

Session 10

Chained Matrix Multiplication

This session explains why the order of multiplying matrices matters. Matrix multiplication is associative, so the final result is the same for any valid parenthesization, but the number of scalar multiplications can change drastically.

Objectives

✕ Do not copy. Read for understanding and the viva

- Understand why matrix-chain order affects multiplication cost.
- Find all possible orders for multiplying five matrices.
- Use dynamic programming to find optimal parenthesization.

Given Dimensions

■ Write in lab record

Matrix	Dimension
A	10 x 4
B	4 x 5
C	5 x 20
D	20 x 2
E	2 x 50

Dimension array:

$$p = [10, 4, 5, 20, 2, 50]$$

Concept

✖ Do not copy. Read for understanding and the viva

Matrix multiplication is associative:

$$1 \quad (A * B) * C = A * (B * C)$$

The result is the same, but the number of scalar multiplications can be very different.

Dynamic Programming Recurrence

✖ Do not copy. Read for understanding and the viva

$$1 \quad m[i][j] = \min(m[i][k] + m[k+1][j] + p[i-1] * p[k] * p[j])$$

where $i \leq k < j$.

Question 1

Problem Statement

■ Write in lab record

List different orders for evaluating the product of A, B, C, D, E matrices.

Explanation / Approach

✖ Do not copy. Read for understanding and the viva

For a fixed sequence of matrices $A \times B \times C \times D \times E$, the order of evaluation refers to how you place parentheses. Due to the associative property of matrix multiplication, all valid parenthesizations yield the same final matrix, but they require vastly different amounts of computational work.

For 5 matrices, there are exactly **14 valid parenthesization orders** (given by the 4th Catalan number, $C_4 = 14$). Below is the complete list, along with the **scalar multiplication cost** for each,

which highlights why evaluation order matters in practice.

#	Evaluation Order (Parenthesization)	Scalar Multiplication Cost
1	$((((A \cdot B) \cdot C) \cdot D) \cdot E)$	"2,600"
2	$((((A \cdot (B \cdot C)) \cdot D) \cdot E)$	"2,600"
3	$((A \cdot ((B \cdot C) \cdot D)) \cdot E)$	"1,640"
4	$((A \cdot (B \cdot (C \cdot D))) \cdot E)$	"1,320"
5	$((((A \cdot B) \cdot (C \cdot D)) \cdot E)$	"1,500"
6	$((A \cdot B) \cdot ((C \cdot D) \cdot E))$	"3,400"
7	$((A \cdot B) \cdot (C \cdot (D \cdot E)))$	"9,700"
8	$((A \cdot (B \cdot C)) \cdot (D \cdot E))$	"13,200"
9	$(A \cdot (((B \cdot C) \cdot D) \cdot E))$	"2,960"
10	$(A \cdot ((B \cdot (C \cdot D)) \cdot E))$	"2,640"
11	$(A \cdot ((B \cdot C) \cdot (D \cdot E)))$	"8,400"
12	$(A \cdot (B \cdot ((C \cdot D) \cdot E)))$	"3,700"
13	$(A \cdot (B \cdot (C \cdot (D \cdot E))))$	"10,000"
14	$((((A \cdot B) \cdot C) \cdot (D \cdot E))$	"13,200"

- **Cost Calculation:** Multiplying an $m \times n$ matrix by an $n \times p$ matrix requires $m \cdot n \cdot p$ scalar multiplications. The total cost is the sum of costs at each multiplication step.
- **Performance Gap:** The worst valid orders (#8 and #14) require **10×** more operations than the optimal order (#4).
- **Why #4 is optimal:** It computes the narrowest intermediate matrices first ($C \cdot D \rightarrow 5 \times 2$, then $B \cdot (C \cdot D) \rightarrow 4 \times 2$, then $A \cdot \dots \rightarrow 10 \times 2$), keeping the inner dimension small before multiplying by the large 50 column of E .

Question 2

Problem Statement

■ Write in lab record

Find the optimal order for evaluating $A * B * C * D * E$ using the given dimensions.

Explanation / Approach

✗ Do not copy. Read for understanding and the viva

The optimal parenthesization is: $((A \cdot (B \cdot (C \cdot D))) \cdot E)$

Minimum scalar multiplication cost: 1,320

Step	Operation	Input Dimensions	Output Dimensions	Cost (m×n×p)
1	$C \cdot D$	$5 \times 20 \times 20 \times 2$	5×2	$5 \times 20 \times 2 = 200$
2	$B \cdot (CD)$	$4 \times 5 \times 5 \times 2$	4×2	$4 \times 5 \times 2 = 40$
3	$A \cdot (B(CD))$	$10 \times 4 \times 4 \times 2$	10×2	$10 \times 4 \times 2 = 80$
4	$(A(B(CD))) \cdot E$	$10 \times 2 \times 2 \times 50$	10×50	$10 \times 2 \times 50 = 1,000$
Total				1,320

Why This Order is Optimal

✗ Do not copy. Read for understanding and the viva

- Matrix chain multiplication is all about controlling the size of intermediate matrices. $C \cdot D$ collapses the large 20 inner dimension early, producing a skinny 5×2 matrix.
- Subsequent multiplications with B and A keep the intermediate column dimension at 2, avoiding expensive operations with the 50 columns of E until the very last step.
- Multiplying the final 10×2 intermediate by E (2×50) is cheap because the shared dimension is only 2.

Any other parenthesization forces a larger inner dimension to be multiplied earlier, drastically increasing the total operation count (the worst valid order requires **13,200** operations, exactly **10×** more).

Question 3

Problem Statement

■ Write in lab record

Implement chained matrix multiplication and print the optimal parentheses. Study the performance on different problem instances.

Answer

■ Write in lab record

Aspect	Complexity / Behavior
Time Complexity	$O(n^3)$ due to 3 nested loops (chain_len, i, k)
Space Complexity	$O(n^2)$ for DP tables m and s
Empirical Scaling	Matches $O(n^3)$: doubling n roughly increases runtime by 8× (e.g., 50→100: ~1.3ms → ~9.8ms)
Pattern Impact on Cost	Dimension distribution drastically changes optimal cost, but not DP runtime. Increasing/Decreasing chains have highly structured optimal splits, while random chains require full DP evaluation.
Practical Limit	Python's pure loops cap out around $n \approx 300-400$ (~1-2 sec). For $n > 1000$, use C/C++ extensions or iterative reconstruction to avoid recursion limits.

Implementation

■ Write in lab record



Lab record: write one language only. Pick yours once and every page opens on it; the other tabs are the same solution for comparison.

Python

matrix-chain-multiplication.py

```

1 import random
2 import time
3 from typing import List, Tuple
4
5
6 def matrix_chain_order(dims: List[int]) → Tuple[List[List[int]], List[List[
7     """
8     Computes minimum multiplication cost and optimal split points using DP.
9
10    Args:
11        dims: List of dimensions where matrix i has shape dims[i] x dims[i+1]
12              Length must be n + 1 for n matrices.
13
14    Returns:
15        m: DP table for minimum costs
16        s: DP table for optimal split indices
17    """
18    n = len(dims) - 1 # Number of matrices
19    m = [[0] * n for _ in range(n)]
20    s = [[0] * n for _ in range(n)]
21
22    # l = chain length (subproblem size)
23    for l in range(2, n + 1):
24        # i = left boundary of subchain
25        for i in range(n - l + 1):
26            j = i + l - 1 # j = right boundary
27            m[i][j] = float("inf")
28
29            # k = split point
30            for k in range(i, j):
31                # Cost = left + right + merge
32                cost = m[i][k] + m[k + 1][j] + dims[i] * dims[k + 1] * dims[
33                if cost < m[i][j]:
34                    m[i][j] = cost
35                    s[i][j] = k
36
37    return m, s
38
39
40 def construct_parens(s: List[List[int]], i: int, j: int) → str:
41     """Reconstructs optimal parenthesization string from split table."""
42     if i == j:
43         return f"M{i + 1}"

```

```

44     k = s[i][j]
45     return f"({construct_parens(s, i, k)} · {construct_parens(s, k + 1, j)})"
46
47
48 def run_performance_study():
49     """Benchmarks algorithm across varying chain lengths."""
50     print(f'{'N':<5} | {'Pattern':<12} | {'Cost':<10} | {'Time(ms)':<8}")
51     print("-" * 45)
52
53     test_cases = [
54         (5, "Fixed", [10, 4, 5, 20, 2, 50]),
55         (20, "Uniform", [random.randint(5, 100) for _ in range(21)]),
56         (50, "Uniform", [random.randint(5, 100) for _ in range(51)]),
57         (100, "Uniform", [random.randint(5, 100) for _ in range(101)]),
58     ]
59
60     for n, pattern, dims in test_cases:
61         start = time.perf_counter()
62         m, s = matrix_chain_order(dims)
63         end = time.perf_counter()
64         cost = m[0][n - 1]
65         parens = construct_parens(s, 0, n - 1)
66
67         print(f'{'n':<5} | {'pattern':<12} | {'cost':<10} | {(end - start) * 1000:}
68
69         if n == 5:
70             print(f'\n🔍 Verified: {parens} (Cost: {cost})\n')
71
72
73 if __name__ == "__main__":
74     run_performance_study()

```

Practical Limits

- Overhead of list-of-lists and interpreter slows $n > 300$. Use numpy arrays or lru_cache for memoization if recursion is preferred.

Sample Output

■ Write in lab record

```

1 N      | Pattern      | Cost      | Time(ms)
2 -----
3 5      | Fixed        | 1320      | 0.012
4
5 🔍 Verified: ((M1 · (M2 · (M3 · M4))) · M5) (Cost: 1320)
6
7 20     | Uniform      | 241405    | 0.176
8 50     | Uniform      | 929424    | 2.231
9 100    | Uniform      | 1535972   | 16.081

```

SH

Optimizations

Knuth Optimization: If the split table satisfies monotonicity ($s[i][j-1] \leq s[i][j] \leq s[i+1][j]$), time can be reduced to $O(n^2)$. This holds for many real-world dimension sequences but requires proof per instance.

[✎ Edit this page](#)

< Previous
Session 9

Next >
Session 11