Matrix-chain Multiplication by Dynamic Programming

Multiply n compatible matrices $A_1 A_2 \cdots A_n$. It can be done in various ways, e.g.

$A_1A_2A_3A_4$ can be done as $(A_1(A_2(A_3A_4)))$ or $(A_1((A_2A_3)A_4))$ or $((A_1A_2)(A_3A_4))$ or $((A_1(A_2A_3))A_4)$ or $(((A_1A_2)A_3)A_4)$. since matrix multiplication is associative.

We intend to parenthesize the product in such a way that the total number of scalar multiplication is minimized, e.g.,

$A_{10\times100} \times A_{100\times5} \times A_{5\times50}$ can be parenthesized to perform $10\times100\times5 + 10\times5\times50 = 7500$ multiplications or $100\times5\times50 + 10\times100\times50 = 75000$ multiplications.

The problem:

Given a chain $A_1, A_2, \ldots, A_n$ of n matrices where for $i = 1, 2, \ldots, n$ matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

Solution:

Let P(n) be the number of alternative parenthesization of a sequence of n matrices.

We can split the sequence between k^{th} and $(k+1)^{th}$ matrices for any $k = 1, 2, \ldots, n-1$ and then parenthesize the two resulting sequences independently,

$$(A_1 A_2 A_3 \cdots A_k)\overset{\downarrow}{}(A_{k+1} A_{k+2} \cdots A_n)$$

We then get the recurrence

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

This recurrence is related to a famous function in combinatorics called the Catalan numbers, which are related to the number of different binary trees of n nodes. So,

$P(n) = C(n-1)$ where $C(n)$ is the n^{th} Catalan number. By applying Stirling's formula,

$$C(n) = \frac{1}{n+1}\binom{2n}{n}$$
$$= \Omega\left(4^n/n^{3/2}\right).$$

Since 4^n is exponential, the function grows very rapidly. Thus, the number of solutions is exponential in n and brute-force method of exhaustive search is a poor strategy for determining the optimal parenthesization of a matrix chain. Therefore, the naive algorithm will not be practical except for very small n.

Dynamic Programming approach:

Let us write

$$A_{i..j} = A_i A_{i+1} \cdots A_j \text{ where } i \leq j.$$

So, for the highest level of parenthesizing,

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

There are now two questions:

- * How to decide where to split the chain i.e what is k?
- * How do we parenthesize the subchains $A_{1..k}$ and $A_{k+1..n}$

The Principle of Optimality now comes into play:

Principle of Optimality: "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision." — Bellman

We say, the problem can be broken apart into optimal substructure. The present problem satisfies the Principle of Optimality, because once we decide to break the sequence into the product, we must compute each subsequence optimally. So, for solving the global problem optimally, we must solve the subproblems optimally first. So, $A_{1..k}$ must also be parenthesized optimally. Similarly, the other subchain $A_{k+1..n}$ must also be optimally parenthesized.

The next step of the dynamic programming paradigm is to define the value of an optimal solution recursively in terms of the optimal solutions to subproblems. We will build a table in a bottom-up manner that will keep track of solutions to subproblems.

For $1 \leq i \leq j \leq n$, let $m[i,j]$ be the minimum number of scalar multiplications needed to compute the $A_{i..j}$.

Observe that if $i=j$ then the problem is trivial; the sequence contains only one matrix, there is nothing to multiply and so the cost is 0.

Thus, $m[i,i] = 0$ for $i = 1, 2, .., n$.

If $i \neq j$ then it is the product of the subchain $A_{i..j}$ and we take advantage of the Principle of Optimality and assuming that the optimal parenthesization splits the product $A_{i..j}$ into

$A_{i..k} A_{k+1..j}$ for each value of k, $1 \leq k \leq n-1$.

Therefore,

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

To keep track of optimal subsolutions, we store the value of k in a table $s[i,j]$. That is,

$$s[i,j] = k \text{ such that } m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$$

We now implement the above recurrence into a procedure.

In the process of computing $m[i,j]$ we will need to access values $m[i,k]$ and $m[k+1,j]$ for each value of k lying between i and j. This suggests that we should organize 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 i.e $m[i,i]$ are trivial. Then we build up by computing the subchains of length $2, 3, .., n$. Finally $m[1,n]$.

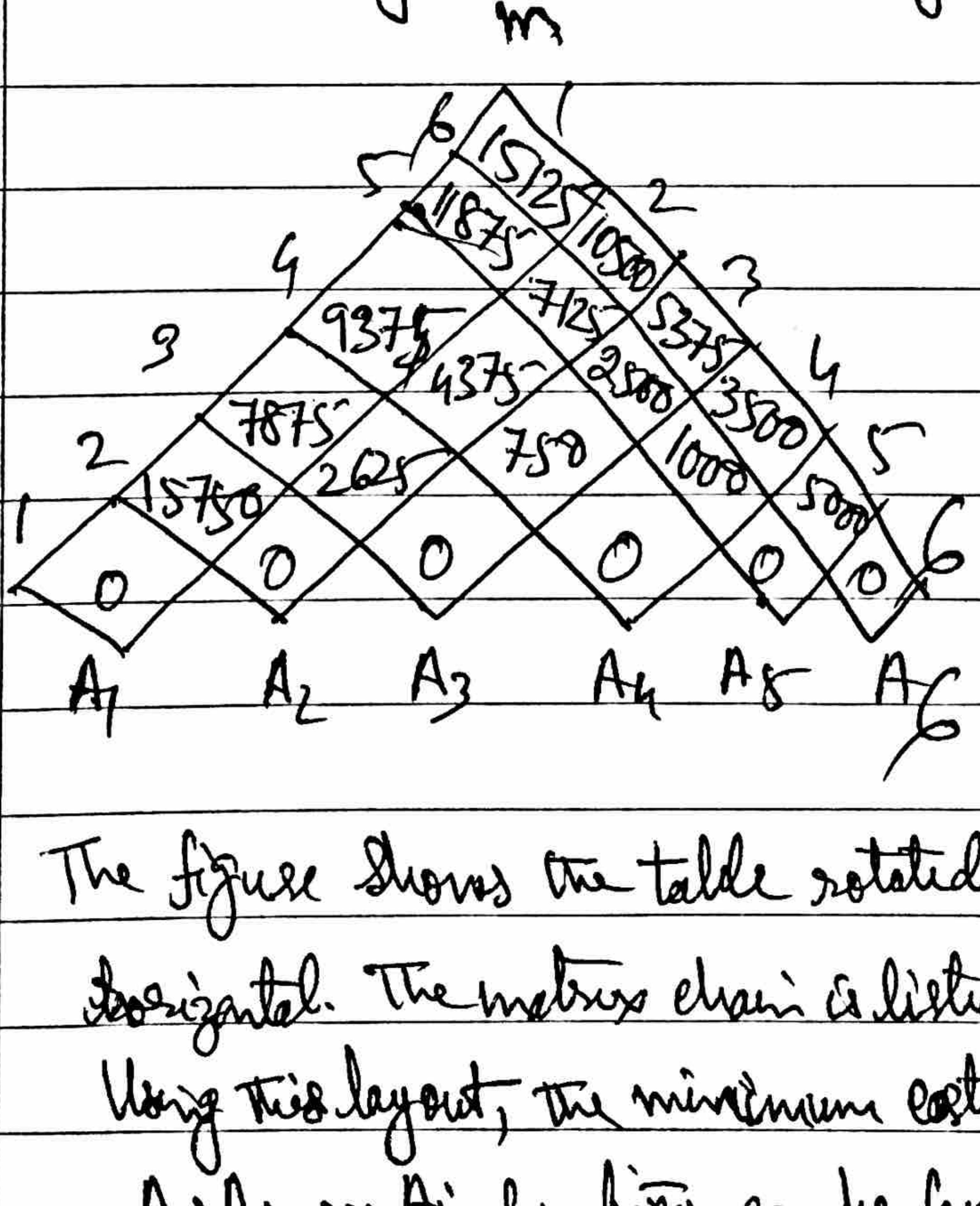
```
Matrix-Chain (array p[1..n], int n) {
    array s [1..n-1, 2..n];
    for i = 1 to n m[i,i] = 0;
    for L = 2 to n {
        for i = 1 to n-L+1 {
            j = i+L-1;
            m[i,j] = infinity;
            for k = i to j-1 {
                q = m[i,k] + m[k+1,j] + p[i-1] p[k] p[j];
                if (q < m[i,j]) {
                    m[i,j] = q;
                    s [i,j] = k;
                }
            }
        }
    }
    return m [1,n] // final cost
        s // splitting markers.
}
```

The input is a sequence {$p_0, p_1, .., p_n$} where length [p] = n+1. The procedure uses an auxiliary table m [1..n, 1..n] for storing the m[i,j] costs and an auxiliary table s [1..n, 1..n] that records which index of k achieved the optimum cost in computing m[i,j]. The algorithm first computes m[i,i] = 0 for i = 1,2, .., n. I + then uses the recurrence relation to compute m[i, i+1] for i = 1,2, .., n-1 during the first execution of the loop. The second time through the loop, it computes m[i, i+2] for i=1,2,..,n-2 and so on. At each step, the m[i,j] cost computed depends only on table entries m[i,k] and m[k+1,j] already computed.

Example:

Matrix	Dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

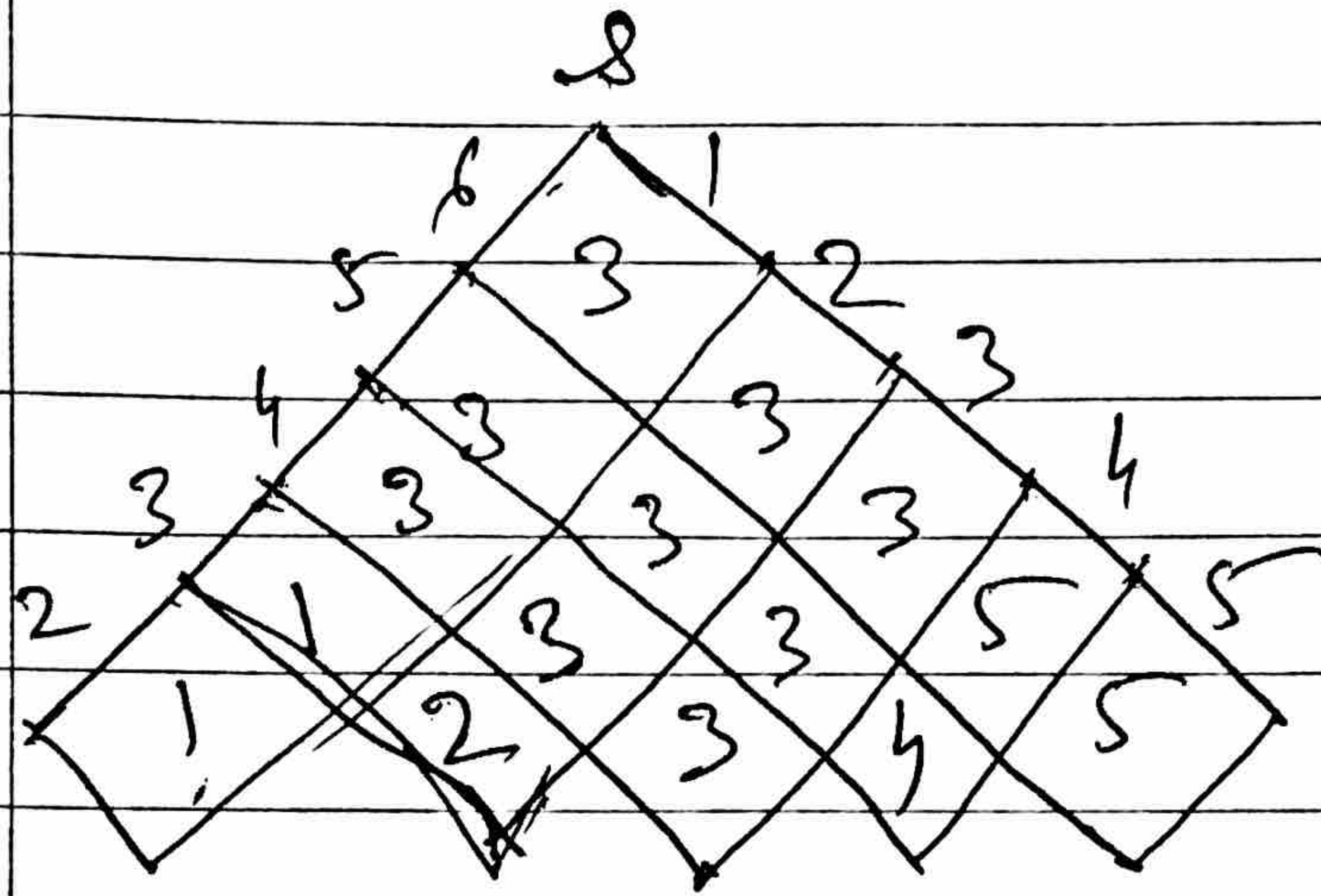
Since we have defined m[i,j] only for i≤j, the portion of the table m strictly above the main diagonal is used.



The figure shows the table rotated to make the main diagonal horizontal. The matrix chain is listed at the bottom.
Using this layout, the minimum cost m[i,j] for multiplying a subchain $A_i A_{i+1} \cdots A_j$ of matrices can be found at the intersection of lines

__/__/__

running northeast from A_i and northwest from A_j.
The minimum number of scalar multiplications to multiply the 6 matrices is $m[1, 6] = 15125$.



The actual chain or sequence is given by the following procedure with an initial call to Chain-Order(A, s, 1, n)

```
Chain-Order (A, s, i, j) {
    if i<j then {
        X = Chain-Order (A, s, i, s[i, j]);
        Y = Chain-order (A, s, s[i, j]+1, j);
        return Chain-Order (X, Y);
    }
    else return $A_i$
}
```

In the example given, the parenthesization given by the procedure is $((A_1(A_2A_3))((A_4A_5)A_6))$.