20. Dynamic Programming I

Memoization, Optimal Substructure, Overlapping Sub-Problems, Dependencies, General Procedure. Examples: Fibonacci, Rod Cutting, Longest Ascending Subsequence, Longest Common Subsequence, Edit Distance, Matrix Chain Multiplication (Strassen)

[Ottman/Widmayer, Kap. 1.2.3, 7.1, 7.4, Cormen et al, Kap. 15]

Fibonacci Numbers

 (again)

$$F_n := \begin{cases} n & \text{if } n < 2 \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

Analysis: why ist the recursive algorithm so slow?

Algorithm FibonacciRecursive(n)

Input: $n \geq 0$
Output: n-th Fibonacci number

if $n < 2$ **then**
  $f \leftarrow n$
else
  $f \leftarrow$ FibonacciRecursive$(n-1)$ + FibonacciRecursive$(n-2)$
return f

Analysis

$T(n)$: Number executed operations.

- $n = 0, 1$: $T(n) = \Theta(1)$

Analysis

$T(n)$: Number executed operations.

- $n = 0, 1$: $T(n) = \Theta(1)$
- $n \geq 2$: $T(n) = T(n-2) + T(n-1) + c$.

Analysis

$T(n)$: Number executed operations.

- $n = 0, 1$: $T(n) = \Theta(1)$
- $n \geq 2$: $T(n) = T(n-2) + T(n-1) + c$.

$$T(n) = T(n-2) + T(n-1) + c \geq 2T(n-2) + c \geq 2^{n/2}c' = (\sqrt{2})^n c'$$

Analysis

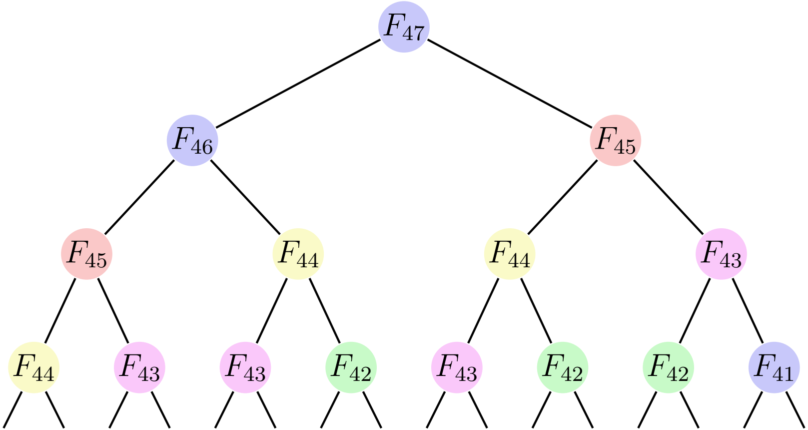
$T(n)$: Number executed operations.

- $n = 0, 1$: $T(n) = \Theta(1)$
- $n \geq 2$: $T(n) = T(n-2) + T(n-1) + c$.

$$T(n) = T(n-2) + T(n-1) + c \geq 2T(n-2) + c \geq 2^{n/2}c' = (\sqrt{2})^n c'$$

Algorithm is **exponential** in n.

Reason (visual)



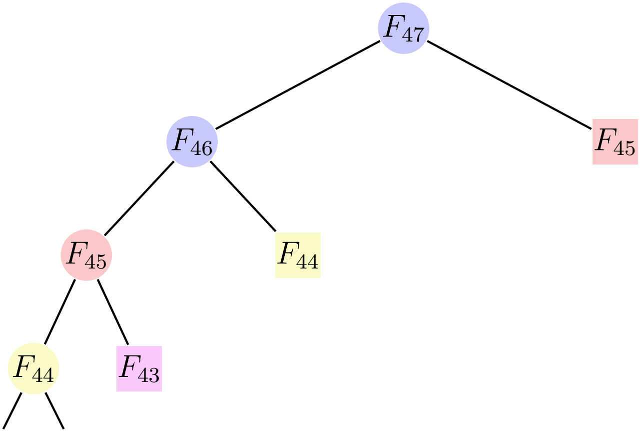
Nodes with same values are evaluated (too) often.

Memoization

Memoization (sic) saving intermediate results.

- Before a subproblem is solved, the existence of the corresponding intermediate result is checked.
- If an intermediate result exists then it is used.
- Otherwise the algorithm is executed and the result is saved accordingly.

Memoization with Fibonacci



Rectangular nodes have been computed before.

Algorithm FibonacciMemoization(n)

Input: $n \geq 0$
Output: n-th Fibonacci number

if $n \leq 2$ **then**
    $f \leftarrow 1$
else if $\exists$memo$[n]$ **then**
    $f \leftarrow$ memo$[n]$
else
    $f \leftarrow$ FibonacciMemoization$(n-1)$ + FibonacciMemoization$(n-2)$
    memo$[n] \leftarrow f$

return f

Analysis

Computational complexity:

$$T(n) = T(n-1) + c = ... = \mathcal{O}(n).$$

because after the call to $f(n-1)$, $f(n-2)$ has already been computed.
A different argument: $f(n)$ is computed exactly once recursively for each n.
Runtime costs: n calls with $\Theta(1)$ costs per call $n \cdot c \in \Theta(n)$. The recursion vanishes from the running time computation.
Algorithm requires $\Theta(n)$ memory.[29]

[29] But the naive recursive algorithm also requires $\Theta(n)$ memory implicitly.

Looking closer ...

... the algorithm computes the values of F_1, F_2, F_3,... in the **top-down** approach of the recursion.

Can write the algorithm **bottom-up**. This is characteristic for **dynamic programming**.

Algorithm FibonacciBottomUp(n)

Input: $n \geq 0$
Output: n-th Fibonacci number

$F[1] \leftarrow 1$
$F[2] \leftarrow 1$
for $i \leftarrow 3, \ldots, n$ **do**
  $F[i] \leftarrow F[i-1] + F[i-2]$
return $F[n]$

Dynamic Programming: Idea

- Divide a complex problem into a reasonable number of sub-problems
- The solution of the sub-problems will be used to solve the more complex problem
- Identical problems will be computed only once

Dynamic Programming Consequence

Identical problems will be computed only once

$\Rightarrow$ Results are saved

We trade speed against memory consumption

Dynamic Programming = Divide-And-Conquer ?

- In both cases the original problem can be solved (more easily) by utilizing the solutions of sub-problems. The problem provides **optimal substructure**.
- Classical Divide-And-Conquer algorithms (such as Mergesort): sub-problems are independent; their solutions are required only once in the algorithm.
- DP: sub-problems are dependent. The problem is said to have **overlapping sub-problems** that are required multiple-times in the algorithm.
- In order to avoid redundant computations, results are tabulated. For **sub-problems there must not be any circular dependencies**.

Dynamic Programming: Description

1. Use a **DP-table** with information to the subproblems.
   Dimension of the table? Semantics of the entries?
2. Computation of the **base cases**.
   Which entries do not depend on others?
3. Determine **computation order**.
   In which order can the entries be computed such that dependencies are fulfilled?
4. Read-out the **solution**
   How can the solution be read out from the table?

Runtime (typical) = number entries of the table times required operations per entry.

Dynamic Programing: Description (Fibonacci)

1. Dimension of the table? Semantics of the entries?
2. Which entries do not depend on other entries?
3. Computation order?
4. Reconstruction of a solution?

Dynamic Programing: Description (Fibonacci)

1.
Dimension of the table? Semantics of the entries?

$n \times 1$ table. nth entry contains nth Fibonacci number.

2.
Which entries do not depend on other entries?

3.
Computation order?

4.
Reconstruction of a solution?

Dynamic Programing: Description (Fibonacci)

1. Dimension of the table? Semantics of the entries?

   $n \times 1$ table. nth entry contains nth Fibonacci number.

2. Which entries do not depend on other entries?

   Values F_1 and F_2 can be computed easily and independently.

3. Computation order?

4. Reconstruction of a solution?

Dynamic Programing: Description (Fibonacci)

1. **Dimension of the table? Semantics of the entries?**

   $n \times 1$ table. nth entry contains nth Fibonacci number.

2. **Which entries do not depend on other entries?**

   Values F_1 and F_2 can be computed easily and independently.

3. **Computation order?**

   F_i with increasing i.

4. **Reconstruction of a solution?**

Dynamic Programing: Description (Fibonacci)

1. **Dimension of the table? Semantics of the entries?**

   $n \times 1$ table. nth entry contains nth Fibonacci number.

2. **Which entries do not depend on other entries?**

   Values F_1 and F_2 can be computed easily and independently.

3. **Computation order?**

   F_i with increasing i.

4. **Reconstruction of a solution?**

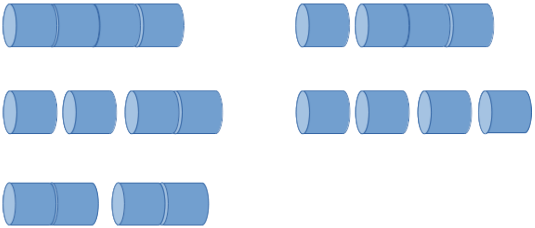
   F_n is the n-th Fibonacci number.

Rod Cutting

- Rods (metal sticks) are cut and sold.
- Rods of length $n \in \mathbb{N}$ are available. A cut does not provide any costs.
- For each length $l \in \mathbb{N}$, $l \leq n$ known is the value $v_l \in \mathbb{R}^+$
- Goal: cut the rods such (into $k \in \mathbb{N}$ pieces) that

$$\sum_{i=1}^{k} v_{l_i} \text{ is maximized subject to } \sum_{i=1}^{k} l_i = n.$$

Rod Cutting: Example



Possibilities to cut a rod of length 4 (without permutations)

Length	0	1	2	3	4
Price	0	2	3	8	9

$\Rightarrow$ Best cut: 3 + 1 with value 10.

How to Find the DP Algorithm.

0. Exact formulation of the wanted solution
1. Define sub-problems, reformulate (0.) as sub-problem
2. Recursion: relate subproblems by enumerating of local properties
3. Determine the dependencies of the sub-problems
4. Solve the problem
   Running time = #sub-problems $\times$ time/sub-problem

Structure of the problem

0. **Wanted:** r_n = maximal value of rod (cut or as a whole) with length n.
1. **sub-problems**: maximal value r_k for each $0 \leq k < n$
2. Local property: length of the first piece
   Recursion

$$r_k = \max\{v_i + r_{k-i} : 0 < i \leq k\}, \quad k > 0$$
$$r_0 = 0$$

3. **Dependency:** r_k depends (only) on values v_i, $1 \leq i \leq k$ and the optimal cuts r_i, $i < k$.
4. **Solution** in r_n. DP running time: $\Theta(n^2)$

Algorithm RodCut(v,n) (without memoization)

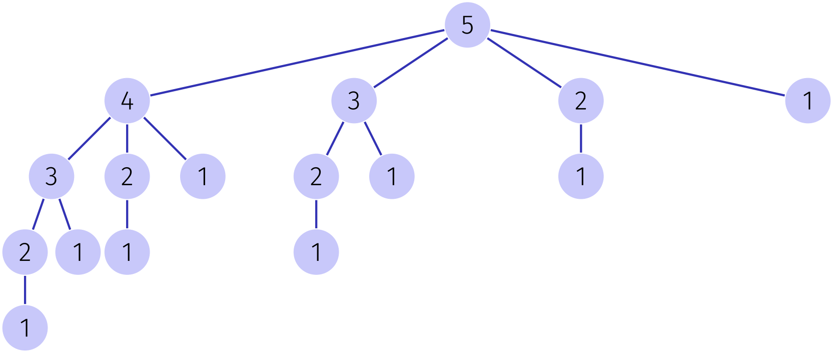
Input: $n \geq 0$, Prices v
Output: best value

$q \leftarrow 0$
if $n > 0$ **then**
  for $i \leftarrow 1, \ldots, n$ **do**
    $q \leftarrow \max\{q, v_i + \text{RodCut}(v, n-i)\}$;
return q

Running time $T(n) = \sum_{i=0}^{n-1} T(i) + c \quad \Rightarrow$[30] $\quad T(n) \in \Theta(2^n)$

[30] $T(n) = T(n-1) + \sum_{i=0}^{n-2} T(i) + c = T(n-1) + (T(n-1) - c) + c = 2T(n-1) \quad (n > 0)$

Recursion Tree



Algorithm RodCutMemoized(m, v, n)

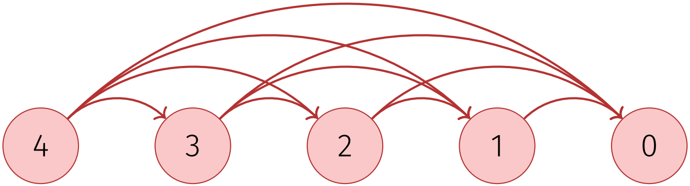
Input: $n \geq 0$, Prices v, Memoization Table m
Output: best value

$q \leftarrow 0$
if $n > 0$ **then**
  if $\exists\ m[n]$ **then**
    $q \leftarrow m[n]$
  else
    for $i \leftarrow 1, \ldots, n$ **do**
      $q \leftarrow \max\{q, v_i +$ RodCutMemoized$(m, v, n - i)\}$;
    $m[n] \leftarrow q$
return q

Running time $\sum_{i=1}^{n} i = \Theta(n^2)$

Subproblem-Graph

Describes the mutual dependencies of the subproblems



and must not contain cycles

Construction of the Optimal Cut

- During the (recursive) computation of the optimal solution for each $k \leq n$ the recursive algorithm determines the optimal length of the first rod
- Store the lenght of the first rod in a separate table of length n

Bottom-up Description with the example

1. Dimension of the table? Semantics of the entries?

2. Which entries do not depend on other entries?

3. Computation order?

4. Reconstruction of a solution?

Bottom-up Description with the example

1. **Dimension of the table? Semantics of the entries?**

   $n \times 1$ table. nth entry contains the best value of a rod of length n.

2. **Which entries do not depend on other entries?**

3. **Computation order?**

   .

4. **Reconstruction of a solution?**

Bottom-up Description with the example

1. Dimension of the table? Semantics of the entries?

   $n \times 1$ table. nth entry contains the best value of a rod of length n.

2. Which entries do not depend on other entries?

   Value r_0 is 0

3. Computation order?

   .

4. Reconstruction of a solution?

Bottom-up Description with the example

1. Dimension of the table? Semantics of the entries?

   $n \times 1$ table. nth entry contains the best value of a rod of length n.

2. Which entries do not depend on other entries?

   Value r_0 is 0

3. Computation order?

   r_i, $i = 1, \dots, n$.

4. Reconstruction of a solution?

Bottom-up Description with the example

1. **Dimension of the table? Semantics of the entries?**

   $n \times 1$ table. nth entry contains the best value of a rod of length n.

2. **Which entries do not depend on other entries?**

   Value r_0 is 0

3. **Computation order?**

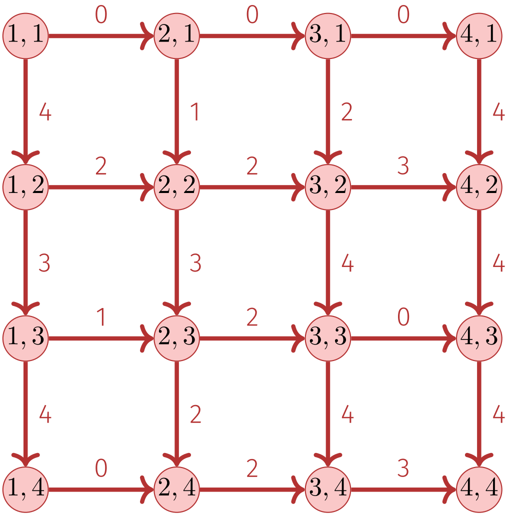
   r_i, $i = 1, \ldots, n$.

4. **Reconstruction of a solution?**

   r_n is the best value for the rod of length n.

Rabbit!

A rabbit sits on cite $(1,1)$ of an $n \times n$ grid. It can only move to east or south. On each pathway there is a number of carrots. How many carrots does the rabbit collect maximally?

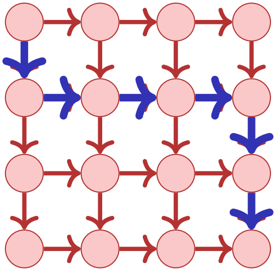


Rabbit!

Number of possible paths?

- Choice of $n-1$ ways to south out of $2n-2$ ways overal.

$\Rightarrow$ No chance for a naive algorithm



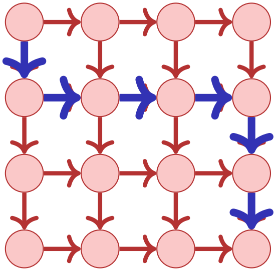
The path 100011
(1:to south, 0: to east)

Rabbit!

Number of possible paths?

- Choice of $n-1$ ways to south out of $2n-2$ ways overal.
- $$\binom{2n-2}{n-1} \in \Omega(2^n)$$

$\Rightarrow$ No chance for a naive algorithm



The path 100011
(1:to south, 0: to east)

Recursion

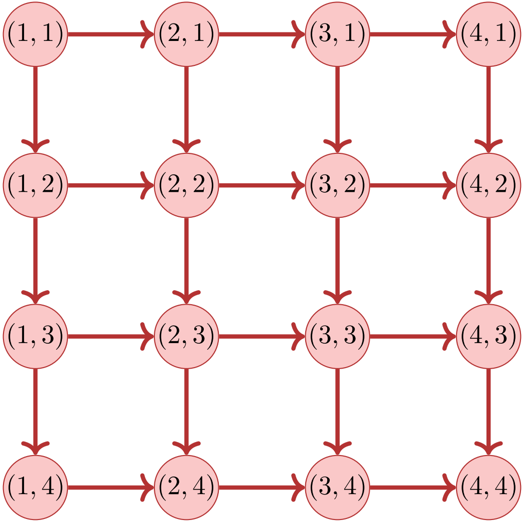
Wanted: $T_{1,1}$ **= maximal number carrots from** $(1,1)$ **to** (n,n)**.**

Let $w_{(i,j)-(i',j')}$ number of carrots on egde from (i,j) to (i',j').

Recursion (maximal number of carrots from (i,j) to (n,n)

$$T_{ij} = \begin{cases} \max\{w_{(i,j)-(i,j+1)} + T_{i,j+1}, w_{(i,j)-(i+1,j)} + T_{i+1,j}\}, & i < n, j < n \\ w_{(i,j)-(i,j+1)} + T_{i,j+1}, & i = n, j < n \\ w_{(i,j)-(i+1,j)} + T_{i+1,j}, & i < n, j = n \\ 0 & i = j = n \end{cases}$$

Graph of Subproblem Dependencies



Bottom-up Description with the example

1. **Dimension of the table? Semantics of the entries?**

   Table T with size $n \times n$. Entry at i, j provides the maximal number of carrots from (i, j) to (n, n).

2. **Which entries do not depend on other entries?**

   Value $T_{n,n}$ is 0

3. **Computation order?**

   $T_{i,j}$ with $i = n \searrow 1$ and for each i: $j = n \searrow 1$, (or vice-versa: $j = n \searrow 1$ and for each j: $i = n \searrow 1$).

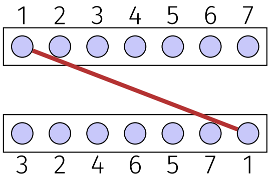
4. **Reconstruction of a solution?**

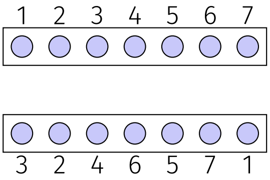
   $T_{1,1}$ provides the maximal number of carrots.

Longest Ascending Sequence (LAS)

Connect as many as possible fitting ports without lines crossing.

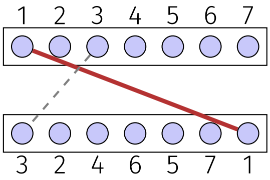
Longest Ascending Sequence (LAS)

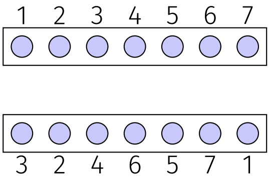




Connect as many as possible fitting ports without lines crossing.

Longest Ascending Sequence (LAS)





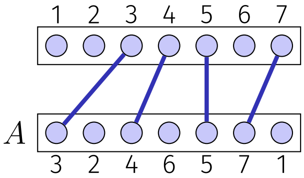
Connect as many as possible fitting ports without lines crossing.

Longest Ascending Sequence (LAS)

Connect as many as possible fitting ports without lines crossing.

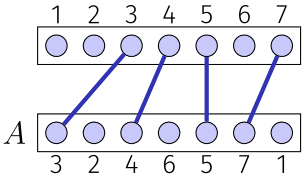
Formally

- Consider Sequence $A_n = (a_1, \dots, a_n)$.
- Search for a longest increasing subsequence of A_n.
- Examples of increasing subsequences: $(3, 4, 5)$, $(2, 4, 5, 7)$, $(3, 4, 5, 7)$, $(3, 7)$.



Formally

- Consider Sequence $A_n = (a_1, \ldots, a_n)$.
- Search for a longest increasing subsequence of A_n.
- Examples of increasing subsequences: $(3, 4, 5)$, $(2, 4, 5, 7)$, $(3, 4, 5, 7)$, $(3, 7)$.



Generalization: allow any numbers, even with duplicates (still only strictly increasing subsequences permitted). Example: $(2, 3, 3, 3, 5, 1)$ with increasing subsequence $(2, 3, 5)$.

First idea (Greedy)

Let L_i **= longest ascending subsequence of** A_i $(1 \leq i \leq n)$

Assumption: LAS L_k of A_k known. Compute L_{k+1} for A_{k+1}.

First idea (Greedy)

Let L_i **= longest ascending subsequence of** A_i $(1 \leq i \leq n)$

Assumption: LAS L_k of A_k known. Compute L_{k+1} for A_{k+1}.

Idea

$$L_{k+1} = \begin{cases} L_k \oplus a_{k+1} & \text{if } a_k > \max(L_k) \\ L_k & \text{otherwise?} \end{cases}$$

First idea (Greedy)

Let L_i = **longest ascending subsequence of** A_i $(1 \le i \le n)$

Assumption: LAS L_k of A_k known. Compute L_{k+1} for A_{k+1}.

Idea

$$L_{k+1} = \begin{cases} L_k \oplus a_{k+1} & \text{if } a_k > \max(L_k) \\ L_k & \text{otherwise?} \end{cases}$$

Counterexample

$A_5 = (1, 2, 5, 3, 4)$.
$A_3 = (1, 2, 5)$ with $L_3 = A_3$ and $L_4 = A_3$.

First idea (Greedy)

Let L_i = **longest ascending subsequence of** A_i $(1 \leq i \leq n)$

Assumption: LAS L_k of A_k known. Compute L_{k+1} for A_{k+1}.

Idea

$$L_{k+1} = \begin{cases} L_k \oplus a_{k+1} & \text{if } a_k > \max(L_k) \\ L_k & \text{otherwise?} \end{cases}$$

Counterexample

$A_5 = (1, 2, 5, 3, 4)$.
$A_3 = (1, 2, 5)$ with $L_3 = A_3$ and $L_4 = A_3$.

Greedy idea fails here: we cannot directly infer L_{k+1} from L_k.

Second idea. (Prefix)

Let L_i **= longest ascending subsequence of** A_i $(1 \leq i \leq n)$

Assumption: a LAS L_j that ends in a_j is known for each $j \leq k$. Now compute LAS L_{k+1} for $k+1$.

Second idea. (Prefix)

Let L_i **= longest ascending subsequence of** A_i $(1 \leq i \leq n)$

Assumption: a LAS L_j that ends in a_j is known for each $j \leq k$. Now compute LAS L_{k+1} for $k+1$.

Look at all fitting $L_{k+1} = L_j \oplus a_{k+1}$ $(j \leq k)$ and choose a longest sequence.

Second idea. (Prefix)

Let L_i **= longest ascending subsequence of** A_i $(1 \leq i \leq n)$

Assumption: a LAS L_j that ends in a_j is known for each $j \leq k$. Now compute LAS L_{k+1} for $k+1$.

Look at all fitting $L_{k+1} = L_j \oplus a_{k+1}$ $(j \leq k)$ and choose a longest sequence.

Example

$A_5 = (1, 2, 5, 3, 4)$.
$L_1 = (1)$, $L_2 = (1, 2)$, $L_3 = (1, 2, 5)$, $L_4 = (1, 2, 3)$, $L_5 = (1, 2, 3, 4)$.

This works with running time n^2 (and requires access to all sequences L_i.

Third approach

Let $M_{n,i}$ **= longest ascending subsequence of** A_i $(1 \leq i \leq n)$

Assumption: the LAS M_j for A_k, **that end with smallest element** are known for each of the lengths $1 \leq j \leq k$.

Third approach

Let $M_{n,i}$ **= longest ascending subsequence of** A_i $(1 \leq i \leq n)$

Assumption: the LAS M_j for A_k, **that end with smallest element** are known for each of the lengths $1 \leq j \leq k$.

Consider all fitting $M_{k,j} \oplus a_{k+1}$ $(j \leq k)$ and update the table of the LAS,that end with smallest possible element.

Third approach Example

Example: $A = (1, 1000, 1001, 4, 5, 2, 6, 7)$

A	LAT $M_{k,\cdot}$
1	(**1**)

Third approach Example

Example: $A = (1, 1000, 1001, 4, 5, 2, 6, 7)$

A	LAT $M_{k,\cdot}$
1	$(\mathbf{1})$
$+\ 1000$	$(1), (1, \mathbf{1000})$

Third approach Example

Example: $A = (1, 1000, 1001, 4, 5, 2, 6, 7)$

A	LAT $M_{k,\cdot}$
1	$(\mathbf{1})$
$+\ 1000$	$(1), (1, \mathbf{1000})$
$+\ 1001$	$(1), (1, 1000), (1, 1000, \mathbf{1001})$

Third approach Example

Example: $A = (1, 1000, 1001, 4, 5, 2, 6, 7)$

A	LAT $M_{k,\cdot}$
1	$(\mathbf{1})$
$+\,1000$	$(1), (1, \mathbf{1000})$
$+\,1001$	$(1), (1, 1000), (1, 1000, \mathbf{1001})$
$+\,4$	$(1), (1, \mathbf{4}), (1, 1000, 1001)$

Third approach Example

Example: $A = (1, 1000, 1001, 4, 5, 2, 6, 7)$

A	LAT $M_{k,\cdot}$
1	$(\mathbf{1})$
$+\ 1000$	$(1), (1, \mathbf{1000})$
$+\ 1001$	$(1), (1, 1000), (1, 1000, \mathbf{1001})$
$+\ 4$	$(1), (1, \mathbf{4}), (1, 1000, 1001)$
$+\ 5$	$(1), (1, 4), (1, 4, \mathbf{5})$

Third approach Example

Example: $A = (1, 1000, 1001, 4, 5, 2, 6, 7)$

A	LAT $M_{k,\cdot}$
1	$(\mathbf{1})$
$+\ 1000$	$(1), (1, \mathbf{1000})$
$+\ 1001$	$(1), (1, 1000), (1, 1000, \mathbf{1001})$
$+\ 4$	$(1), (1, \mathbf{4}), (1, 1000, 1001)$
$+\ 5$	$(1), (1, 4), (1, 4, \mathbf{5})$
$+\ 2$	$(1), (1, \mathbf{2}), (1, 4, 5)$

Third approach Example

Example: $A = (1, 1000, 1001, 4, 5, 2, 6, 7)$

A	LAT $M_{k,\cdot}$
1	$(\mathbf{1})$
$+\ 1000$	$(1), (1, \mathbf{1000})$
$+\ 1001$	$(1), (1, 1000), (1, 1000, \mathbf{1001})$
$+\ 4$	$(1), (1, \mathbf{4}), (1, 1000, 1001)$
$+\ 5$	$(1), (1, 4), (1, 4, \mathbf{5})$
$+\ 2$	$(1), (1, \mathbf{2}), (1, 4, 5)$
$+\ 6$	$(1), (1, 2), (1, 4, 5), (1, 4, 5, \mathbf{6})$

Third approach Example

Example: $A = (1, 1000, 1001, 4, 5, 2, 6, 7)$

A	LAT $M_{k,\cdot}$
1	$(\mathbf{1})$
$+\ 1000$	$(1), (1, \mathbf{1000})$
$+\ 1001$	$(1), (1, 1000), (1, 1000, \mathbf{1001})$
$+\ 4$	$(1), (1, \mathbf{4}), (1, 1000, 1001)$
$+\ 5$	$(1), (1, 4), (1, 4, \mathbf{5})$
$+\ 2$	$(1), (1, \mathbf{2}), (1, 4, 5)$
$+\ 6$	$(1), (1, 2), (1, 4, 5), (1, 4, 5, \mathbf{6})$
$+\ 7$	$(1), (1, 2), (1, 4, 5), (1, 4, 5, 6), (1, 4, 5, 6, \mathbf{7})$

DP Table

- Idea: save the last element of the increasing sequence $M_{k,j}$ at slot j.

DP Table

- Idea: save the last element of the increasing sequence $M_{k,j}$ at slot j.
- Example:
  13 12 15 11 16 14

i	1	2	3	4	5	6
value a_i	13	12	15	11	16	14

DP Table

- Idea: save the last element of the increasing sequence $M_{k,j}$ at slot j.
- Example:

  13 12 15 11 16 14

i	1	2	3	4	5	6
value a_i	13	12	15	11	16	14

j	0	1	2	3	4	...
$(L_j)_j$	$-\infty$	∞	∞	∞	∞	

DP Table

- Idea: save the last element of the increasing sequence $M_{k,j}$ at slot j.
- Example:
  13 12 15 11 16 14

i	1	2	3	4	5	6
value a_i	13	12	15	11	16	14

j	0	1	2	3	4	...
$(L_j)_j$	$-\infty$	13	∞	∞	∞	

DP Table

- Idea: save the last element of the increasing sequence $M_{k,j}$ at slot j.
- Example:
  13 12 15 11 16 14

i	1	2	3	4	5	6
value a_i	13	12	15	11	16	14

j	0	1	2	3	4	...
$(L_j)_j$	$-\infty$	12	∞	∞	∞	

DP Table

- Idea: save the last element of the increasing sequence $M_{k,j}$ at slot j.
- Example:
  13 12 15 11 16 14

i	1	2	3	4	5	6
value a_i	13	12	15	11	16	14

j	0	1	2	3	4	...
$(L_j)_j$	$-\infty$	12	15	∞	∞	

DP Table

- Idea: save the last element of the increasing sequence $M_{k,j}$ at slot j.
- Example:
  13 12 15 11 16 14

i	1	2	3	4	5	6
value a_i	13	12	15	11	16	14

j	0	1	2	3	4	...
$(L_j)_j$	$-\infty$	11	15	∞	∞	

DP Table

- Idea: save the last element of the increasing sequence $M_{k,j}$ at slot j.
- Example:
  13 12 15 11 16 14

i	1	2	3	4	5	6
value a_i	13	12	15	11	16	14

j	0	1	2	3	4	...
$(L_j)_j$	$-\infty$	11	15	16	∞	

DP Table

- Idea: save the last element of the increasing sequence $M_{k,j}$ at slot j.
- Example:
  13 12 15 11 16 14
- Problem: Table does not contain the subsequence, only the last value.

i	1	2	3	4	5	6
value a_i	13	12	15	11	16	14

j	0	1	2	3	4	...
$(L_j)_j$	$-\infty$	11	14	16	∞	

DP Table

- Idea: save the last element of the increasing sequence $M_{k,j}$ at slot j.
- Example:
  13 12 15 11 16 14
- Problem: Table does not contain the subsequence, only the last value.
- Solution: second table with the values of the predecessors.

i	1	2	3	4	5	6
value a_i	13	12	15	11	16	14

j	0	1	2	3	4	...
$(L_j)_j$	$-\infty$	11	14	16	∞	

DP Table

- Idea: save the last element of the increasing sequence $M_{k,j}$ at slot j.
- Example:
  13 12 15 11 16 14
- Problem: Table does not contain the subsequence, only the last value.
- Solution: second table with the values of the predecessors.

i	1	2	3	4	5	6
value a_i	13	12	15	11	16	14
Predecessor	$-\infty$	$-\infty$	12	$-\infty$	15	11

j	0	1	2	3	4	...
$(L_j)_j$	-∞	11	14	16	∞	

Dynamic Programming Algorithm LAS

Table dimension? Semantics?

1. Two tables $T[0,\dots,n]$ and $V[1,\dots,n]$. $T[j]$: last Element of the increasing subequence $M_{n,j}$
$V[j]$: Value of the predecessor of a_j.
Start with $T[0] \leftarrow -\infty$, $T[i] \leftarrow \infty \; \forall i > 1$

Computation of an entry

2. Entries in T sorted in ascending order. For each new entry a_k binary search for l, such that $T[l] < a_k < T[l+1]$. Set $T[l+1] \leftarrow a_k$. Set $V[k] = T[l]$.

Dynamic Programming algorithm LAS

3. **Computation order**

   Traverse the list anc compute $T[k]$ and $V[k]$ with ascending k

   Reconstruction of a solution?

4. Search the largest l with $T[l] < \infty$. l is the last index of the LAS. Starting at l search for the index $i < l$ such that $V[l] = a_i$, i is the predecessor of l. Repeat with $l \leftarrow i$ until $T[l] = -\infty$

Analysis

- Computation of the table:
  - Initialization: $\Theta(n)$ Operations
  - Computation of the kth entry: binary search on positions $\{1, \ldots, k\}$ plus constant number of assignments.

$$\sum_{k=1}^{n} (\log k + \mathcal{O}(1)) = \mathcal{O}(n) + \sum_{k=1}^{n} \log(k) = \Theta(n \log n).$$

- Reconstruction: traverse A from right to left: $\mathcal{O}(n)$.

Overal runtime:

$$\Theta(n \log n).$$

20.7 Editing Distance

Minimal Editing Distance

Editing distance of two sequences $A_n = (a_1, \ldots, a_n)$, $B_m = (b_1, \ldots, b_m)$.

Editing operations:

- Insertion of a character
- Deletion of a character
- Replacement of a character

Question: how many editing operations at least required in order to transform string A into string B.

TIGER $\rightarrow$ ZIGER $\rightarrow$ ZIEGER $\rightarrow$ ZIEGE

Minimal Editing Distance

Wanted: cheapest character-wise transformation $A_n \rightarrow B_m$ with costs

operation	Levenshtein	LCS[31]	general
Insert c	1	1	ins(c)
Delete c	1	1	del(c)
Replace $c \rightarrow c'$	$\mathbb{1}(c \neq c')$	$\infty \cdot \mathbb{1}(c \neq c')$	repl(c, c')

Beispiel

T	I	G	E	R		T	I	_	G	E	R		T→Z	+E	-R
Z	I	E	G	E		Z	I	E	G	E	_		Z→T	-E	+R

[31]Longest common subsequence – A special case of an editing problem

Idea

Z I E G E → T I G E R

Possibilities

1. $$c(\texttt{'ZIEG'} \to \texttt{'TIGE'}) + c(\texttt{'E'} \to \texttt{'R'})$$

   Z I E G **E** → T I G E **R**

2. $$c(\texttt{'ZIEGE'} \to \texttt{'TIGE'}) + c(\mathsf{ins}(\texttt{'R'}))$$

   Z I E G E → T I G E **+ R**

3. $$c(\texttt{'ZIEG'} \to \texttt{'TIGER'}) + c(\mathsf{del}(\texttt{'E'}))$$

   Z I E G E **- E** → T I G E R

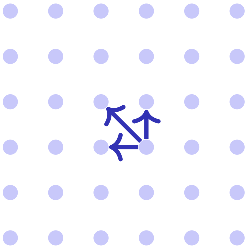
DP

0. $E(n,m)$ = mimimum number edit operations (ED cost) $a_{1\ldots n} \rightarrow b_{1\ldots m}$
1. Subproblems $E(i,j)$ = ED of $a_{1\ldots i}$, $b_{1\ldots j}$. #SP $= n \cdot m$
2. Guess Costs$\Theta(1)$
   - $a_{1..i} \rightarrow a_{1\ldots i-1}$ (delete)
   - $a_{1..i} \rightarrow a_{1\ldots i}b_j$ (insert)
   - $a_{1..i} \rightarrow a_{1\ldots i-1}b_j$ (replace)
3. Rekursion

$$E(i,j) = \min \begin{cases} \text{del}(a_i) + E(i-1,j), \\ \text{ins}(b_j) + E(i,j-1), \\ \text{repl}(a_i,b_j) + E(i-1,j-1) \end{cases}$$

DP

4. Dependencies



$\Rightarrow$ Computation from left top to bottom right. Row- or column-wise.

5. Solution in $E(n, m)$

Example (Levenshtein Distance)

$$E[i,j] \leftarrow \min\left\{E[i-1,j]+1, E[i,j-1]+1, E[i-1,j-1]+\mathbb{1}(a_i \neq b_j)\right\}$$

	∅	Z	I	E	G	E
∅	0	1	2	3	4	5
T	1	1	2	3	4	5
I	2	2	1	2	3	4
G	3	3	2	2	2	3
E	4	4	3	2	3	2
R	5	5	4	3	3	3

Editing steps: from bottom right to top left, following the recursion.

Bottom-Up DP algorithm ED

Dimension of the table? Semantics?

1. Table $E[0,\dots,m][0,\dots,n]$. $E[i,j]$: minimal edit distance of the strings $(a_1,\dots,a_i)$ and $(b_1,\dots,b_j)$

Computation of an entry

2. $E[0,i] \leftarrow i \; \forall 0 \leq i \leq m$, $E[j,0] \leftarrow i \; \forall 0 \leq j \leq n$. Computation of $E[i,j]$ otherwise via $E[i,j] = \min\{\text{del}(a_i)+E(i-1,j), \text{ins}(b_j)+E(i,j-1), \text{repl}(a_i,b_j)+E(i-1,j-1)\}$

Bottom-Up DP algorithm ED

3\.

Computation order

Rows increasing and within columns increasing (or the other way round).

4\.

Reconstruction of a solution?

Start with $j = m$, $i = n$. If $E[i, j] = \text{repl}(a_i, b_j) + E(i-1, j-1)$ then output $a_i \rightarrow b_j$ and continue with $(j, i) \leftarrow (j-1, i-1)$; otherwise, if $E[i, j] = \text{del}(a_i) + E(i-1, j)$ output $\text{del}(a_i)$ and continue with $j \leftarrow j-1$ otherwise, if $E[i, j] = \text{ins}(b_j) + E(i, j-1)$, continue with $i \leftarrow i-1$. Terminate for $i = 0$ and $j = 0$.

Analysis ED

- Number table entries: $(m+1) \cdot (n+1)$.
- Constant number of assignments and comparisons each. Number steps: $\mathcal{O}(mn)$
- Determination of solition: decrease i or j. Maximally $\mathcal{O}(n+m)$ steps.

Runtime overal:

$$\mathcal{O}(mn).$$

Matrix-Chain-Multiplication

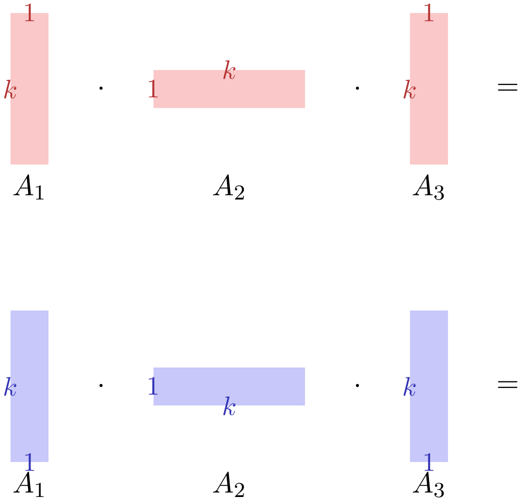
Task: Computation of the product $A_1 \cdot A_2 \cdot \ldots \cdot A_n$ of matrices $A_1, \ldots, A_n$.

Matrix multiplication is associative, i.e. the order of evaluation can be chosen arbitrarily

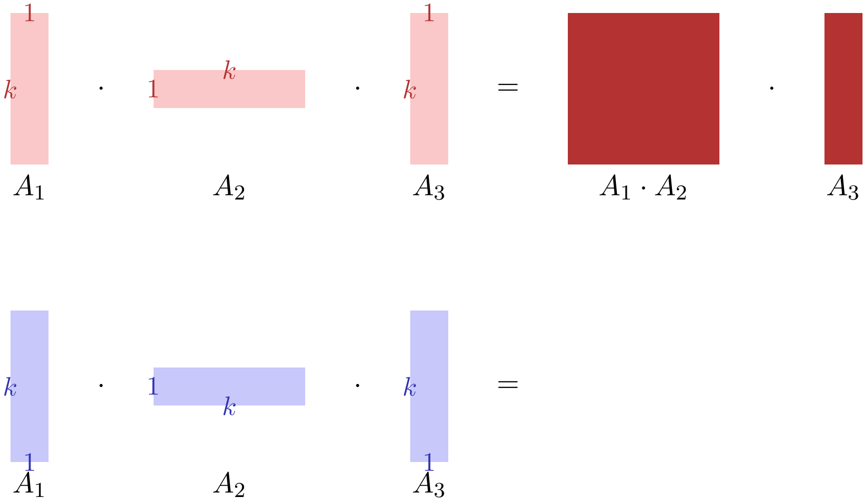
Goal: efficient computation of the product.

Assumption: multiplication of an $(r \times s)$-matrix with an $(s \times u)$-matrix provides costs $r \cdot s \cdot u$.

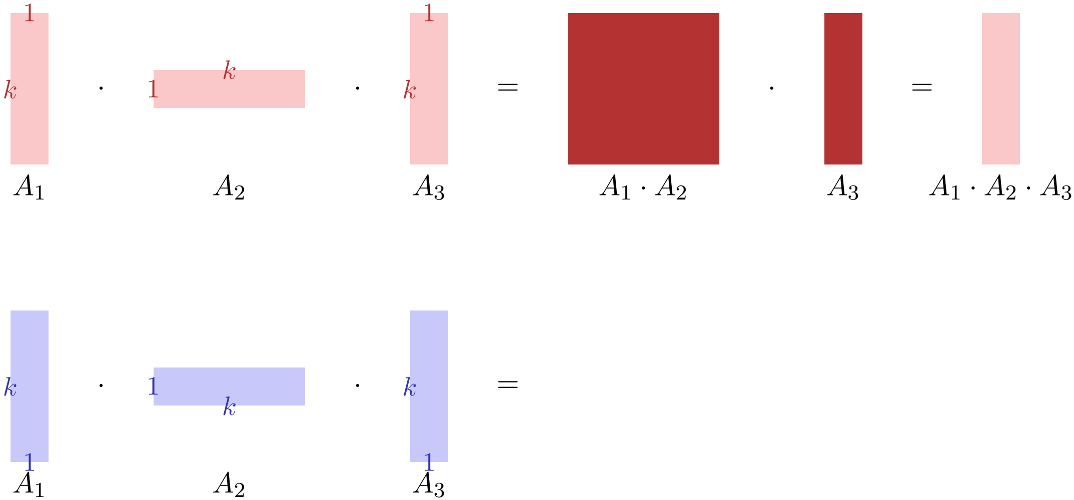
Does it matter?



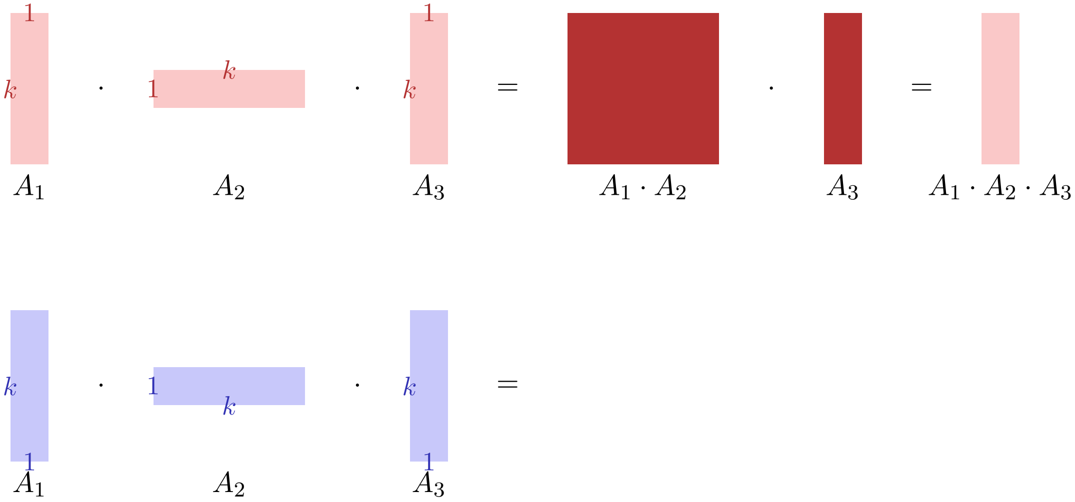
Does it matter?



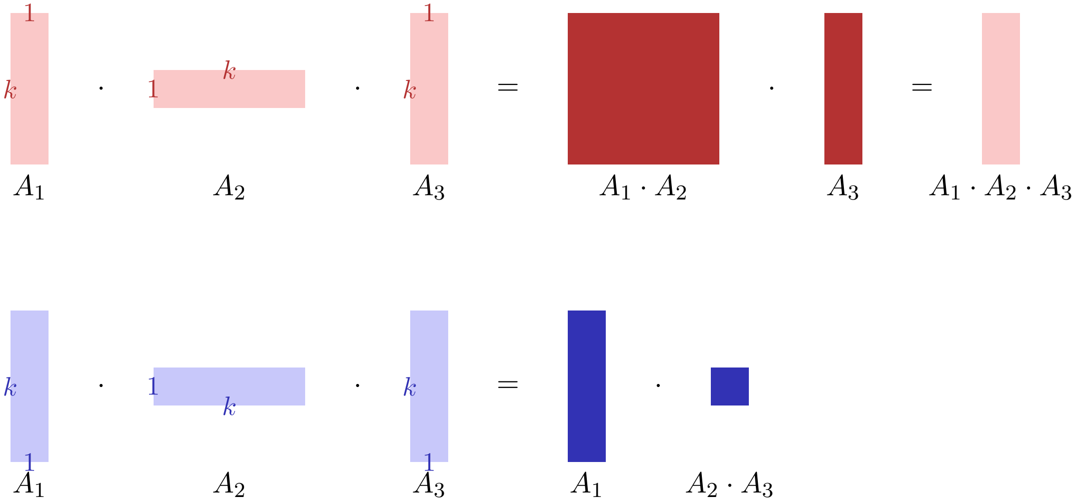
Does it matter?



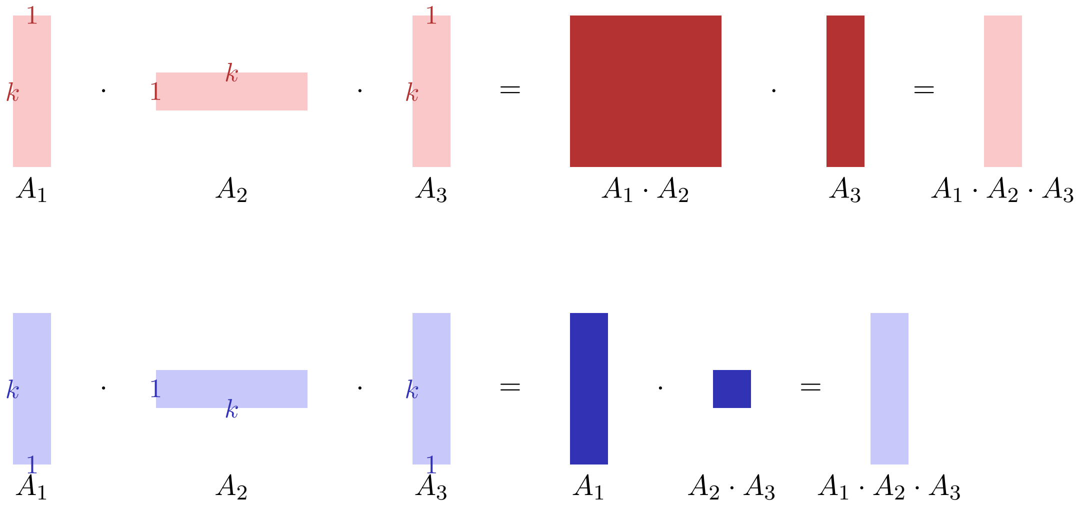
Does it matter?



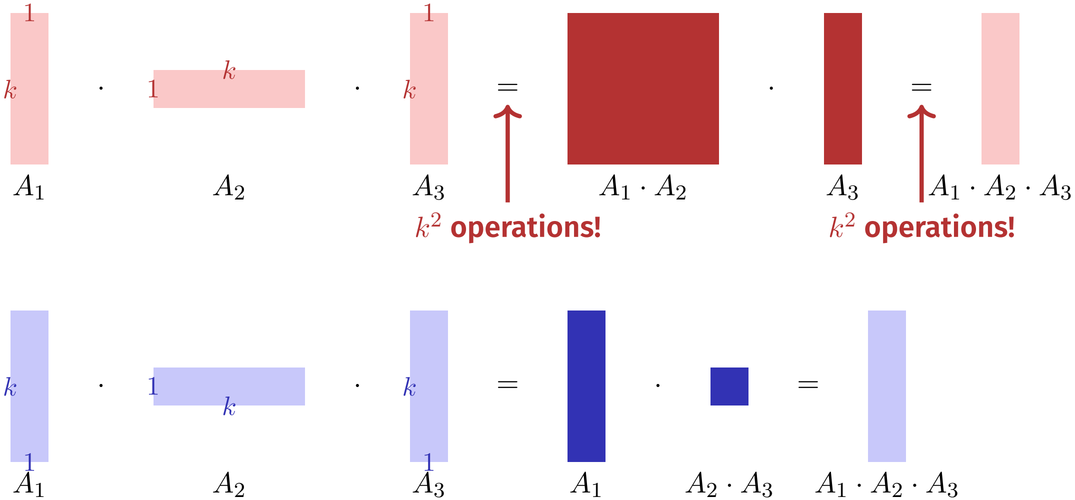
Does it matter?



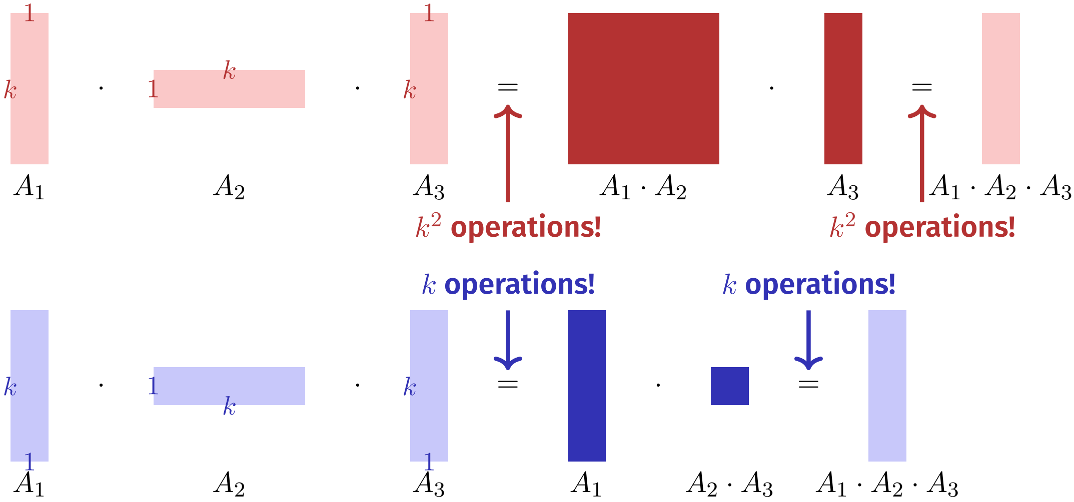
Does it matter?



Does it matter?



Does it matter?



Recursion

- Assume that the best possible computation of $(A_1 \cdot A_2 \cdots A_i)$ and $(A_{i+1} \cdot A_{i+2} \cdots A_n)$ is known for each i.
- Compute best i, done.

$n \times n$-table M. entry $M[p, q]$ provides costs of the best possible bracketing $(A_p \cdot A_{p+1} \cdots A_q)$.

$$M[p, q] \leftarrow \min_{p \leq i < q} (M[p, i] + M[i + 1, q] + \text{costs of the last multiplication})$$

Computation of the DP-table

- Base cases $M[p, p] \leftarrow 0$ for all $1 \leq p \leq n$.
- Computation of $M[p, q]$ depends on $M[i, j]$ with $p \leq i \leq j \leq q$, $(i, j) \neq (p, q)$.
  In particular $M[p, q]$ depends at most from entries $M[i, j]$ with $i - j < q - p$.
  Consequence: fill the table from the diagonal.

Analysis

DP-table has n^2 entries. Computation of an entry requires considering up to $n-1$ other entries.
Overal runtime $\mathcal{O}(n^3)$.

Readout the order from M: exercise!

Digression: matrix multiplication

Consider the multiplication of two $n \times n$ matrices.
Let

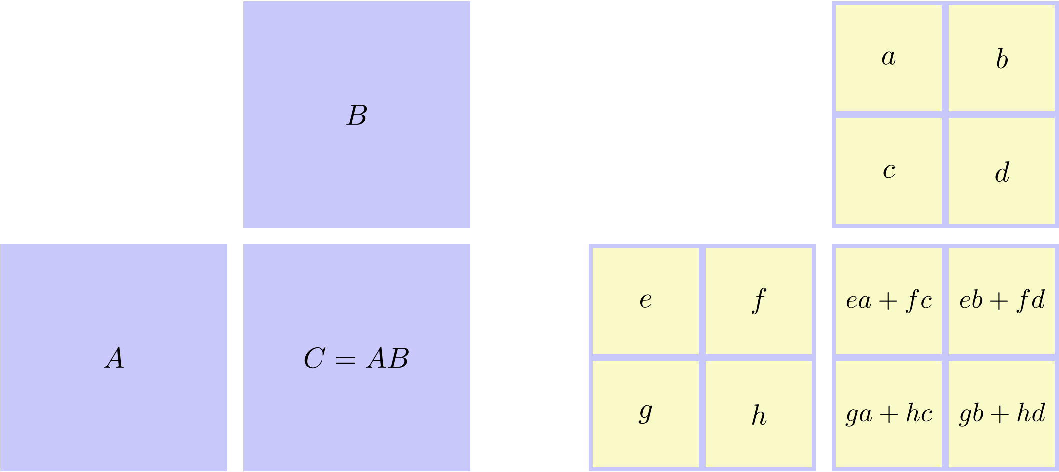
$$A = (a_{ij})_{1 \le i,j \le n}, B = (b_{ij})_{1 \le i,j \le n}, C = (c_{ij})_{1 \le i,j \le n},$$
$$C = A \cdot B$$

then

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

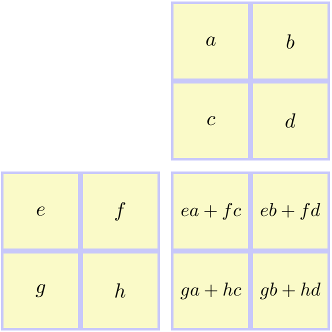
Naive algorithm requires $\Theta(n^3)$ elementary multiplications.

Divide and Conquer



Divide and Conquer

- Assumption $n = 2^k$.
- Number of elementary multiplications: $M(n) = 8M(n/2)$, $M(1) = 1$.
- yields $M(n) = 8^{\log_2 n} = n^{\log_2 8} = n^3$. No advantage ☹



Strassen's Matrix Multiplication

- Nontrivial observation by Strassen (1969): It suffices to compute the seven products $A = (e+h)\cdot(a+d)$, $B = (g+h)\cdot a$, $C = e\cdot(b-d)$, $D = h\cdot(c-a)$, $E = (e+f)\cdot d$, $F = (g-e)\cdot(a+b)$, $G = (f-h)\cdot(c+d)$. Because:
  $ea + fc = A + D - E + G$, $eb + fd = C + E$,
  $ga + hc = B + D$, $gb + hd = A - B + C + F$.
- This yields $M'(n) = 7M(n/2)$, $M'(1) = 1$.
  Thus $M'(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807}$.
- Fastest currently known algorithm: $\mathcal{O}(n^{2.37})$