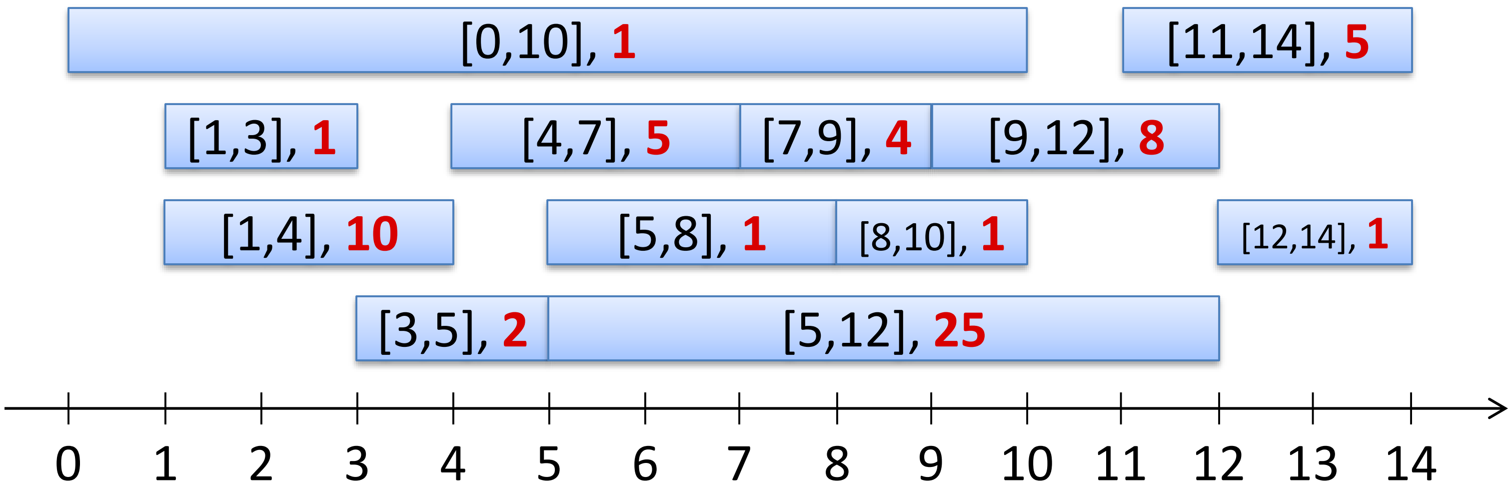
Chapter 3
Dynamic Programming

Algorithm Theory
WS 2018/19

Fabian Kuhn

Weighted Interval Scheduling

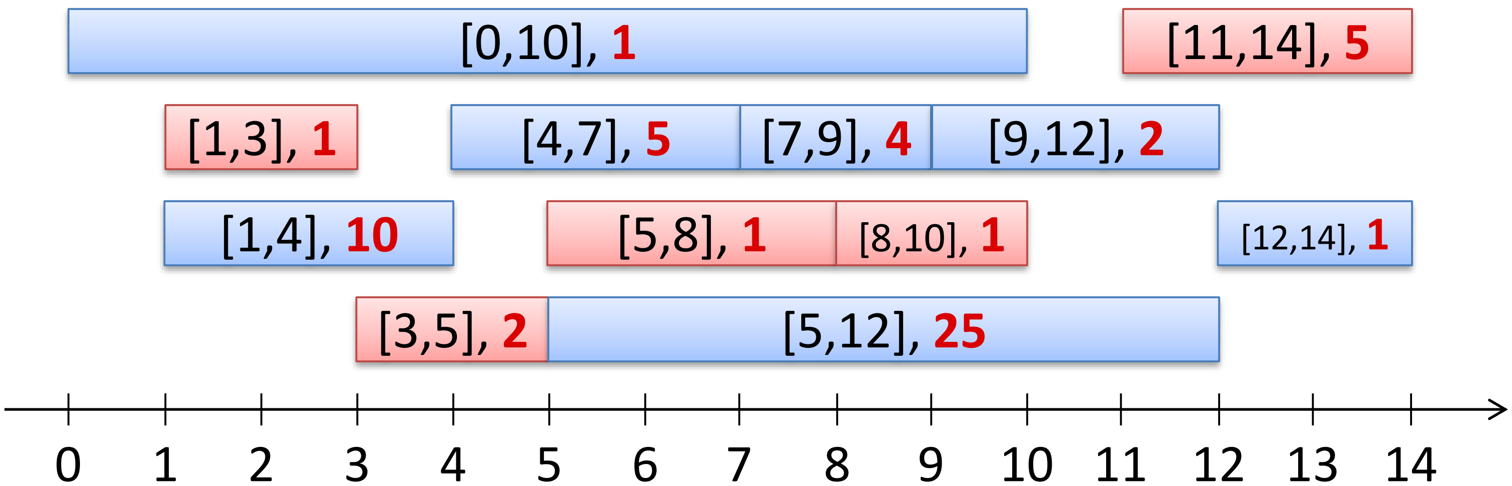
- **Given:** Set of intervals, e.g.
  [0,10],[1,3],[1,4],[3,5],[4,7],[5,8],[5,12],[7,9],[9,12],[8,10],[11,14],[12,14]
- Each interval has a **weight w**



- **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

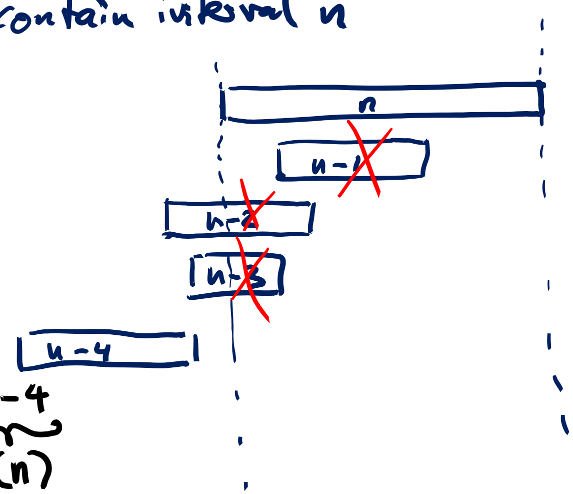
Solving Weighted Interval Scheduling

- Interval i: start time $s(i)$, finishing time: $f(i)$, weight: $w(i)$
- Assume intervals $1, \dots, n$ are sorted by increasing $f(i)$
  - $0 < f(1) \leq f(2) \leq \cdots \leq f(n)$, for convenience: $f(0) = 0$
- Simple observation:
  Opt. solution contains interval n or it doesn't contain interval n

Case 1: opt. solution does not contain interval n
$\Longrightarrow$ opt. sol. for intervals $1, \dots, n$ = opt. sol. for intervals $1, \dots, n-1$

Case 2: opt. solution does contain interval n

in example:
opt. solution consists of
interval n
+
opt. sol. for intervals $1, \dots, n-4$ ($n-4 = p(n)$)



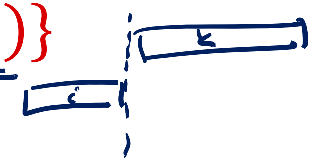
Solving Weighted Interval Scheduling

- Interval i: start time $s(i)$, finishing time: $f(i)$, weight: $w(i)$

- Assume intervals $1, \dots, n$ are sorted by increasing $f(i)$
  - $0 < f(1) \le f(2) \le \dots \le f(n)$, 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 $p(k) := \max\{i \in \{0, \dots, k-1\} : f(i) \le s(k)\}$



$$W(k) = \max\{W(k-1), w(k) + W(p(k))\}$$

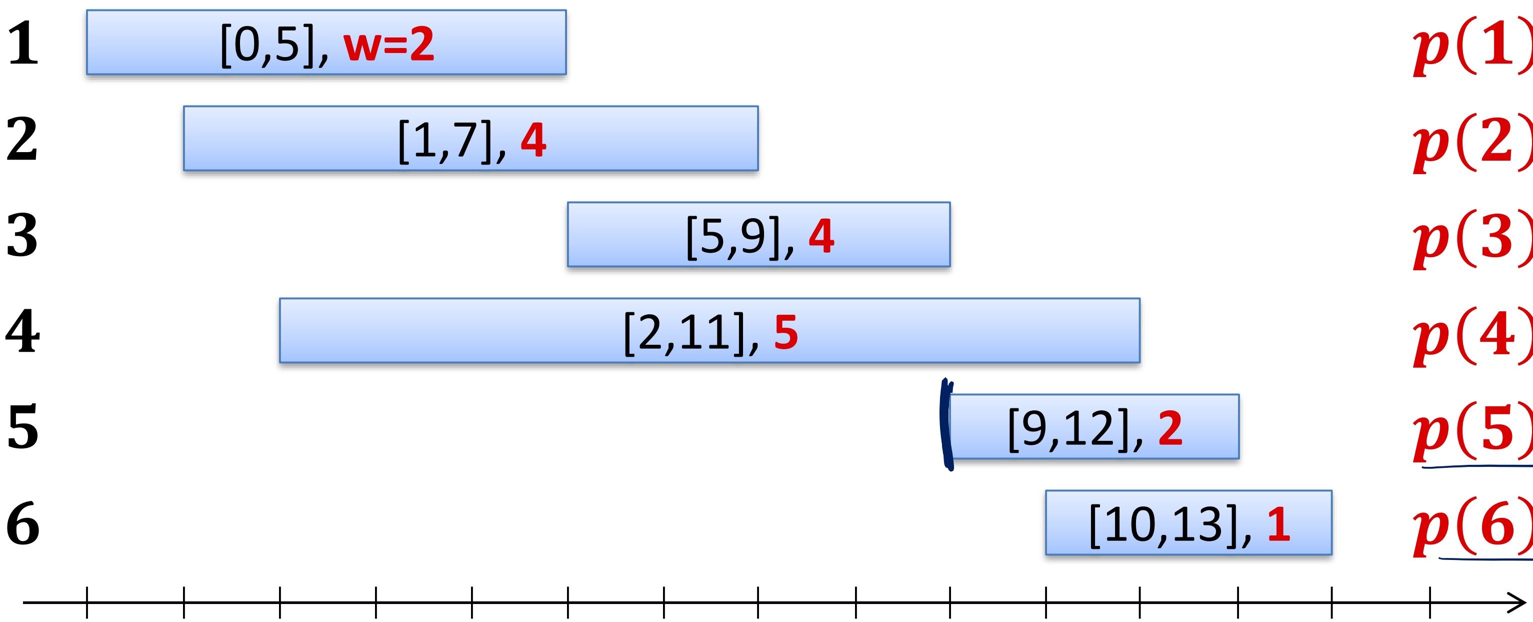
opt. sol. does not contain k — opt. sol. cont. k

- 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:



$p(1) = 0$

$p(2) = 0$

$p(3) = 1$

$p(4) = 0$

$p(5) = 3$

$p(6) = 3$

compute $p(k)$:

using binary search : time $O(\log k)$

all k: $O(n \log n)$

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:

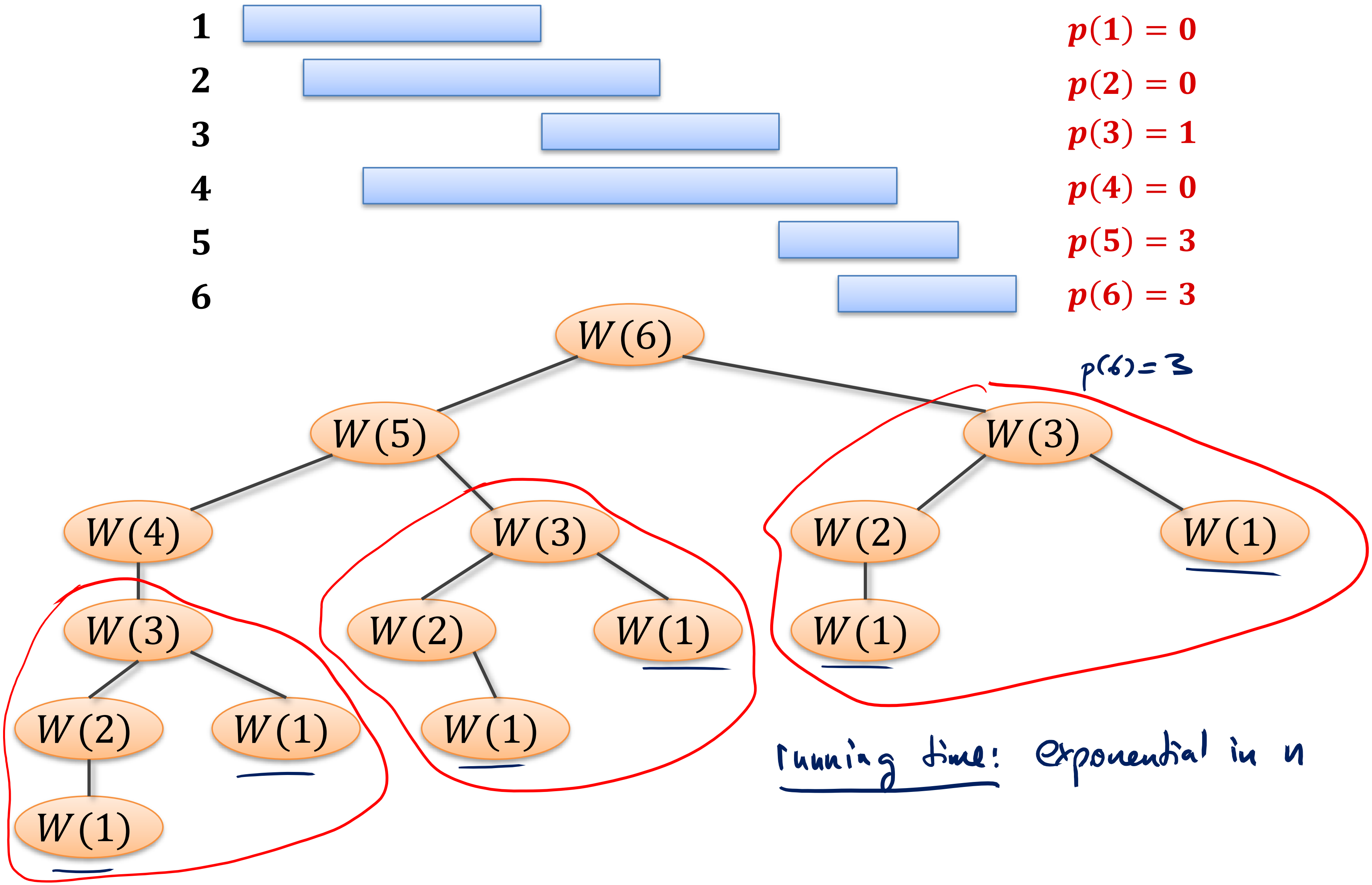
$$\forall k > 1:\ W(k) = \max\{W(k-1), w(k) + W(p(k))\}$$

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

Immediately gives a simple, recursive algorithm

```
Compute p(k) values for all k
W(k):
    if k == 1:
        x = w(1)
    else:
        x = max{W(k-1), w(k) + W(p(k))}
    return x
```

Running Time of Recursive Algorithm



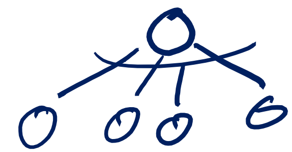
running time: exponential in n

Memoizing the Recursion

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

Memoization: Store already computed values for future rec. calls

```
Compute p(k) for all k
memo = {};
W(k):
    if k in memo: return memo[k]
    if k == 1:
        x = w(1)
    else:
        x = max{W(k-1), w(k) + W(p(k))}
    memo[k] = x
    return x
```



Dynamic Programming (DP)

DP ≈ Recursion + Memoization

Recursion: Express problem *recursively* in terms of (a 'small' number of) *subproblems* (of the same kind)

Memoize: *Store* solutions for *subproblems* reuse the stored solutions if the same subproblems has to be solved again

Weighted interval scheduling: subproblems $W(1), W(2), W(3), \dots$

runtime = #subproblems ⋅ time per subproblem

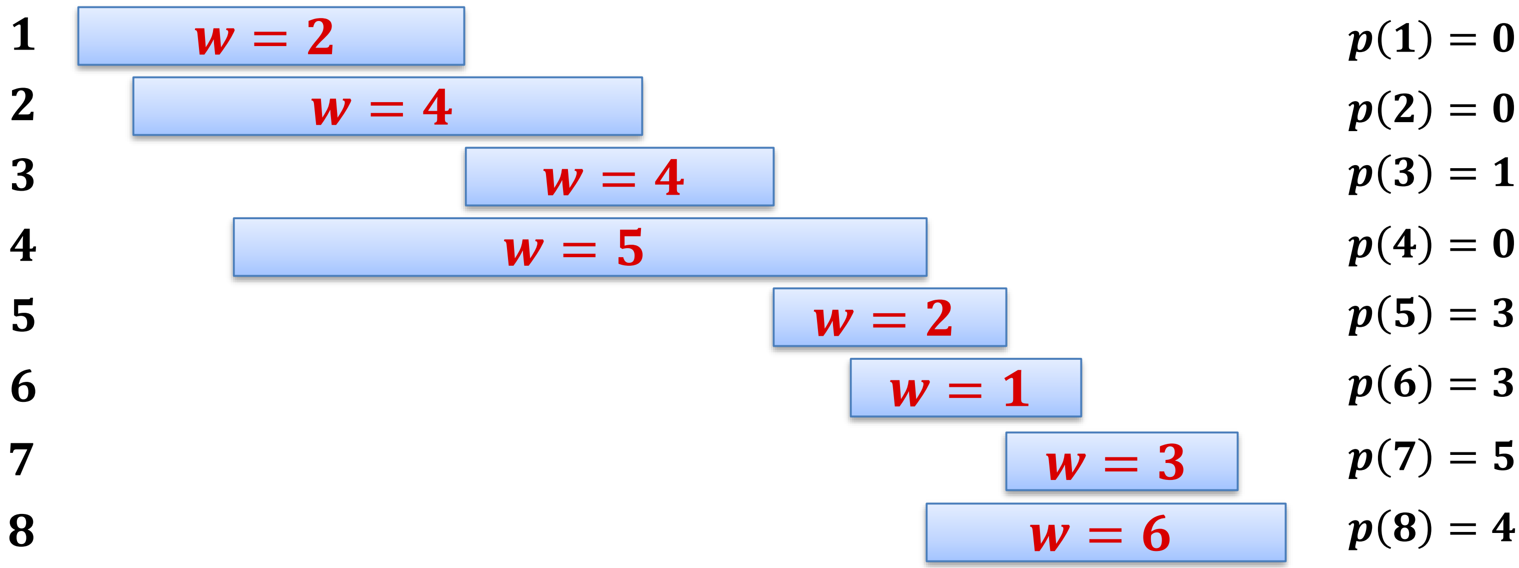
typically # subproblems considered in each call to the rec. func.

DP: Some History …

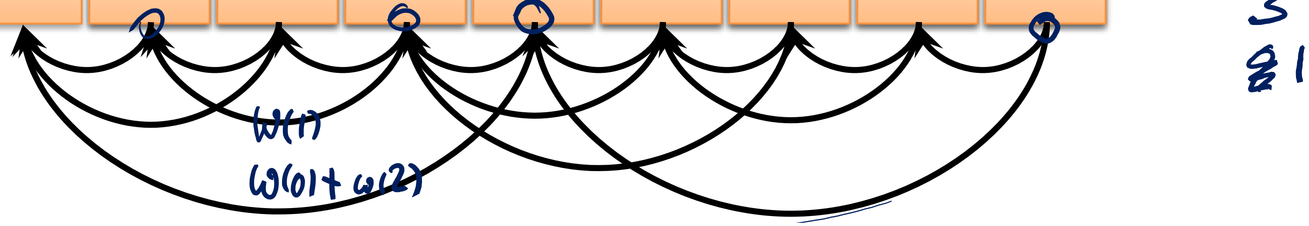
- Where das does the name come from?
- DP was developed by Richard E. Bellman in 1940s/1950s.
- In his autobiography, it says:

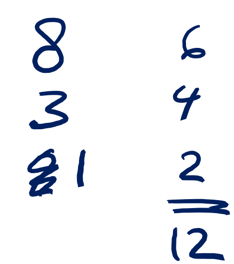
  "I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. … The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word research. … His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. … Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. … It also has a very interesting property as an adjective, and that it's impossible to use the word dynamic in a pejorative sense. … Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. …"

Example



	$W[0]$	$W[1]$	$W[2]$	$W[3]$	$W[4]$	$W[5]$	$W[6]$	$W[7]$	$W[8]$
W:	0	2	4	6	6	8	8	11	12





Computing the schedule: **store where you come from!**

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

Matrix-chain multiplication

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

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

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 $\langle A_1, A_2, A_3, A_4 \rangle$:

$$(A_1 \cdot (A_2 \cdot (A_3 \cdot A_4)))$$

$$(A_1 \cdot ((A_2 \cdot A_3) \cdot A_4))$$

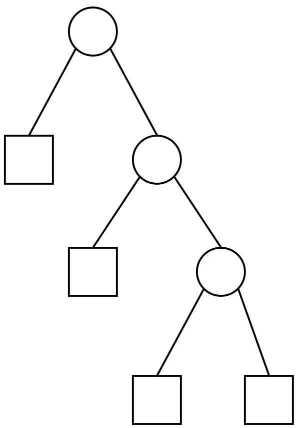
$$((A_1 A_2)(A_3 A_4))$$

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

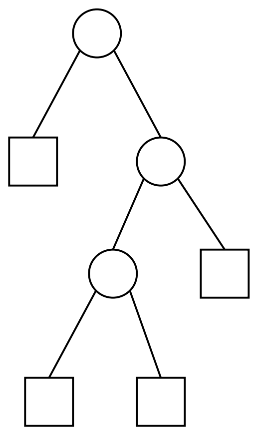
$$(((A_1 A_2) A_3) A_4)$$

Different parenthesizations

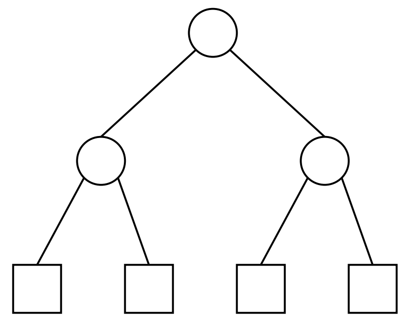
Different parenthesizations correspond to different trees:



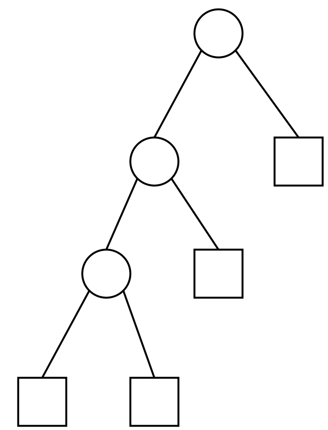
$$\big(A_1\big(A_2(A_3A_4)\big)\big)$$



$$\big(A_1\big((A_2A_3)A_4\big)\big)$$



$$\big((A_1A_2)(A_3A_4)\big)$$



$$\big(\big((A_1A_2)A_3\big)A_4\big)$$

Number of different parenthesizations

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

$$P(1) = 1$$

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

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

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

- 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] · B[k,j]
```

Remark:

Using this algorithm, multiplying two $(n \times n)$ matrices requires n^3 multiplications. This can also be done using $O(n^{2.373})$ 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 : (50×5) matrix

A_2 : (5×100) matrix

A_3 : (100×10) matrix

a) Parenthesization $((A_1 A_2) A_3)$ and $\left(A_1 (A_2 A_3)\right)$ require:

$A' = (A_1 A_2)$: 50 × 5 × 100 = 25'000 $\quad$ $A'' = (A_2 A_3)$: 5 × 100 × 10 = 5000

$A' A_3$: 50 × 100 × 10 = 50'000 $\quad$ $A_1 A''$: 50 × 5 × 10 = 2'500

Sum: 75'000 $\quad$ 7'500

Structure of an Optimal Parenthesization

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

  For some $1 \leq k < n$: $(\boldsymbol{A_{1\dots n}}) = \big((\boldsymbol{A_{1\dots k}}) \cdot (\boldsymbol{A_{k+1\dots n}})\big)$

- Any optimal solution contains optimal solutions for sub-problems

- Assume matrix A_i is a $(d_{i-1} \times d_i)$-matrix

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

- Then:

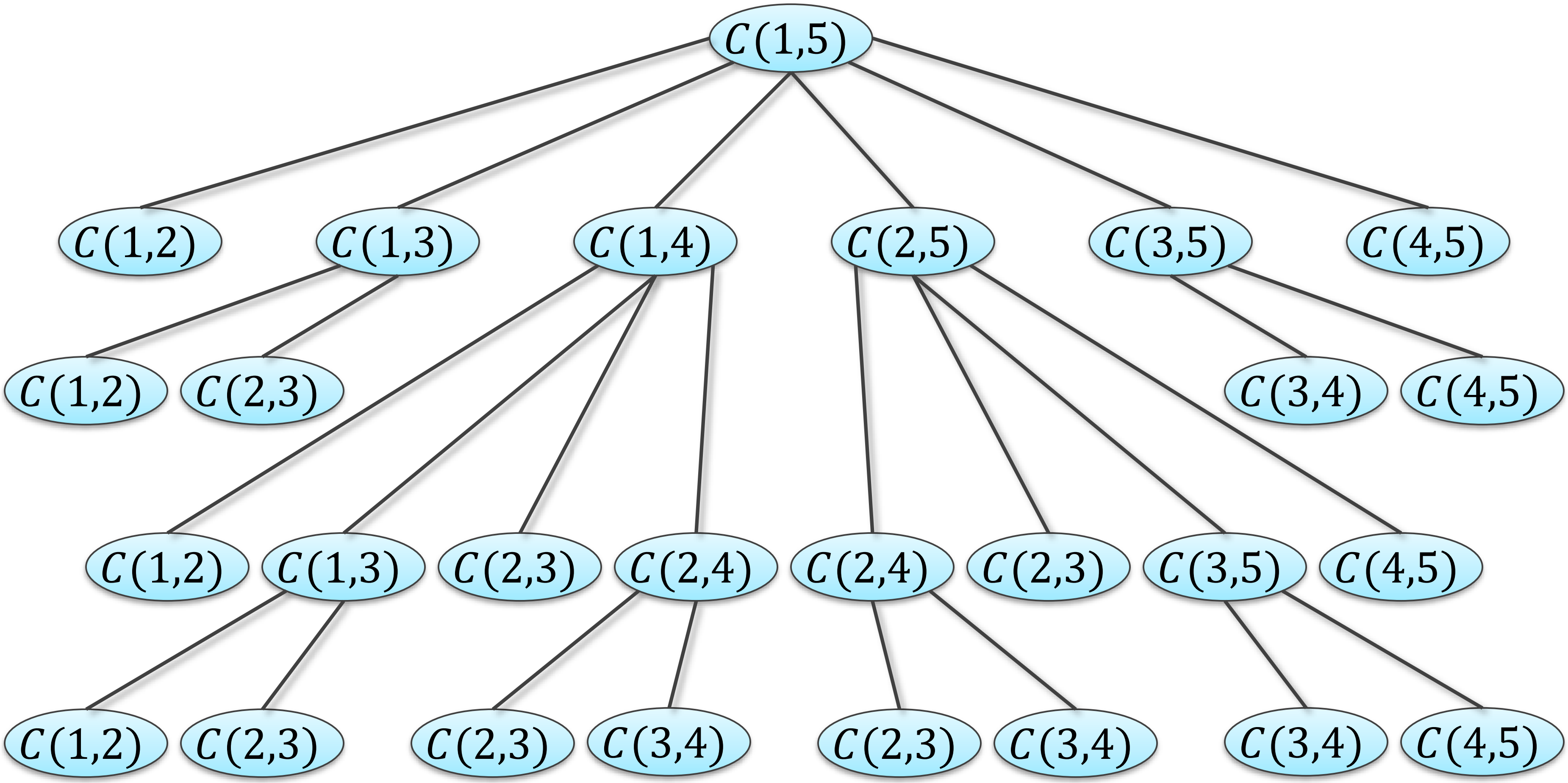
$$\boldsymbol{C(a,b) = \min_{a \leq k < b} C(a,k) + C(k+1,b) + d_{a-1} d_k d_b}$$

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

$a, \dots, k \bullet k+1, \dots, b$

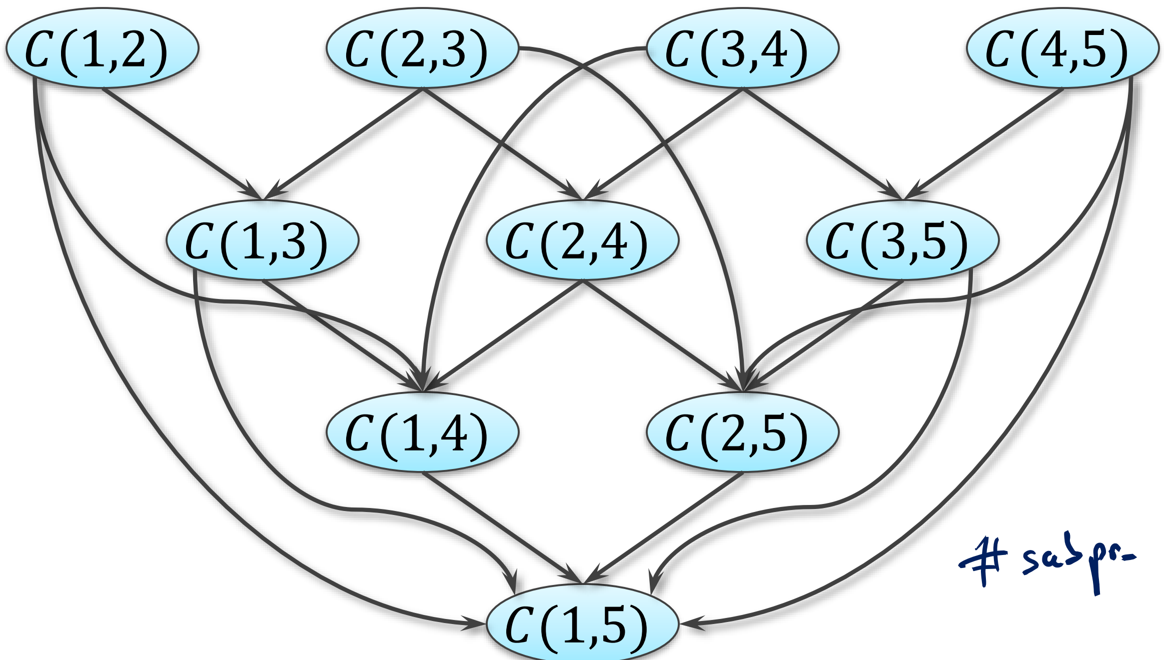
Recursive Computation of Opt. Solution

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



Using Meomization

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



subpr. : $O(n^2)$

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: $\boldsymbol{O(n^3)}$

Remarks about matrix-chain multiplication

1. There is an algorithm that determines an optimal parenthesization in time

$$O(n \cdot \log n).$$

2. There is a linear time algorithm that determines a parenthesization using at most

$$1.155 \cdot C(1, n)$$

multiplications.