$ and its resulting matrix it's of size 3×2 , and the total of scalar operations for this configurations are $30 + 12 = 42$. We can find this configuration in Figure 4.2.1 in the row 2, i.e. $((A_1 A_2) A_3)$.

Then, the other configuration it's $(A_1 (A_2 A_3))$, so lets multiply first A_2 and A_3 which scalar operations are $5 \times 2 \times 2 = 20$ and the resulting matrix it's of size 5×2 . Then, let's multiply this resulting matrix with A_1 , which scalar operations are $3 \times 5 \times 2 = 30$, and the total of scalar operations for this configurations are $30 + 20 = 50$. We can find this configuration in Figure 4.2.1 in the row 4.

4.3 Matrix-Chain of Size $n = 4$:

The third test consist in calculate the optimal parenthesization for the matrices in Table 4, The output of our program it's captured in Figure 4.3.0.

Matrix	A_1	A_2	A_3	A_4
Dimensions	10 x 8	8 x 12	12 x 2	2 x 5

Table 4: Matrix-chain of $n = 4$.

```

Matrix Chain Order:

Matrix chain: [ 10, 8, 12, 2, 5]

Table m:
[ 0, 'x', 'x', 'x', 'x' ]
['x', 0, 960, 352, 452]
['x', 'x', 0, 192, 272]
['x', 'x', 'x', 0, 120]
['x', 'x', 'x', 'x', 0]

Table s:
['x', 'x', 'x', 'x', 'x' ]
['x', 'x', 1, 1, 3]
['x', 'x', 'x', 2, 3]
['x', 'x', 'x', 'x', 3]
['x', 'x', 'x', 'x', 'x' ]

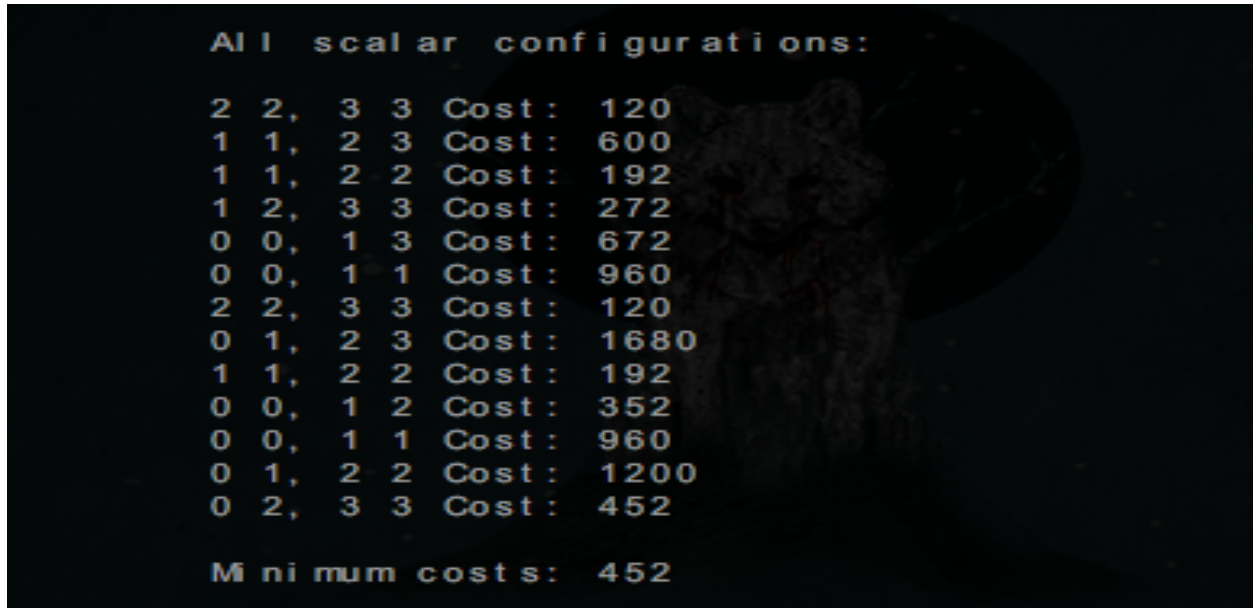
Parent hesi zation: (( A1( A2A3) ) A4)

```

Figure 4.3.0: Console output for Table 4 matrices.

As we can see in Figure 4.3.0 we have 2 tables; m and s . As we have explained in section 3, the table s are the pivots for the optimal parenthesization and table m it's the one that contains the minimum cost. This minimum cost it's stored in row 2, column 5 which is 452. Finally in the bottom we can visualize the correct parenthesization.

Also for this test, we don't capture the others parenthesizations as in Figure 4.3.0, but we captured all the costs for all configurations as we can see in Figure 4.3.1, and also we corroborate that the configuration in Figure 4.3.0 it's the optimal.



All scalar configurations:			
2	2,	3	3 Cost: 120
1	1,	2	3 Cost: 600
1	1,	2	2 Cost: 192
1	2,	3	3 Cost: 272
0	0,	1	3 Cost: 672
0	0,	1	1 Cost: 960
2	2,	3	3 Cost: 120
0	1,	2	3 Cost: 1680
1	1,	2	2 Cost: 192
0	0,	1	2 Cost: 352
0	0,	1	1 Cost: 960
0	1,	2	2 Cost: 1200
0	2,	3	3 Cost: 452
Minimum costs: 452			

Figure 4.3.1: All configurations output.

In Figure 4.3.1 are displayed all the scalar product for all the configurations. From table 4 lets make the product of the first 2 matrices, the scalar operations are: $10 \times 8 \times 12 = 960$ which resulting matrix it's of size 10×12 . Then, lets multiply the matrices A_3 and A_4 , the scalar operations are: $12 \times 2 \times 5 = 120$. Finally lets multiply this 2 resulting matrices, which scalar operations are $10 \times 12 \times 5 = 600$ and the total of scalar operations it's $600 + 960 + 120 = 1680$ and the configuration is: $((A_1 A_2)(A_3 A_4))$. We can find this result in Figure 4.3.1 row 8.

Then, the other configuration it's $((A_1(A_2 A_3))A_4)$, so lets multiply first A_2 and A_3 which scalar operations are $8 \times 12 \times 2 = 192$ and the resulting matrix it's of size 8×2 . Then, let's multiply this resulting matrix with A_1 , which scalar operations are $10 \times 8 \times 2 = 160$. Finally lets multiply this resulting matrix with A_4 which scalar operations are $10 \times 2 \times 5 = 100$ and the total of scalar operations for this configuration it's $192 + 160 + 100 = 452$ making this parenthesization the optimal.

5 Annexes:

In the following section we will formally demonstrate the complexity of section's 2 equation (2) using the substitution method.

Solution:

Let assume that $P(k) \geq c2^k$ for all $k < n$. Let us prove that $P(n) \geq c2^n$. For $n > 1$.

$$\begin{aligned} P(n) &= \sum_{k=1}^{n-1} P(k)P(n-k) \geq \sum_{k=1}^{n-1} c2^k \cdot c2^{n-k} \\ &= \sum_{k=1}^{n-1} c^2 2^n \\ &= (n-1) \cdot c^2 2^n \\ &\geq c2^n. \end{aligned}$$

Note: Provided that $(n-1) \cdot c > 1$.

Finally: $P(n)$ is $\Omega(2^n)$.

6 Conclusion:

Maybe the first dynamic algorithm program that I ever have seen in all my computing career. It's very interesting to see that if we have a chain of matrices, depending on how we associate to multiply them, it will result in an optimal or not configuration. Dynamic programming works particularly well for optimization problems with inherent left to right ordering among elements. The resulting global optimum often much better than solution found with heuristics. And once understood, dynamic programming is usually easier to work algorithm out from scratch than to look up algorithm.

- Hernandez Martinez Carlos David.

7 Bibliography References:

- [1] Baase and Van Gelder. "Computer Algorithms: Introduction to Design and Analysis". Addison-Wesley.
- [2] Thomas H. Cormen. "Introduction to Algorithms". The MIT press.