ALGORITHMS AND DATA STRUCTURES · LECTURE 17

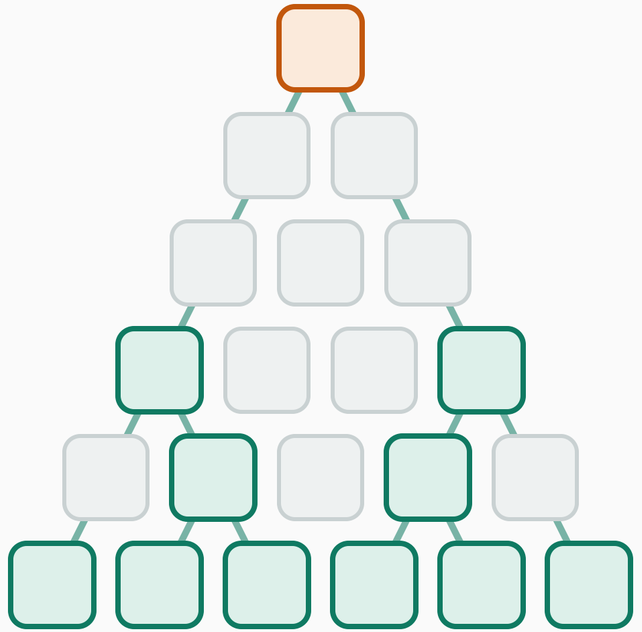
Dynamic programming

Chain matrix multiplication

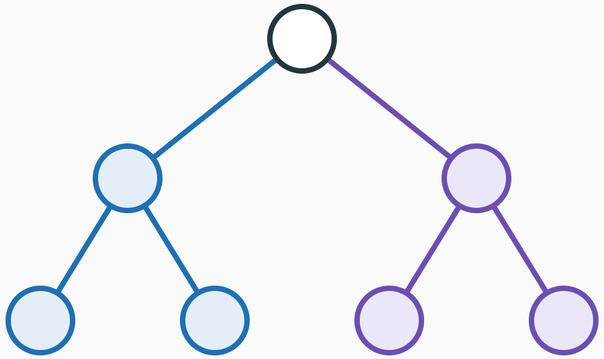
Heng Guo

School of Informatics, University of Edinburgh

Monday 16 November 2026 · 16:10–17:00

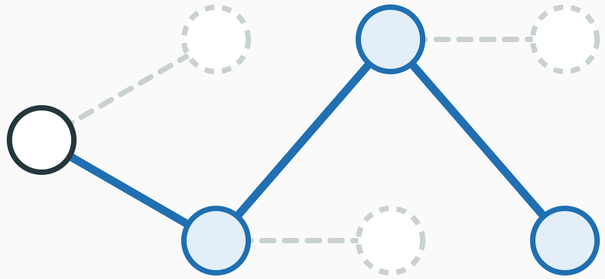


A new technique: dynamic programming



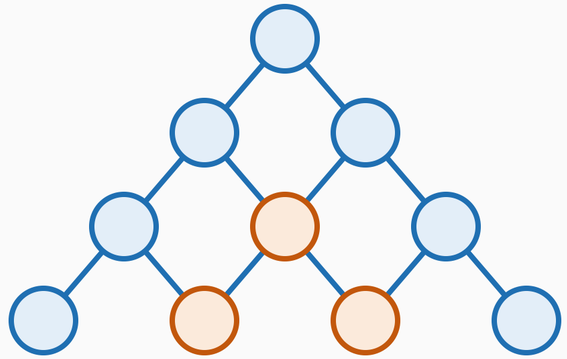
Divide and conquer

independent subproblems



Greedy

one choice at a time, never undone



Dynamic programming

overlapping subproblems,
each solved once

Today's example: given matrices $A_1, \ldots, A_n$, in which order should we multiply them?

The problem

In which order should we multiply a chain of matrices?

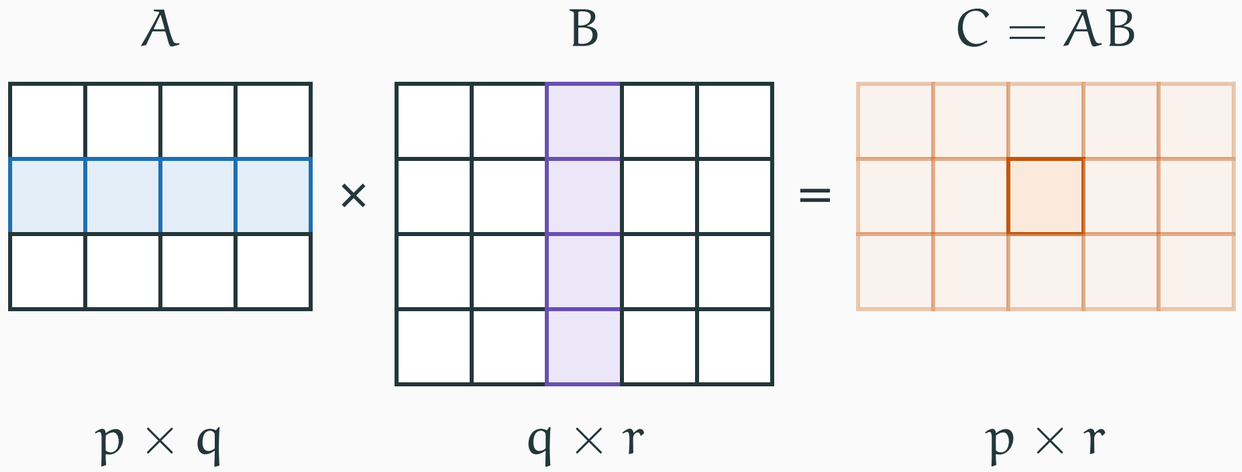
The cost of one product

The entries of $C = AB$ are

$$c_{ij} = \sum_{k=1}^{q} a_{ik}\, b_{kj}.$$

Each entry takes q multiplications.

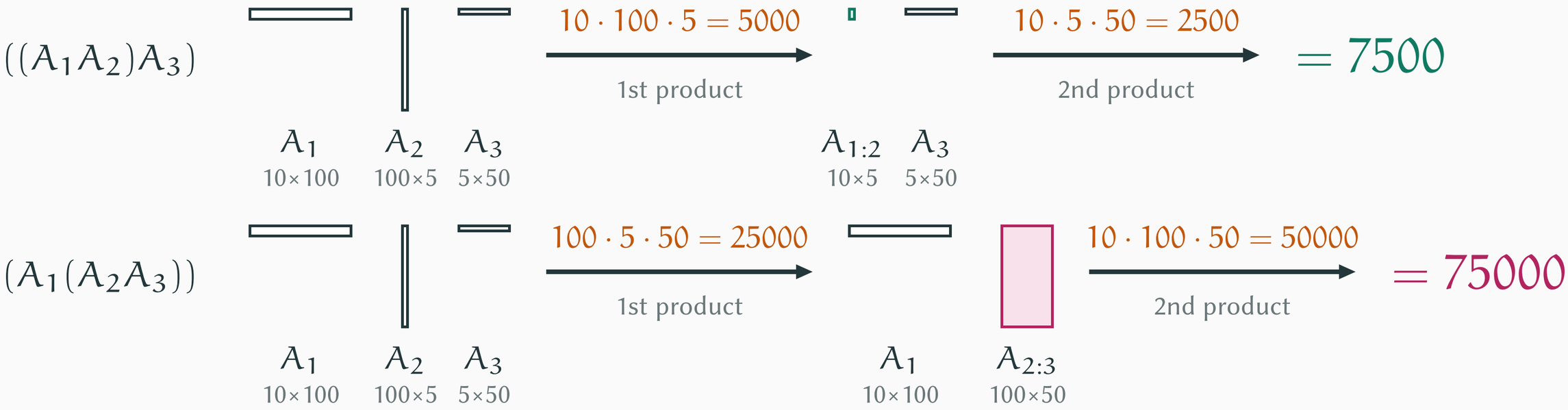
There are $pr = 15$ entries:
$15 \cdot 4 = 60$ multiplications in the picture.



Cost model. Multiplying a $p \times q$ by a $q \times r$ matrix costs pqr scalar multiplications.

The order matters

? *Which order of $A_1A_2A_3$ is cheaper?*



The rectangles are drawn to scale: the area is the number of entries.

Same result, ten times the work: **the order matters**.

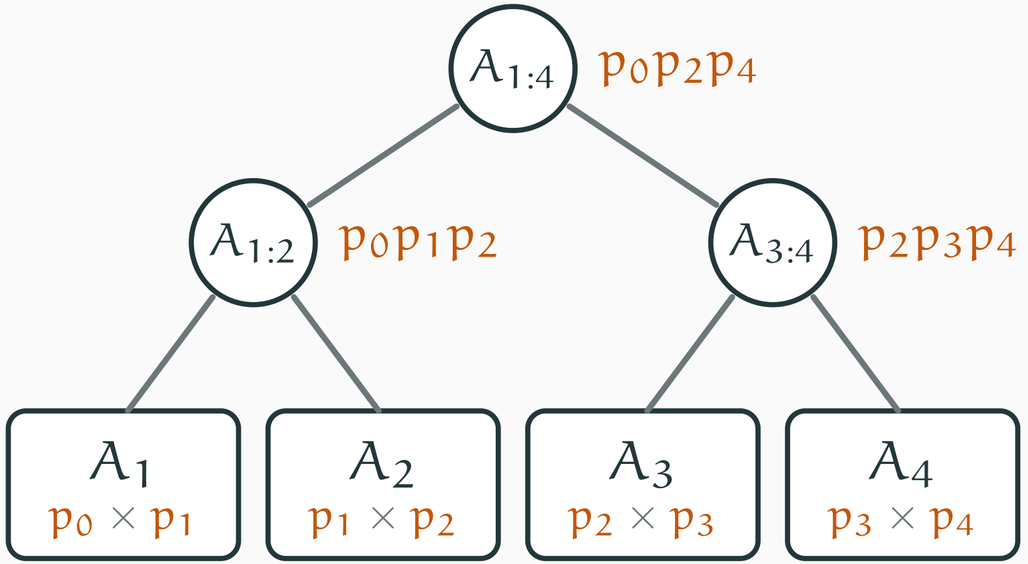
The problem

> **Definition.** A product of matrices is **fully parenthesised** if it is a single matrix, or the product of two fully parenthesised products, in parentheses.

Input: $p_0, p_1, \ldots, p_n$,
where A_i is a $p_{i-1} \times p_i$ matrix.

Output: a full parenthesisation
of $A_1 A_2 \cdots A_n$ that needs
the fewest scalar multiplications.

We only choose the order.
Nothing is multiplied yet.



Brute force: how many parenthesisations?

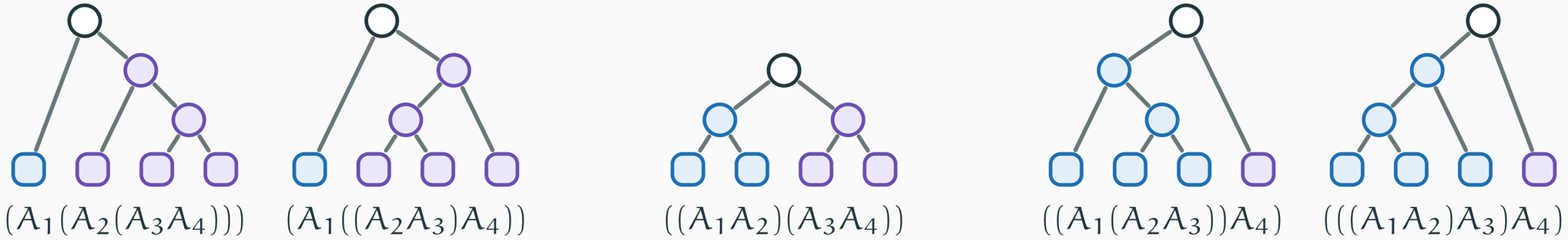
? *How many full parenthesisations has a chain of n matrices?*

$$P(1) = 1, \text{ and } P(n) = \sum_{k=1}^{n-1} P(k)\,P(n-k) \text{ for } n \geqslant 2.$$

The last product splits the chain after matrix k.

The two sides are independent.

$P(4) = 5$ $\quad k = 1 :\ P(1)\,P(3) = 2$ $\quad k = 2 :\ P(2)\,P(2) = 1$ $\quad k = 3 :\ P(3)\,P(1) = 2$



$P(n)$: 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, ..., $P(20)$ = 1 767 263 190.

$P(n)$ is the Catalan number $C_{n-1} = \frac{1}{n}\binom{2n-2}{n-1}$, and $P(n) \geqslant 2^{n-2}$:
for $n \geqslant 3$ the terms $k = 1$ and $k = n-1$ give $P(n) \geqslant 2P(n-1)$.

Optimal substructure

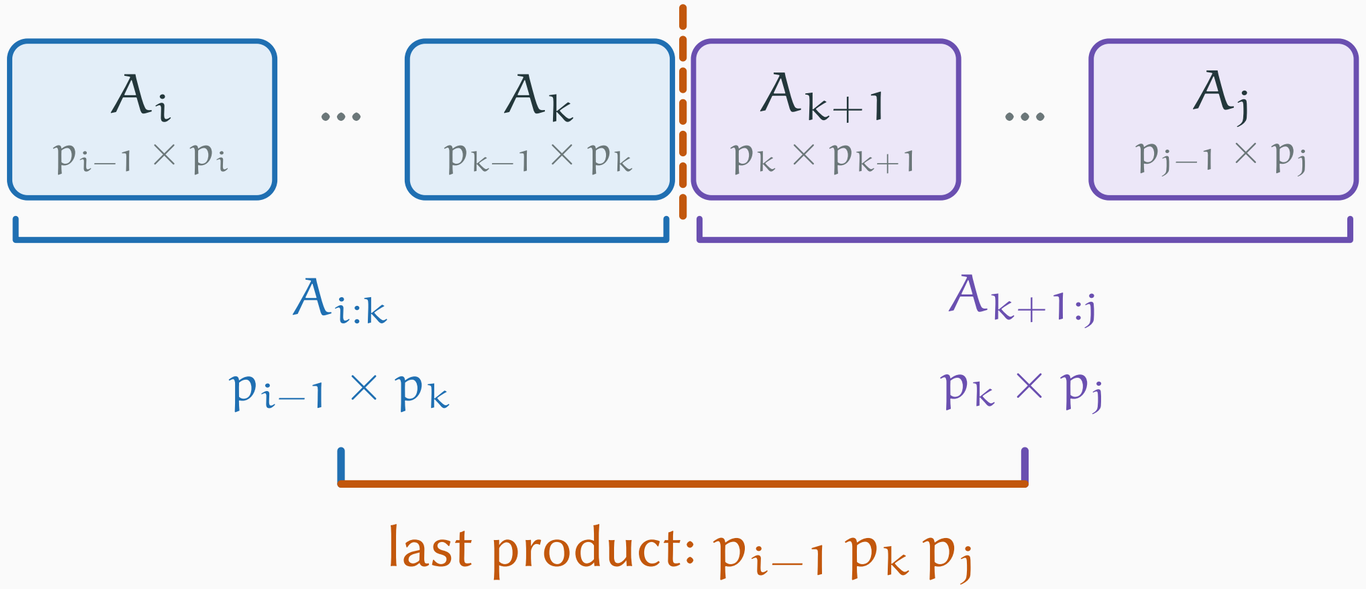
Is an optimal order made of optimal orders of shorter chains?

Where is the last product?

Let $A_{i:j} = A_i A_{i+1} \cdots A_j$:
a $p_{i-1} \times p_j$ matrix, however
it is parenthesised.

In a parenthesisation P of $A_{i:j}$ the **last product** multiplies $A_{i:k}$ by $A_{k+1:j}$, for some k with $i \leqslant k < j$.
Thus $P = (P_L\ P_R)$.

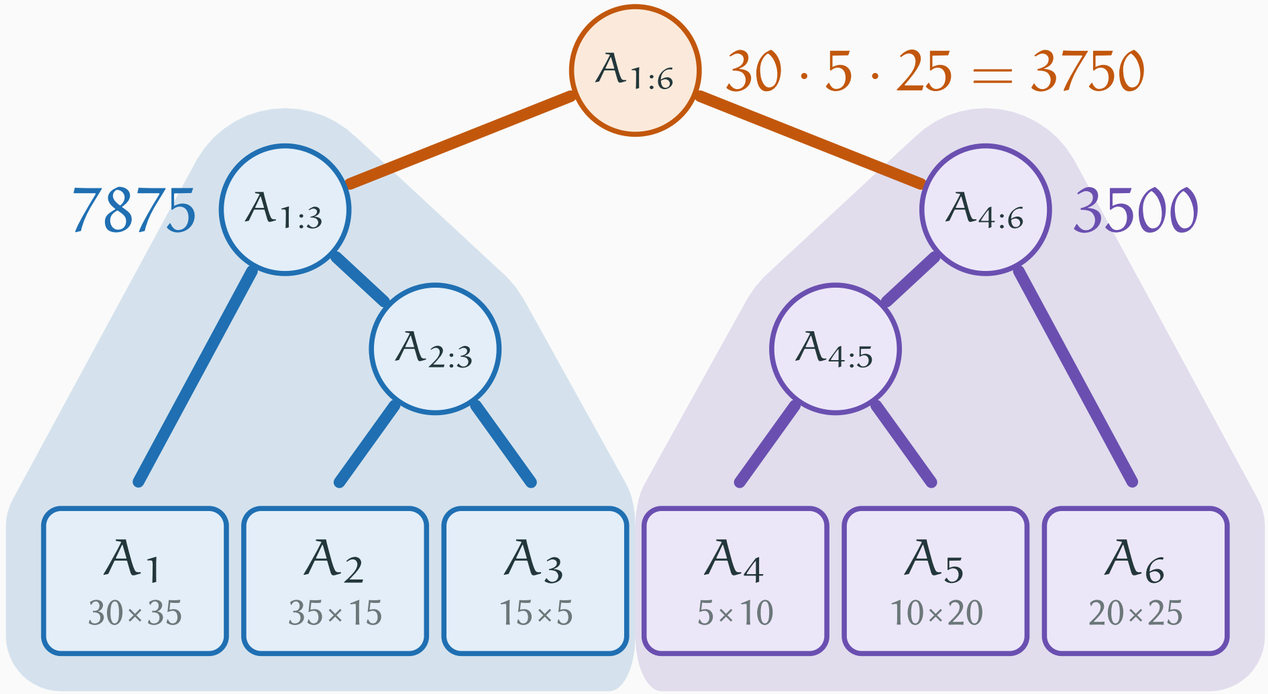
Its cost is $p_{i-1}\, p_k\, p_j$,
whatever happened inside P_L and P_R.



$$\text{cost}(P) = \text{cost}(P_L) + \text{cost}(P_R) + p_{i-1}\, p_k\, p_j$$

Optimal substructure

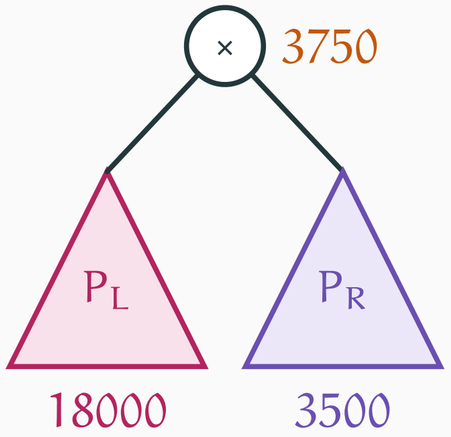
Claim. Let $P = (P_L\ P_R)$ be an **optimal** parenthesisation of $A_{i:j}$ with its last product at k. Then P_L is optimal for $A_{i:k}$, and P_R is optimal for $A_{k+1:j}$.



$$15125 = 7875 + 3500 + 30 \cdot 5 \cdot 25$$

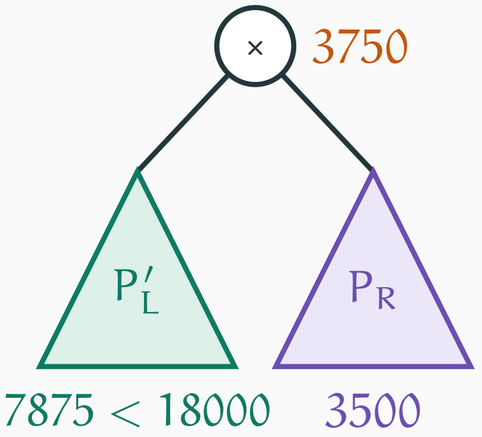
Proof: cut and paste

1. $\text{cost}(P) = \text{cost}(P_L) + \text{cost}(P_R) + p_{i-1}\, p_k\, p_j$.
2. Suppose P_L is not optimal for $A_{i:k}$:
   some P'_L has $\text{cost}(P'_L) < \text{cost}(P_L)$.
3. Cut out P_L, paste in P'_L:
   $P' = (P'_L\ P_R)$ also parenthesises $A_{i:j}$.
4. $\text{cost}(P') = \text{cost}(P'_L) + \text{cost}(P_R) + p_{i-1}p_k p_j$
   $< \text{cost}(P)$: the last product costs the same.
5. Hence P was not optimal: contradiction.
   The same for P_R. □



$\text{cost}(P) = 25250$

P is not optimal



$\text{cost}(P') = 15125$

The recurrence

How does the best cost depend on the best costs of shorter chains?

The recurrence

Let $m[i, j]$ be the **fewest** scalar multiplications that compute $A_{i:j}$.
We want $m[1, n]$.

A single matrix needs no product: $m[i, i] = 0$.

$$m[i, j] = \min_{i \leqslant k < j} \left\{ m[i, k] + m[k+1, j] + p_{i-1}\, p_k\, p_j \right\} \qquad \text{for } i < j.$$

Record the best split: $s[i, j]$ is a k attaining the minimum.

Example: $p = \langle 10, 100, 5, 50 \rangle$,
so $m[1, 2] = 5000$ and $m[2, 3] = 25000$.
Then $m[1, 3]$ is the smaller of two sums:

k	$m[1, k]$	$m[k+1, 3]$	$p_0 p_k p_3$	sum
1	0	25000	10·100·50 = 50000	75000
2	**5000**	**0**	**10·5·50 = 2500**	**7500 ✓**

Recursion straight from the recurrence

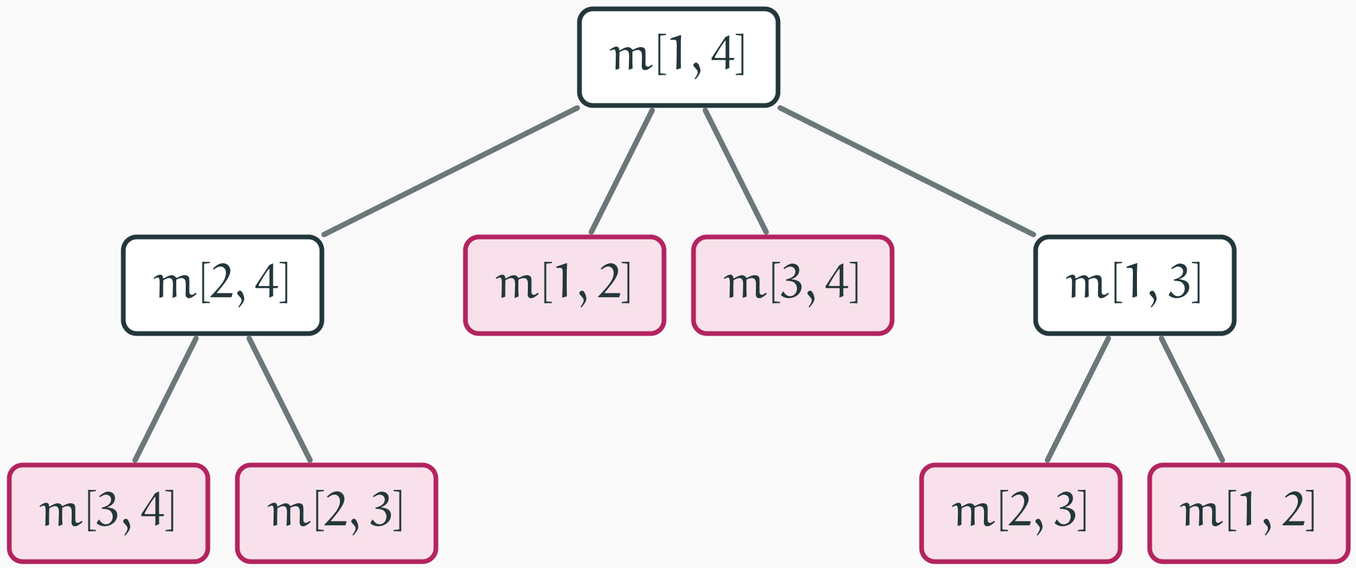

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1 if i = j
2     return 0
3 m[i, j] ← ∞
4 for k = i to j − 1
5     q ← RECURSIVE-MATRIX-CHAIN(p, i, k)
            + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
            + p_{i−1} p_k p_j
6     if q < m[i, j]
7         m[i, j] ← q
8 return m[i, j]
```

Number of calls for n matrices:

$$
\begin{aligned}
C(1) &= 1, \\
C(n) &= 1 + \sum_{k=1}^{n-1} \big(C(k) + C(n-k)\big) && \text{for } n \geqslant 2 \\
&= 1 + 2 \sum_{k=1}^{n-1} C(k) \\
&= 1 + 2 \sum_{k=1}^{n-1} 3^{k-1} && \text{by induction} \\
&= 1 + (3^{n-1} - 1) = 3^{n-1}. && \text{geometric sum}
\end{aligned}
$$

Time $\Omega(3^n)$, yet there are only $n(n+1)/2$ different subproblems (i, j).

The recursion tree



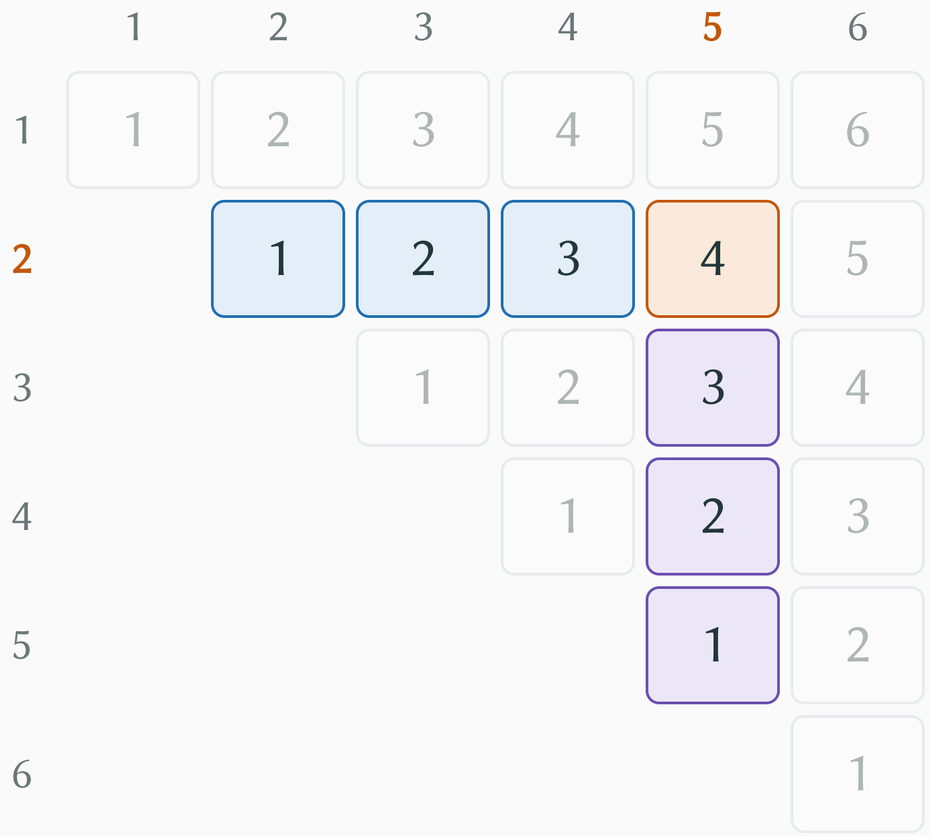
Calls $m[i, i]$ on a single matrix return 0 at once and are not drawn.

? *What do you notice?*

- $m[3, 4]$ is computed twice
- $m[2, 3]$ is computed twice
- $m[1, 2]$ is computed twice

n	calls	subproblems
4	27	10
5	81	15
6	243	21
10	19683	55
20	1 162 261 467	210

One subproblem for every pair (i, j)



Subproblems: the chains $A_{i:j}$ with $1 \leqslant i \leqslant j \leqslant n$.

$\binom{n}{2} + n = \dfrac{n(n+1)}{2} = \Theta(n^2)$ of them (21 for $n = 6$): **solve each once, store the answer.**

$m[i, j]$ needs $m[i, k]$, left in its row, and $m[k+1, j]$, below in its column.

> Fill the table by increasing **chain length** $\ell = j - i + 1$: an entry only needs shorter chains.

The algorithm

In which order do we fill the table, and how fast is it?

Bottom-up: Matrix-Chain-Order


```
    MATRIX-CHAIN-ORDER(p, n)
1   let m[1:n, 1:n] and s[1:n−1, 2:n] be new tables
2   for i = 1 to n   // chains of length 1
3       m[i, i] ← 0
4   for ℓ = 2 to n   // ℓ is the chain length
5       for i = 1 to n − ℓ + 1
6           j ← i + ℓ − 1   // the chain A_{i:j}
7           m[i, j] ← ∞
8           for k = i to j − 1   // try every split
9               q ← m[i, k] + m[k+1, j] + p_{i−1} p_k p_j
10              if q < m[i, j]
11                  m[i, j] ← q   // record the cost
12                  s[i, j] ← k   // record the split
13  return m and s
```

	1	2	3
1	0	5000 k=1	7500 k=2
2		0	25000 k=2
3			0

Number of runs of line 9:

$$\sum_{\ell=2}^{n} (n - \ell + 1)(\ell - 1) = \frac{n^3 - n}{6}$$

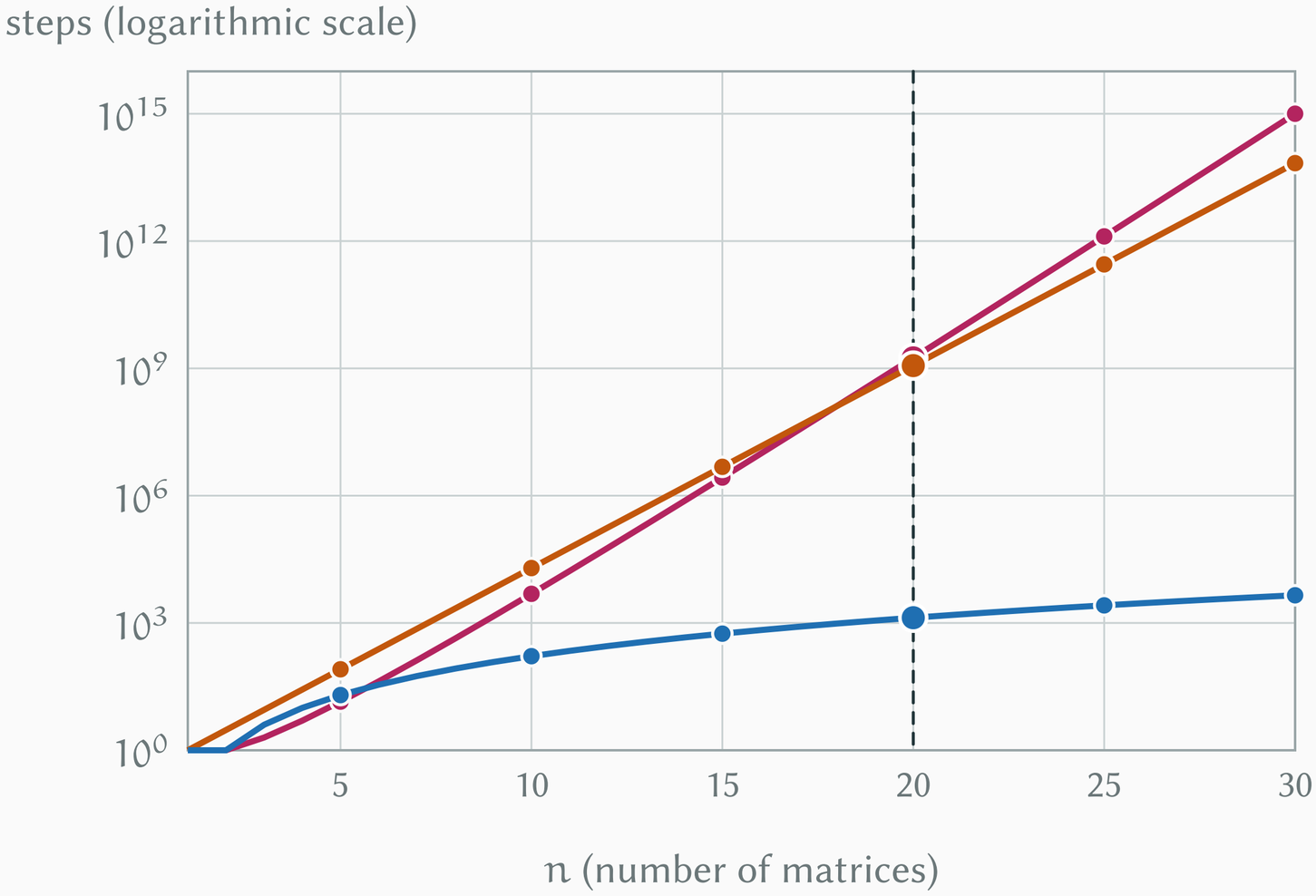
$O(n^3)$ **time**, $\Theta(n^2)$ space.

Worked example

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

$m[i,j]$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$s[i,j]$
$m[1,2]$	**15750**					**1**
$m[2,3]$		**2625**				**2**
$m[3,4]$			**750**			**3**
$m[4,5]$				**1000**		**4**
$m[5,6]$					**5000**	**5**
$m[1,3]$	**7875**	18000				**1**
$m[2,4]$		6000	**4375**			**3**
$m[3,5]$			**2500**	3750		**3**
$m[4,6]$				6250	**3500**	**5**
$m[1,4]$	14875	21000	**9375**			**3**
$m[2,5]$		13000	**7125**	11375		**3**
$m[3,6]$			**5375**	9500	10000	**3**
$m[1,5]$	28125	27250	**11875**	15375		**3**
$m[2,6]$		18500	**10500**	18125	24625	**3**
$m[1,6]$	36750	32375	**15125**	21875	26875	**3**

How much faster?

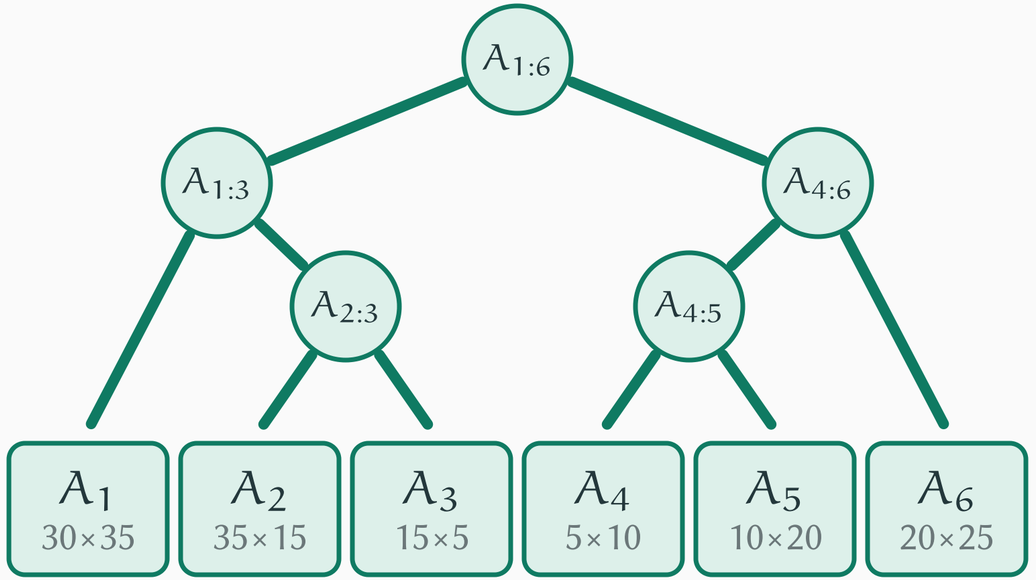


- every parenthesisation: $P(n)$
- naive recursion: 3^{n-1} calls
- the table: $(n^3 - n)/6$ splits

At $n = 20$: **1 767 263 190** parenthesisations
1 162 261 467 recursive calls
1330 splits in the table

Traceback: the optimal order from the table

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



$((A_1(A_2A_3))((A_4A_5)A_6))$

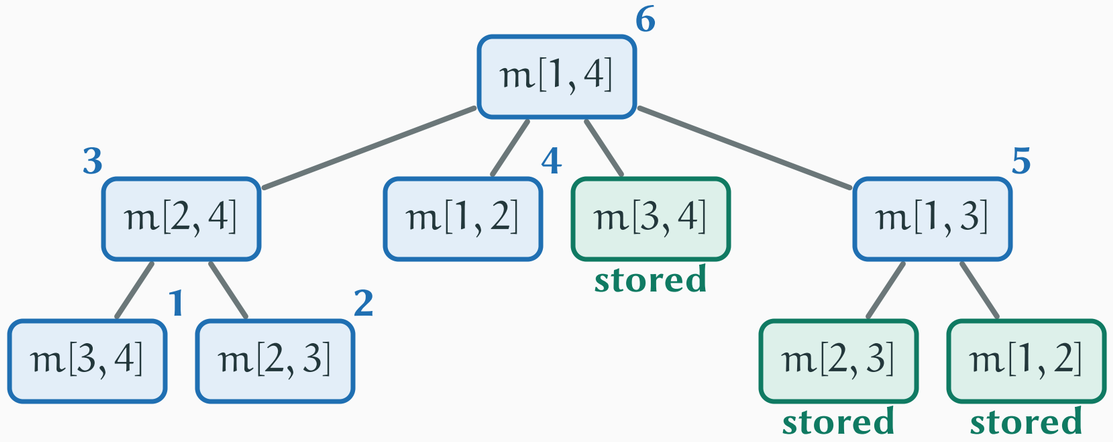
15125 scalar multiplications

from left to right: 40500

Memoisation: top-down

```
MEMOIZED-MATRIX-CHAIN(p, n)
1  m[i, j] ← ∞ for all 1 ⩽ i ⩽ j ⩽ n
2  return LOOKUP-CHAIN(m, p, 1, n)
```

```
LOOKUP-CHAIN(m, p, i, j)
1  if m[i, j] < ∞   // already computed?
2      return m[i, j]
3  if i = j then m[i, j] ← 0
4  else for k = i to j − 1
5      q ← LOOKUP-CHAIN(m, p, i, k)
             + LOOKUP-CHAIN(m, p, k+1, j)
             + p_{i−1} p_k p_j
6      if q < m[i, j] then m[i, j] ← q
7  return m[i, j]
```



For $n = 6$: **71** calls instead of 243:
21 computed, 50 looked up.

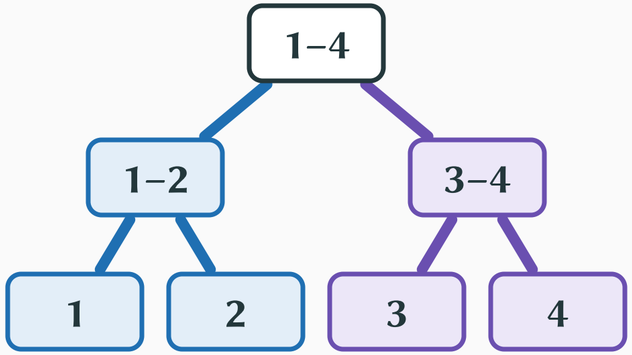
Each of the $\Theta(n^2)$ subproblems is computed once, in $O(n)$ time: $O(n^3)$ as well.

Dynamic programming

When does dynamic programming work, and how do we use it?

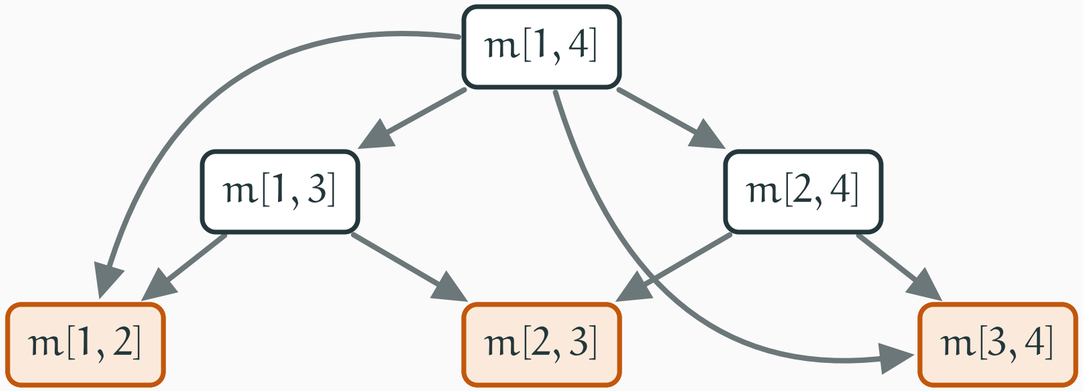
Dynamic programming versus divide and conquer

Divide and conquer



- **independent** subproblems
- dependencies form a **tree**
- sorting, FFT: not about optimisation

Dynamic programming



- **overlapping** subproblems: solve each once
- dependencies form a **DAG**
- **optimisation**, with optimal substructure

If the subproblems overlap, store their answers.

The recipe

	in general	chain matrix multiplication
1 Subproblems	smaller versions of the problem	$A_{i:j}$, $1 \leqslant i \leqslant j \leqslant n$
2 Optimal substructure	an optimal solution contains optimal solutions of subproblems	parts of an optimal parenthesisation are optimal
3 Recurrence	the value from smaller subproblems; base cases	$m[i,j] = \min_k\{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\}$, $m[i,i] = 0$
4 Table	store the answers; fill in dependency order (or memoise)	$\Theta(n^2)$ states, $O(n^3)$ time
5 Traceback	keep the choices, read the solution off the table	$s[i,j]$ gives the parenthesisation

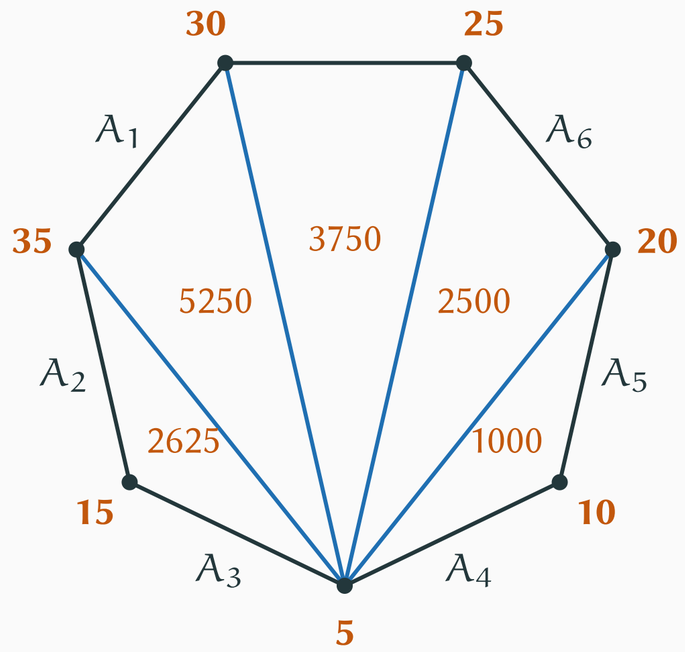
DP works when you can find optimal substructure and can reuse work.

Historical notes

1. **Richard Bellman, RAND, early 1950s:**
   the name "dynamic programming" (his autobiography, 1984).
   "Programming" as in planning, as in "linear programming".
   "Dynamic": a word no one could object to,
   at a time when the Secretary of Defense disliked "research".

2. **Hu and Shing, 1982 and 1984:**
   the best matrix multiplication order in $O(n \log n)$ time.
   They used connections to convex polygon triangulations.

3. **Schwartz and Weiss, 2019:**
   a gap in Hu and Shing's correctness proof.
   They found a new proof: the algorithm is correct.
   (Shing also provided a corrected proof.)



$((A_1(A_2A_3))((A_4A_5)A_6))$

the five triangles cost 15125 in total

Summary

1. **The order of a matrix product matters.**
2. **Optimal substructure:** the parts of an optimal parenthesisation are optimal.
   Hence $m[i, j] = \min_k \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\}$.
3. **Few distinct subproblems:** $n(n+1)/2$ against 3^{n-1} calls.
   Store the answers in a table, or memoise.
   Optimal solution can be worked out by traceback.

> Optimal matrix multiplication order can be found
> in $O(n^3)$ time and $O(n^2)$ space.

Reference: CLRS, Chapters 14.2, 14.3