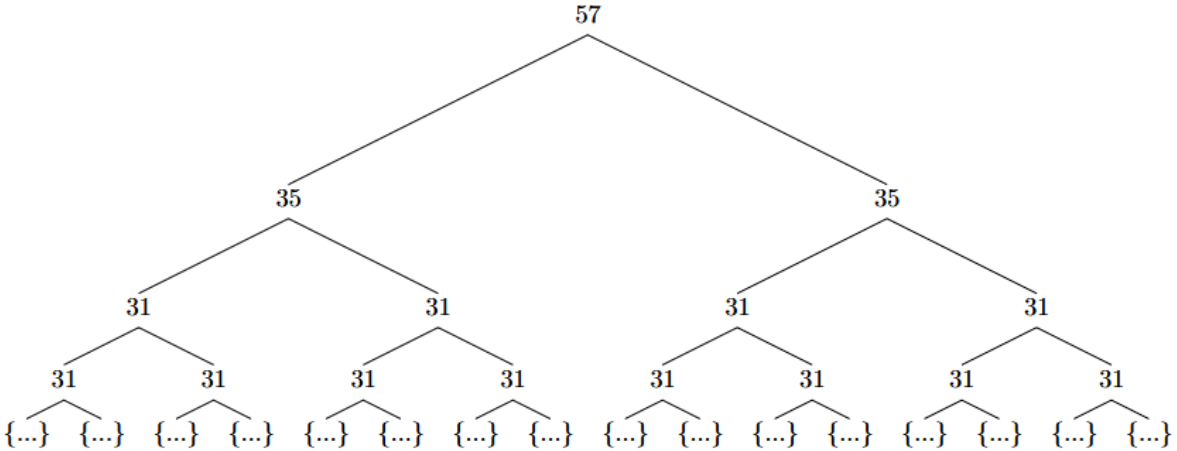
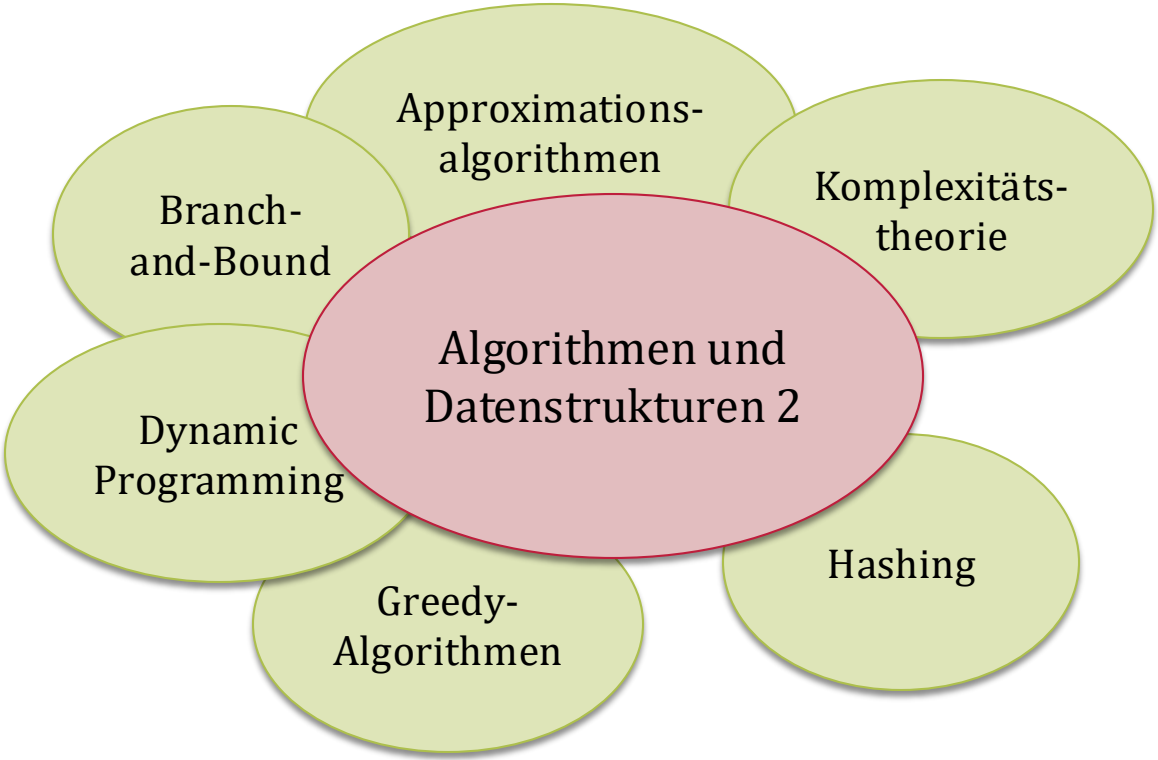
Technische Universität Braunschweig

Algorithmen und Datenstrukturen 2

Arne Schmidt

Enumerationsbaum




Approximations-
algorithmen
Branch-
and-Bound
Komplexitäts-
theorie
Algorithmen und
Datenstrukturen 2
Dynamic
Programming
Hashing
Greedy-
Algorithmen

Kapitel 2 – Dynamic Programming

Kapitel 2.1 – Dynamic Programming
Subset Sum

Fragen


Können wir Lösungen nach und nach konstruieren?
Was macht eine Lösung von Knapsack aus?

Überlegung

Betrachte die Instanz aus Beispiel 1.10 als Subset-Sum-Instanz:
$\{z_1, \dots, z_9\} = \{7, 13, 17, 20, 29, 31, 31, 35, 57\}$ mit $Z = 120$.

Welche Optionen gibt es?

Lassen wir die 57 aus, muss der Rest 120 ergeben.

Nehmen wir die 57, muss der Rest 63 ergeben.

Geht eine der beiden Optionen, geht es komplett.

Eine Rekursionsgleichung

$$\mathcal{S}(x,i) \coloneqq \begin{cases} 1, & \text{falls } x \text{ mit } z_1, \dots, z_i \text{ erzeugt werden kann.} \\ 0, & \text{sonst} \end{cases}$$

Basisfälle

$$\mathcal{S}(x,0) \coloneqq \begin{cases} 1, & \text{falls } x = 0. \\ 0, & \text{sonst} \end{cases}$$

$$\mathcal{S}(x,i) = \begin{cases} \mathcal{S}(x,i-1), & \text{falls } x < z_i \\ \max\left(\mathcal{S}(x,i-1), \mathcal{S}(x-z_i,i-1)\right), & \text{falls } x \geq z_i. \end{cases}$$

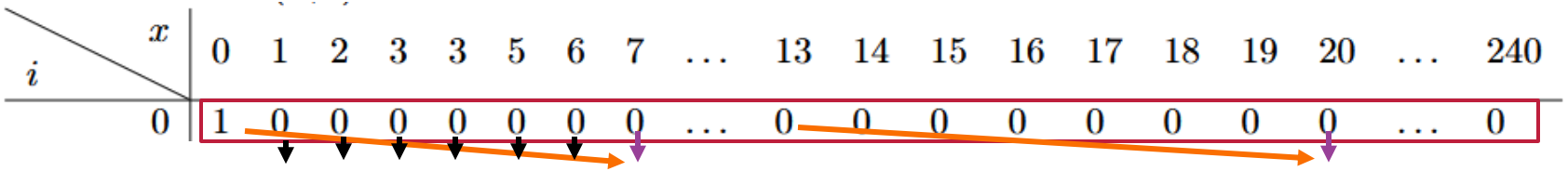
Was passiert, wenn wir es nicht nehmen?

Was passiert, wenn wir es nehmen?

Objekt passt eh nicht.

Wertetabelle

$\{z_1, \ldots, z_9\} = \{7, 13, 17, 20, 29, 31, 31, 35, 57\}$

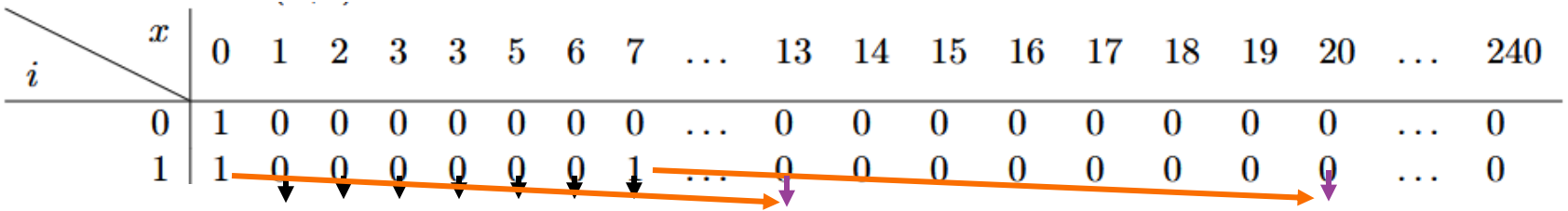


i \ x	0	1	2	3	3	5	6	7	…	13	14	15	16	17	18	19	20	…	240
0	1	0	0	0	0	0	0	0	…	0	0	0	0	0	0	0	0	…	0

Wertetabelle

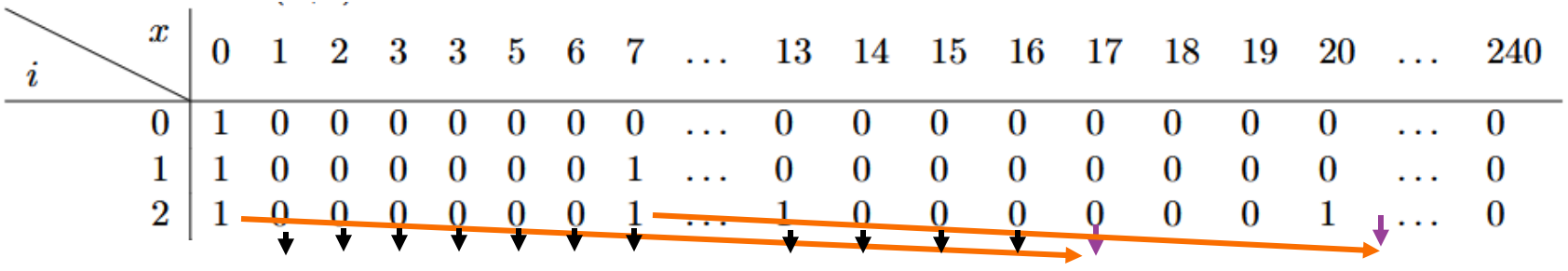
$\{z_1, \ldots, z_9\} = \{7, 13, 17, 20, 29, 31, 31, 35, 57\}$

i \ x	0	1	2	3	3	5	6	7	...	13	14	15	16	17	18	19	20	...	240
0	1	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	...	0
1	1	0	0	0	0	0	0	1	...	0	0	0	0	0	0	0	0	...	0



Wertetabelle

$\{z_1, \ldots, z_9\} = \{7, 13, 17, 20, 29, 31, 31, 35, 57\}$

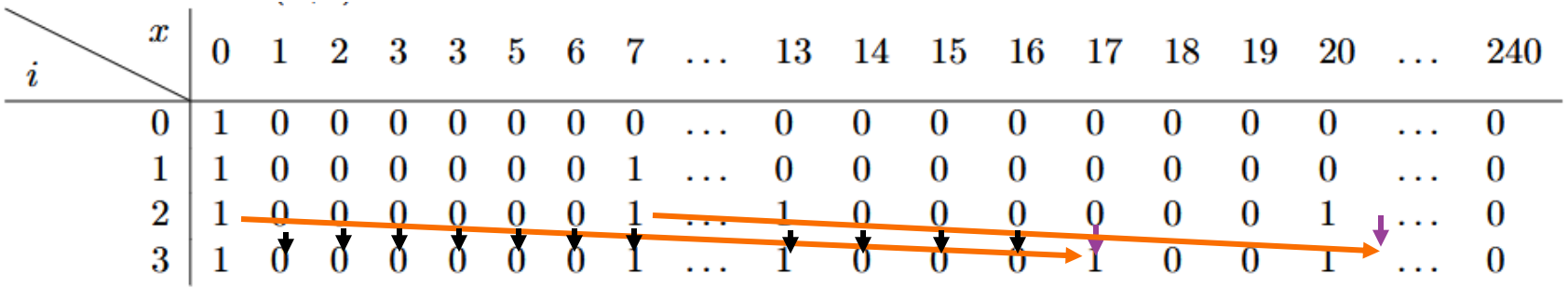


i \ x	0	1	2	3	3	5	6	7	…	13	14	15	16	17	18	19	20	…	240
0	1	0	0	0	0	0	0	0	…	0	0	0	0	0	0	0	0	…	0
1	1	0	0	0	0	0	0	1	…	0	0	0	0	0	0	0	0	…	0
2	1	0	0	0	0	0	0	1	…	1	0	0	0	0	0	0	1	…	0

Wertetabelle

$\{z_1, \ldots, z_9\} = \{7, 13, 17, 20, 29, 31, 31, 35, 57\}$

i \ x	0	1	2	3	3	5	6	7	...	13	14	15	16	17	18	19	20	...	240
0	1	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	...	0
1	1	0	0	0	0	0	0	1	...	0	0	0	0	0	0	0	0	...	0
2	1	0	0	0	0	0	0	1	...	1	0	0	0	0	0	0	1	...	0
3	1	0	0	0	0	0	0	1	...	1	0	0	0	1	0	0	1	...	0



Wir müssen also nur eine Tabelle mit Werten füllen und brauchen keine Rekursion!
Das halten wir als Algorithmus fest.

Algorithmus für Subset Sum

Satz 2.2:
Algorithmus 2.1 löst das Problem Subset Sum in Zeit $O(nZ)$.

Laufzeit ist im Wesentlichen klar. Wir füllen eine nZ große Tabelle aus. Für jede Zelle haben wir konstanten Aufwand.

Folgt über einen Induktionsbeweis aus vorherigen Ideen.

Algorithmus 2.1 Dynamic Programming für SUBSET SUM

Eingabe: Zahlen $z_1, ..., z_n$ mit $\sum_{i=1}^{n} z_i = Z$

Ausgabe: Boolesche Funktion $\mathcal{S} : \{0, ..., Z\} \times \{0, ..., n\} \to \{0, 1\}$

$$\text{mit } \mathcal{S}(x,i) = \begin{cases} 1, & \text{falls x mit } z_1, ..., z_i \text{ erzeugt werden kann} \\ 0, & \textit{sonst} \end{cases}$$

```
1:  S(0,0) := 1
2:  for (x=1) to Z do
3:      S(x,0) := 0;
4:  for (i = 1) to n do
5:      for (x = 0) to z_i − 1 do
6:          S(x,i) := S(x,i−1);
7:      for (x = z_i) to Z do
8:          if ((S(x,i−1) = 1) OR (S((x−z_i),i−1) = 1)) then
9:              S(x,i) := 1;
10:         else
11:             S(x,i) := 0;
12: return
```

Fragen II

Sparen wir uns hier überhaupt Zeit?

Müssen wir nicht wieder alle 2^n Möglichkeiten durchprüfen?!

Laufzeit

$\{z_1, \ldots, z_9\} = \{7, 13, 17, 20, 29, 31, 31, 35, 57\}$

i \ x	0	1	2	3	3	5	6	7	...	13	14	15	16	17	18	19	20	...	240
0	1	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	...	0
1	1	0	0	0	0	0	0	1	...	0	0	0	0	0	0	0	0	...	0
2	1	0	0	0	0	0	0	1	...	1	0	0	0	0	0	0	1	...	0
3	1	0	0	0	0	0	0	1	...	1	0	0	0	1	0	0	1	...	0

Wie viele Einsen enthält eine Zeile maximal?
Sie verdoppeln sich maximal jede Zeile, können aber Z nicht übersteigen!

→ Wir sparen Zeit, wenn $2^n > Z$ ist!

Moment!

Wie groß kann Z überhaupt werden?!

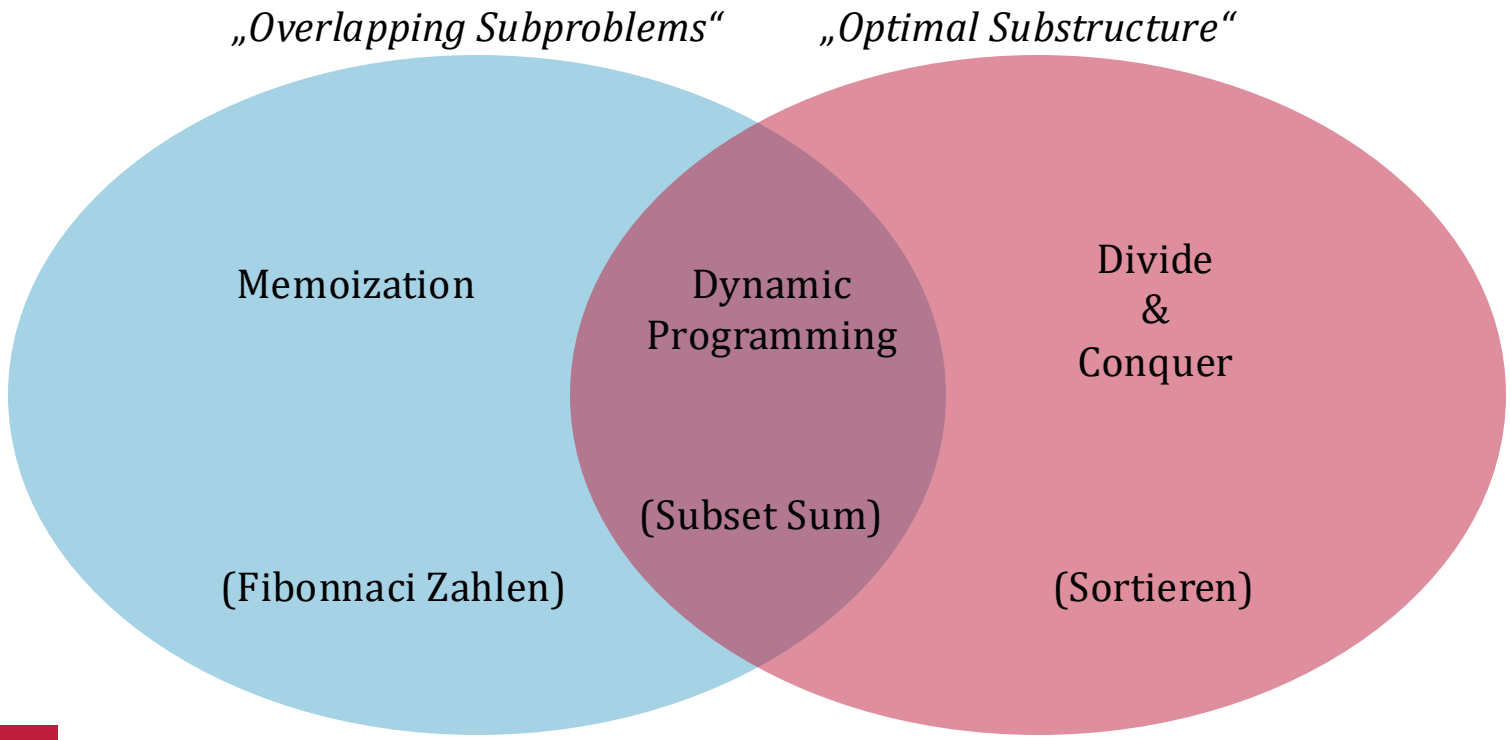
Z wird binär codiert, hat im Input also Größe $m := \log Z$.

Damit ist die Laufzeit des Algorithmus $O(n \cdot 2^m)$, ist also nicht polynomiell?!

Definition:
Ein Algorithmus besitzt *pseudopolynomielle* Laufzeit, wenn die Laufzeit polynomiell im Wert der Eingabe, aber nicht polynomiell in der Größe der Eingabe ist.

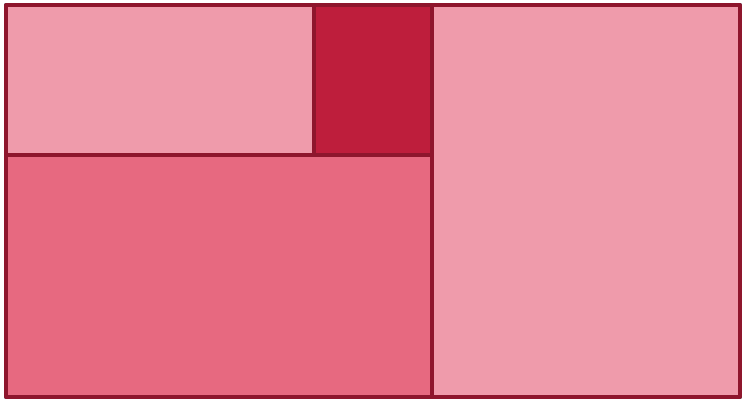
Dynamic Programming im Allgemeinen

Dynamic Programming



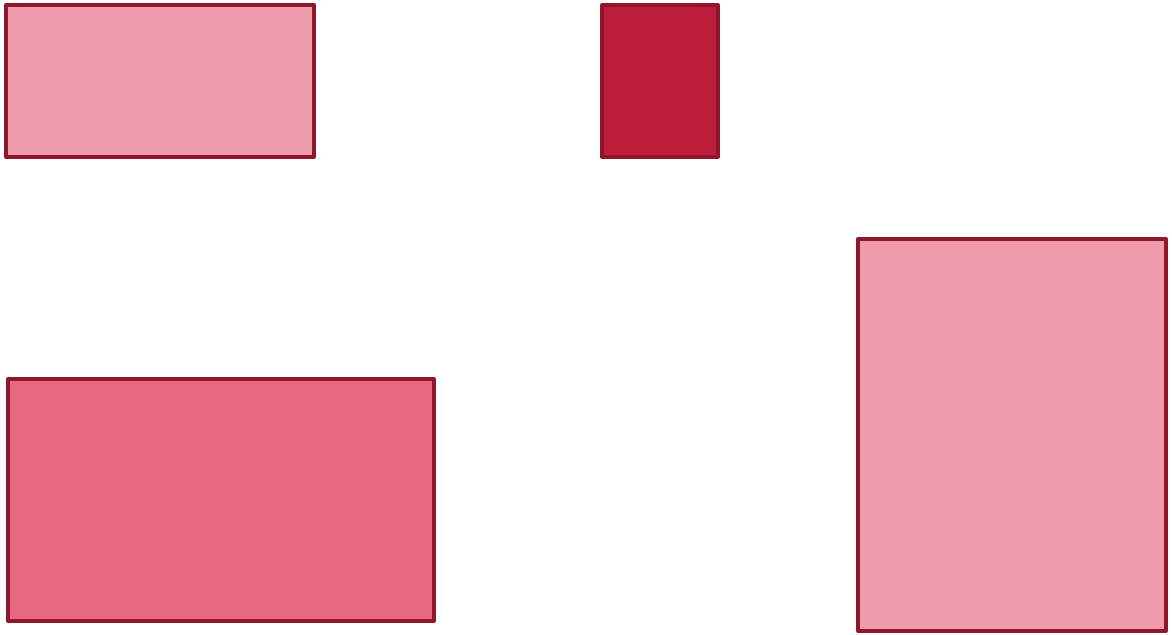
Optimal Substructure

Aufteilen in Teilprobleme



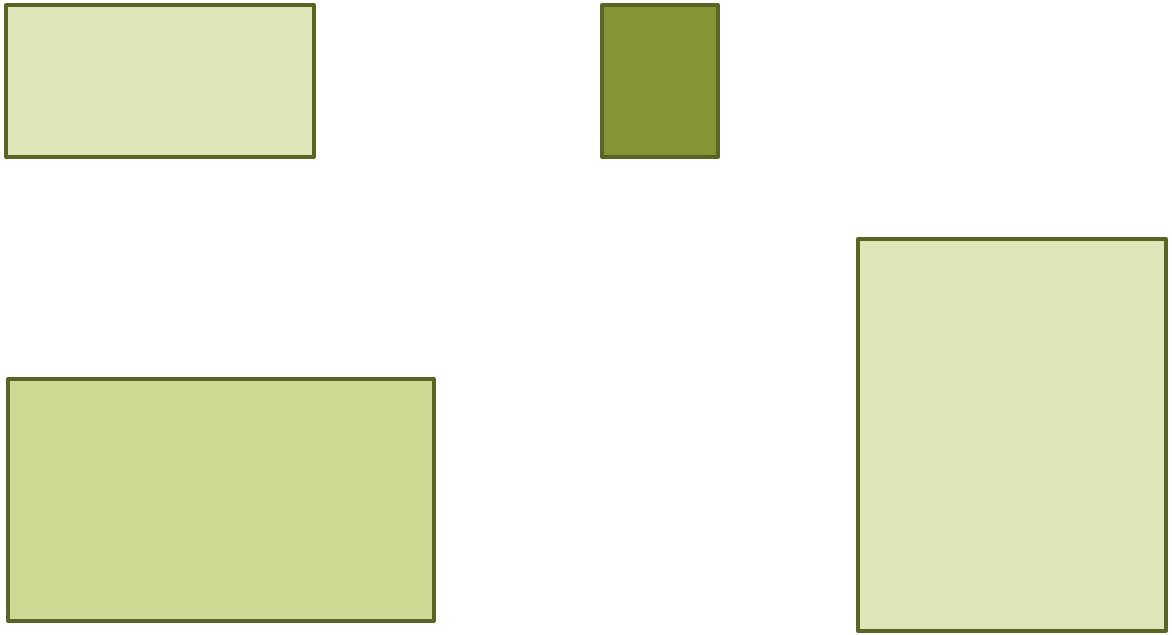
Optimal Substructure

Lösen der Teilprobleme



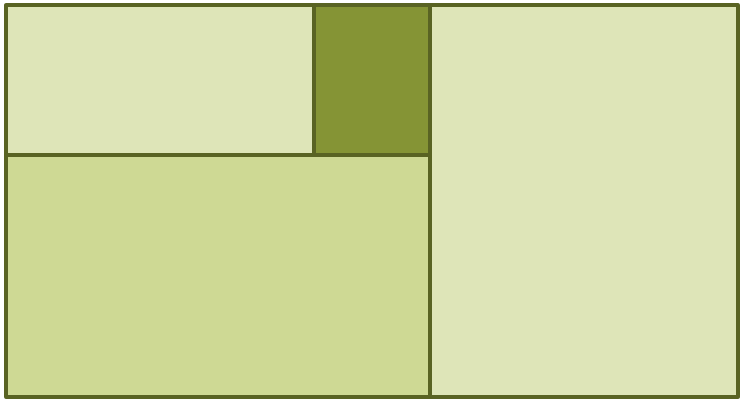
Optimal Substructure

Lösen des Hauptproblems



Optimal Substructure

Lösen des Hauptproblems



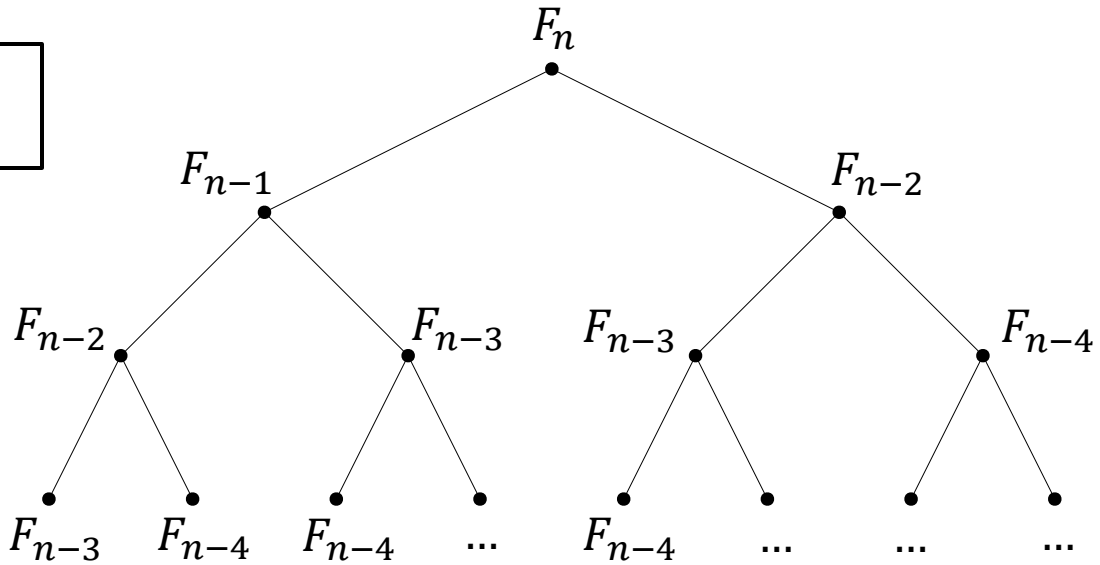
Optimal Substructure

Lösen des Hauptproblems

Overlapping Subproblems

Fibonnaci-Zahlen:

$F_n := F_{n-1} + F_{n-2}, \quad F_0 = 0, \quad F_1 = 1$



Input : $n \geq 0$
Output : n-te Fibonacci Zahl

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

Fragen III

Wie lässt sich Dynamic Programming für andere Knapsack Probleme anwenden?

Wie sieht das für andere Probleme aus?