universität freiburg

Algorithm Theory – WS 2024/25

Chapter 3 : Dynamic Programming 2
(Knapsack)

Fabian Kuhn
Dept. of Computer Science
Algorithms and Complexity

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).

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!

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

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

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

opt. solution when using item k, only possible if $x \geq w_k$

opt. solution when not using item k

remaining capacity

Initialization:

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

Number of subproblems:

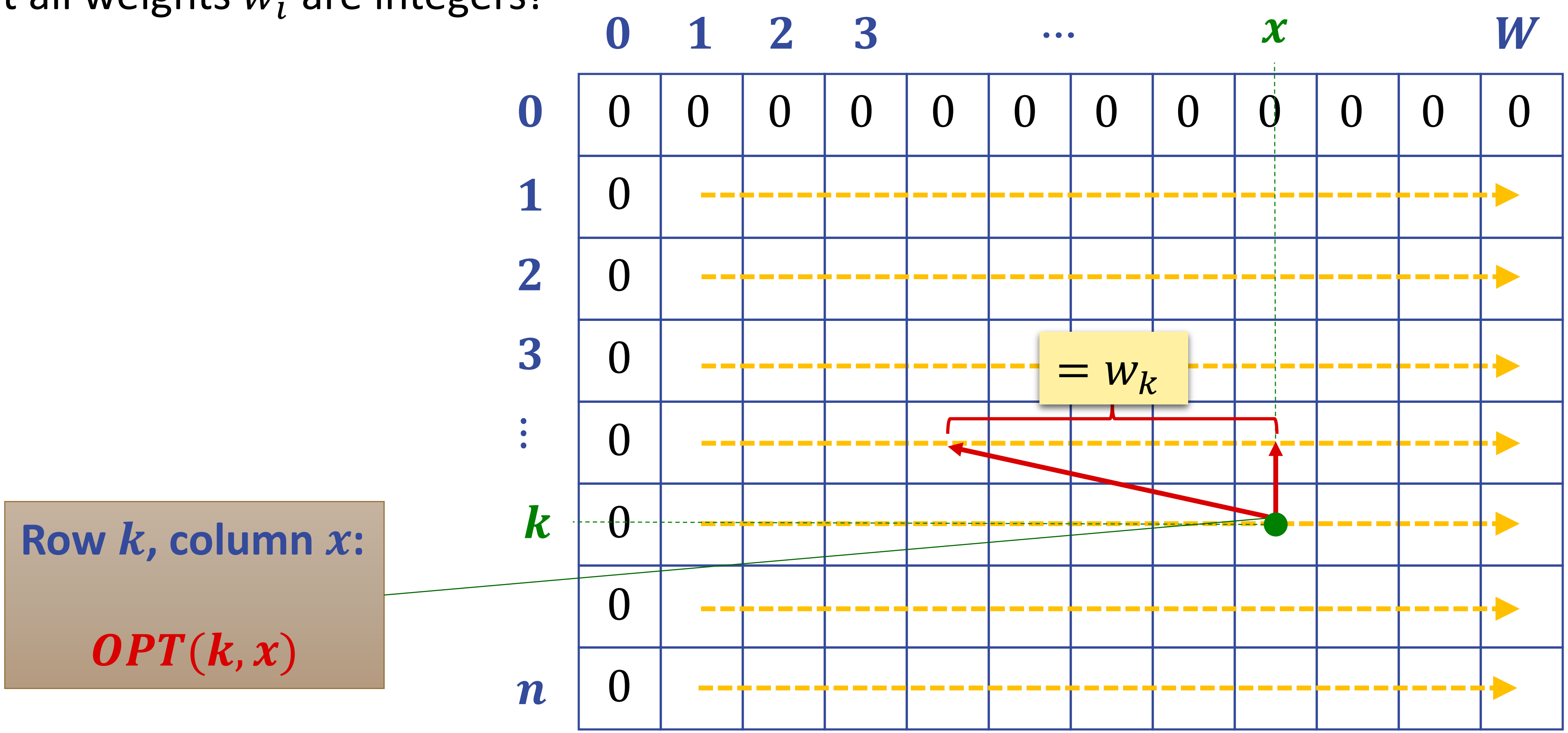
- arbitrary weights: $\leq n \cdot 2^n$
  - In this case, the problem is NP-hard.
- integer weights: $n \cdot W$
  - 2 cases per subproblem

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

Dynamic Programming Algorithm

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

- Assume that all weights w_i are integers!



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

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

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** $\boldsymbol{O(n^2 \cdot V)}$
  - 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 …