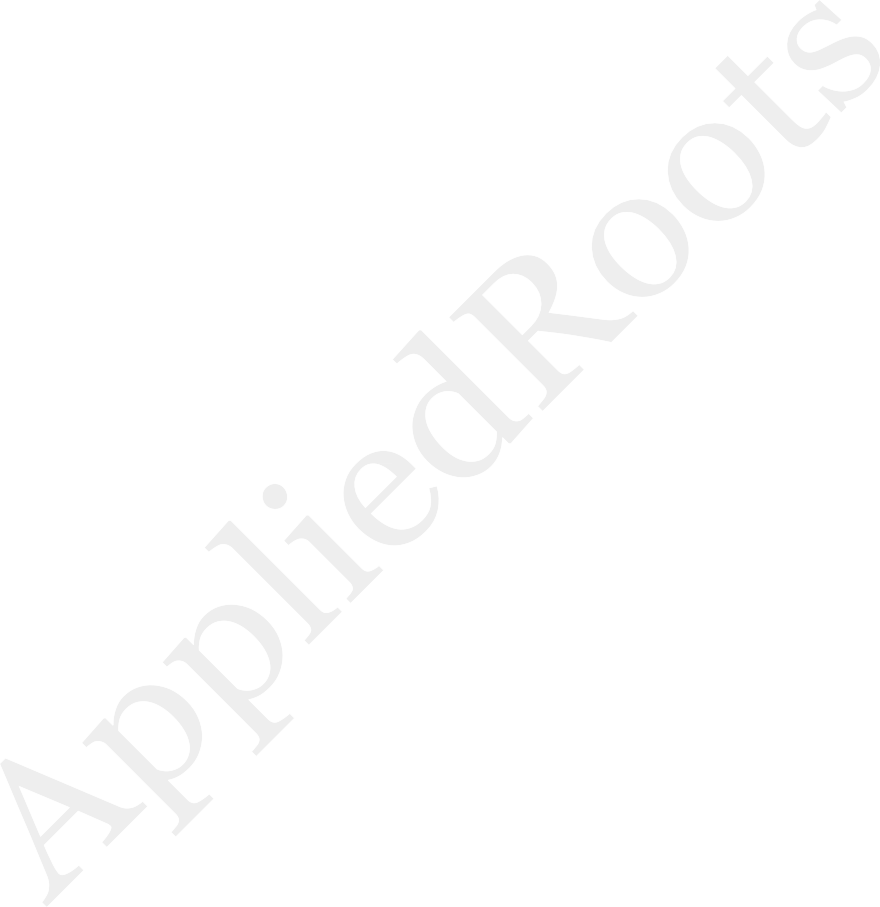


Dynamic Programming: Matrix chainmultiplication

(CLRS 15.2)

# The problem

Given a sequence of matrices *A*1*, A*2*, A*3*, ..., An*, find the best way (using the minimal number of multiplications) to compute their product.

* + Isn’t there only one way? ((· · · ((*A*1 · *A*2) · *A*3) · · ·) · *An*)



* + No, matrix multiplication is *associative*.

e.g. *A*1 · (*A*2 · (*A*3 · (· · · (*An−*1 · *An*) · · ·))) yields the same matrix.

* + Different multiplication orders do not cost the same:
    - Multiplying *p* × *q* matrix *A* and *q* × *r* matrix *B* takes *p* · *q* · *r* multiplications; result is a

*p* × *r* matrix.

* + - Consider multiplying 10 × 100 matrix *A*1 with 100 × 5 matrix *A*2 and 5 × 50 matrix *A*3.

– (*A*1 · *A*2) · *A*3 takes 10 · 100 · 5 + 10 · 5 · 50 = 7500 multiplications.

– *A*1 · (*A*2 · *A*3) takes 100 · 5 · 50 + 10 · 50 · 100 = 75000 multiplications.

# Notation

* + In general, let *Ai* be *pi−*1 × *pi* matrix.
  + Let *m*(*i, j*) denote minimal number of multiplications needed to compute *Ai* · *Ai*+1 · ·· *Aj*
  + We want to compute *m*(1*, n*).

# Recursive algorithm

* + Assume that someone tells us the position of the **last** product, say *k*. Then we have to compute recursively the best way to multiply the chain from *i* to *k*, and from *k* + 1 to *j*, and add the cost of the final product. This means that

*m*(*i, j*) = *m*(*i, k*) + *m*(*k* + 1*, j*) + *pi−*1 · *pk* · *pj*

* + If noone tells us *k*, then we have to try all possible values of *k* and pick the best solution.
  + Recursive formulation of *m*(*i, j*):

*m*(*i, j*) = 0 If *i* = *j*

min*i≤k<j*{*m*(*i, k*) + *m*(*k* + 1*, j*) + *pi−*1 · *pk* · *pj*} If *i < j*

* + To go from the recursive formulation above to a program is pretty straightforward:



Matrix-chain(*i, j*)

IF *i* = *j* THEN return 0

*m* = ∞

FOR *k* = *i* TO *j* − 1 DO

q = Matrix-chain(*i, k*) + Matrix-chain(*k* + 1*, j*) +*pi−*1 · *pk* · *pj*

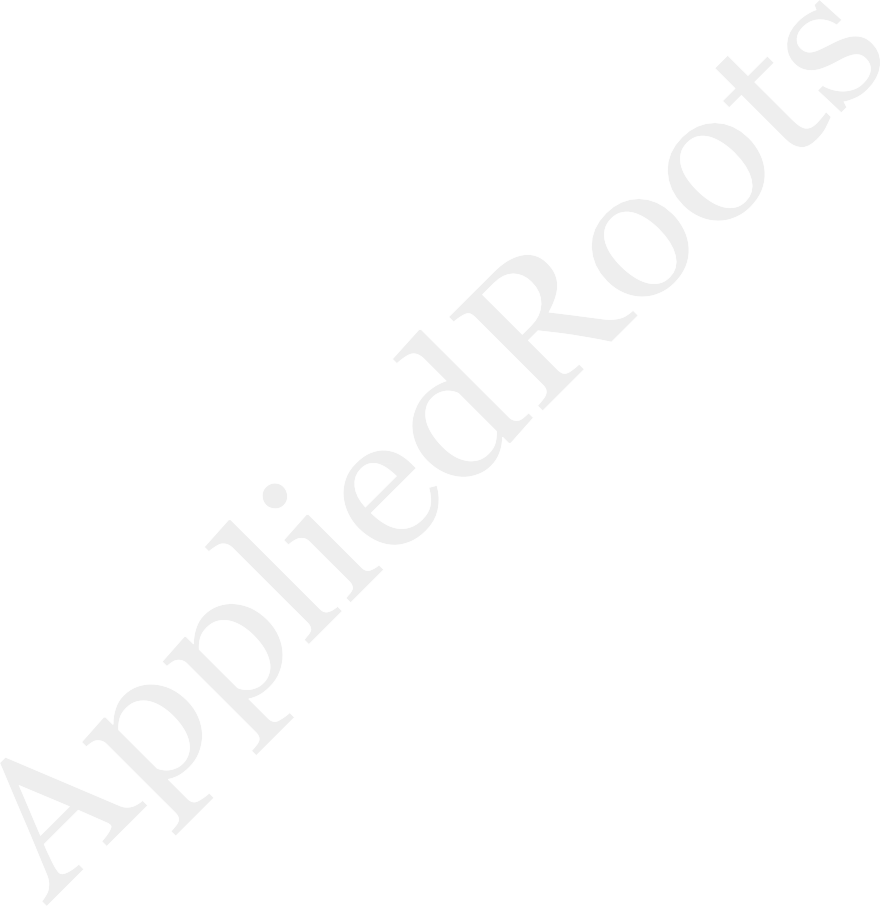
IF *q < m* THEN *m* = *q*

OD

Return *m*

END Matrix-chain

Return Matrix-chain(1*, n*)



* Running time:

*n−*1

*T* (*n*) = (*T*(*k*) + *T* (*n* − *k*) + *O*(1))

*k*=1

*n−*1

= 2 · *T*(*k*) + *O*(*n*)

*k*=1

≥ 2 · *T* (*n* − 1)

≥ 2 · 2 · *T* (*n* − 2)

≥ 2 · 2 · 2 *. . .*

= 2*n*

* Exponential is ... SLOW!
* Problem is that we compute the same result over and over again.

**–** Example: Recursion tree for Matrix-chain(1*,* 4)

1,4

1,1 2,4 1,2 3,4 1,3 4,4

2,2 3,4 2,3 4,4 1,1 2,2 3,3 4,4 1,1 2,3 1,2 3,3

3,3 4,4 2,2 3,3 2,2 3,3 1,1 2,2

For example, we compute Matrix-chain(3*,* 4) twice.

# Dynamic programming with a table and recursion

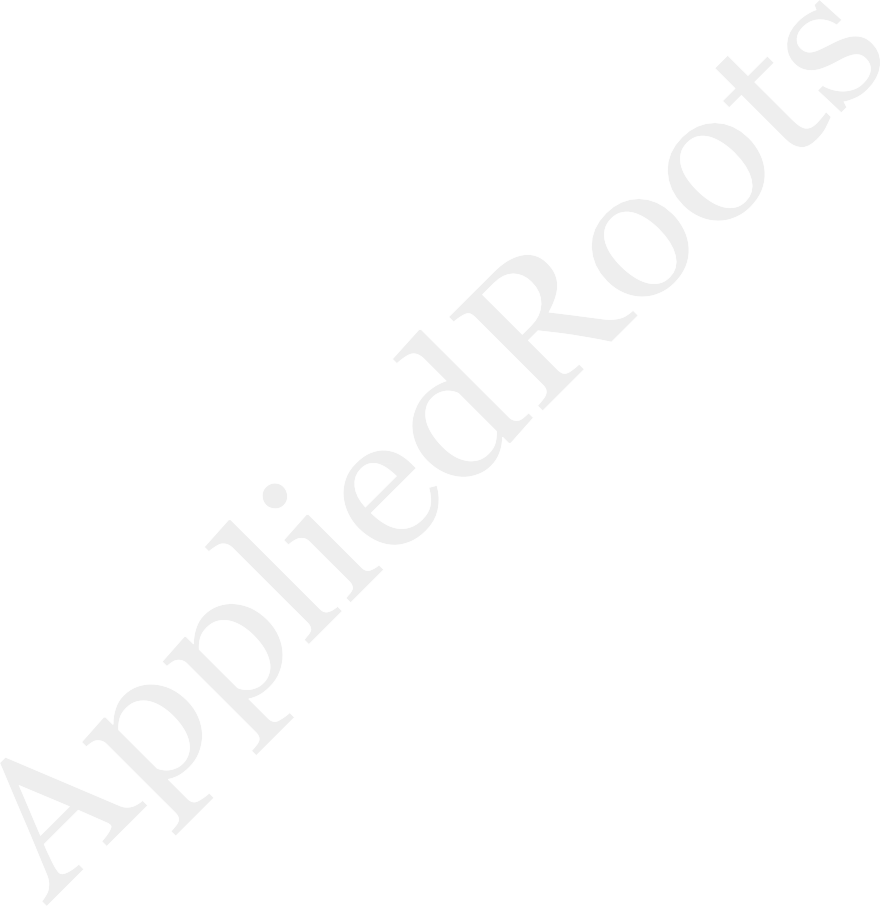
* + Solution is to “remember” the values we have already computed in a table. This is called *memoization*. We’ll have a table T[1..n][1..n] such that T[i][j] stores the solution to problem Matrix-CHAIN(i,j). Initially all entries will be set to ∞.

FOR *i* = 1 to *n* DO FOR *j* = *i* to *n* DO

*T* [*i*][*j*] = ∞

OD

OD

* + The code for MATRIX-CHAIN(i,j) stays the same, except that it now uses the table. The first thing MATRIX-CHAIN(i,j) does is to check the table to see if *T* [*i*][*j*] is already computed. Is so, it returns it, otherwise, it computes it and writes it in the table. Below is the updated

code.

Matrix-chain(*i, j*)

IF *T* [*i*][*j*] *<* ∞ THEN return *T* [*i*][*j*] IF *i* = *j* THEN *T* [*i*][*j*] = 0, return 0 *m* = ∞

FOR *k* = *i* to *j* − 1 DO

q = Matrix-chain(*i, k*) + Matrix-chain(*k* + 1*, j*)+*pi−*1 · *pk* · *pj*

IF *q < m* THEN *m* = *q*

OD

*T* [*i*][*j*] = *m*

return *m*

END Matrix-chain

return Matrix-chain(1*, n*)

* + The table will prevent a subproblem MATRIX-CHAIN(i,j) to be computed more than once.
  + Running time:
    - Θ(*n*2) different calls to matrix-chain(*i, j*).
    - The first time a call is made it takes *O*(*n*) time, *not* counting recursive calls.
    - When a call has been made once it costs *O*(1) time to make it again.

⇓

*O*(*n*3) time

* + - Another way of thinking about it: Θ(*n*2) total entries to fill, it takes *O*(*n*) to fill one.

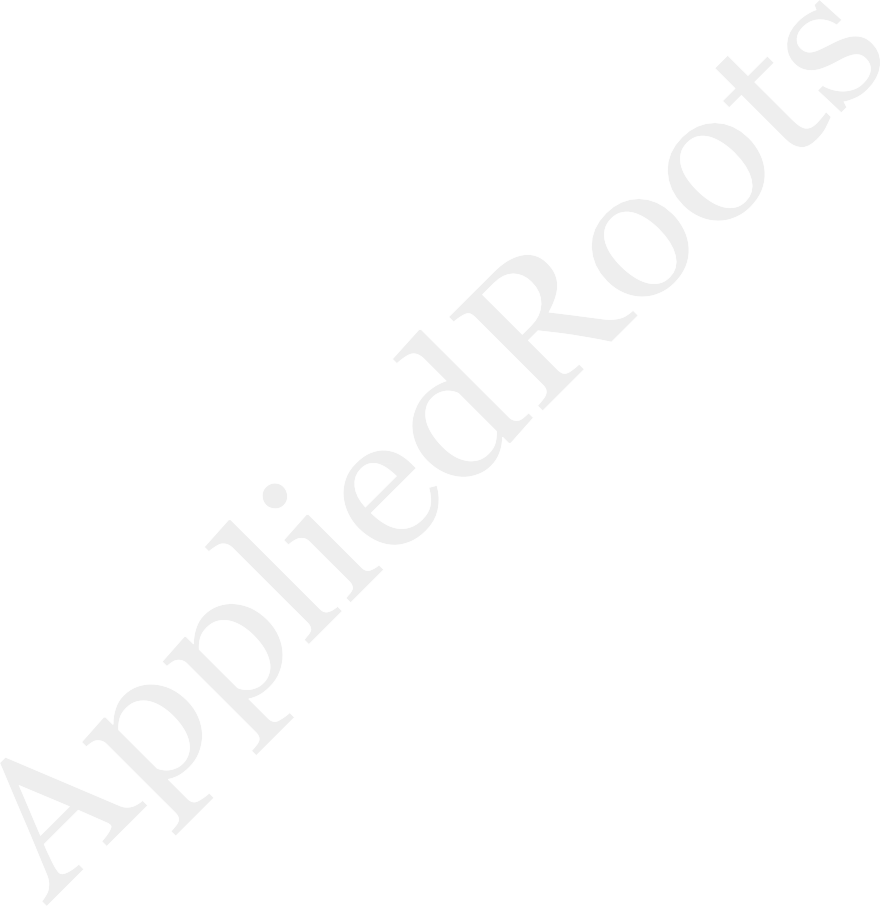
# Dynamic Programming without recursion

* + Often dynamic programming is presented as filling up a table from the bottom, in such a way that makes recursion unnecessary. Avoiding recursion is perhaps elegant, and necessary 20-30 years ago when programming languages did not include support for recursion.
  + Personally, I like top-down recursive thinking, and I think compilers are pretty effective at optimizing recursion.
  + Solution for Matrix-chain without recursion: Key is that *m*(*i, j*) only depends on *m*(*i, k*) and

*m*(*k* + 1*, j*) where *i* ≤ *k < j* ⇒ if we have computed them, we can compute *m*(*i, j*)

* + - We can easily compute *m*(*i, i*) for all 1 ≤ *i* ≤ *n* (*m*(*i, i*) = 0)
    - Then we can easily compute *m*(*i, i* + 1) for all 1 ≤ *i* ≤ *n* − 1

*m*(*i, i* + 1) = *m*(*i, i*) + *m*(*i* + 1*, i* + 1) + *pi−*1 · *pi* · *pi*+1



* + - Then we can compute *m*(*i, i* + 2) for all 1 ≤ *i* ≤ *n* − 2

*m*(*i, i* + 2) = min{*m*(*i, i*) + *m*(*i* + 1*, i* + 2) + *pi−*1 · *pi* · *pi*+2*, m*(*i, i* + 1) + *m*(*i* + 2*, i* + 2) +

*pi−*1 · *pi*+1 · *pi*+2}

.

* + - Until we compute *m*(1*, n*)
    - Computation order:

j

1 2 3 4 5 6 7

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|  | 1 | 2 | 3 | 4 | 5 | 6 |
|  |  | 1 | 2 | 3 | 4 | 5 |
|  |  |  | 1 | 2 | 3 | 4 |
|  |  |  |  | 1 | 2 | 3 |
|  |  |  |  |  | 1 | 2 |
|  |  |  |  |  |  | 1 |

1

2

3

i

4

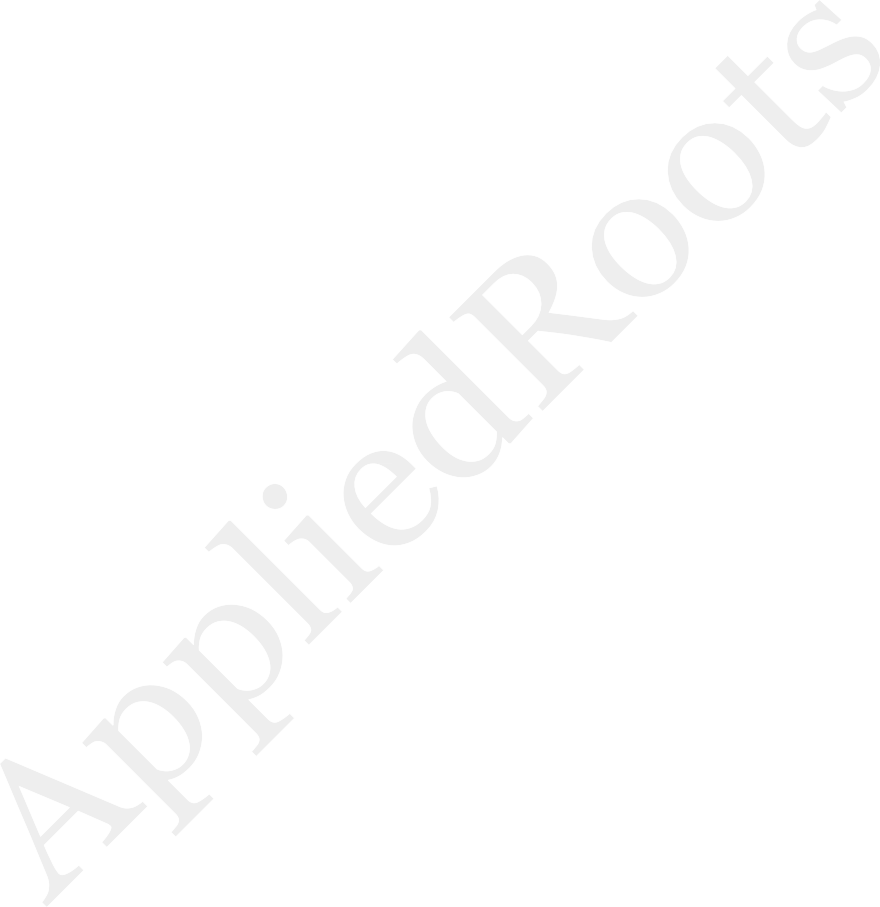
5

6

7

* + Program:

− Computation order

* + Analysis: *O*(*n*2) entries, *O*(*n*) time to compute each ⇒ *O*(*n*3).

FOR *i* = 1 to *n* DO

*T* [*i*][*i*] = 0

OD

FOR *l* = 1 to *n* − 1 DO FOR *i* = 1 to *n* − *l* DO

*j* = *i* + *l T*

[*i*][*j*] = ∞

FOR *k* = 1 to *j* −1 DO

*q* = *T* [*i*][*k*] + *T* [*k* + 1][*j*] + *pi−*1 · *pk* · *pj*

IF *q < T* [*i*[*j*] THEN *T* [*i*][*j*] = *q*

OD

OD

OD

# Extensions

Above we only computed the best way to multiply the chain (with the smallest number of opera- tions). The algorithm can be extended to compute the actual order of multiplications corresponding to this optimal cost (we’ll do this as homework or in-class exercise).

This is a simplification that is often done: focus on computing the optimal cost, and leave the details of computing the solution corresponding to that optimal cost for later.