Chapter 14. Dynamic Programming

Joon Soo Yoo

May 20, 2025

Assignment

- Read §14.2
- Problems
  - §14.2 - 1, 3

Chapter 14: Dynamic Programming

- Chapter 14.1: Rod Cutting
- **Chapter 14.2: Matrix-Chain Multiplication**
- Chapter 14.3: Elements of Dynamic Programming
- Chapter 14.4: Longest Common Subsequence
- Chapter 14.5: Optimal Binary Search Trees

What is the Problem?

- Given a chain of matrices $A_1, A_2, \ldots, A_n$
- Goal: Compute the product $A_1 A_2 \ldots A_n$
- Matrix multiplication is **associative**, so we can parenthesize it in many ways
- But: **Different parenthesizations lead to different computational costs!**

Why Does Parenthesization Matter?

- Consider three matrices:
  - $A_1 : 10 \times 100$
  - $A_2 : 100 \times 5$
  - $A_3 : 5 \times 50$
- Two ways to compute:
  1. $((A_1A_2)A_3)$: $10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 7500$
  2. $(A_1(A_2A_3))$: $100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 75000$
- **10$\times$ more efficient!**

How Do We Measure Cost?

- Scalar multiplication: one operation of the form $a_{ik} \cdot b_{kj}$
- Standard algorithm for multiplying $p \times q$ by $q \times r$:

$$\text{Cost} = p \cdot q \cdot r$$

- Total cost of parenthesization is the sum of scalar multiplications

What is the Input?

- Input is a sequence of dimensions:

$$\langle p_0, p_1, p_2, \ldots, p_n \rangle$$

- Matrix A_i has dimensions $p_{i-1} \times p_i$
- We do **not** multiply matrices — we only choose the best parenthesization

Why Use Dynamic Programming?

- Many overlapping subproblems: e.g., A_2A_3 appears in many splits
- Brute-force: exponential number of parenthesesizations
- Dynamic Programming:
  - Use a table to store best cost for each subchain
  - Solve smaller subproblems first
  - Build up to the full problem

What Are We Counting?

- Given: a sequence of n matrices $A_1, A_2, \ldots, A_n$
- Goal: Count the number of ways to fully parenthesize $A_1 A_2 \ldots A_n$
- Matrix multiplication is associative $\rightarrow$ multiple valid parenthesizations
- But the number of possibilities grows rapidly as n increases

Examples of Parenthesization Counts

- $n = 1$: just one matrix $\rightarrow$ 1 way
- $n = 2$: $(A_1 A_2) \rightarrow 1$ way
- $n = 3$:
  - $((A_1 A_2) A_3)$
  - $(A_1 (A_2 A_3))$
- $n = 4$: 5 ways
- $n = 5$: 14 ways

Number of ways increases super-exponentially!

Recursive Formula

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

- For each split point k, split into:
  - Left subchain: $A_1 \dots A_k$
  - Right subchain: $A_{k+1} \dots A_n$
- Multiply number of ways to parenthesize left and right
- Asymptotically:

$$P(n) = \Omega(2^n)$$

Why Not Brute Force?

- For $n = 10$: Over 1000 ways to try
- For $n = 20$: Over a million
- Each possibility requires:
  - Evaluating subchains
  - Tracking multiplication costs
- Exhaustive search is impractical

We need a smarter way $\rightarrow$ Dynamic Programming!

Dynamic Programming Strategy

Four Steps for DP:

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 (bottom-up)
4. Construct the optimal solution from saved data

Step 1: Optimal Substructure

- To compute $A_i A_{i+1} \ldots A_j$, we must split at some $k \in [i, j-1]$
- That is:

$$A_i \ldots A_j = (A_i \ldots A_k) \cdot (A_{k+1} \ldots A_j)$$

- So the total cost:

$$\text{cost} = \text{left} + \text{right} + \text{merge}$$

- Merging cost:

$$p_{i-1} \cdot p_k \cdot p_j$$

Step 2: Recursive Definition

Define $m[i][j]$: minimum cost to multiply $A_i, \cdots, A_j$

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

- Try every split point $k \in [i, j-1]$
- Recursively compute cost of left and right chains
- Add cost to multiply the resulting two matrices

Example: Matrix Dimensions

$$p = [10, 100, 5, 50]$$

- A_1: 10×100
- A_2: 100×5
- A_3: 5×50

Compare:

- $((A_1A_2)A_3) : 10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 7500$
- $(A_1(A_2A_3)) : 100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 75000$

$$\Rightarrow \quad m[1][3] = 7500$$

Step 3: Computing the optimal costs

Why Not Recursion?

- Recursive solution based on:

$$m[i][j] = \min_{i \le k < j} (m[i][k] + m[k+1][j] + p_{i-1} \cdot p_k \cdot p_j)$$

- Inefficient: overlaps subproblems $\rightarrow$ exponential time
- Solution: use bottom-up dynamic programming

The DP Tables

We define:

- $m[i][j]$: minimum scalar multiplications for $A_i \dots A_j$
- $s[i][j]$: index k where split yields minimal cost

Base Case:

$$m[i][i] = 0 \quad \text{(a single matrix requires no multiplications)}$$

Bottom-Up DP: MATRIX-CHAIN-ORDER

- Loop over chain lengths $l = 2$ to n
- For each subchain $A_i \dots A_j$ of length l:

$$j = i + l - 1$$

- Try all split points $k \in [i, j-1]$
- Update:

$$m[i][j] = \min_{i \le k < j} (m[i][k] + m[k+1][j] + p_{i-1} \cdot p_k \cdot p_j)$$

Order of Computation

- Fill table by increasing chain length l
- For chain length $l = 2$: fill $m[i][i+1]$
- Then $l = 3$: fill $m[i][i+2]$, etc.
- Ensures all required subproblems $m[i][k], m[k+1][j]$ are filled before use

Visualize diagonals of the DP table being filled from bottom to top

Bottom-Up DP Table

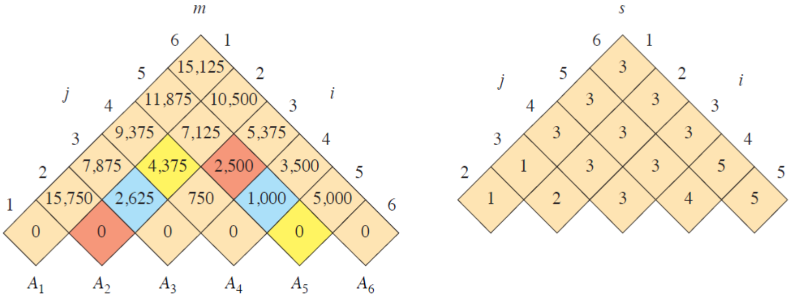

Figure 14.5 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

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

Example: Matrix Dimensions

$$p = [30, 35, 15, 5, 10, 20, 25]$$

- A_1: 30×35
- A_2: 35×15
- A_3: 15×5
- A_4: 5×10
- A_5: 10×20
- A_6: 20×25

To compute $m[2][5]$, try:

- $k = 2$: $\text{cost} = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000$
- $k = 3$: $\text{cost} = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125$
- $k = 4$: $\text{cost} = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375$

$$\Rightarrow m[2][5] = 7125, \quad s[2][5] = 3$$

Bottom-Up DP Algorithm

```
MATRIX-CHAIN-ORDER(p, n)
let m[1 : n, 1 : n] and s[1 : n − 1, 2 : n] be new tables
for i = 1 to n                              // chain length 1
    m[i, i] = 0
for l = 2 to n                              // l is the chain length
    for i = 1 to n − l + 1                  // chain begins at A_i
        j = i + l − 1                       // chain ends at A_j
        m[i, j] = ∞
        for k = i to j − 1                  // try A_{i:k} A_{k+1:j}
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q                 // remember this cost
                s[i, j] = k                 // remember this index
return m and s
```

Step 4: Constructing an optimal solution

- DP table $m[i][j]$: min cost to compute $A_i \dots A_j$
- Split table $s[i][j]$: best split point k for subchain $A_i \dots A_j$
- But: we don't yet know the **full parenthesis structure**

Goal: Print optimal parenthesization using $s[i][j]$

What Does s[i][j] Mean?

- $s[i][j] = k \Rightarrow$ optimal split is between A_k and A_{k+1}
- So:

$$A_i \dots A_j = (A_i \dots A_k) \cdot (A_{k+1} \dots A_j)$$

- To fully parenthesize $A_i \dots A_j$, we:
  - Recursively parenthesize $A_i \dots A_k$
  - Recursively parenthesize $A_{k+1} \dots A_j$

Recursive Construction Algorithm

PRINT-OPTIMAL-PARENS(s, i, j)

if $i == j$
    print "A"$_i$
else print "("
    PRINT-OPTIMAL-PARENS$(s, i, s[i, j])$
    PRINT-OPTIMAL-PARENS$(s, s[i, j] + 1, j)$
    print ")"

Example Output

Suppose the table gives:

- $s[1][6] = 3 \rightarrow$ split as $(A_1 A_2 A_3)(A_4 A_5 A_6)$
- $s[1][3] = 1 \rightarrow$ split as $A_1(A_2 A_3)$
- $s[4][6] = 5 \rightarrow$ split as $(A_4 A_5)A_6$

Then:

$$\text{Output: } ((A_1(A_2 A_3))((A_4 A_5)A_6))$$

Summary of Step 4

- Table $s[i][j]$ stores optimal split positions
- Recursively use $s[i][j]$ to reconstruct full parenthesis structure
- Base case: when $i = j$, print "$\mathrm{A_i}$"
- Final output is the full multiplication order with minimal cost

Complexity

- Time complexity:

$$O(n^3) \quad \text{(3 nested loops: } l, i, k\text{)}$$

- Space complexity:

$$O(n^2) \quad \text{(tables } m \text{ and } s\text{)}$$

- Much more efficient than brute-force exponential time

Complexity of `MATRIX-CHAIN-ORDER`

Time Complexity:

- `for l = 2 to n` // chain length $\to O(n)$
- `for i = 1 to n - l + 1` // start index $\to O(n)$
- `for k = i to j - 1` // split points $\to O(n)$

$$\Rightarrow \text{Total time: } O(n) \cdot O(n) \cdot O(n) = O(n^3)$$

Space Complexity:

- Table $m[i][j]$: stores optimal costs for $A_i \ldots A_j$
- Table $s[i][j]$: stores optimal split points

Each table is of size $\Rightarrow O(n^2)$ space

Question?