Matrix-Chain Multiplication

Dynamic Programming Lecture Notes (CS5800)

Virgil Pavlu

notes written by Claude

The problem, and why the order matters

We are given a chain of n matrices $A_1, A_2, \ldots, A_n$ to multiply together. Matrix A_t has dimension $p_{t-1} \times p_t$, so consecutive dimensions match and the product $A_1 A_2 \cdots A_n$ is defined. Matrix multiplication is *associative* — every way of parenthesizing gives the same matrix — but the *cost* (number of scalar multiplications) depends heavily on *how* we parenthesize. Multiplying a $p \times q$ by a $q \times r$ matrix costs $p\,q\,r$ scalar multiplications. **We want the parenthesization that minimizes the total scalar multiplications.**

A tiny example. Take $A_1\,(10 \times 100)$, $A_2\,(100 \times 5)$, $A_3\,(5 \times 50)$.

$$(A_1A_2)A_3 : \underbrace{10 \cdot 100 \cdot 5}_{5000} + \underbrace{10 \cdot 5 \cdot 50}_{2500} = 7500, \qquad A_1(A_2A_3) : \underbrace{100 \cdot 5 \cdot 50}_{25000} + \underbrace{10 \cdot 100 \cdot 50}_{50000} = 75000.$$

Same answer matrix, but one order is *ten times* cheaper. With n matrices the choice matters a lot.

Why not just try them all? Let $P(n)$ be the number of full parenthesizations of n matrices. Splitting at the top level between matrix k and $k+1$ gives $P(n) = \sum_{k=1}^{n-1} P(k)\,P(n-k)$ with $P(1) = 1$ — the Catalan recurrence. A quick induction gives $P(n) \geq 2^{n-2}$, and in fact $P(n) = \Omega(4^n/n^{3/2})$. Exhaustive search is hopeless; but the same "split at the top" idea, *memoized*, is exactly the DP.

Running example. We carry the classic $n = 6$ instance through the document:

$$(p_0, p_1, \ldots, p_6) = (30,\ 35,\ 15,\ 5,\ 10,\ 20,\ 25),$$

so $A_1{:}\,30{\times}35$, $A_2{:}\,35{\times}15$, $A_3{:}\,15{\times}5$, $A_4{:}\,5{\times}10$, $A_5{:}\,10{\times}20$, $A_6{:}\,20{\times}25$.

The DP is built in five steps. We mark each one explicitly.

Step 1 — Solution characterization (divide & conquer)

Goal of this step: show that an optimal parenthesization is built from optimal parenthesizations of two sub-chains.

Any parenthesization of $A_i A_{i+1} \cdots A_j$ (with $i < j$) makes *one* multiplication last: the outermost product splits the chain between some A_k and A_{k+1}, $i \leq k < j$,

$$\big(\underbrace{A_i \cdots A_k}_{\text{left}}\big)\big(\underbrace{A_{k+1} \cdots A_j}_{\text{right}}\big).$$

Claim 1 (Optimal substructure)**.** *If a parenthesization of $A_i \cdots A_j$ is optimal and its top-level split is at k, then the way it parenthesizes the left sub-chain $A_i \cdots A_k$ is itself optimal for that sub-chain, and likewise for the right sub-chain $A_{k+1} \cdots A_j$.*

Proof. Cut-and-paste. The total cost is (cost of left) + (cost of right) + $p_{i-1}p_kp_j$, where the last term is the price of the single final multiply (a $p_{i-1} \times p_k$ matrix times a $p_k \times p_j$ matrix) and does *not* depend on how the two halves are internally parenthesized. So if some cheaper parenthesization of the left sub-chain existed, swapping it in would lower the total — contradicting optimality. Same for the right. □

So an optimal solution for $A_i \cdots A_j$ is built from optimal solutions of two strictly shorter sub-chains. That is the divide-and-conquer shape DP needs — the only thing we don't know yet is *where* to split.

Step 2A — The DP recurrence

Define the table in English first:

$$\mathrm{DP}[i][j] \;=\; \text{the minimum number of scalar multiplications to compute } A_iA_{i+1}\cdots A_j.$$

A single matrix costs nothing, so $\mathrm{DP}[i][i] = 0$. For $i < j$ we don't know the top-level split k, so by Claim 1 we *try them all* and keep the cheapest:

$$\boxed{\mathrm{DP}[i][j] = \begin{cases} 0 & i = j, \\ \displaystyle\min_{i \le k < j} \Big(\mathrm{DP}[i][k] + \mathrm{DP}[k+1][j] + p_{i-1}\, p_k\, p_j \Big) & i < j. \end{cases}}$$

We also store $S[i][j] = \arg\min_k$, the split that achieved the minimum — needed to reconstruct the parenthesization in Step 4. The answer is $\mathrm{DP}[1][n]$.

Worked entry. On the running example, $\mathrm{DP}[2][5]$ (the chain $A_2A_3A_4A_5$):

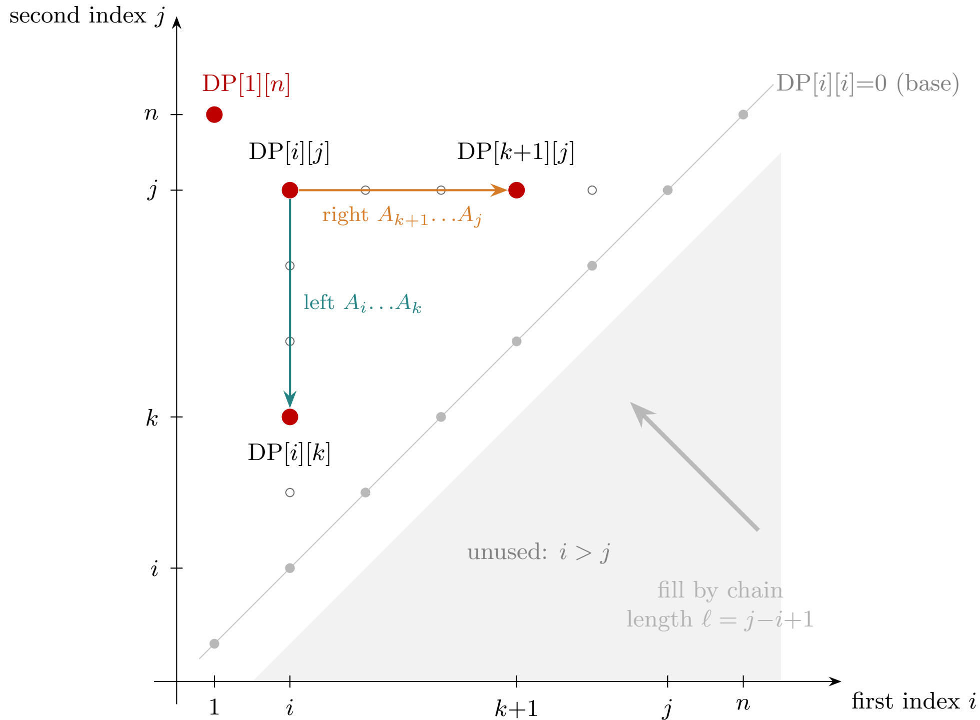
$$\mathrm{DP}[2][5] = \min \begin{cases} k{=}2: \;\; \mathrm{DP}[2][2] + \mathrm{DP}[3][5] + p_1p_2p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000, \\ k{=}3: \;\; \mathrm{DP}[2][3] + \mathrm{DP}[4][5] + p_1p_3p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = \mathbf{7125}, \\ k{=}4: \;\; \mathrm{DP}[2][4] + \mathrm{DP}[5][5] + p_1p_4p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375, \end{cases}$$

so $\mathrm{DP}[2][5] = 7125$ with best split $S[2][5] = 3$.

Step 2B — Subproblem dependence and the fill order

Goal of this step: lay the subproblems out and see which ones each cell needs, so we know a safe order to fill the table bottom-up.

A subproblem is an interval $[i, j]$ of the chain, so we index the table by its two endpoints: the **first index** i on the horizontal axis, the **second index** j on the vertical axis. Only cells with $i \le j$ exist (an interval can't run backwards), so the table is the *upper-left triangle*; the main diagonal $i = j$ holds the base cases $\mathrm{DP}[i][i] = 0$. The full problem is the far corner $\mathrm{DP}[1][n]$.



To compute DP$[i][j]$ the recurrence pairs, for each split k, one cell *straight down its column* (DP$[i][k]$, same i, lower j) with one cell *straight right along its row* (DP$[k+1][j]$, same j, larger i). As k runs from i to $j-1$, those two markers slide toward the diagonal, sweeping the whole column-below and row-right. Every cell it reads is strictly *closer to the diagonal* — i.e. has a *shorter* chain. So the safe **bottom-up order is by increasing chain length** $\ell = j - i + 1$: first the diagonal ($\ell = 1$, all 0), then $\ell = 2$, and so on, each diagonal using only shorter ones already computed, until the lone corner DP$[1][n]$ ($\ell = n$). This diagonal order is the one genuinely non-obvious thing about matrix chain — you cannot just sweep rows or columns.

Step 3 — Bottom-up pseudocode

```
MATRIX-CHAIN-ORDER(p, n):       # p = (p0, p1, ..., pn);  A_t is p[t-1] x p[t]
    for i in 1..n:
        DP[i][i] = 0                           # base: a single matrix costs 0
    for L in 2..n:                             # L = chain length, increasing
        for i in 1..(n-L+1):
            j = i + L - 1
            DP[i][j] = +infinity
            for k in i..(j-1):                 # try every top-level split
                q = DP[i][k] + DP[k+1][j] + p[i-1]*p[k]*p[j]
                if q < DP[i][j]:
                    DP[i][j] = q
                    S[i][j]  = k               # remember the best split
    return DP, S
```

Note 1 (Memoization here helps with *order*, not with skipping)**.** Computing DP$[1][n]$ needs *every* interval $[i, j]$ — all $\Theta(n^2)$ of them — so top-down memoization skips no subproblems; it is

the same $\Theta(n^3)$ work. What it *does* buy is freedom from the diagonal bookkeeping: a recursive `MEMO-CHAIN(i,j)` that returns $\mathrm{DP}[i][j]$, recursing on (i, k) and $(k+1, j)$ and caching, evaluates shorter chains first *automatically*, so you never have to write the "`for L`" loop. (Contrast knapsack, where the natural order is trivial.)

Step 4 — Tracing the optimal parenthesization

$\mathrm{DP}[1][n]$ is the optimal *cost*; the actual parenthesization is read from S. Since $S[i][j]$ is the top-level split, recurse: parenthesize $A_i \cdots A_{S[i][j]}$, then $A_{S[i][j]+1} \cdots A_j$, wrapped in one pair of parentheses.

```
PRINT-PARENS(S, i, j):
    if i == j:
        print "A" + i                         # a single matrix
    else:
        print "("
        PRINT-PARENS(S, i, S[i][j])            # left sub-chain
        PRINT-PARENS(S, S[i][j] + 1, j)        # right sub-chain
        print ")"
```

Filled table on the running example ($n = 6$), same axes as Step 2B: first index i along the bottom, second index j up the side, the answer $\mathrm{DP}[1][6]$ in the top-left corner. The diagonal is the base; shaded cells are the sub-chains used by the optimal parenthesization.

second index j	i=1	i=2	i=3	i=4	i=5	i=6
j=6	15125	10500	5375	3500	5000	0
j=5	11875	7125	2500	1000	0	
j=4	9375	4375	750	0		
j=3	7875	2625	0			
j=2	15750	0				
j=1	0					

first index i

Reconstruction from S. The relevant splits are $S[1][6] = 3$, $S[1][3] = 1$, $S[4][6] = 5$:

$$\underbrace{S[1][6] = 3}_{(A_1A_2A_3)(A_4A_5A_6)} \quad \Rightarrow \underbrace{S[1][3] = 1}_{A_1(A_2A_3)} \quad \text{and} \quad \underbrace{S[4][6] = 5}_{(A_4A_5)A_6}.$$

So the optimal parenthesization is $\big(A_1(A_2A_3)\big)\big((A_4A_5)A_6\big)$, at cost $\mathrm{DP}[1][6] = 15125$ scalar multiplications (versus $30 \cdot 35 \cdot 15 \cdots$ far more for a naive left-to-right product).

Step 5 — Running time and memory

phase	cost
number of cells (intervals $i \le j$)	$\Theta(n^2)$
work per cell (the min over splits k)	$O(n)$
filling the table	$\Theta(n^3)$ time
the DP and S tables	$\Theta(n^2)$ space
`PRINT-PARENS`	$\Theta(n)$ time
total	$\boldsymbol{\Theta(n^3)}$ time, $\boldsymbol{\Theta(n^2)}$ space

The $\Theta(n^3)$ is tight: one can show the work is also $\Omega(n^3)$ (the inner min over k really does $\Theta(j-i)$ work, and summing over all intervals gives $\Theta(n^3)$).

Can we do better? Yes — the **Hu–Shing** algorithm solves matrix-chain ordering in $O(n \log n)$ by turning it into an optimal polygon-triangulation problem. Note this is *not* Knuth's $O(n^2)$ speedup: Knuth's trick needs the added weight to depend only on the interval endpoints, but here the extra term $p_{i-1}p_kp_j$ depends on the split k itself. Its sibling, **Optimal BST**, has the same interval-DP shape *and* a split-independent weight, so there Knuth's monotonicity does give $O(n^2)$. For a first course, $\Theta(n^3)$ is the algorithm to know.