

Analysis of Algorithms, I

CSOR W4231.002

Eleni Drinea

Computer Science Department

Columbia University

Thursday, February 11, 2016

Outline

- 1 Recap
- 2 Matrix chain multiplication
- 3 A first attempt: brute-force
- 4 A second attempt: divide and conquer
- 5 A Dynamic Programming (DP) solution
- 6 Organizing DP computations

Today

- 1 Recap
- 2 Matrix chain multiplication
- 3 A first attempt: brute-force
- 4 A second attempt: divide and conquer
- 5 A Dynamic Programming (DP) solution
- 6 Organizing DP computations

Greedy algorithms: cache maintenance

- ▶ The offline problem
- ▶ An optimal algorithm for the offline problem:
Farthest-in-Future (FF)
- ▶ Proof of optimality of FF
- ▶ The online problem

Today

- 1 Recap
- 2 Matrix chain multiplication**
- 3 A first attempt: brute-force
- 4 A second attempt: divide and conquer
- 5 A Dynamic Programming (DP) solution
- 6 Organizing DP computations

Matrix chain multiplication example

Example 1.

Input: matrices A_1 , A_2 , A_3 of dimensions 6×1 , 1×5 , 5×2

Output:

- ▶ a way to compute the product $A_1A_2A_3$ so that the number of arithmetic operations performed is **minimized**;
- ▶ the minimum number of arithmetic operations required.

Remark 1.

- ▶ *We do not want to compute the actual product.*
- ▶ *Matrix multiplication is associative but not commutative (in general). Hence a solution to our problem corresponds to a **parenthesization** of the product.*
- ▶ *We want the **optimal parenthesization** and its **cost**, that is, the parenthesization that minimizes the number of arithmetic operations, as well as that number.*

Estimating #arithmetic operations

- ▶ Let A, B be matrices of dimensions $m \times n, n \times p$.
- ▶ Let $C = AB$. Then C is an $m \times p$ matrix such that

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

- $\Rightarrow c_{ij}$ requires n scalar multiplications, $n - 1$ additions
- \Rightarrow #arithmetic operations to compute c_{ij} is **dominated** by #scalar multiplications
- ▶ Total #scalar multiplications to fill in C is mnp

Minimizing #scalar multiplications for $A_1A_2A_3$

Input: A_1, A_2, A_3 of dimensions $6 \times 1, 1 \times 5, 5 \times 2$ respectively

Recall that, given a parenthesization of the input matrices, its cost is the total # scalar multiplications to compute the product.

Two ways of computing $A_1A_2A_3$:

1. $(A_1A_2)A_3$: first compute A_1A_2 , then multiply it by A_3
 - ▶ $6 \cdot 1 \cdot 5$ scalar multiplications for A_1A_2
 - ▶ $6 \cdot 5 \cdot 2$ scalar multiplications for $(A_1A_2)A_3$
 - ⇒ 90 scalar multiplications in total
2. $A_1(A_2A_3)$: first compute A_2A_3 , then multiply A_1 by A_2A_3
 - ▶ $1 \cdot 5 \cdot 2$ scalar multiplications for A_2A_3
 - ▶ $6 \cdot 1 \cdot 2$ scalar multiplications for $A_1(A_2A_3)$
 - ⇒ 22 scalar multiplications in total

Remark 2.

Solution $A_1(A_2A_3)$ improves over $(A_1A_2)A_3$ by over 75%.

(Fully) Parenthesized products of matrices

Definition 2.

A product of matrices is fully parenthesized if it is

1. a single matrix; or
2. the product of two fully parenthesized matrices, surrounded by parentheses.

Examples: $((A_1A_2)A_3)$ and $(A_1(A_2A_3))$ are fully parenthesized.

Remark: we will henceforth refer to a *full parenthesization* simply as a *parenthesization*.

Matrix chain multiplication

Input: n matrices A_1, A_2, \dots, A_n , with dimensions $p_{i-1} \times p_i$, for $1 \leq i \leq n$.

Output:

1. the **optimal** parenthesization of the input (that is, the one incurring the minimum cost);
2. its cost.

Example: the optimal parenthesization for Example 1 is $(A_1(A_2A_3))$ and its cost is 22.

Remark 3.

- ▶ *We might want the optimal solution and its cost, or just the cost.*
- ▶ *The optimal solution might not be unique; of course, the optimal cost **is** unique.*

Today

- 1 Recap
- 2 Matrix chain multiplication
- 3 A first attempt: brute-force**
- 4 A second attempt: divide and conquer
- 5 A Dynamic Programming (DP) solution
- 6 Organizing DP computations

Brute-force approach

- ▶ A_1, \dots, A_n are matrices of dimensions $p_{i-1} \times p_i$ for $1 \leq i \leq n$.
- ▶ Consider the product $A_1 \cdots A_n$.
- ▶ Let $P(n) = \#$ parenthesizations of the product $A_1 \cdots A_n$.
- ▶ Then $P(0) = 0, P(1) = 1, P(2) = 1$
- ▶ For $n > 2$, by Definition 2, for every possible parenthesization, there is a $1 \leq k \leq n - 1$ such that the parenthesized product looks like

$$((A_1 A_2 \cdots A_k)(A_{k+1} \cdots A_n))$$

Computing #possible parenthesizations

- ▶ Given k , the #parenthesizations for the product

$$((A_1 A_2 \cdots A_k)(A_{k+1} \cdots A_n))$$

can be computed **recursively**:

$$P(k) \cdot P(n - k)$$

- ▶ There are $n - 1$ possible values for k . Hence

$$P(n) = \sum_{k=1}^{n-1} P(k) \cdot P(n - k), \text{ for } n > 1$$

Bounding $P(n)$

- ▶ We may obtain a crude yet sufficient for our purposes **lower bound** for $P(n)$ as follows

$$\begin{aligned} P(n) &\geq P(1) \cdot P(n-1) + P(2) \cdot P(n-2) \\ &\geq P(n-1) + P(n-2) \end{aligned} \tag{1}$$

- ▶ By strong induction on n , we can show that $P(n) \geq F_n$, the n -th Fibonacci number.
- ▶ By Problem 6a in Homework 1, $P(n) = \Omega(2^{n/2})$.
 - ▶ In fact, $P(n) = \Omega(2^{2n}/n^{3/2})$ (e.g., see your textbook).

⇒ Brute force requires exponential time.

Today

- 1 Recap
- 2 Matrix chain multiplication
- 3 A first attempt: brute-force
- 4 A second attempt: divide and conquer**
- 5 A Dynamic Programming (DP) solution
- 6 Organizing DP computations

A second attempt: divide and conquer

Notation:

1. $(A_i \cdots A_j)$ is a parenthesization of the product $A_i \cdots A_j$.
 2. $A_{1,n}$ is the **optimal** parenthesization of the product $A_1 \cdots A_n$, that is, the one that incurs the minimum cost.
- Consider a parenthesization for $A_1 \cdots A_n$. By Definition 2, it is the product of two fully parenthesized subproducts; hence for some $1 \leq k \leq n - 1$

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

- In particular, there exists $1 \leq k^* \leq n - 1$ such that

$$A_{1,n} = (A_1 \cdots A_{k^*})(A_{k^*+1} \cdots A_n)$$

Notation: $A_{i,j}$ is the optimal parenthesization of the product $A_i \cdots A_j$.

Fact 3.

There exists k^ such that $1 \leq k^* \leq n - 1$ and*

$$A_{1,n} = A_{1,k^*} A_{k^*+1,n}.$$

Hence the **optimal parenthesization of the input** can be decomposed into the **optimal parenthesizations of two subproblems**.

The cost of multiplying two matrices

- ▶ Recall that matrix A_i has dimensions $p_{i-1} \times p_i$. Hence
 - ▶ $(A_1 \cdots A_k)$ is a $p_0 \times p_k$ matrix,
 - ▶ $(A_{k+1} \cdots A_n)$ is a $p_k \times p_n$ matrix.
- ▶ The #scalar multiplications required for multiplying matrix $(A_1 \cdots A_k)$ by matrix $(A_{k+1} \cdots A_n)$ is

$$p_0 p_k p_n.$$

Proof of optimal substructure

Notation: $A_{i,j}$ is the optimal parenthesization of $A_i \cdots A_j$.

- ▶ By Definition 2, exists k^* such that

$$A_{1,n} = ((A_1 \cdots A_{k^*})(A_{k^*+1} \cdots A_n))$$

Then the cost of $A_{1,n}$ is the **sum** of

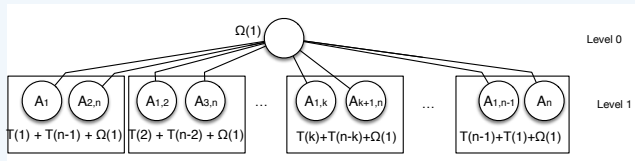
1. the costs of the subproblems $A_1 \cdots A_{k^*}$, $A_{k^*+1} \cdots A_n$;
2. the **fixed** cost $p_0 p_{k^*} p_n$ of multiplying $(A_1 \cdots A_{k^*})$ by $(A_{k^*+1} \cdots A_n)$.

- ▶ *If a solution to a subproblem was not optimal, replacing it by a better one in the overall solution would yield a cheaper overall solution, thus contradicting optimality of $A_{1,n}$.*

⇒ Hence $(A_1 \cdots A_{k^*})$, $(A_{k^*+1} \cdots A_n)$ must be **optimal** parenthesizations themselves.

Recursive computation of $A_{1,n}$

- ▶ **Idea:** compute the cost of the optimal parenthesization **recursively**.
- ▶ **Issue:** we do not know k^* !
- ▶ **Solution:** consider **every** possible value of k .
 - ▶ So we must solve $n - 1$ *large* subproblems, one for every $1 \leq k \leq n - 1$; each *large* subproblem involves solving two subproblems, that is, $A_{1,k}$, $A_{k+1,n}$, and combining them.



Exponential-time recursion

Notation: $T(n)$ = **time** required to optimally parenthesize a product of n matrices.

- ▶ At level 0, there are $n - 1$ *large* subproblems. Finding the one that incurs the minimum cost requires time $\Omega(1)$.
- ▶ The k -th *large* subproblem at level 1 requires time $T(k) + T(n - k) + \Omega(1)$.
- ▶ Note that $T(1) \geq 1$, $T(2) \geq 2$.
- ▶ Therefore,

$$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n - k) + 1).$$

- ▶ Then $T(n) \geq T(n - 1) + T(n - 2)$.
 - ▶ Hence $T(n) \geq F_n$ (see our argument for $P(n)$).
- \Rightarrow The recursive algorithm requires $\Omega(2^{n/2})$ time.

Today

- 1 Recap
- 2 Matrix chain multiplication
- 3 A first attempt: brute-force
- 4 A second attempt: divide and conquer
- 5 A Dynamic Programming (DP) solution**
- 6 Organizing DP computations

Are we really that far from an efficient solution?

Recall Fibonacci problem from HW1: exponential recursive algorithm, **polynomial** iterative solution

How?

1. **Overlapping subproblems:** spectacular redundancy in computations of recursion tree
2. **Easy-to-compute recurrence** for combining the smaller subproblems: $F_n = F_{n-1} + F_{n-2}$
3. **Small number of subproblems:** only solved $n - 1$ subproblems.
4. **Iterative, bottom-up computations:** we computed the subproblems from smallest (F_0, F_1) to largest (F_n) , iteratively.

Elements of DP in matrix chain multiplication

Our problem exhibits similar properties.

1. We showed **overlapping subproblems**.
2. We have implicitly formulated a **recurrence** for the cost of the optimal parenthesization in terms of the costs of the optimal parenthesizations of appropriate subproblems. We will show that the recurrence can be computed in **polynomial** time, given solutions to subproblems.
3. We will show a **polynomial** number of subproblems..
4. We will solve the subproblems in a **bottom-up** fashion, from smallest to largest.

The cost of multiplying two matrices

- ▶ Recall that A_i is a $p_{i-1} \times p_i$ matrix. Hence
 - ▶ $(A_i \cdots A_k)$ is a $p_{i-1} \times p_k$ matrix,
 - ▶ $(A_{k+1} \cdots A_j)$ is a $p_k \times p_j$ matrix.
- ▶ Then the #scalar multiplications required for computing the product $(A_i \cdots A_k)(A_{k+1} \cdots A_j)$ is

$$p_{i-1}p_kp_j.$$

Introducing subproblems: a first attempt

For $1 \leq j \leq n$, define

$OPT(1, j) =$ **optimal cost** for computing $A_1 \cdots A_j$

$$OPT(1, j) = \begin{cases} 0 & , \text{ if } j = 1 \\ \min_{1 \leq k < j} \{ OPT(1, k) + OPT(k + 1, j) + p_0 p_k p_j \} & , \text{ if } j > 1 \end{cases}$$

△ Does not work: the subproblems are **not** both of the same form as the original problem (*why?*).

Introducing more subproblems

For $1 \leq i \leq j \leq n$, define

$OPT(i, j) =$ **optimal cost** for computing $A_i \cdots A_j$

$$OPT(i, j) = \begin{cases} 0 & , \text{ if } i = j \\ \min_{i \leq k < j} \left\{ OPT(i, k) + OPT(k + 1, j) + p_{i-1}p_kp_j \right\} & , \text{ if } i < j \end{cases}$$

Remark 4.

- ▶ Only $\Theta(n^2)$ subproblems.
- ▶ If subproblems are computed from smaller to larger, then only $\Theta(j - i) = \Theta(n)$ work per subproblem: each term inside the min computation requires time $O(1)$ (why?).

Today

- 1 Recap
- 2 Matrix chain multiplication
- 3 A first attempt: brute-force
- 4 A second attempt: divide and conquer
- 5 A Dynamic Programming (DP) solution
- 6 Organizing DP computations**

Bottom-up computation of subproblems

Define matrix $M[1 : n, 1 : n]$, $S[1 : n - 1, 2 : n]$ such that

$$\begin{aligned}M[i, j] &= OPT(i, j), && \text{for } 1 \leq i \leq j \leq n \\S[i, j] &= k, \text{ if } A_{i,j} = A_{i,k}A_{k+1,j}, && \text{for } 1 \leq i < j \leq n\end{aligned}$$

- ▶ Only need fill in the **upper triangle** of M , where $i \leq j$
- ▶ Start from the main diagonal, proceed diagonal by diagonal
- ▶ Last entry to fill in: $M[1, n]$, the cost of the optimal parenthesization of the entire product $A_1 \cdots A_n$
- ▶ **Running time:** $O(n^3)$
 - ▶ $\Theta(n^2)$ entries to fill in
 - ▶ each entry requires $\Theta(j - i) = O(n)$ work
- ▶ **Space:** $\Theta(n^2)$

Example

Input

- ▶ 6×1 matrix A_1
- ▶ 1×5 matrix A_2
- ▶ 5×2 matrix A_3
- ▶ 2×3 matrix A_4

Output

- ▶ the cost of the optimal parenthesization of $A_1A_2A_3A_4$
(by filling in the dynamic programming table M)

Computing the cost of the optimal parenthesization in $O(n^3)$ (from CLRS)

MATRIX-CHAIN-ORDER (p)

```
1   $n = p.length - 1$ 
2  let  $m[1..n, 1..n]$  and  $s[1..n - 1, 2..n]$  be new tables
3  for  $i = 1$  to  $n$ 
4       $m[i, i] = 0$ 
5  for  $l = 2$  to  $n$            //  $l$  is the chain length
6      for  $i = 1$  to  $n - l + 1$ 
7           $j = i + l - 1$ 
8           $m[i, j] = \infty$ 
9          for  $k = i$  to  $j - 1$ 
10              $q = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$ 
11             if  $q < m[i, j]$ 
12                  $m[i, j] = q$ 
13                  $s[i, j] = k$ 
14  return  $m$  and  $s$ 
```


Reconstructing the optimal parenthesization (from CLRS)

```
PRINT-OPTIMAL-PARENS ( $s, i, j$ )  
1  if  $i == j$   
2     print " $A$ ";  
3  else print "("  
4     PRINT-OPTIMAL-PARENS ( $s, i, s[i, j]$ )  
5     PRINT-OPTIMAL-PARENS ( $s, s[i, j] + 1, j$ )  
6     print ")"
```

Memoized recursion

Use the original recursive algorithm together with M :

- ▶ initialize M to ∞ above the main diagonal and to 0 on the main diagonal.
- ▶ to solve a subproblem, look up its value in M
 - ▶ if it is ∞ , solve the subproblem **and** store its cost in M ;
 - ▶ else, directly use its value from M .

Remark 5.

- ▶ *The memoized recursive algorithm solves every subproblem **once**, thus overcoming the main source of inefficiency of the original recursive algorithm.*
- ▶ *Running time: $O(n^3)$.*

Memoized recursion pseudocode (from CLRS)

MEMOIZED-MATRIX-CHAIN(p)

```
1  $n = p.length - 1$ 
2 let  $m[1..n, 1..n]$  be a new table
3 for  $i = 1$  to  $n$ 
4   for  $j = i$  to  $n$ 
5      $m[i, j] = \infty$ 
6 return LOOKUP-CHAIN( $m, p, 1, n$ )
```

LOOKUP-CHAIN(m, p, i, j)

```
1 if  $m[i, j] < \infty$ 
2   return  $m[i, j]$ 
3 if  $i == j$ 
4    $m[i, j] = 0$ 
5 else for  $k = i$  to  $j - 1$ 
6    $q =$  LOOKUP-CHAIN( $m, p, i, k$ )
       + LOOKUP-CHAIN( $m, p, k + 1, j$ ) +  $p_{i-1} p_k p_j$ 
7   if  $q < m[i, j]$ 
8      $m[i, j] = q$ 
9 return  $m[i, j]$ 
```

Dynamic programming vs Divide & Conquer

- ▶ They both combine solutions to subproblems to generate a solution to the whole problem.
- ▶ However, divide and conquer starts with a large problem and divides it into small pieces.
- ▶ While dynamic programming works from the bottom up, solving the smallest subproblems first and building optimal solutions to steadily larger problems.