

# Optimal Parenthesization of a Chain of Matrix Multiplications

## 1 Problem Statement

**Chain of Matrices** We have  $n$  matrices  $M_1, M_2, \dots, M_n$ , with dimensions stored in an integer array  $d[0..n]$ . Thus:

$$M_i \text{ has dimension } d[i-1] \times d[i], \quad \text{for } i = 1, \dots, n.$$

We wish to compute the product:

$$M_1 \times M_2 \times \dots \times M_n.$$

Matrix multiplication is associative: the product  $M_1 \times M_2 \times \dots \times M_n$  is well-defined regardless of how we place parentheses. *However*, the total number of scalar multiplications required *does* depend on the chosen parenthesization. The core problem is to find a way of inserting parentheses (i.e., deciding the order of multiplications) to minimize the total cost of forming the product.

**Cost Function and Objective** Multiplying a  $p \times q$  matrix by a  $q \times r$  matrix costs  $p \cdot q \cdot r$  scalar operations. For a chain of  $n$  matrices, we sum these costs according to the chosen parenthesization. Formally, if  $C(P)$  denotes the total scalar multiplications for a parenthesization  $P$ , then we want:

$$c^* = \min_{P \in \mathcal{P}} \text{Cost}(P) \quad \text{and} \quad P^* = \operatorname{argmin}_{P \in \mathcal{P}} \text{Cost}(P)$$

where  $\mathcal{P}$  is the set of all valid parenthesizations of  $M_1 \dots M_n$ .

**All Possible Parenthesizations** A *parenthesization* of  $M_1 \dots M_n$  specifies the order in which these matrices are multiplied. For example, with  $n = 4$ , some ways to parenthesize include:

$$(M_1(M_2(M_3M_4))), \quad ((M_1M_2)(M_3M_4)), \quad (M_1((M_2M_3)M_4)), \dots$$

Each way can incur a different total cost, despite producing the same final matrix.

The number of ways to fully parenthesize a chain of  $n$  matrices is given by the  $(n-1)$ -th *Catalan number*. The sequence of Catalan numbers  $\{C_0, C_1, C_2, \dots\}$  is typically defined by

$$C_0 = 1, \quad \text{and for } n \geq 1, \quad C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k}.$$

In other words, the Catalan numbers obey the recursive equation:

$$C_n = C_0C_{n-1} + C_1C_{n-2} + \dots + C_{n-1}C_0,$$

reflecting the idea that to form a full binary tree on  $n+1$  leaves, you can choose a split for the root in all possible ways and combine the corresponding left and right subtrees.

An alternative *closed-form* expression is:

$$C_n = \frac{1}{n+1} \binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}.$$

As  $n$  grows large,  $C_n$  grows asymptotically like

$$C_n \sim \frac{4^n}{n^{3/2} \sqrt{\pi}}.$$

Thus, the number of distinct ways to parenthesize  $n$  matrices, or to build a full binary tree on  $n$  leaves, increases *exponentially* with  $n$ . Concretely, for small values, we have:

$$C_1 = 1, \quad C_2 = 2, \quad C_3 = 5, \quad C_4 = 14, \quad C_5 = 42, \quad \dots$$

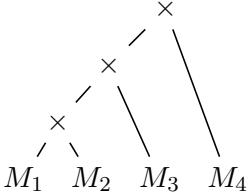
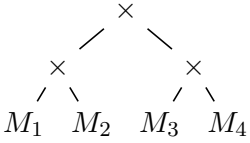
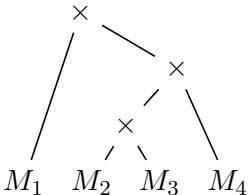
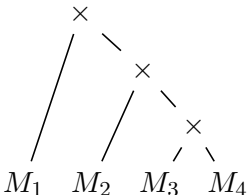
This exponential growth in the number of parenthesizations underlies the impracticality of naive exhaustive search (backtracking) for large  $n$ .

**Correspondence with Full Binary Trees** Every parenthesization corresponds to a *full binary tree* whose leaves are the matrices  $M_1, M_2, \dots, M_n$ . Each internal node represents a multiplication of two sub-products. Consequently, enumerating all parenthesizations is equivalent to enumerating all full binary trees on  $n$  leaves.

**Example:** Consider four matrices  $M_1, M_2, M_3, M_4$  with dimensions:

$$d = [10, 100, 5, 50, 20].$$

We show all five distinct parenthesizations in the following table, with their costs and representative binary trees.

Parenthesization	Cost	Binary Tree
$((M_1 M_2) M_3) M_4$	$(10 \times 100 \times 5) + (10 \times 5 \times 50) + (10 \times 50 \times 20) = 17,500$	
$((M_1 M_2) (M_3 M_4))$	$(10 \times 100 \times 5) + (5 \times 50 \times 20) + (10 \times 5 \times 20) = 11,000$	
$(M_1 ((M_2 M_3) M_4))$	$(100 \times 5 \times 50) + (100 \times 50 \times 20) + (10 \times 100 \times 20) = 145,000$	
$(M_1 (M_2 (M_3 M_4)))$	$(5 \times 50 \times 20) + (100 \times 5 \times 20) + (10 \times 100 \times 20) = 35,000$	

Here, the **optimal** parenthesization is  $P^* = ((M_1 M_2) (M_3 M_4))$ , with cost  $c^* = 11,000$  scalar multiplications.

## 2 Complete Enumeration via Stack-Based Backtracking

A direct way to systematically *enumerate every* parenthesization is a backtracking approach using a stack, with two main operations:

- **SHIFT**: Push the next matrix  $M_i$  onto the stack (if it exists).
- **REDUCE**: Pop the top two sub-products, multiply them into a new sub-product, and push that result back onto the stack.

Each distinct sequence of **SHIFT** and **REDUCE** steps that leaves a single item on the stack at the end corresponds to one valid parenthesization of the chain.

**Backtracking Pseudocode** Algorithm 1 outlines the idea. We keep:

- A **stack** to accumulate partial products,
- A pointer  $i$  for the next matrix to possibly **SHIFT**,
- Global variables **bestCost** and **bestTree** to record the best solution found.

---

### Algorithm 1 SHIFTREDUCEOPTIMAL( $d[0..n]$ )

---

```

1: bestCost  $\leftarrow +\infty$ , bestTree  $\leftarrow$  None
2: stack  $\leftarrow \emptyset$ 
3: function BACKTRACK( $i$ )                                 $\triangleright i = \text{index of next matrix to SHIFT, } 1 \leq i \leq n$ 
4:    $\triangleright$  Case 1: SHIFT if  $i \leq n$ 
5:   if  $i \leq n$  then
6:     push Leaf( $i$ ) on stack                                 $\triangleright$  Matrix  $M_i$  has cost=0
7:     BACKTRACK( $i+1$ )
8:     pop stack                                              $\triangleright$  undo SHIFT
9:   end if
10:   $\triangleright$  Case 2: REDUCE if at least 2 items on stack
11:  if stack.size()  $\geq 2$  then
12:     $B \leftarrow \text{stack.pop}()$ 
13:     $A \leftarrow \text{stack.pop}()$ 
14:     $C \leftarrow \text{Combine}(A, B)$ 
15:    push  $C$  on stack
16:    BACKTRACK( $i$ )
17:    pop stack                                               $\triangleright$  undo REDUCE
18:    push  $A$ , push  $B$                                         $\triangleright$  restore original top items
19:  end if
20:   $\triangleright$  Case 3: Completed parse if  $i > n$  and stack.size = 1
21:  if  $i > n$  and stack.size() = 1 then
22:    candidate  $\leftarrow \text{stack.top}()$ 
23:    if candidate.cost < bestCost then
24:      bestCost  $\leftarrow \text{candidate.cost}$ 
25:      bestTree  $\leftarrow \text{candidate.repr}$ 
26:    end if
27:  end if
28: end function
29: BACKTRACK(1)
30: return (bestCost, bestTree)

```

---

**Complexity:** Since the number of ways to parenthesize  $n$  matrices is the  $(n-1)$ -th Catalan number, which is exponential in  $n$ . This method becomes impractical for large  $n$ , but it is conceptually simple and helpful for verifying correctness with small instances.

**Small Example for  $n = 3$**  Let  $M_1 : 2 \times 3$ ,  $M_2 : 3 \times 4$ ,  $M_3 : 4 \times 5$ . Possible SHIFT/REDUCE sequences:

SHIFT  $M_1 \rightarrow$  SHIFT  $M_2 \rightarrow$  REDUCE  $(M_1 M_2) \rightarrow$  SHIFT  $M_3 \rightarrow$  REDUCE  $((M_1 M_2) M_3)$ ,  
 SHIFT  $M_1 \rightarrow$  SHIFT  $M_2 \rightarrow$  SHIFT  $M_3 \rightarrow$  REDUCE  $(M_2 M_3) \rightarrow$  REDUCE  $(M_1 (M_2 M_3))$ .

These yield  $((M_1 M_2) M_3)$  or  $(M_1 (M_2 M_3))$ . Counting their costs lets us pick the cheaper.

**Implementation and Experiment Pipeline** Download the provided code form the course Moodle, in which we provide:

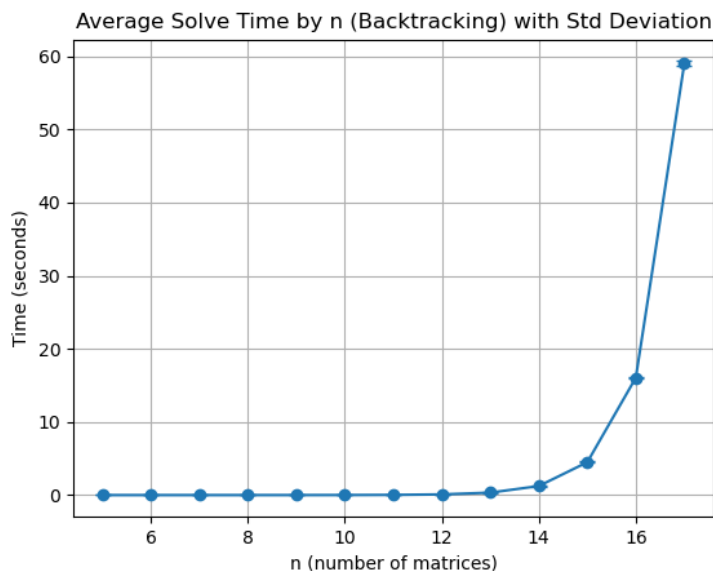
1. **A generator** to produce random matrix-chain instances,
2. **A solver** implementing the stack-based backtracking algorithm,
3. **A plotting script** in Python (`matplotlib`) to visualize performance,
4. **Instructions** in a `README.md` file.

In short:

- `generate_instances` creates an `instances.txt` with multiple sizes and random dimensions.
- `solve_backtracking` read `instances.txt`, solve each, and record results.
- `compare_algos_plot.py` parse result files and generate time-vs-size graphs.

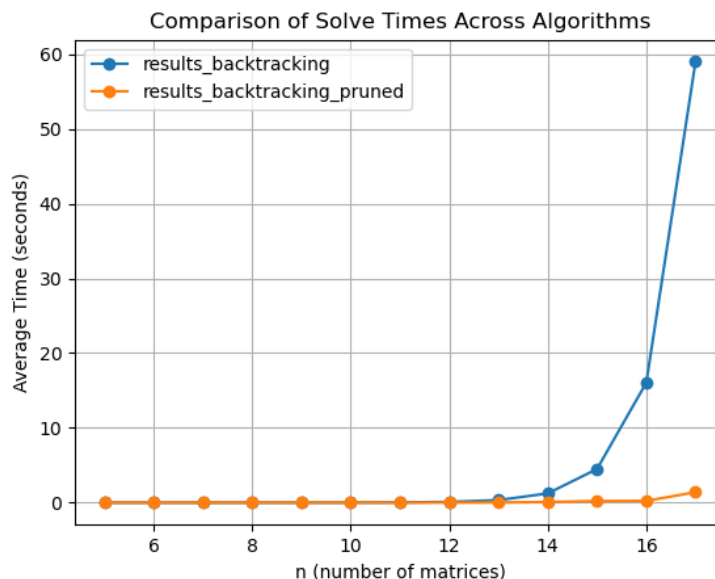
## 2.1 Exercises

**Exponential Growth** Run the backtracking solution for different values of  $n = 2, 3, 4, 5, \dots$  and measure total enumeration time. Observe how rapidly it grows. Make plots to illustrate your findings and familiarize yourself with the code. You should be able to reproduce the following graph.



## Pruning the Search Space

- **Idea:** If a partial cost already exceeds the best known cost so far, prune that branch.
- **Implementation:** Modify the backtracking code to check partial costs. If  $\text{partial} \geq \text{bestCost}$ , skip deeper exploration.
- **Evaluation:** Compare pruned backtracking vs. naive backtracking on the same instances. Count how many partial solutions were pruned. Observe the difference in runtime for bigger  $n$ . You should obtain results similar to the following figure:



What is the largest value for  $n$  for which the running time remains “reasonable”.

## Dimension Patterns

- What if all matrices are  $10 \times 10$  (square)? Does parenthesization matter?
- What about strictly increasing or decreasing dimensions (e.g.  $10 \times 20, 20 \times 30, \dots$ )?

## 3 Dynamic Programming (DP) Approach

Backtracking enumerates all parenthesizations (exponential in  $n$ ). A more efficient solution in  $\mathcal{O}(n^3)$  time uses dynamic programming.

**Defining the Subproblems** Let  $\text{dp}[i, j]$  denote the *minimum cost* (number of scalar multiplications) to multiply the sub-chain  $M_i \times M_{i+1} \times \dots \times M_j$ , where  $1 \leq i \leq j \leq n$ . If  $i = j$ , there is only a single matrix, so  $\text{dp}[i, i] = 0$  (no cost is needed to form one matrix).

**Recurrence Relation** When  $i < j$ , we can split the product  $(M_i \dots M_j)$  at any intermediate position  $k$  where  $i \leq k < j$ . We thus form two sub-chains:

$$\underbrace{(M_i \dots M_k)}_{\text{left sub-product}} \quad \text{and} \quad \underbrace{(M_{k+1} \dots M_j)}_{\text{right sub-product}},$$

and we compute the cost of multiplying these two sub-products once they are formed. The dimension of the left product is  $d_{i-1} \times d_k$ , and the dimension of the right product is  $d_k \times d_j$ . Therefore, the cost of multiplying these two resulting matrices is  $d_{i-1} \cdot d_k \cdot d_j$ .

Hence, the total cost for a particular split  $k$  is:

$$\underbrace{\text{dp}[i, k]}_{\text{cost of left}} + \underbrace{\text{dp}[k+1, j]}_{\text{cost of right}} + \underbrace{d_{i-1} \times d_k \times d_j}_{\text{cost to combine}}.$$

Taking the minimum over all possible  $k$  yields:

$$\text{dp}[i, j] = \min_{i \leq k < j} \left[ \text{dp}[i, k] + \text{dp}[k+1, j] + (d_{i-1} \cdot d_k \cdot d_j) \right].$$

Storing `split[i, j]` records which  $k$  gave the best split.

**Order of Filling** We compute  $\text{dp}[i, i] = 0$  for all  $i$ . Then for sub-chains of length  $\ell = 2$  up to  $n$ :

- For each  $i$  such that  $j = i + \ell - 1 \leq n$ , compute  $\text{dp}[i, j]$  by trying all possible splits  $k \in [i..j-1]$ .
- Record the split  $k$  that yields the minimum cost in `split[i, j]`.

**Example** With  $n = 4$ ,  $d = [10, 100, 5, 50, 20]$ , we get:

		$j$			
		1	2	3	4
$i$	1	0 (−)	5000 (1)	7500 (2)	11000 (2)
	2		0 (−)	25000 (2)	15000 (2)
	3			0 (−)	5000 (3)
	4				0 (−)

Each cell shows  $\text{dp}[i, j]$  and `split[i, j]`. The optimal parenthesization is  $((M_1 M_2)(M_3 M_4))$  with cost 11,000.

**Reconstructing the Parenthesization.** From `split[1, 4] = 2`, we know to split  $(M_1 \cdots M_4)$  between  $M_2$  and  $M_3$ , i.e.  $(M_1 M_2)$  and  $(M_3 M_4)$ .

- For the left part (1, 2), `split[1, 2] = 1` means it is simply  $(M_1 M_2)$ .
- For the right part (3, 4), `split[3, 4] = 3` means it is simply  $(M_3 M_4)$ .

Hence the optimal parenthesization is

$$((M_1 M_2)(M_3 M_4))$$

with a total cost of 11,000 scalar multiplications. This matches our earlier conclusion via enumerating all parenthesizations.

### 3.1 DP Implementation and Comparison

- Write `solve_dp.c` to read the same `instances.txt` and compute `dp` and `split`.
- Reconstruct the best parenthesization from `split`.
- Compare run times and costs with naive/pruned backtracking. Show that for larger  $n$ , DP is significantly faster, yet yields the same optimal cost.

## 4 Greedy Heuristics

Sometimes, for very large  $n$ , even  $\mathcal{O}(n^3)$  might be too large, or we might want a near-optimal solution quickly. Two simple greedy approaches:

### Greedy 1: Multiply the Smallest Adjacent Pair First

- **Heuristic:** At each step, find the adjacent pair in the current chain that yields the smallest immediate multiplication cost  $p \times q \times r$ . Multiply them, shrink the chain by one. Repeat until only one sub-product remains.
- **Caveat:** This local choice can be short-sighted.

### Greedy 2: Divide-and-Conquer by Minimizing Border Cost

- **Heuristic:** At a sub-chain  $(i, j)$ , pick the split  $k$  that minimizes  $d_{i-1} \times d_k \times d_j$  (the immediate border cost). Then recursively do the same on the two sub-chains.
- **Caveat:** Ignores internal sub-chain costs, so not always near-optimal. However, it is easy to implement top-down.

### 4.1 Exercise

1. Implement both heuristics in, e.g., `solve_greedy1.c` and `solve_greedy2.c`.
2. Compare their running times and final costs against DP, backtracking, etc.
3. Propose and implement your own new heuristic. Do you find a scenario where your heuristic outperforms these two in cost or speed?
4. **Optional Rank Gap:** For small  $n$ , extend your backtracking to *enumerate all parenthesizations* and sort them by cost (lowest to highest). Then for each heuristic solution:
  - Determine the *rank* of its cost in that sorted list (where rank 1 = optimal).
  - Define the *rank gap* as how many positions away from 1 the heuristic solution stands. For instance, if rank is 5, the gap is  $5 - 1 = 4$ .
  - Report the average rank gap or its percentage relative to the total number of solutions.