universität freiburg

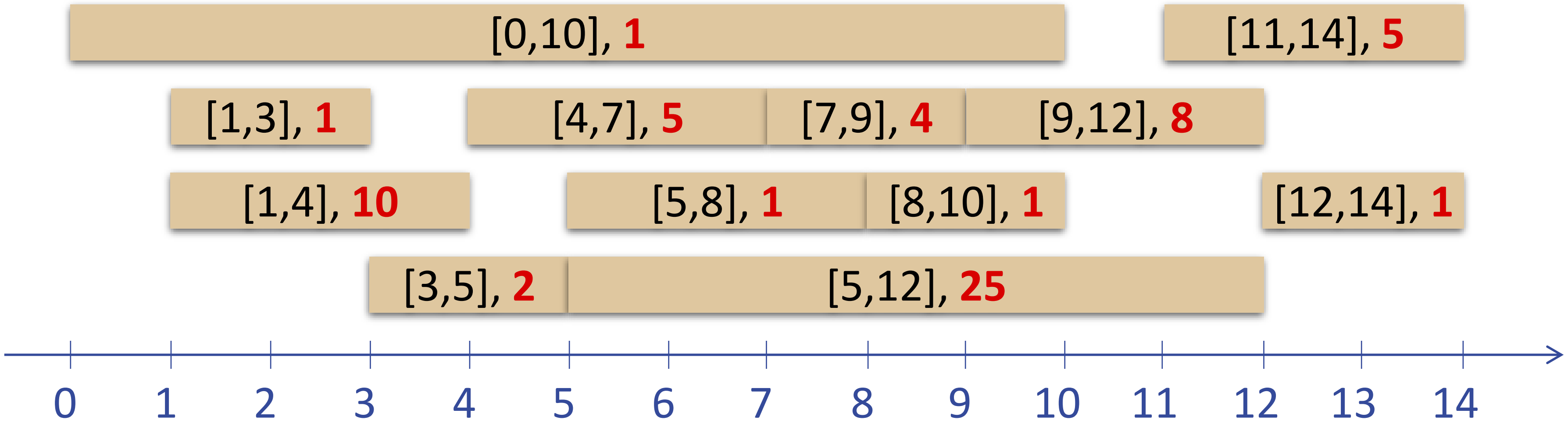
Algorithm Theory – WS 2024/25

Chapter 3 : Dynamic Programming

Fabian Kuhn
Dept. of Computer Science
Algorithms and Complexity

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**

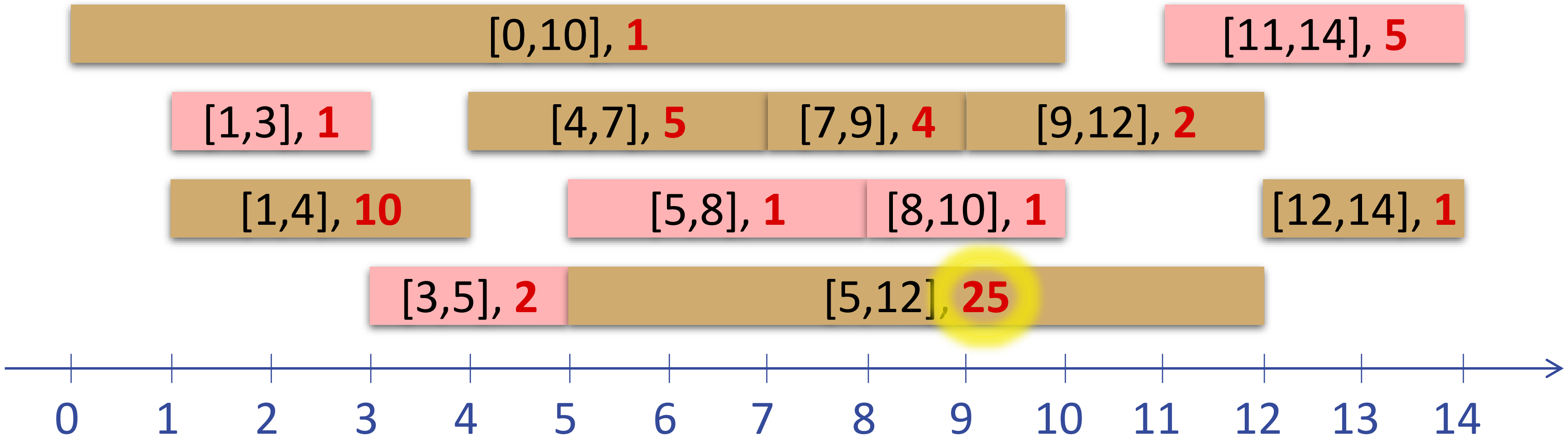


- **Goal:** Non-overlapping set of intervals of largest possible weight
  - Overlap at boundary ok, i.e., [4,7] and [7,9] are non-overlapping

- **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 for $i = 1, \dots, n$:**
  - start time $s(i) \geq 0$, finishing time $f(i) > s(i)$, weight $w(i) \geq 0$
- 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 does or does not 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 **contains** interval n

In example:

Opt. sol. consists of interval n

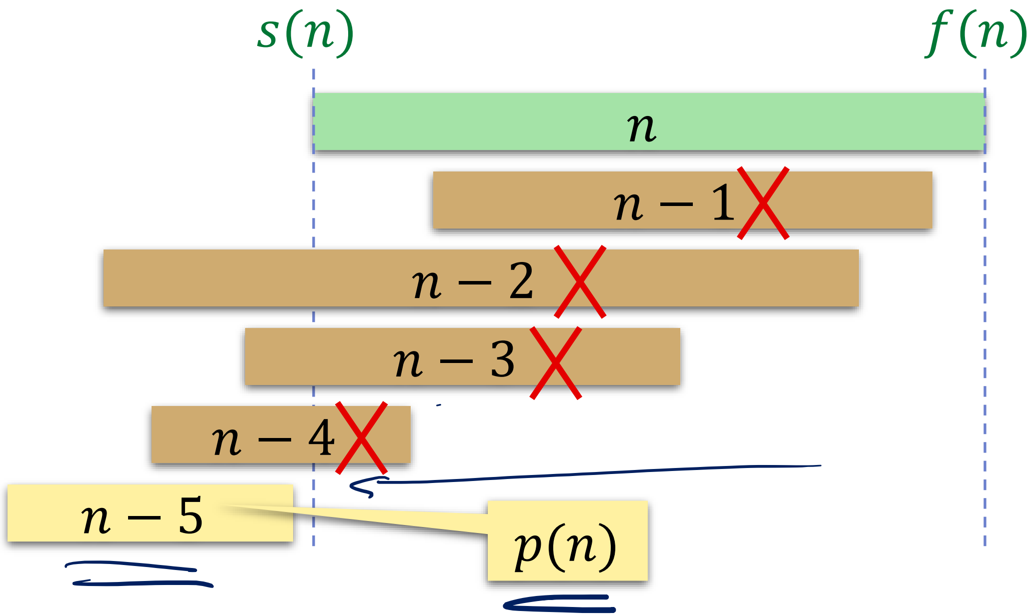
+

opt. sol. for intervals $1, \dots, n-5$

(last)

$p(n)$: first non-conflicting interval
here, $p(n) = n - 5$

$p(n) = \max_i \{ f(i) \leq s(n) \}$



Solving Weighted Interval Scheduling

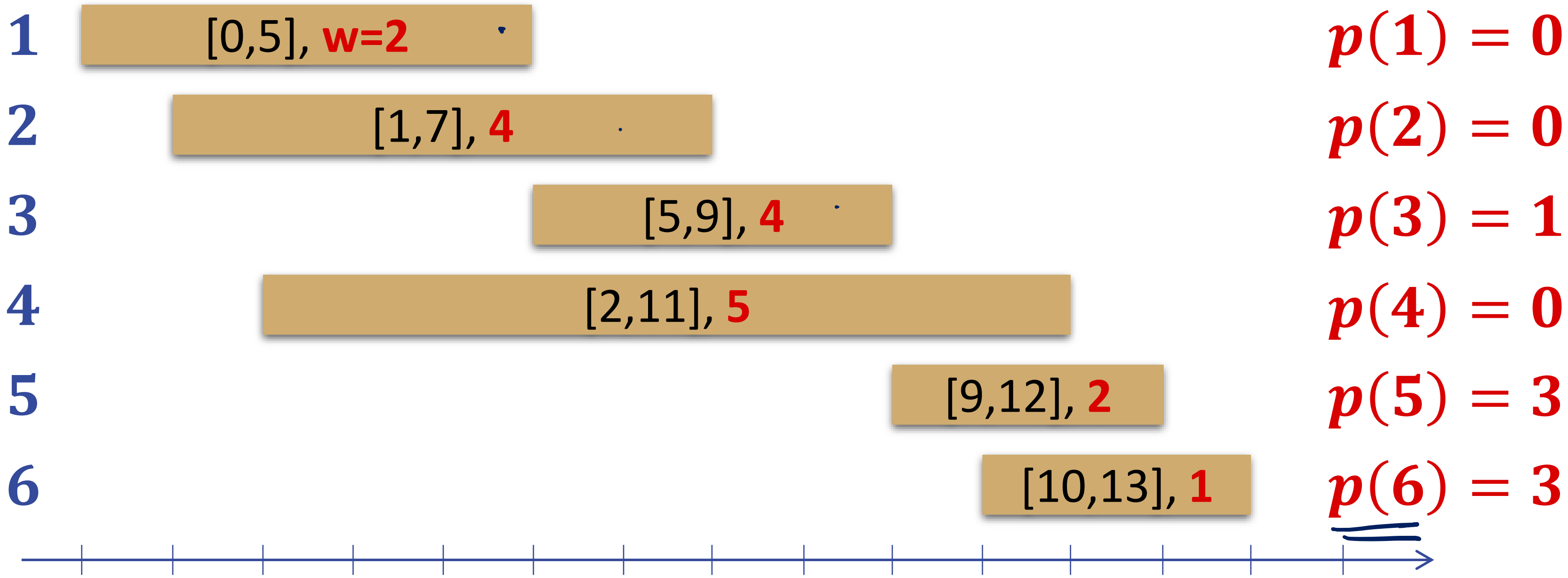
- **Interval $\boldsymbol{i}$ for $\boldsymbol{i = 1, \dots, n}$:**
  - start time $s(i) \geq 0$, finishing time $f(i) > s(i)$, weight $w(i) \geq 0$
- Assume intervals $1, \dots, n$ are **sorted by increasing $\boldsymbol{f(i)}$**
  - $0 < f(1) \leq f(2) \leq \cdots \leq f(n)$, for convenience: $f(0) = 0$

Simple observation: Opt. solution does or does not contain interval n

- Define $\underline{p(k) \coloneqq \max\{i \in \{0, \dots, k-1\} : f(i) \leq s(k)\}} \leftarrow$
- Weight of optimal solution $\underline{\mathrm{OPT}_k}$ for only intervals $1, \dots, k$: $\underline{W(k)}$
- Solution OPT_k does not contain interval k: $\boldsymbol{\underline{W(k)} = \underline{W(k-1)}}$
  - Weight of optimal solution with only first $k-1$ intervals
- Solution OPT_k contains interval k: $\boldsymbol{\underline{W(k)} = \underline{w(k)} + \underline{W(p(k))}}$
  - Weight of interval k plus weight of optimal solution of non-conflicting earlier intervals

Example

Interval:



1 [0,5], w=2 $p(1) = 0$

2 [1,7], 4 $p(2) = 0$

3 [5,9], 4 $p(3) = 1$

4 [2,11], 5 $p(4) = 0$

5 [9,12], 2 $p(5) = 3$

6 [10,13], 1 $\underline{p(6)} = 3$

Time to compute values $\underline{p(k)}$:

- Assume that intervals are already sorted by finishing time.
- For each k, do a binary search $\Longrightarrow$ time $O(\log k)$
- Overall time: $O(n \log n)$

Recursive Definition of Optimal Solution

- Recall:
  - $W(k)$: weight of optimal solution with intervals $1, \dots, k$
  - $p(k)$: last interval that finishes before interval k starts
    - $p(k) = 0$ if there is no interval that finishes before interval k starts

- Recursive definition of optimal weight:

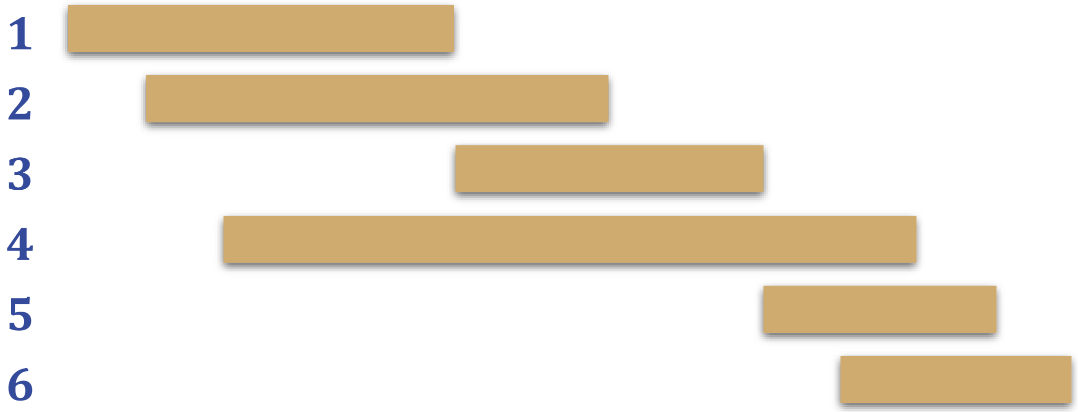
$$\forall k > 1:\ \underline{W(k)} = \max\underline{\{\underline{W(k-1)}, \underline{w(k) + W\big(p(k)\big)}\}}$$

$$\underline{W(0) = 0}$$

Immediately gives a simple, recursive algorithm

```
Compute p(k) values for all k
W(k):
    if k == 0:
        x = 0
    else:
        x = max{W(k-1), w(k) + W(p(k))}
    return x
```

Running Time of Recursive Algorithm



$p(1) = 0$

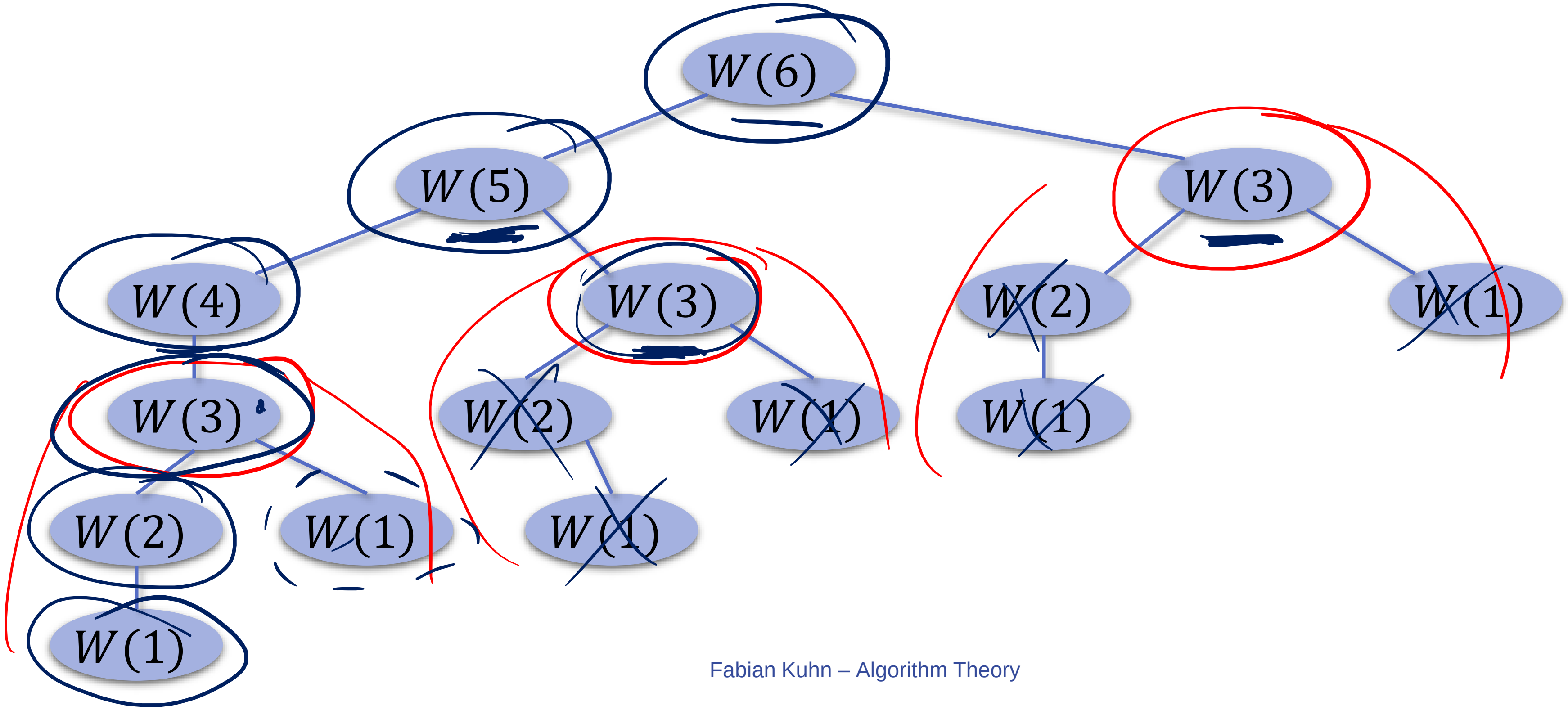
$p(2) = 0$

$p(3) = 1$

$p(4) = 0$

$p(5) = 3$

$p(6) = 3$



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 rec. calls

```
Compute p(k) for all k
memo = {};
W(k):
    if k in memo: return memo[k]
    if k == 0:
        x = 0
    else:
        x = max{W(k-1), w(k) + W(p(k))}
    memo[k] = x
    return x
```

Dynamic Programming (DP)

DP ≈ Recursion + Memoization

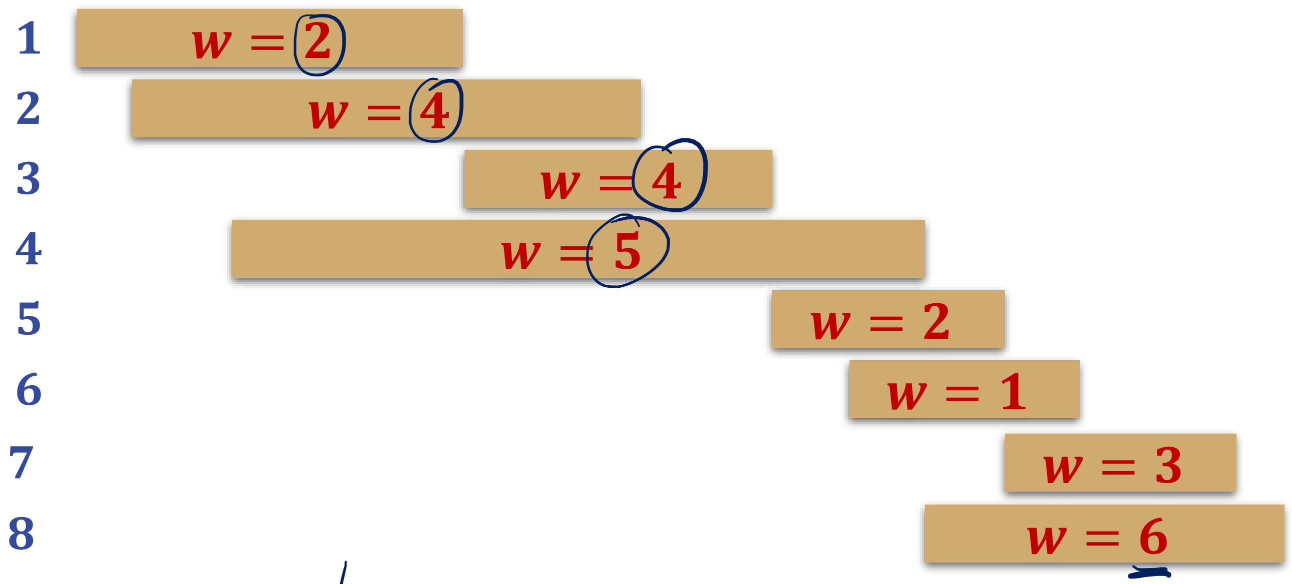
Recursion: Express problem *recursively* in terms of
(a 'small' number of) *subproblems* (of the same kind)

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

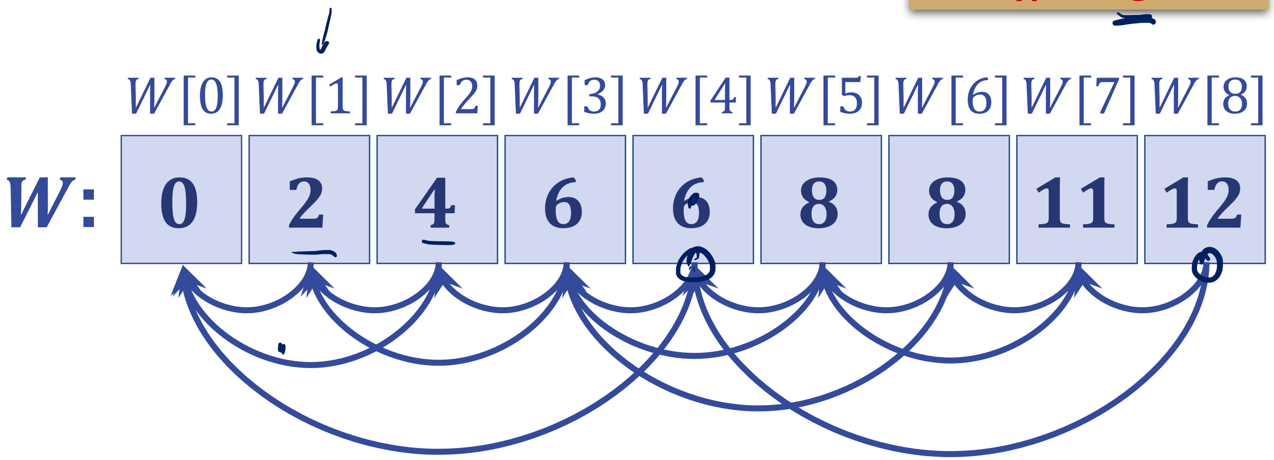
Weighted interval scheduling: subproblems $W(1), W(2), W(3), \dots$

runtime $=$ #subproblems $\cdot$ time per subproblem

Bottom-Up & Computing the Solution



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



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

take 8
$p(8) = 4$
not take 4
take 3
take 1

Computing the schedule: store where you come from!

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. …"

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!).

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

  Time is at least linear in the number of subproblems.

Computing the solution:

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

Matrix-chain multiplication

$$A_1 \cdot A_2 \cdot A_3$$

$$(A_1 \cdot A_2) \cdot A_3 \quad \leftarrow$$

$$A_1 \cdot (\underline{A_2} \cdot \underline{A_3}) \quad \leftarrow$$

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

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

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

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

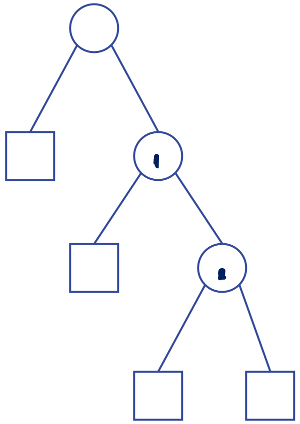
$$(A_1((A_2A_3)A_4))$$

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

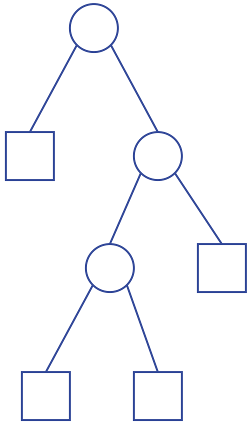
$$((A_1(A_2A_3))A_4)$$

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

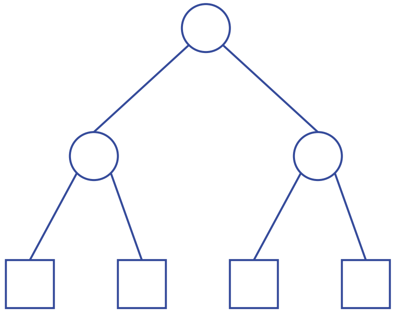
Different parenthesizations correspond to different trees



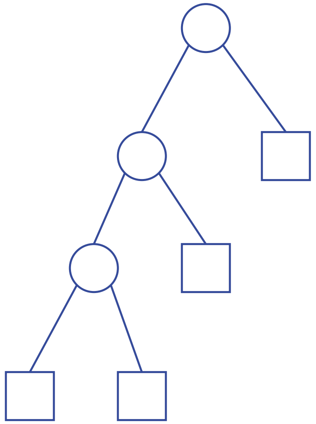
$$\left(A_1\left(A_2\left(\underline{A_3A_4}\right)\right)\right)$$



$$\left(A_1\left(\left(A_2A_3\right)A_4\right)\right)$$



$$\left(\left(A_1A_2\right)\left(A_3A_4\right)\right)$$



$$\left(\left(\left(A_1A_2\right)A_3\right)A_4\right)$$

Number of different parenthesizations

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

$$P(1) = 1$$

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

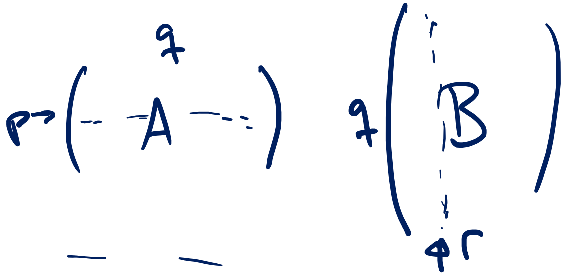
$$P(n+1) = \frac{1}{n+1}\binom{2n}{n} \approx \frac{4^n}{n\sqrt{\pi n}} + O\left(\frac{4^n}{\sqrt{n^5}}\right)$$

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

- 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
            $C[i,j] := C[i,j] + A[i,k]\cdot B[k,j]$
```

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

Remark:

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

using divide-and-conquer

Matrix-chain multiplication: Example

Computation of the product $A_1 A_2 A_3$, where

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

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

A_3 : $(\underline{100 \times 10})$ matrix

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

$A' = (A_1 A_2)$: $50 \cdot 5 \cdot 100 = 25'000$ $\boxed{50 \times 100}$

$A'' = (A_2 A_3)$: $5 \cdot 100 \cdot 10 = 5'000$ $\boxed{5 \times 10}$

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

$A_1 A''$: $50 \cdot 5 \cdot 10 = 2'500$

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 $1 \le k < n$: $\underline{(\boldsymbol{A_{1\ldots n}})} = \Big(\underline{(\boldsymbol{A_{1\ldots k}})} \overset{\downarrow}{\cdot} \underline{(\boldsymbol{A_{k+1\ldots n}})}\Big)$

- Any optimal solution contains optimal solutions for sub-problems

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

$(A_{\ell\ldots r})$

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

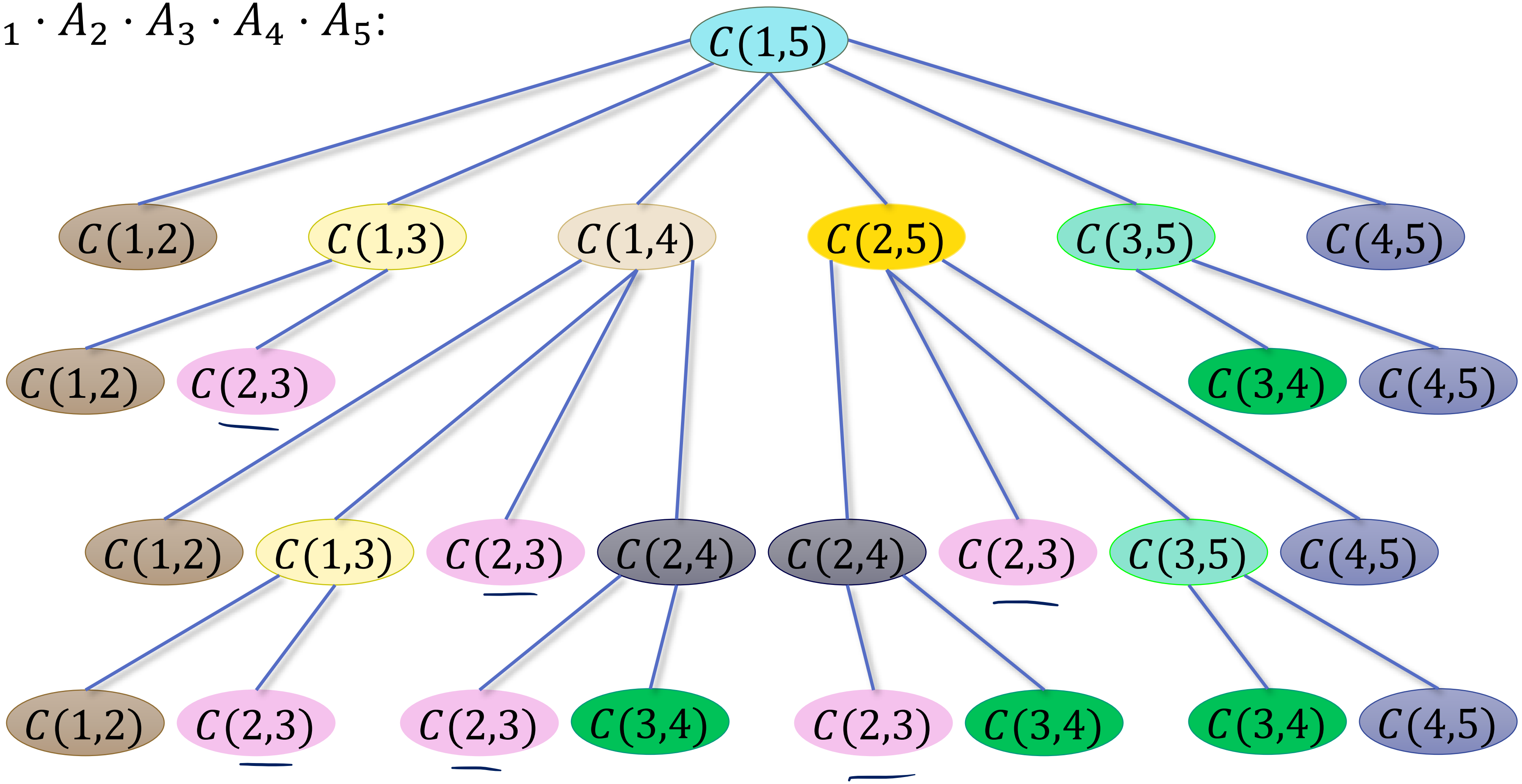
- Then:

$$\underline{C(\ell, r)} = \min_{\underline{\ell \le k < r}} \{C(\ell, k) + C(k+1, r) + \underline{d_{\ell-1} d_k d_r}\}$$

$$C(\ell, \ell) = 0$$

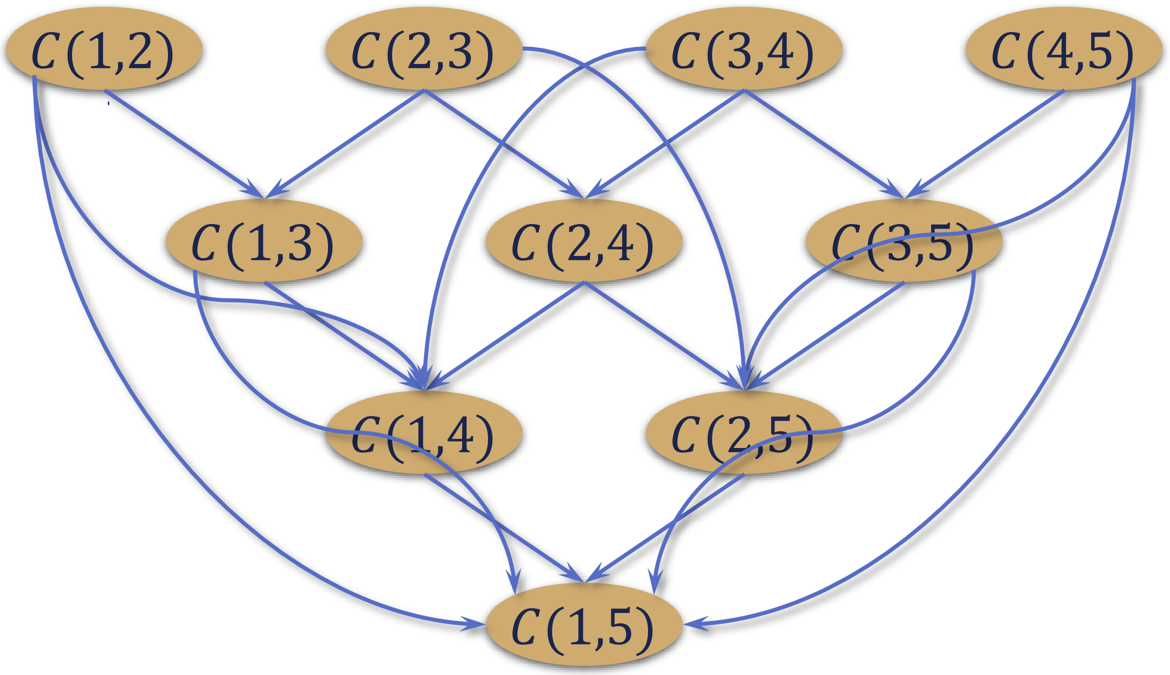
Recursive Computation of Opt. Solution

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



Using Meomization

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



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

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

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

Remarks about matrix-chain multiplication

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

$$\underline{O(n \cdot \log n)}.$$

[Hu, Shing; 1980]

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

$$\underline{1.155 \cdot C(1, n)}$$

multiplications.

[Hu, Shing; 1981]

Knapsack

- n items $1, \dots, n$, 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:

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

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

- 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?

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

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

This is not just $\mathrm{OPT}(n-1)$.

A More Complicated Recursion

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

Recursion: $$\mathbf{OPT}(k,x) = \max\Big\{\underbrace{\mathbf{OPT}(k-1,x)}_{\text{opt. solution when not using item } k},\ \overbrace{v_k + \mathbf{OPT}(k-1,\underbrace{x-w_k}_{\text{remaining capacity}})}^{\text{opt. solution when using item } k\text{, only possible if } x \geq w_k}\Big\}$$

Initialization:

- $\mathrm{OPT}(0,x) = 0$
  - no items $\Longrightarrow$ value 0
- $\mathrm{OPT}(k,0) = 0$
  - capacity $0 \Longrightarrow$ value 0

Number of subproblems:

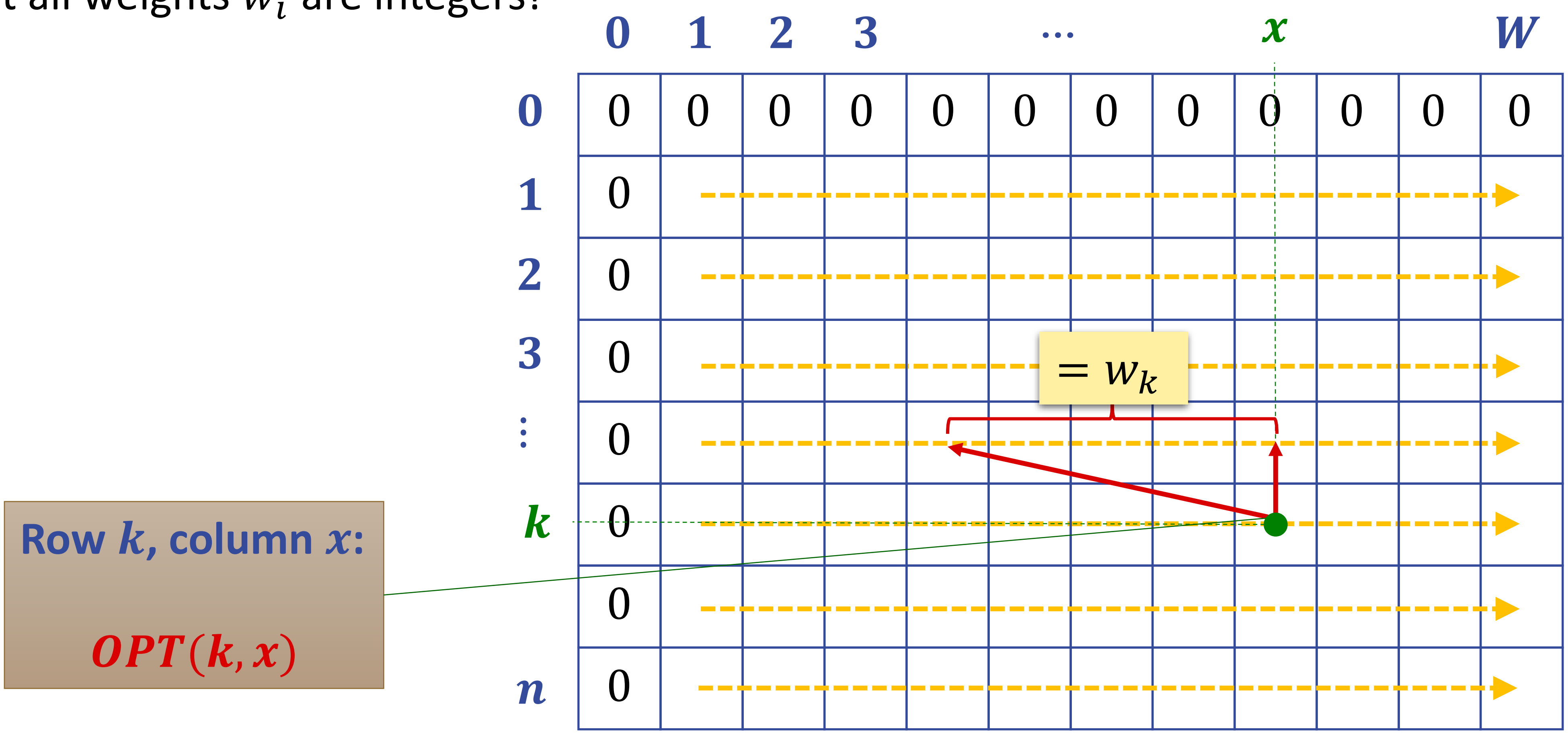
- arbitrary weights: $\leq n \cdot 2^n$
  - In this case, the problem is NP-hard.
- integer weights: $n \cdot W$
  - 2 cases per subproblem

$\Longrightarrow$ **running time: $O(n \cdot W)$**

Dynamic Programming Algorithm

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

- Assume that all weights w_i are integers!



Row k, column x:

$OPT(k, x)$

Example

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

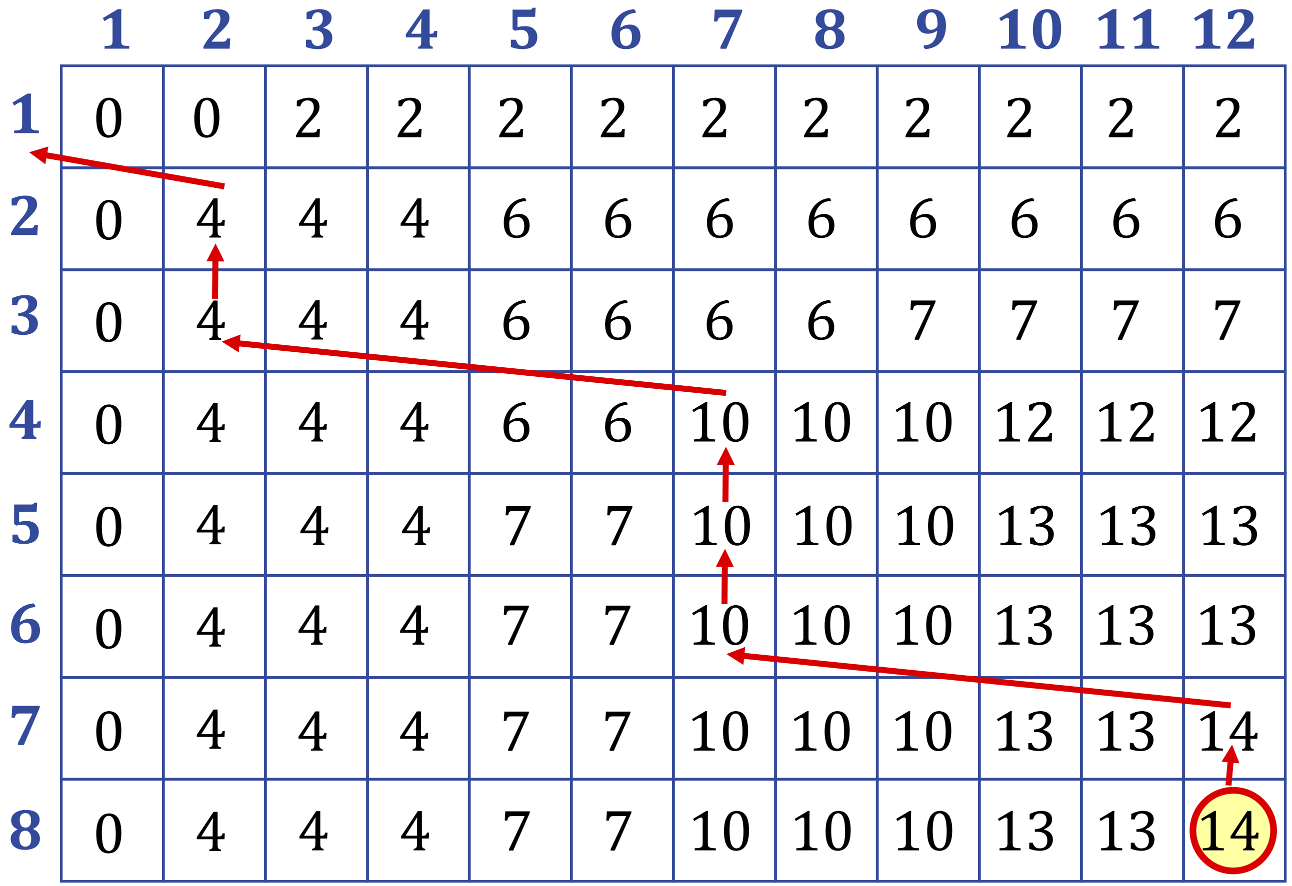
  weight **value**

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

Optimal solution:
Items 2, 4, and 7

Total weight:
$2 + 5 + 5 = 12$

Total value:
$4 + 6 + 4 = 14$



	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	2	2	2	2	2	2	2	2	2	2
2	0	4	4	4	6	6	6	6	6	6	6	6
3	0	4	4	4	6	6	6	6	7	7	7	7
4	0	4	4	4	6	6	10	10	10	12	12	12
5	0	4	4	4	7	7	10	10	10	13	13	13
6	0	4	4	4	7	7	10	10	10	13	13	13
7	0	4	4	4	7	7	10	10	10	13	13	14
8	0	4	4	4	7	7	10	10	10	13	13	14

Running Time of Knapsack Algorithm

- **Size of table:** $O(n \cdot W)$

- Time per table entry: $O(1)$ → **overall time:** $\boldsymbol{O(n \cdot W)}$

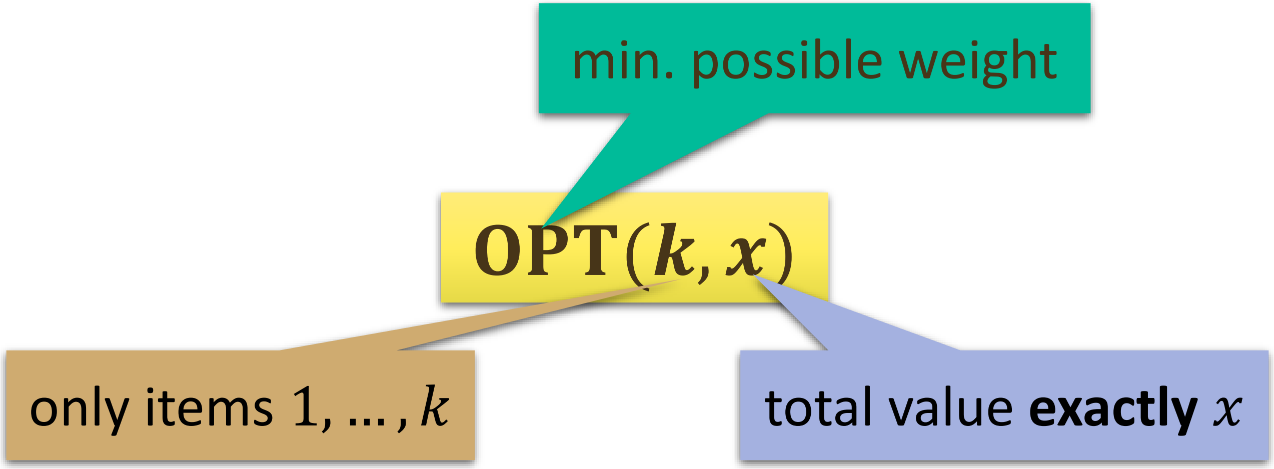
- Computing solution (set of items to pick):
  Follow $\leq n$ arrows → $O(n)$ time (after filling table)

- Note: Time depends on W → can be exponential in n…
- And it only works if all weights are integers
  - … or can be scaled so that they are integers

Knapsack with Integer Values

- Let's also consider the case that weights are arbitrary and the values are integers…
- Assume that all item values are integers $\in \{1, \dots, V\}$
- Again distinguish two cases depending on if the last item is part of an optimal solution or it isn't.

Recursive Function:



min. possible weight

$\mathbf{OPT}(k, x)$

only items $1, \dots, k$

total value **exactly** x

Knapsack with Integer Values

- Assume that all item values are integers $\in \{1, \dots, V\}$

Recursive Function:

- $\mathbf{OPT}(\boldsymbol{k}, \boldsymbol{x})$**:** min. possible weight to achieve exactly value x with only items $1, \dots, k$
- **Recursive definition of function** $\mathbf{OPT}(\boldsymbol{k}, \boldsymbol{x})$

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

$$\mathrm{OPT}(k, 0) = 0$$
$$\mathrm{OPT}(0, x) = \infty \ \text{ for } x > 0$$

only possible if $x \geq v_k$

- At the end, find maximum x such that $\mathrm{OPT}(n, x) \leq W$
- Number of subproblems $\leq n^2 \cdot V \Longrightarrow$ **running time** $\boldsymbol{O}(\boldsymbol{n}^2 \cdot \boldsymbol{V})$
  - Max. required x-value: $x \leq \sum_{i=1}^{n} v_k \leq n \cdot V$

Dynamic Programming : Summary

Dynamic Programming:

- Use recursion together with memorization
- Applicable if #recursive subproblems is moderately small

Additional Applications of Dynamic Programming:

- The idea can be applied to a wide range of problems
- Examples, beyond what we already saw:
  - Shortest path algorithms such as Bellman-Ford and Dijkstra can be seen as applications of DP
  - String comparison & matching problems such as edit distance, approximate text search, Biological sequence alignment problems, etc.
  - Further string problems: longest common subsequence, etc.
  - Hidden Markov model analysis
  - And many more …