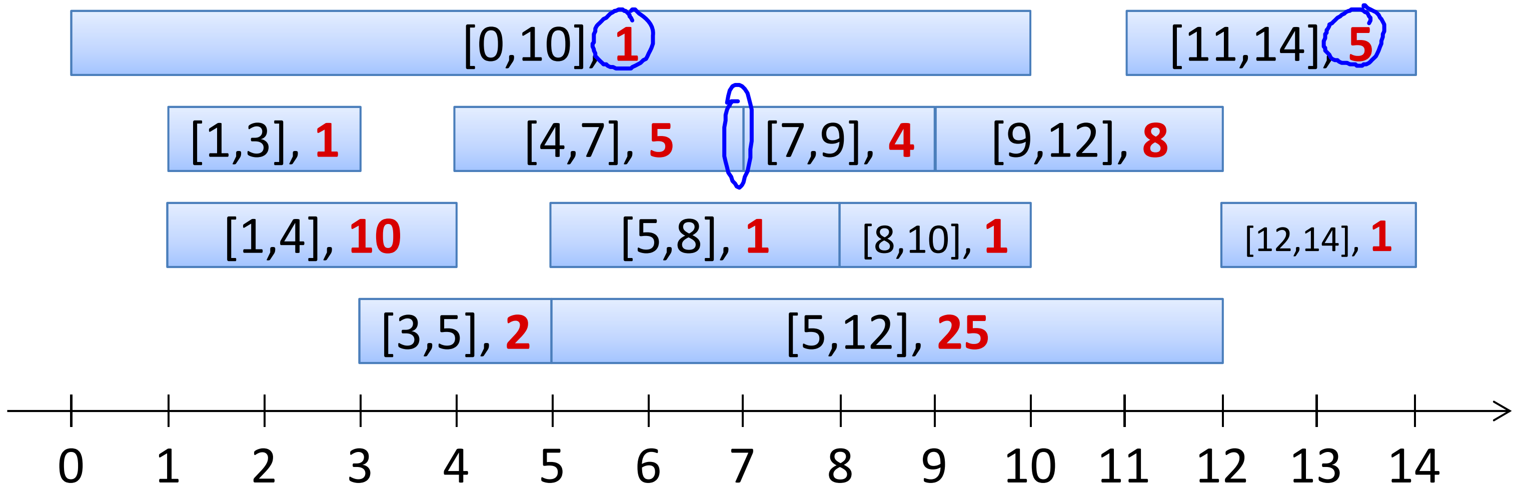
Chapter 3
Dynamic Programming

Algorithm Theory
WS 2012/13

Fabian Kuhn

Weighted Interval Scheduling

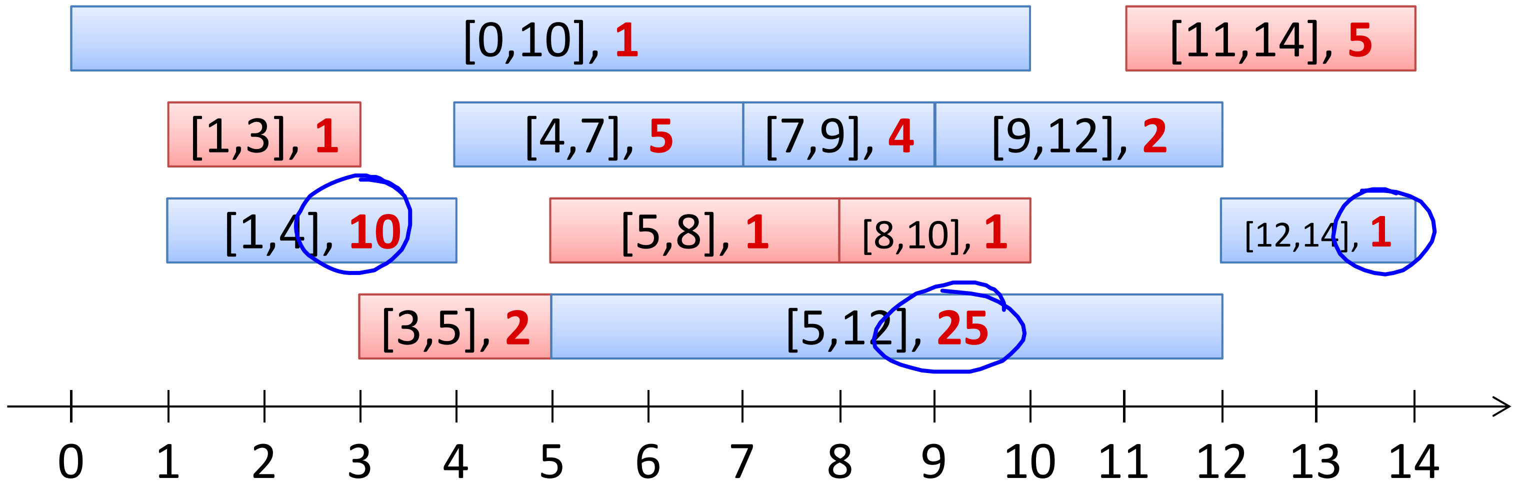
- **Given:** Set of intervals, e.g.
  [0,10],[1,3],[1,4],[3,5],[4,7],[5,8],[5,12],[7,9],[9,12],[8,10],[11,14],[12,14]
- Each interval has a **weight *w***



- **Goal:** Non-overlapping set of intervals of largest possible weight
  - Overlap at boundary ok, i.e., [4,7] and [7,9] are non-overlapping
- **Example:** Intervals are room requests of different importance

Greedy Algorithms

Choose available request with earliest finishing time:



- Algorithm is not optimal any more
  - It can even be arbitrarily bad…

- No greedy algorithm known that works

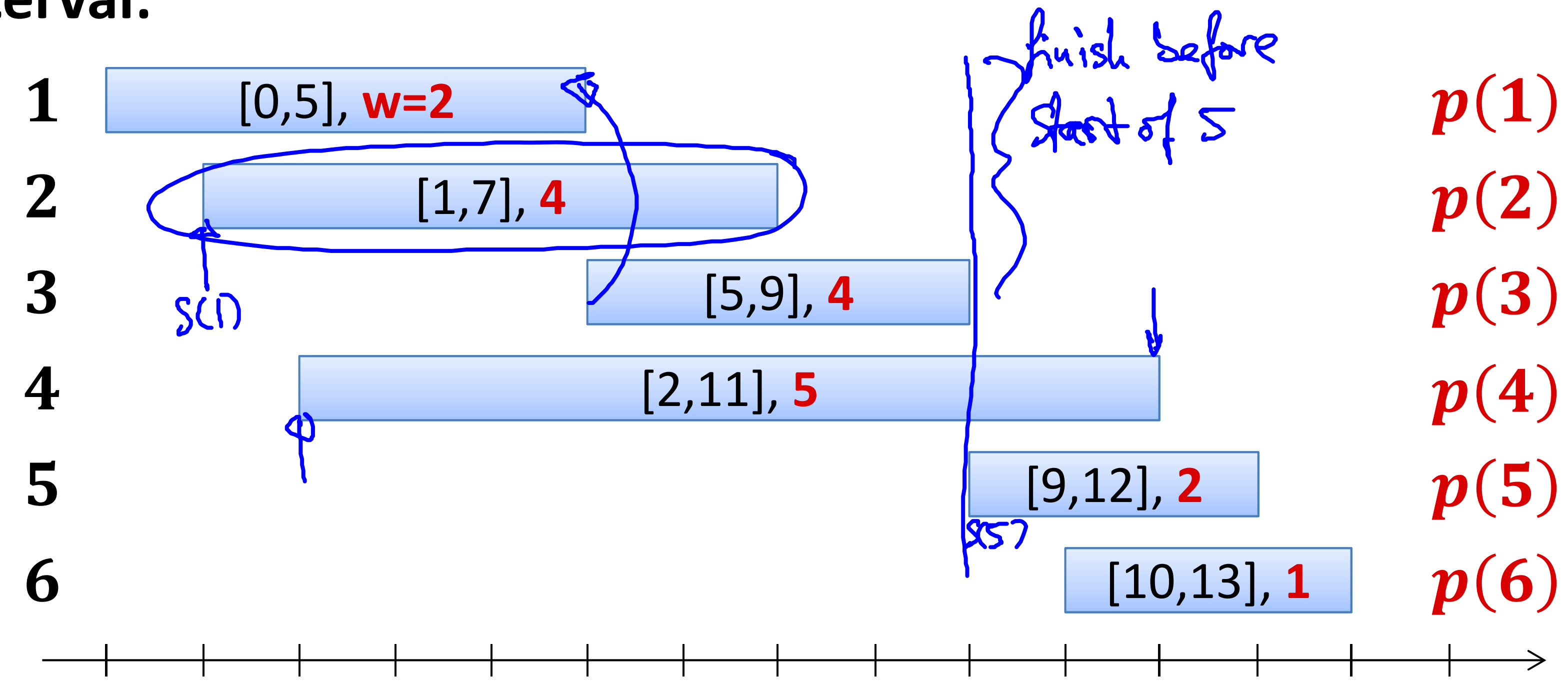
Solving Weighted Interval Scheduling

- Interval i: start time $s(i)$, finishing time: $t(i)$, weight: $w(i)$
  $i \in \{1, \dots, n\}$
- Assume intervals $1, \dots, n$ are sorted by increasing $t(i)$
  - $0 < f(1) \leq f(2) \leq \cdots \leq f(n)$, for convenience: $f(0) = 0$
- Simple observation:
  Opt. solution contains interval n or it doesn't contain interval n
- Weight of optimal solution for only intervals $1, \dots, k$: $W(k)$
  Define $p(k) \coloneqq \max\{i \in \{0, \dots, k-1\} : f(i) \leq s(k)\}$
- Opt. solution does not contain interval n: $\boldsymbol{W(n) = W(n-1)}$

  Opt. solution contains interval n: $\boldsymbol{W(n) = w(n) + W(p(n))}$

Example

Interval:



$p(1) = 0$

$p(2) = 0$

$p(3) = 1$

$p(4) = 0$

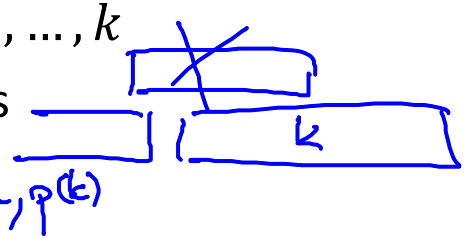
$p(5) = 3$

$p(6) = 3$

$p(k)$: largest $i<k$ s.t. $f(i) \leq s(k)$ (# intervals before ind. k)

Recursive Definition of Optimal Solution

- Recall:
  - $W(k)$: weight of optimal solution with intervals $1, \dots, k$
  - $p(k)$: last interval to finish before interval k starts



- Recursive definition of optimal weight:

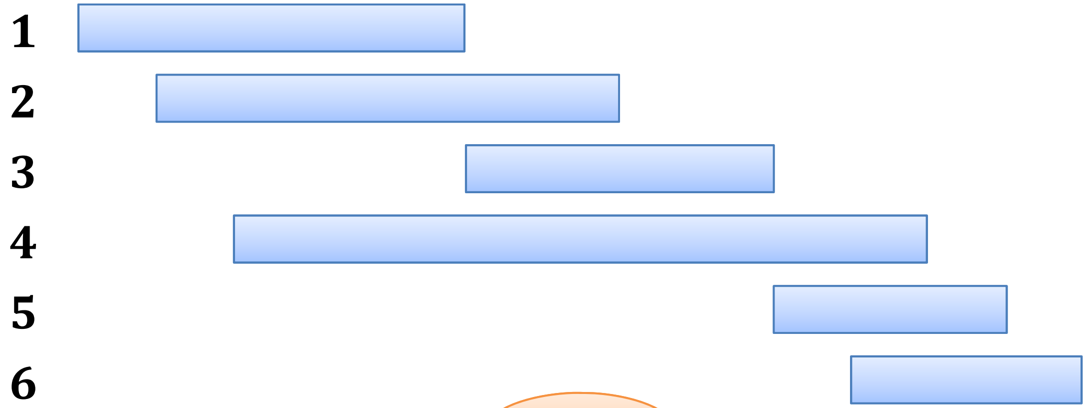
$$\forall k > 1:\ W(k) = \max\{\underbrace{W(k-1)}_{k \notin OPT},\ \underbrace{w(k) + W(p(k))}_{\text{OPT contains interval } k}\}$$

$$W(1) = w(1)$$

↳ $w(k) + W(p(k))$

- Immediately gives a simple, recursive algorithm

Running Time of Recursive Algorithm

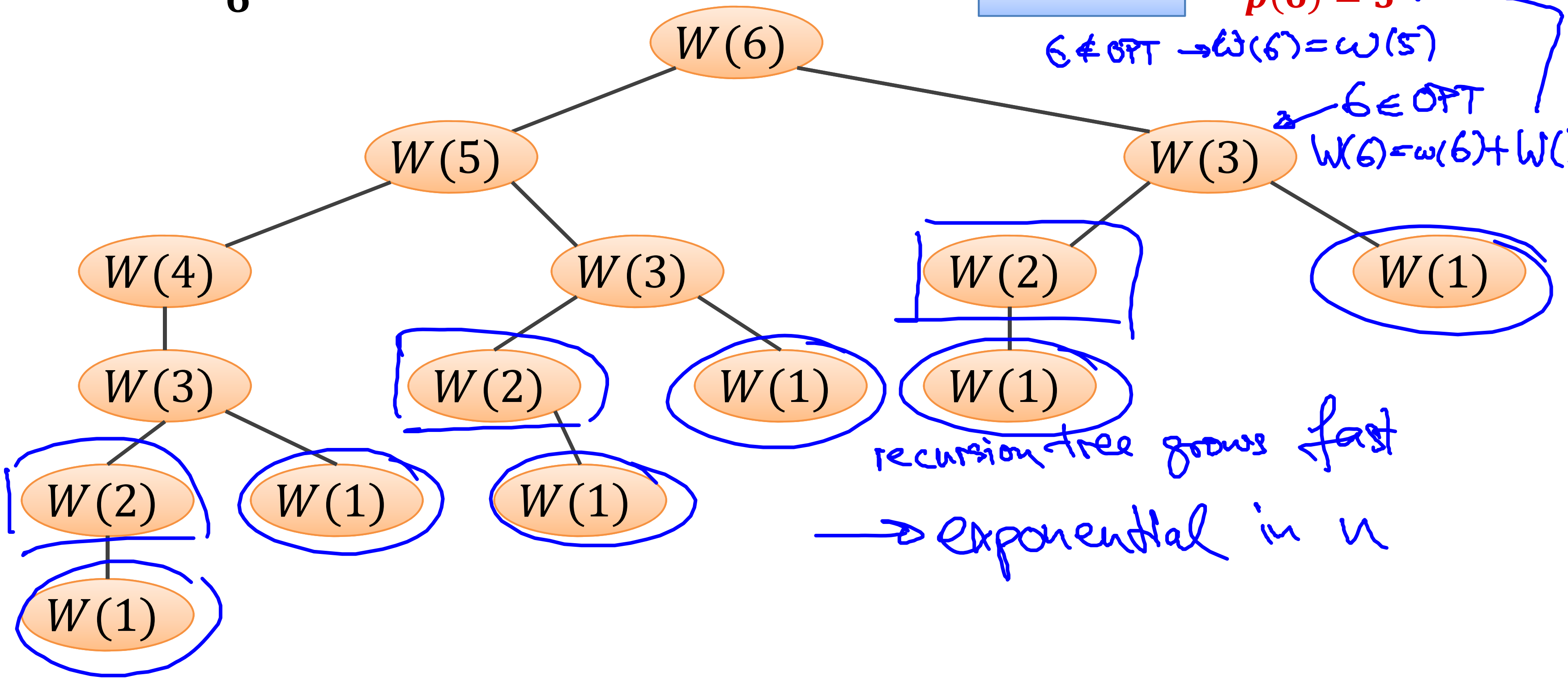


$p(1) = 0$
$p(2) = 0$
$p(3) = 1$
$p(4) = 0$
$p(5) = 3$
$p(6) = 3$

$6 \notin OPT \to W(6) = W(5)$

$6 \in OPT$

$W(6) = w(6) + W(3)$



recursion tree grows fast

$\to$ exponential in n

Memoizing the Recursion

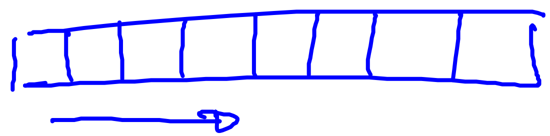
- Running time of recursive algorithm: exponential!
- But, alg. only solves n different sub=problems: $W(1), \dots, W(n)$
- There is no need to compute them multiple times

Memoization:

- **Store already computed values** for future use (recursive calls)

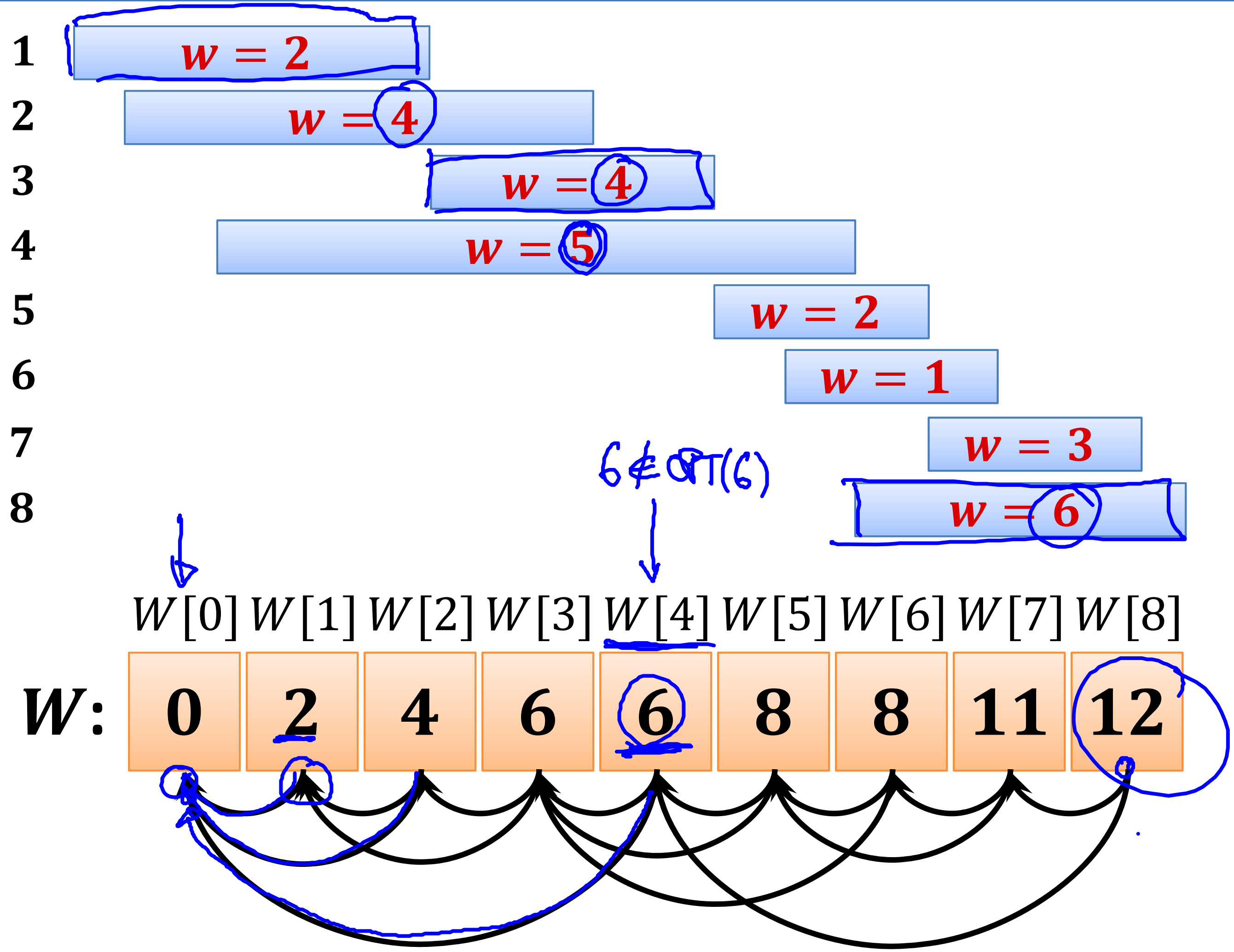
Efficient algorithm:

1. $W[0] := 0$; compute values $\boxed{p(i)}$



2. **for** $i := 1$ **to** n **do**
3. $\quad W[i] := \max\{\underbrace{W[i-1]}, w(i) + \underbrace{W[p(i)]}_{p(i)<i}\}$
4. **end**

Example



$p(1) = 0$
$p(2) = 0$
$p(3) = 1$
$p(4) = 0$
$p(5) = 3$
$p(6) = 3$
$p(7) = 5$
$p(8) = 4$

weight of OPT

1, 3, 8

Computing the schedule: store where you come from!

$W[i] = \max\{W[i-1], w(i) + W[p(i)]\}$

Matrix-chain multiplication

Given: sequence (chain) $\langle A_1, A_2, \dots, A_n \rangle$ of matrices

Goal: compute the product $A_1 \cdot A_2 \cdot \dots \cdot A_n$

$(((A_1 A_2) A_3) A_4) \dots)$ $\quad$ $((A_1 A_2)(A_3 A_4)(A_5 \dots A_n))$

Problem: Parenthesize the product in a way that minimizes the number of scalar multiplications.

Definition: A product of matrices is *fully parenthesized* if it is

- a single matrix
- or the product of two fully parenthesized matrix products, surrounded by parentheses.

Example

All possible fully parenthesized matrix products of the chain $\langle A_1, A_2, A_3, A_4 \rangle$:

$$(A_1(A_2(A_3A_4)))$$

$$(A_1(\underline{(A_2A_3)}\,\underline{A_4}))$$

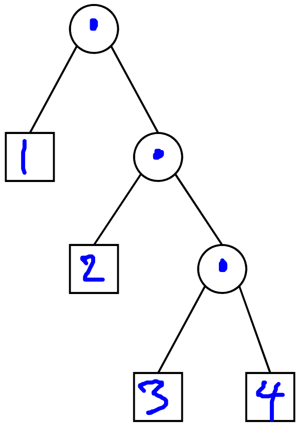
$$((A_1A_2)(A_3A_4))$$

$$((A_1(A_2A_3))A_4)$$

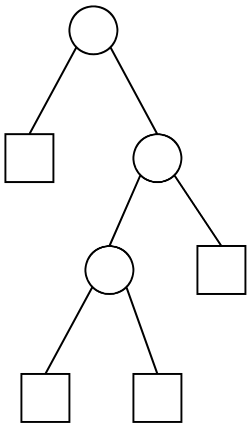
$$(((A_1A_2)A_3)A_4)$$

Different parenthesizations

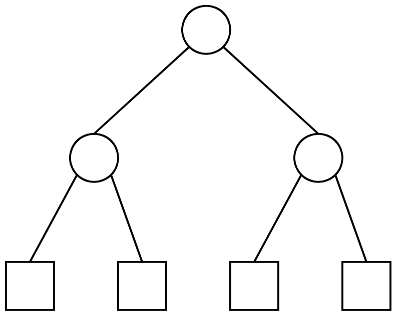
Different parenthesizations correspond to different trees:



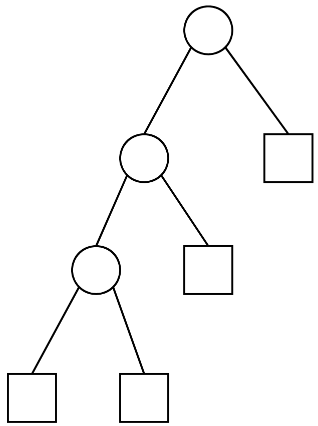
$(A_1(A_2(A_3A_4)))$



$(A_1((A_2A_3)A_4))$



$((A_1A_2)(A_3A_4))$



$(((A_1A_2)A_3)A_4)$

Number of different parenthesizations

- Let $P(n)$ be the number of alternative parenthesizations of the product $A_1 \cdot \ldots \cdot A_n$:

$A_1, \ldots, A_k$ $\qquad$ $A_{k+1}, \ldots, A_n$

-

$$P(1) = 1$$

$$P(n) = \sum_{k=1}^{n-1} P(k) \cdot P(n-k), \qquad \text{for } n \geq 2$$

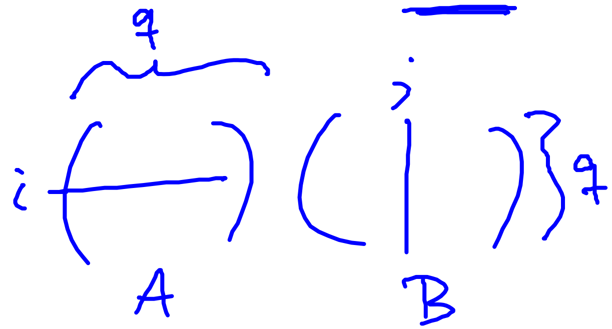
$$P(n+1) = \frac{1}{n+1}\binom{2n}{n} \approx \frac{4^n}{n\sqrt{\pi n}} + O\left(\frac{4^n}{\sqrt{n^5}}\right)$$

$$P(n+1) = C_n \qquad (n^{th} \text{ Catalan number})$$

- Thus: Exhaustive search needs exponential time!

Multiplying Two Matrices

$$A = (a_{ij})_{p \times q}, \qquad B = (b_{ij})_{q \times r}, \qquad A \cdot B = C = (c_{ij})_{p \times r}$$



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

Algorithm *Matrix-Mult*

Input: $(p \times q)$ matrix A, $(q \times r)$ matrix B

Output: $(p \times r)$ matrix $C = A \cdot B$

for $i := 1$ **to** p **do**
    for $j := 1$ **to** r **do**
        $C[i, j] := 0;$
        for $k := 1$ **to** q **do**
            $C[i, j] := C[i, j] + A[i, k] \cdot B[k, j]$

Remark:

Using this algorithm, multiplying two $(n \times n)$ matrices requires n^3 multiplications. This can also be done using $O(n^{2.376})$ multiplications.

Number of multiplications and additions: $\boldsymbol{p \cdot q \cdot r}$

Matrix-chain multiplication: Example

Computation of the product $A_1 A_2 A_3$, where

A_1 : $(\underline{50} \times 5)$ matrix

A_2 : $(5 \times \underline{100})$ matrix

A_3 : (100×10) matrix

a) Parenthesization $((\underline{A_1 A_2})A_3)$ and $(A_1(\underline{A_2 A_3}))$ require:

$A' = (A_1 A_2)$: (50×100)-matrix

$50 \cdot 5 \cdot 100 = 25'000$

$A' A_3$: $50 \cdot 100 \cdot 10 = 50'000$

$A'' = (A_2 A_3)$: (5×10)-matrix

$5 \cdot 100 \cdot 10 = 5000$

$A_1 A''$: $50 \cdot 5 \cdot 10 = 2500$

Sum: 75'000 7'500

Structure of an Optimal Parenthesization

- $(A_{\ell\ldots r})$: optimal parenthesization of $A_\ell \cdot \ldots \cdot A_r$

  For some $1 \le k < n$: $\underbrace{(\boldsymbol{A_{1\ldots n}})}$ $= \big(\underbrace{(\boldsymbol{A_{1\ldots k}})}_{\text{OPT}} \cdot \underbrace{(\boldsymbol{A_{k+1\ldots n}})}_{\text{OPT}}\big)$

- Any optimal solution contains optimal solutions for sub-problems

- Assume matrix A_i is a $(d_{i-1} \times d_i)$-matrix $d_{i-1}\left\{\underbrace{\big(\quad A_1 \quad\big)}_{d_i}\right.$

- Cost to solve sub-problem $A_\ell \cdot \ldots \cdot A_r, \ell \le r$ optimally: $C(\ell, r)$

- Then: $A_a \cdot \ldots \cdot A_b$, $\uparrow k$

$$\boldsymbol{C(a,b) = \min_{a \le k < b} C(a,k) + C(k+1,b) + d_{a-1} d_k d_b}$$

cost for mult. $A_{a..k}$ with $A_{k+1..b}$

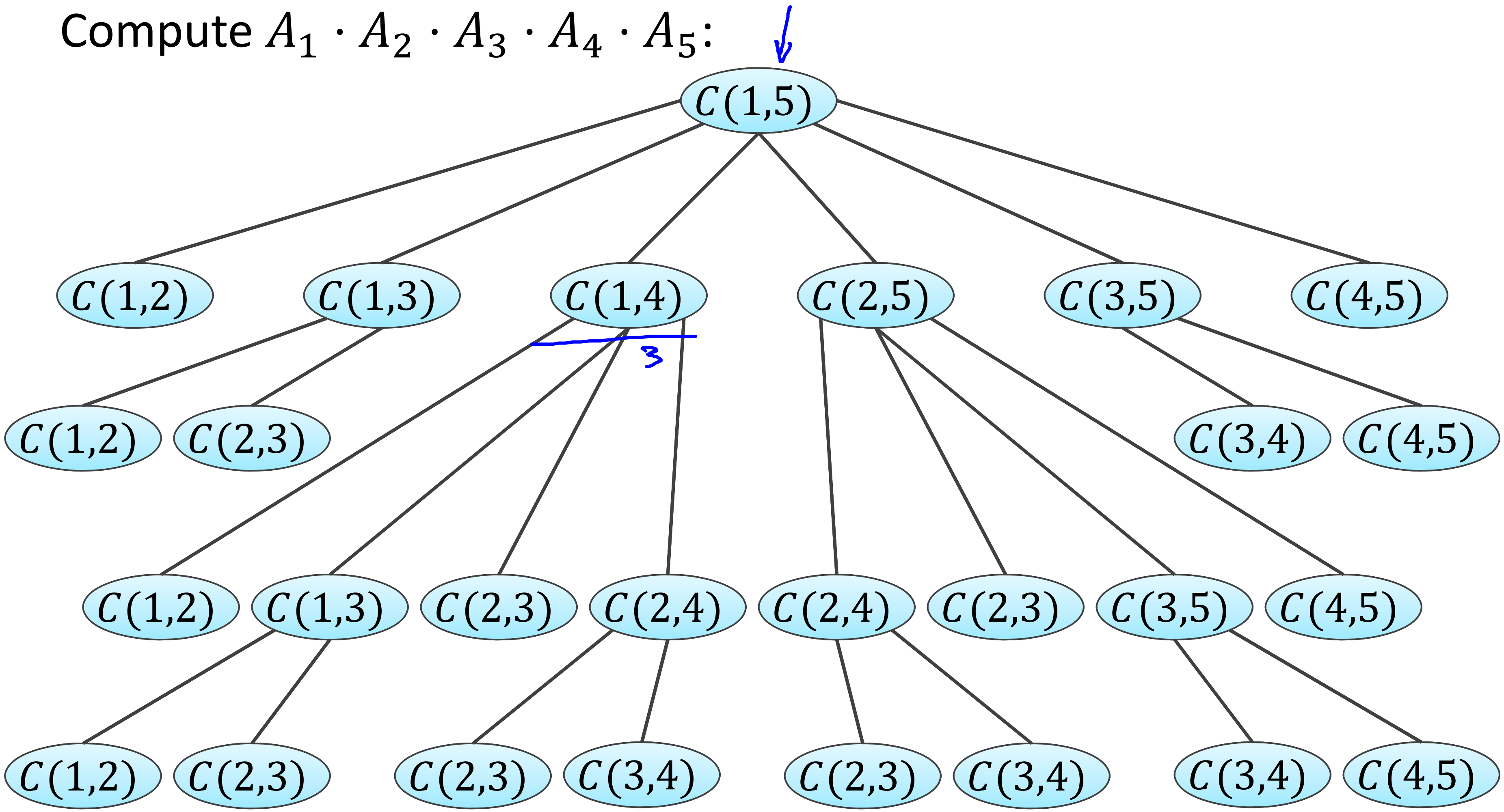
$$\boldsymbol{C(a,a) = 0}$$

recursive rule

↳ recursive alg.

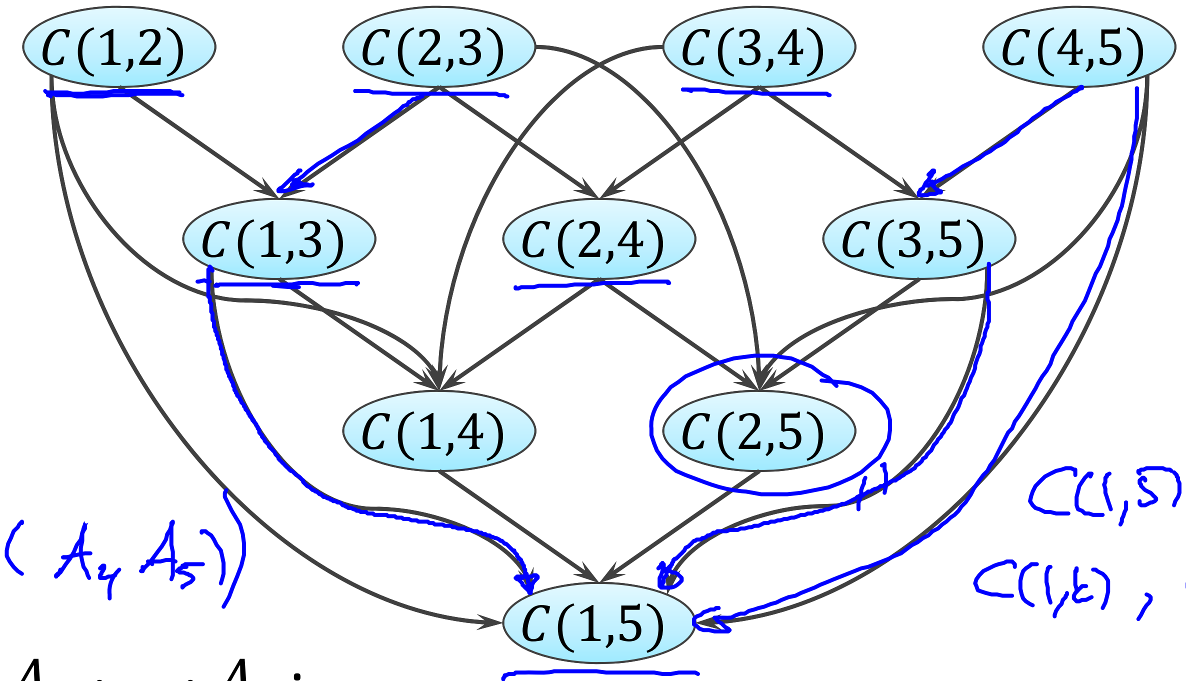
Recursive Computation of Opt. Solution

Compute $A_1 \cdot A_2 \cdot A_3 \cdot A_4 \cdot A_5$:



Using Meomization $A_{\ell..r}$

Compute $A_1 \cdot A_2 \cdot A_3 \cdot A_4 \cdot A_5$:



← $n-1$

← $n-2$

⋮

$((A_1(A_2 A_3))(A_4 A_5))$

$C(1,5)$

$C(1,k)$, $C(k,5)$

Compute $A_1 \cdot \ldots \cdot A_n$:

- Each $C(i,j)$, $i < j$ is computed exactly once → $O(n^2)$ values
- Each $C(i,j)$ dir. depends on $C(i,k)$, $C(k,j)$ for $i < k < j$

Cost for each $C(i,j)$: $O(n)$ → overall time: $\boldsymbol{O(n^3)}$

Dynamic Programming

„Memoization“ for increasing the efficiency of a recursive solution:

- Only the *first time* a sub-problem is encountered, its solution is computed and then stored in a table. Each subsequent time that the subproblem is encountered, the value stored in the table is simply looked up and returned
  (without repeated computation!).

- *Computing the solution*: For each sub-problem, store how the value is obtained (according to which recursive rule).

Dynamic Programming

Dynamic programming / memoization can be applied if

- Optimal solution contains optimal solutions to sub-problems (recursive structure)

- Number of sub-problems that need to be considered is small

$$\underbrace{(A_1 \cdot \ldots A_k)}_{\text{sub-problem } 1} \cdot \underbrace{(A_{k+1} \ldots A_n)}_{2}$$

Remarks about matrix-chain multiplication

1. There is an algorithm that determines an optimal parenthesization in time
$$\boxed{O(n \cdot \log n).}$$

2. There is a linear time algorithm that determines a parenthesization using at most
$$\underline{1.155} \cdot C(1, n)$$
multiplications.

Knapsack

- n items $1, \ldots, n$, each item has weight w_i and value v_i
- Knapsack (bag) of capacity W

- Goal: pack items into knapsack such that total weight is at most W and total value is maximized:

$$\boxed{\max \sum_{i \in S} v_i}$$

$$\text{s.t. } S \subseteq \{1, \ldots, n\} \text{ and } \sum_{i \in S} w_i \leq W$$

- E.g.: jobs of length w_i and value v_i, server available for W time units, try to execute a set of jobs that maximizes the total value

Recursive Structure?

items: $1, \dots, n$

- Optimal solution: $\mathcal{O}$
- If $n \notin \mathcal{O}$: $\mathrm{OPT}(n) = \mathrm{OPT}(n-1)$

$n \in \mathcal{O}$ or $n \notin \mathcal{O}$

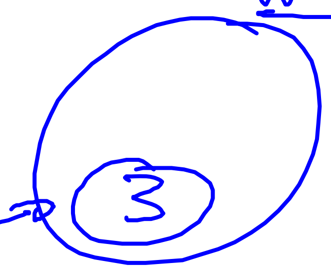
- What if $n \in \mathcal{O}$?
  - Taking n gives value v_n
  - But, n also occupies space w_n in the bag (knapsack)
  - There is space for $W - w_n$ total weight left!

$$\mathrm{OPT}(n) = w_n + \text{optimal solution with first } n-1 \text{ items and knapsack of capacity } W - w_n$$

get value v_n

$W = 6$

1. 1 2
2. 2 4
3. 1 3



A More Complicated Recursion

$\mathbf{OPT}(\boldsymbol{k}, \boldsymbol{x})$: value of optimal solution with items $1, \dots, k$
and knapsack of capacity x

Recursion:

1) k not in opt solution items 1,...,k, capacity x

$$\mathrm{OPT}(k,x) = \mathrm{OPT}(k-1,x)$$

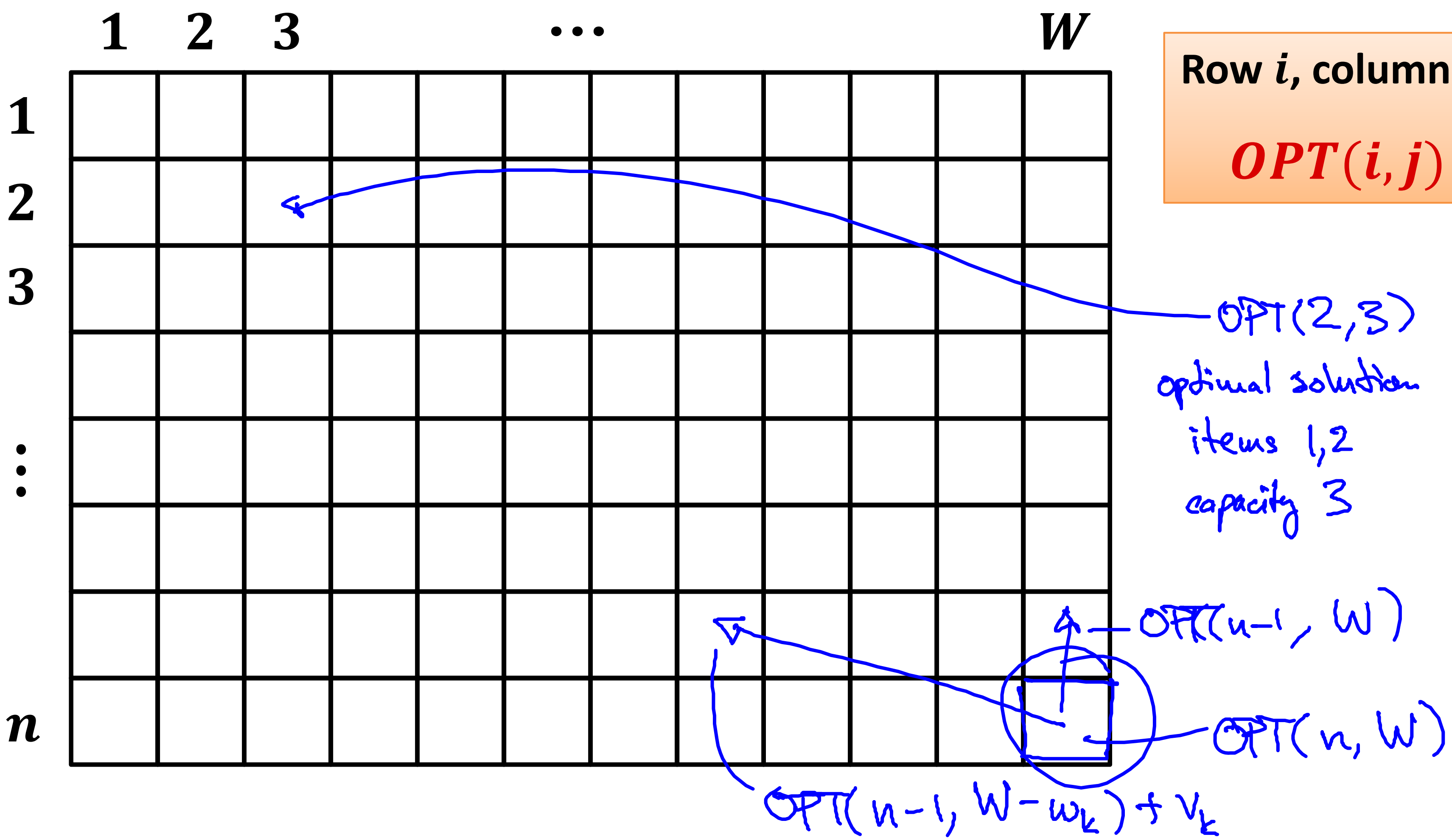
2) k in opt. solution (items 1,...,k, capacity x)

$$\mathrm{OPT}(k,x) = v_k + \mathrm{OPT}(k-1, x-w_k)$$

Dynamic Programming Algorithm

Set up table for all possible $\mathrm{OPT}(k, W')$-values

- Assume that all weights w_i are integers!



Row i, column j:

$OPT(i, j)$

Example

weight value

- 8 items: (3,2), (2,4), (4,1), (5,6), (3,3), (4,3), (5, 4), (6,6)
  Knapsack capacity: 12

  weight **value**

- $\boldsymbol{OPT(k,x) = \max\{OPT(k-1,x), OPT(k-1,x-w_k)+v_k\}}$

capacity

items

		1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	2	2	2	2	2	2	2	2	2	2
2	0	0	4	4	4	6	6	6	6	6	6	6	6
3	0	0	4	4	4	6	6	6	6	7	7	7	7
4													
5													
6													
7													
8													

Running Time of Knapsack Algorithm

- **Size of table:** $O(n \cdot W)$
- Time per table entry: $O(1)$ → **overall time:** $\boldsymbol{O(nW)}$
- Computing solution (set of items to pick):
  Follow $\leq n$ arrows → $O(n)$ time (after filling table)
- Note: Time depends on W → can be exponential in n…
- And it is problematic if weights are not integers.

W arbitrary : NP hard

$\underbrace{(1+\varepsilon)\text{-approximation}}$

PTAS polynomial-time approx. scheme