Algorithms: Dynamic Programming (Rod Cutting, Matrix Chain Multi.)

Theophile Thiery

EPFL School of Computer and Communication Sciences

Lecture 10, 19.03.2025

DYNAMIC PROGRAMMING

(An algorithmic paradigm not a way of “programming”)

DYNAMIC PROGRAMMING

(An algorithmic paradigm not a way of “programming”)

What is $2^5 + 3 - \sqrt{16}$?

DYNAMIC PROGRAMMING

(An algorithmic paradigm not a way of "programming")

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

DYNAMIC PROGRAMMING

(An algorithmic paradigm not a way of “programming”)

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

DYNAMIC PROGRAMMING

(An algorithmic paradigm not a way of "programming")

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

DYNAMIC PROGRAMMING

(An algorithmic paradigm not a way of "programming")

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

DYNAMIC PROGRAMMING

(An algorithmic paradigm not a way of “programming”)

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

DYNAMIC PROGRAMMING

(An algorithmic paradigm not a way of "programming")

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

Dynamic Programming (DP)

Main idea:

- Remember calculations already made
- Saves enormous amounts of computation

Dynamic Programming (DP)

Main idea:

- Remember calculations already made
- Saves enormous amounts of computation

Allows to solve many optimization problems

- Always at least one question in google code jam needs DP

Key elements in designing a DP-algorithm

Optimal substructure

- Show that a solution to a problem consists of making a choice, which leaves one or several subproblems to solve

Key elements in designing a DP-algorithm

Optimal substructure

- Show that a solution to a problem consists of making a choice, which leaves one or several subproblems to solve and the optimal solution solves the subproblems optimally

Key elements in designing a DP-algorithm

Optimal substructure

- Show that a solution to a problem consists of making a choice, which leaves one or several subproblems to solve and the optimal solution solves the subproblems optimally

Overlapping subproblems

- A naive recursive algorithm may revisit the same (sub)problem over and over.
- Top-down with memoization

  Solve recursively but store each result in a table
- Bottom-up

  Sort the subproblems and solve the smaller ones first; that way, when solving a subproblem, have already solved the smaller subproblems we need

ROD CUTTING

Rod cutting - Reminder

Instance:
- A length n of a metal rod.
- A table of prices p_i for rods of lengths $i = 1, \ldots, n$.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Rod cutting - Reminder

Instance:
- A length n of a metal rod.
- A table of prices p_i for rods of lengths $i = 1, \ldots, n$.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Objective: Decide how to cut the rod into pieces and maximize the price.

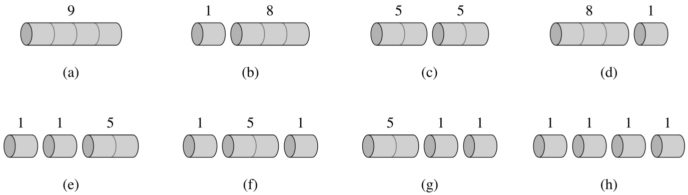
Rod cutting - Reminder

Instance:

- A length n of a metal rod.
- A table of prices p_i for rods of lengths $i = 1, \ldots, n$.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Objective: Decide how to cut the rod into pieces and maximize the price.



Dynamic programming algorithm

Choice:

Dynamic programming algorithm

Choice: where to make the leftmost cut

Dynamic programming algorithm

Choice: where to make the leftmost cut

Optimal substructure:

Dynamic programming algorithm

Choice: where to make the leftmost cut

Optimal substructure: to obtain an optimal solution, we need to cut the remaining piece in an optimal way

Dynamic programming algorithm

Choice: where to make the leftmost cut

Optimal substructure: to obtain an optimal solution, we need to cut the remaining piece in an optimal way

Hence, if we let $r(n)$ be the optimal revenue from a rod of length n, we can express $r(n)$ recursively as follows

$$r(n) = \begin{cases} 0 & \text{if } n = 0 \\ \max_{1 \leq i \leq n} \{p_i + r(n-i)\} & \text{otherwise if } n \geq 1 \end{cases}$$

Dynamic programming algorithm

Choice: where to make the leftmost cut

Optimal substructure: to obtain an optimal solution, we need to cut the remaining piece in an optimal way

Hence, if we let $r(n)$ be the optimal revenue from a rod of length n, we can express $r(n)$ recursively as follows

$$r(n) = \begin{cases} 0 & \text{if } n = 0 \\ \max_{1 \leq i \leq n} \{p_i + r(n-i)\} & \text{otherwise if } n \geq 1 \end{cases}$$

Overlapping subproblems: Solve recurrence using top-down with memoization or bottom-up which yields an algorithm that runs in time $\Theta(n^2)$.

Reconstructing an Optimal Solution

- We saw algorithms that only return the optimal profit.
- Sometimes one needs also to find an optimal solution.

Reconstructing an Optimal Solution

- We saw algorithms that only return the optimal profit.
- Sometimes one needs also to find an optimal solution.

Approach

- Each cell of the memoization table corresponds to a decision: the location of the left most cut.

Reconstructing an Optimal Solution

- We saw algorithms that only return the optimal profit.
- Sometimes one needs also to find an optimal solution.

Approach

- Each cell of the memoization table corresponds to a decision: the location of the left most cut.
- Store the decision corresponding to every cell in a separate table.

Reconstructing an Optimal Solution (cont.)

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  let r[0..n] and s[0..n] be new arrays
  r[0] = 0
  for j = 1 to n
      q = −∞
      for i = 1 to j
          if q < p[i] + r[j − i]
              q = p[i] + r[j − i]
              s[j] = i
      r[j] = q
  return r and s
```

Reconstructing an Optimal Solution (cont.)

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  let r[0..n] and s[0..n] be new arrays
  r[0] = 0
  for j = 1 to n
      q = −∞
      for i = 1 to j
          if q < p[i] + r[j − i]
              q = p[i] + r[j − i]
              s[j] = i
      r[j] = q
  return r and s
```

Output

i	0	1	2	3	4	5	6	7	8
r[i]	0	1	5	8	10	13	17	18	22
s[i]	0	1	2	3	2	2	6	1	2

Reconstructing an Optimal Solution (cont.)

- The table s stores the choices that lead to an optimal solution.
- These decisions can be extracted from the table.

```
PRINT-CUT-ROD-SOLUTION(p, n)
  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  while n > 0
      print s[n]
      n = n - s[n]
```

Reconstructing an Optimal Solution (cont.)

- The table s stores the choices that lead to an optimal solution.
- These decisions can be extracted from the table.

```
PRINT-CUT-ROD-SOLUTION(p, n)
  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  while n > 0
      print s[n]
      n = n − s[n]
```

Output(for $n = 8$)

n	8
output	

Reconstructing an Optimal Solution (cont.)

- The table s stores the choices that lead to an optimal solution.
- These decisions can be extracted from the table.

```
PRINT-CUT-ROD-SOLUTION(p, n)
  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  while n > 0
       print s[n]
       n = n - s[n]
```

Output(for $n = 8$)

n	8
output	2

Reconstructing an Optimal Solution (cont.)

- The table s stores the choices that lead to an optimal solution.
- These decisions can be extracted from the table.

```
PRINT-CUT-ROD-SOLUTION(p, n)
  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  while n > 0
      print s[n]
      n = n - s[n]
```

Output(for $n = 8$)

n	8	6
output	2	

Reconstructing an Optimal Solution (cont.)

- The table s stores the choices that lead to an optimal solution.
- These decisions can be extracted from the table.

```
PRINT-CUT-ROD-SOLUTION(p, n)
  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  while n > 0
      print s[n]
      n = n - s[n]
```

Output(for $n = 8$)

n	8	6
output	2	6

Reconstructing an Optimal Solution (cont.)

- The table s stores the choices that lead to an optimal solution.
- These decisions can be extracted from the table.

```
PRINT-CUT-ROD-SOLUTION(p, n)
  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  while n > 0
      print s[n]
      n = n - s[n]
```

Output(for $n = 8$)

n	8	6	0
output	2	6	

Summary

- We had a recursive formulation for the optimal value for our problem

$$r(n) = \begin{cases} 0 & \text{if } n = 0 \ , \\ \max_{1 \leq i \leq n} \{p_i + r(n-i)\} & \text{otherwise if } n \geq 1 \ . \end{cases}$$

- Speed up the calculations by filling in a table either "top-down with memoization" or with "bottom-up".
- Recovered an optimal solution using an additional table.

When Can Dynamic Programming Be Used?

When Can Dynamic Programming Be Used?

1. Optimal Substructure.
   - An optimal solution can be built by combining optimal solutions for subproblems.

When Can Dynamic Programming Be Used?

1. Optimal Substructure.
   - An optimal solution can be built by combining optimal solutions for subproblems.
   - Implies that the optimal value can be given by a recursive formula.

When Can Dynamic Programming Be Used?

1. Optimal Substructure.
   - An optimal solution can be built by combining optimal solutions for subproblems.
   - Implies that the optimal value can be given by a recursive formula.
2. Overlapping subproblems.

Problem Solving: the Change-Making Problem

- How can a given amount of money be made with the least number of coins of given denominations?

Problem Solving: the Change-Making Problem

- How can a given amount of money be made with the least number of coins of given denominations?

Formally:

Input: n distinct coin denominators (integers) $0 < w_1 < w_2 < \ldots < w_n$ and an amount W (the change) which is also a positive integer.

Output: The minimum number of coins needed in order to make the change:

$$\min \left\{ \sum_{j=1}^{n} x_j : \sum_{j=1}^{n} w_j x_j = W \text{ and } x_j\text{'s are integers} \right\}.$$

Problem Solving: the Change-Making Problem

- How can a given amount of money be made with the least number of coins of given denominations?

Formally:

Input: n distinct coin denominators (integers) $0 < w_1 < w_2 < \ldots < w_n$ and an amount W (the change) which is also a positive integer.

Output: The minimum number of coins needed in order to make the change:

$$\min \left\{ \sum_{j=1}^{n} x_j : \sum_{j=1}^{n} w_j x_j = W \text{ and } x_j\text{'s are integers} \right\}.$$

Example: On input $w_1 = 1, w_2 = 2, w_3 = 5$ and $W = 8$, the output should be 3 since the best way of giving 8 is $x_1 = x_2 = x_3 = 1$.

Problem Solving: the Change-Making Problem

Parenthesization	Cost computation	Cost
$A \times ((B \times C) \times D)$	$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$	$120,200$
$(A \times (B \times C)) \times D$	$20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$	$60,200$
$(A \times B) \times (C \times D)$	$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$	$7,000$

MATRIX-CHAIN MULTIPLICATION

Cost of Matrix Multiplication

$$A_{p,q} \qquad \times \qquad B_{q,r}$$

Cost of Matrix Multiplication

$$A_{p,q} \times B_{q,r}$$

$$p\left\{\left[\overbrace{\begin{matrix}(1,1) & (1,2) & \cdots & (1,q)\\ (2,1) & (2,2) & \cdots & (2,q)\\ \vdots & \vdots & \ddots & \vdots\\ (p,1) & (p,2) & \cdots & (p,q)\end{matrix}}^{q}\right]\right. \qquad q\left\{\left[\overbrace{\begin{matrix}(1,1) & (1,2) & \cdots & (1,r)\\ (2,1) & (2,2) & \cdots & (2,r)\\ \vdots & \vdots & \ddots & \vdots\\ (q,1) & (q,2) & \cdots & (q,r)\end{matrix}}^{r}\right]\right.$$

Cost of Matrix Multiplication

$$A_{p,q} \times B_{q,r}$$

$$p\left\{\begin{bmatrix} \overbrace{\begin{matrix}(1,1) & (1,2) & \cdots & (1,q)\\ (2,1) & (2,2) & \cdots & (2,q)\\ \vdots & \vdots & \ddots & \vdots\\ (p,1) & (p,2) & \cdots & (p,q)\end{matrix}}^{q}\end{bmatrix}\right. \qquad q\left\{\begin{bmatrix} \overbrace{\begin{matrix}(1,1) & (1,2) & \cdots & (1,r)\\ (2,1) & (2,2) & \cdots & (2,r)\\ \vdots & \vdots & \ddots & \vdots\\ (q,1) & (q,2) & \cdots & (q,r)\end{matrix}}^{r}\end{bmatrix}\right.$$

$$\Downarrow$$

$$C_{p,r}$$

$$p\left\{\begin{bmatrix} \overbrace{\begin{matrix}(1,1) & (1,2) & \cdots & (1,r)\\ (2,1) & (2,2) & \cdots & (2,r)\\ \vdots & \vdots & \ddots & \vdots\\ (p,1) & (p,2) & \cdots & (p,r)\end{matrix}}^{r}\end{bmatrix}\right.$$

Cost of Matrix Multiplication

$$A_{p,q} \times B_{q,r}$$

$$p\left\{\begin{bmatrix} \overbrace{\begin{matrix} (1,1) & (1,2) & \cdots & (1,q) \\ (2,1) & (2,2) & \cdots & (2,q) \\ \vdots & \vdots & \ddots & \vdots \\ (p,1) & (p,2) & \cdots & (p,q) \end{matrix}}^{q} \end{bmatrix}\right. \qquad q\left\{\begin{bmatrix} \overbrace{\begin{matrix} (1,1) & (1,2) & \cdots & (1,r) \\ (2,1) & (2,2) & \cdots & (2,r) \\ \vdots & \vdots & \ddots & \vdots \\ (q,1) & (q,2) & \cdots & (q,r) \end{matrix}}^{r} \end{bmatrix}\right.$$

$$\Downarrow$$

$$C_{p,r}$$

$$p\left\{\begin{bmatrix} \overbrace{\begin{matrix} (1,1) & (1,2) & \cdots & (1,r) \\ \boxed{(2,1)} & (2,2) & \cdots & (2,r) \\ \vdots & \vdots & \ddots & \vdots \\ (p,1) & (p,2) & \cdots & (p,r) \end{matrix}}^{r} \end{bmatrix}\right.$$

Cost of Matrix Multiplication

$$A_{p,q} \times B_{q,r}$$

$$p\left\{\overbrace{\begin{bmatrix} (1,1) & (1,2) & \cdots & (1,q) \\ \boxed{(2,1) \quad (2,2) \quad \cdots \quad (2,q)} \\ \vdots & \vdots & \ddots & \vdots \\ (p,1) & (p,2) & \cdots & (p,q) \end{bmatrix}}^{q}\right. \qquad q\left\{\overbrace{\begin{bmatrix} (1,1) & (1,2) & \cdots & (1,r) \\ (2,1) & (2,2) & \cdots & (2,r) \\ \vdots & \vdots & \ddots & \vdots \\ (q,1) & (q,2) & \cdots & (q,r) \end{bmatrix}}^{r}\right.$$

$$\Downarrow$$

$$C_{p,r}$$

$$p\left\{\overbrace{\begin{bmatrix} (1,1) & (1,2) & \cdots & (1,r) \\ \boxed{(2,1)} & (2,2) & \cdots & (2,r) \\ \vdots & \vdots & \ddots & \vdots \\ (p,1) & (p,2) & \cdots & (p,r) \end{bmatrix}}^{r}\right.$$

Cost of Matrix Multiplication

$$A_{p,q} \qquad \times \qquad B_{q,r}$$

$$p\left\{\left[\overbrace{\begin{matrix}(1,1) & (1,2) & \cdots & (1,q)\\ \boxed{(2,1)} & \boxed{(2,2)} & \boxed{\cdots} & \boxed{(2,q)}\\ \vdots & \vdots & \ddots & \vdots\\ (p,1) & (p,2) & \cdots & (p,q)\end{matrix}}^{q}\right]\right. \qquad q\left\{\left[\overbrace{\begin{matrix}\boxed{(1,1)} & (1,2) & \cdots & (1,r)\\ \boxed{(2,1)} & (2,2) & \cdots & (2,r)\\ \boxed{\vdots} & \vdots & \ddots & \vdots\\ \boxed{(q,1)} & (q,2) & \cdots & (q,r)\end{matrix}}^{r}\right]\right.$$

$$\Downarrow$$

$$C_{p,r}$$

$$p\left\{\left[\overbrace{\begin{matrix}(1,1) & (1,2) & \cdots & (1,r)\\ \boxed{(2,1)} & (2,2) & \cdots & (2,r)\\ \vdots & \vdots & \ddots & \vdots\\ (p,1) & (p,2) & \cdots & (p,r)\end{matrix}}^{r}\right]\right.$$

Cost of Matrix Multiplication

$$A_{p,q} \times B_{q,r}$$

$$p\left\{\begin{bmatrix} \overbrace{(1,1) & (1,2) & \cdots & (1,q)}^{q} \\ \boxed{(2,1) \quad (2,2) \quad \cdots \quad (2,q)} \\ \vdots & \vdots & \ddots & \vdots \\ (p,1) & (p,2) & \cdots & (p,q) \end{bmatrix}\right. \quad q\left\{\begin{bmatrix} \overbrace{(1,1) & (1,2) & \cdots & (1,r)}^{r} \\ (2,1) & (2,2) & \cdots & (2,r) \\ \vdots & \vdots & \ddots & \vdots \\ (q,1) & (q,2) & \cdots & (q,r) \end{bmatrix}\right.$$

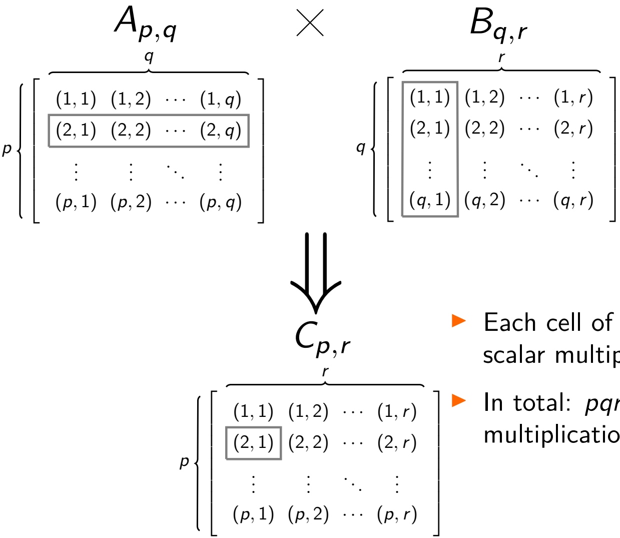
$$\Downarrow$$

$$C_{p,r}$$

$$p\left\{\begin{bmatrix} \overbrace{(1,1) & (1,2) & \cdots & (1,r)}^{r} \\ \boxed{(2,1)} & (2,2) & \cdots & (2,r) \\ \vdots & \vdots & \ddots & \vdots \\ (p,1) & (p,2) & \cdots & (p,r) \end{bmatrix}\right.$$

- Each cell of C requires q scalar multiplications.

Cost of Matrix Multiplication



- Each cell of C requires q scalar multiplications.
- In total: pqr scalar multiplications.

Cost of Matrix Multiplication

$$A_{p,q} \qquad \times \qquad B_{q,r}$$

$$p\left\{\begin{bmatrix} \overbrace{\begin{matrix}(1,1) & (1,2) & \cdots & (1,q)\\ \boxed{(2,1) \quad (2,2) \quad \cdots \quad (2,q)} \\ \vdots & \vdots & \ddots & \vdots \\ (p,1) & (p,2) & \cdots & (p,q)\end{matrix}}^{q}\end{bmatrix}\right. \qquad q\left\{\begin{bmatrix} \overbrace{\begin{matrix}(1,1) & (1,2) & \cdots & (1,r)\\ (2,1) & (2,2) & \cdots & (2,r) \\ \vdots & \vdots & \ddots & \vdots \\ (q,1) & (q,2) & \cdots & (q,r)\end{matrix}}^{r}\end{bmatrix}\right.$$

$$\Downarrow$$

$$C_{p,r}$$

$$p\left\{\begin{bmatrix} \overbrace{\begin{matrix}(1,1) & (1,2) & \cdots & (1,r)\\ \boxed{(2,1)} & (2,2) & \cdots & (2,r) \\ \vdots & \vdots & \ddots & \vdots \\ (p,1) & (p,2) & \cdots & (p,r)\end{matrix}}^{r}\end{bmatrix}\right.$$

- Each cell of C requires q scalar multiplications.
- In total: pqr scalar multiplications.
- The scalar multiplications dominate the time complexity.

Matrix Chain Multiplication

Definition

Input: A chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$.

Output: A full parenthesization of the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

Matrix Chain Multiplication

Definition

Input: A chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$.

Output: A full parenthesization of the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

Remarks

Matrix Chain Multiplication

Definition

Input: A chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$.

Output: A full parenthesization of the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

Remarks

- We are not asked to calculate the product, only find the best parenthesization.

Matrix Chain Multiplication

Definition

Input: A chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$.

Output: A full parenthesization of the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

Remarks

- We are not asked to calculate the product, only find the best parenthesization.
- The parenthesization can significantly affect the number of multiplications.

Matrix Chain Multiplication

Definition

Input: A chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$.

Output: A full parenthesization of the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

Example

Matrix Chain Multiplication

Definition

Input: A chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$.

Output: A full parenthesization of the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

Example

- A product $A_1 A_2 A_3$ with dimensions: 50×5, 5×100 and 100×10.

Matrix Chain Multiplication

Definition

Input: A chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$.

Output: A full parenthesization of the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

Example

- A product $A_1 A_2 A_3$ with dimensions: 50×5, 5×100 and 100×10.
- Calculating $(A_1 A_2) A_3$ requires: $50 \cdot 5 \cdot 100 + 50 \cdot 100 \cdot 10 = 75000$ scalar multiplications.

Matrix Chain Multiplication

Definition

Input: A chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$.

Output: A full parenthesization of the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

Example

- A product $A_1 A_2 A_3$ with dimensions: 50×5, 5×100 and 100×10.
- Calculating $(A_1 A_2) A_3$ requires: $50 \cdot 5 \cdot 100 + 50 \cdot 100 \cdot 10 = 75000$ scalar multiplications.
- Calculating $A_1 (A_2 A_3)$ requires: $5 \cdot 100 \cdot 10 + 50 \cdot 5 \cdot 10 = 7500$ scalar multiplications.

Optimal Substructure

Theorem

If:

- *the outermost parenthesization in an optimal solution is:* $(A_1 A_2 \cdots A_i)(A_{i+1} A_{i+2} \cdots A_n)$.
- P_L *and* P_R *are optimal parenthesizations for* $A_1 A_2 \cdots A_i$ *and* $A_{i+1} A_{i+2} \cdots A_n$*, respectively.*

Then, $((P_L) \cdot (P_R))$ *is an optimal parenthesizations for* $A_1 A_2 \cdots A_n$.

Optimal Substructure

Theorem

If:

- *the outermost parenthesization in an optimal solution is:* $(A_1 A_2 \cdots A_i)(A_{i+1} A_{i+2} \cdots A_n)$.
- P_L *and* P_R *are optimal parenthesizations for* $A_1 A_2 \cdots A_i$ *and* $A_{i+1} A_{i+2} \cdots A_n$*, respectively.*

Then, $((P_L) \cdot (P_R))$ *is an optimal parenthesizations for* $A_1 A_2 \cdots A_n$.

Proof

- Let $((O_L) \cdot (O_R))$ be an optimal parenthesization, where O_L and O_R are parenthesizations for $A_1 A_2 \cdots A_i$ and $A_{i+1} \cdots A_n$, respectively.

Optimal Substructure

Theorem

If:

- *the outermost parenthesization in an optimal solution is:* $(A_1 A_2 \cdots A_i)(A_{i+1} A_{i+2} \cdots A_n)$.
- P_L *and* P_R *are optimal parenthesizations for* $A_1 A_2 \cdots A_i$ *and* $A_{i+1} A_{i+2} \cdots A_n$*, respectively.*

Then, $((P_L) \cdot (P_R))$ *is an optimal parenthesizations for* $A_1 A_2 \cdots A_n$.

Proof

- Let $((O_L) \cdot (O_R))$ be an optimal parenthesization, where O_L and O_R are parenthesizations for $A_1 A_2 \cdots A_i$ and $A_{i+1} \cdots A_n$, respectively.
- Let $M(P)$ be the number of scalar multiplications required by a parenthesization.

Optimal Substructure

Theorem

If:

- *the outermost parenthesization in an optimal solution is:* $(A_1 A_2 \cdots A_i)(A_{i+1} A_{i+2} \cdots A_n)$.
- P_L *and* P_R *are optimal parenthesizations for* $A_1 A_2 \cdots A_i$ *and* $A_{i+1} A_{i+2} \cdots A_n$*, respectively.*

Then, $((P_L) \cdot (P_R))$ *is an optimal parenthesizations for* $A_1 A_2 \cdots A_n$.

Proof

$$M((O_L) \cdot (O_R))$$

Optimal Substructure

Theorem

If:

- *the outermost parenthesization in an optimal solution is:* $(A_1 A_2 \cdots A_i)(A_{i+1} A_{i+2} \cdots A_n)$.
- P_L *and* P_R *are optimal parenthesizations for* $A_1 A_2 \cdots A_i$ *and* $A_{i+1} A_{i+2} \cdots A_n$, *respectively.*

Then, $((P_L) \cdot (P_R))$ *is an optimal parenthesizations for* $A_1 A_2 \cdots A_n$.

Proof

$$M((O_L) \cdot (O_R)) = p_0 \cdot p_i \cdot p_n + M(O_L) + M(O_R)$$

Optimal Substructure

Theorem

If:

- *the outermost parenthesization in an optimal solution is:* $(A_1 A_2 \cdots A_i)(A_{i+1} A_{i+2} \cdots A_n)$.
- P_L *and* P_R *are optimal parenthesizations for* $A_1 A_2 \cdots A_i$ *and* $A_{i+1} A_{i+2} \cdots A_n$*, respectively.*

Then, $((P_L) \cdot (P_R))$ *is an optimal parenthesizations for* $A_1 A_2 \cdots A_n$.

Proof

$$M((O_L) \cdot (O_R)) = p_0 \cdot p_i \cdot p_n + M(O_L) + M(O_R)$$

- Since P_L and P_R are optimal: $M(P_L) \leq M(O_L)$ and $M(P_R) \leq M(O_R)$.

Optimal Substructure

Theorem

If:

- *the outermost parenthesization in an optimal solution is:* $(A_1 A_2 \cdots A_i)(A_{i+1} A_{i+2} \cdots A_n)$.
- P_L *and* P_R *are optimal parenthesizations for* $A_1 A_2 \cdots A_i$ *and* $A_{i+1} A_{i+2} \cdots A_n$*, respectively.*

Then, $((P_L) \cdot (P_R))$ *is an optimal parenthesizations for* $A_1 A_2 \cdots A_n$*.*

Proof

$$\begin{aligned} M((O_L) \cdot (O_R)) &= p_0 \cdot p_i \cdot p_n + M(O_L) + M(O_R) \\ &\geq p_0 \cdot p_i \cdot p_n + M(P_L) + M(P_R) \end{aligned}$$

- Since P_L and P_R are optimal: $M(P_L) \leq M(O_L)$ and $M(P_R) \leq M(O_R)$.

Optimal Substructure

Theorem

If:

- *the outermost parenthesization in an optimal solution is:* $(A_1 A_2 \cdots A_i)(A_{i+1} A_{i+2} \cdots A_n)$.
- P_L *and* P_R *are optimal parenthesizations for* $A_1 A_2 \cdots A_i$ *and* $A_{i+1} A_{i+2} \cdots A_n$*, respectively.*

Then, $((P_L) \cdot (P_R))$ *is an optimal parenthesizations for* $A_1 A_2 \cdots A_n$.

Proof

$$
\begin{aligned}
M((O_L) \cdot (O_R)) &= p_0 \cdot p_i \cdot p_n + M(O_L) + M(O_R) \\
&\geq p_0 \cdot p_i \cdot p_n + M(P_L) + M(P_R) = M((P_L) \cdot (P_R)) \ .
\end{aligned}
$$

- Since P_L and P_R are optimal: $M(P_L) \leq M(O_L)$ and $M(P_R) \leq M(O_R)$.

Recursive Formula

- Let $m[i,j]$ be the optimal number of scalar multiplications for calculating $A_i A_{i+1} \cdots , A_j$.

Recursive Formula

- Let $m[i,j]$ be the optimal number of scalar multiplications for calculating $A_i A_{i+1} \cdots, A_j$.
- $m[i,j]$ can be expressed recursively as follows:

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \ , \\ \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j \ . \end{cases}$$

Recursive Formula

- Let $m[i,j]$ be the optimal number of scalar multiplications for calculating $A_i A_{i+1} \cdots , A_j$.

- $m[i,j]$ can be expressed recursively as follows:

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \ , \\ \min_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j \ . \end{cases}$$

- Each $m[i,j]$ depend only on subproblems with smaller $j - i$.

Recursive Formula

- Let $m[i, j]$ be the optimal number of scalar multiplications for calculating $A_i A_{i+1} \cdots, A_j$.

- $m[i, j]$ can be expressed recursively as follows:

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \ , \\ \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j \ . \end{cases}$$

- Each $m[i, j]$ depend only on subproblems with smaller $j - i$.

- A bottom-up algorithm should solve subproblems in increasing $j - i$ order.

Example

Instance

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimensions	30×35	35×15	15×5	5×10	10×20	20×25

Bottom-Up Algorithm

```
MATRIX-CHAIN-ORDER(p)
n = p.length − 1
let m[1..n, 1..n] and s[1..n, 1..n] be new tables
for i = 1 to n
    m[i, i] = 0
for ℓ = 2 to n                    // ℓ is the chain length
    for i = 1 to n − ℓ + 1
        j = i + ℓ − 1
        m[i, j] = ∞
        for k = i to j − 1
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```

Bottom-Up Algorithm

```
MATRIX-CHAIN-ORDER(p)
n = p.length − 1
let m[1 . . n, 1 . . n] and s[1 . . n, 1 . . n] be new tables
for i = 1 to n
    m[i, i] = 0
for ℓ = 2 to n                    // ℓ is the chain length
    for i = 1 to n − ℓ + 1
        j = i + ℓ − 1
        m[i, j] = ∞
        for k = i to j − 1
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```

⇐ s stores the optimal choice

Example

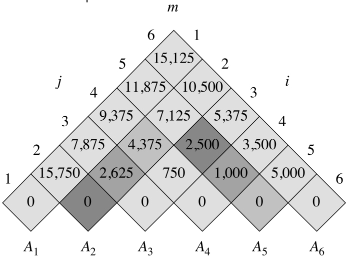
Instance

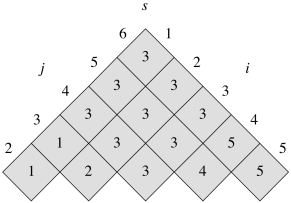
matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimensions	30×35	35×15	15×5	5×10	10×20	20×25

Example

Instance

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimensions	30×35	35×15	15×5	5×10	10×20	20×25

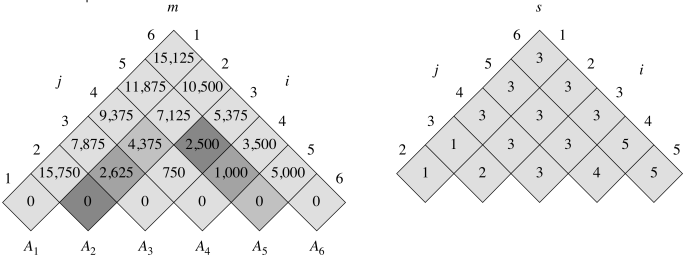




Example

Instance

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimensions	30×35	35×15	15×5	5×10	10×20	20×25

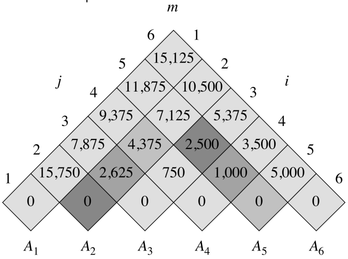


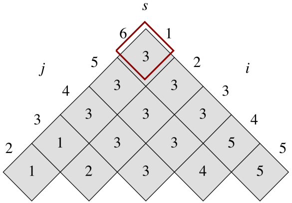
$A_1 \quad A_2 \quad A_3 \qquad A_4 \quad A_5 \quad A_6$

Example

Instance

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimensions	30×35	35×15	15×5	5×10	10×20	20×25



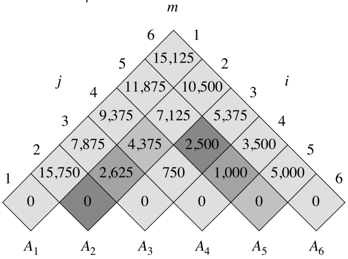


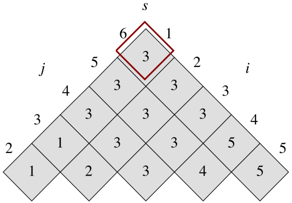
$A_1 \quad A_2 \quad A_3 \qquad A_4 \quad A_5 \quad A_6$

Example

Instance

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimensions	30×35	35×15	15×5	5×10	10×20	20×25



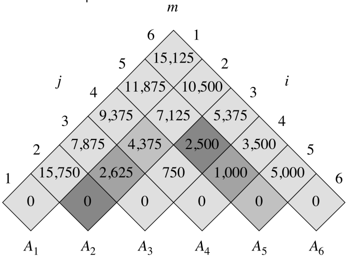


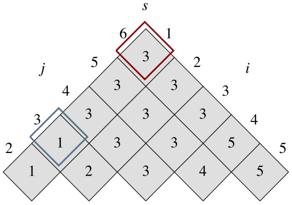
$(A_1 \quad A_2 \quad A_3)(A_4 \quad A_5 \quad A_6)$

Example

Instance

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimensions	30×35	35×15	15×5	5×10	10×20	20×25



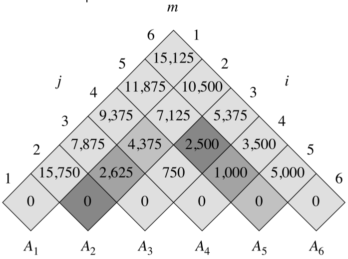


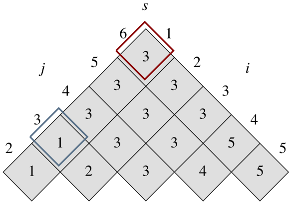
$(A_1 \quad A_2 \quad A_3)(A_4 \quad A_5 \quad A_6)$

Example

Instance

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimensions	30×35	35×15	15×5	5×10	10×20	20×25



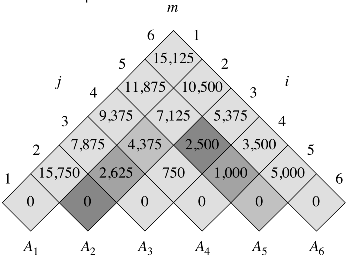


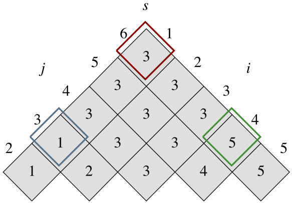
$$(A_1 \quad (A_2 \quad A_3))\,(\,A_4 \quad A_5 \quad A_6)$$

Example

Instance

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimensions	30×35	35×15	15×5	5×10	10×20	20×25



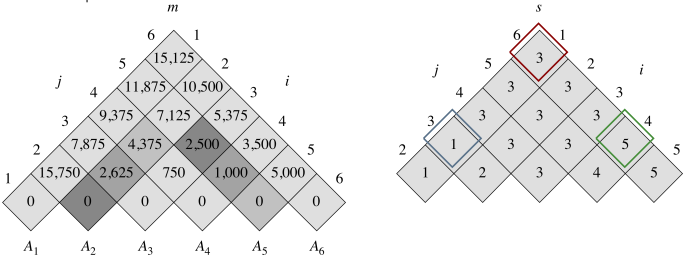


$(A_1 \quad (A_2 \quad A_3))(\; A_4 \quad A_5 \quad A_6)$

Example

Instance

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimensions	30×35	35×15	15×5	5×10	10×20	20×25



$(A_1 \quad (A_2 \quad A_3))((A_4 \quad A_5) \quad A_6)$

Algorithm for Recovering an Optimal Solution

```
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 ")"
```

Summary

Choice:

Summary

Choice: where to make the outermost parenthesis

$$(A_1 \cdots A_k)(A_{k+1} \cdots A_n)$$

Summary

Choice: where to make the outermost parenthesis

$$(A_1 \cdots A_k)(A_{k+1} \cdots A_n)$$

Optimal substructure:

Summary

Choice: where to make the outermost parenthesis

$$(A_1 \cdots A_k)(A_{k+1} \cdots A_n)$$

Optimal substructure: to obtain an optimal solution, we need to parenthesize the two remaining expressions in an optimal way

Summary

Choice: where to make the outermost parenthesis

$$(A_1 \cdots A_k)(A_{k+1} \cdots A_n)$$

Optimal substructure: to obtain an optimal solution, we need to parenthesize the two remaining expressions in an optimal way

Hence, if we let $m[i,j]$ be the optimal value for chain multiplication of matrices $A_i, \ldots, A_j$, we can express $m[i,j]$ recursively as follows

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{otherwise if } i < j \end{cases}$$

Summary

Choice: where to make the outermost parenthesis

$$(A_1 \cdots A_k)(A_{k+1} \cdots A_n)$$

Optimal substructure: to obtain an optimal solution, we need to parenthesize the two remaining expressions in an optimal way

Hence, if we let $m[i,j]$ be the optimal value for chain multiplication of matrices $A_i, \ldots, A_j$, we can express $m[i,j]$ recursively as follows

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{otherwise if } i < j \end{cases}$$

Overlapping subproblem: Solve recurrence using top-down with memoization or bottom-up which yields an algorithm that runs in time $\Theta(n^3)$.