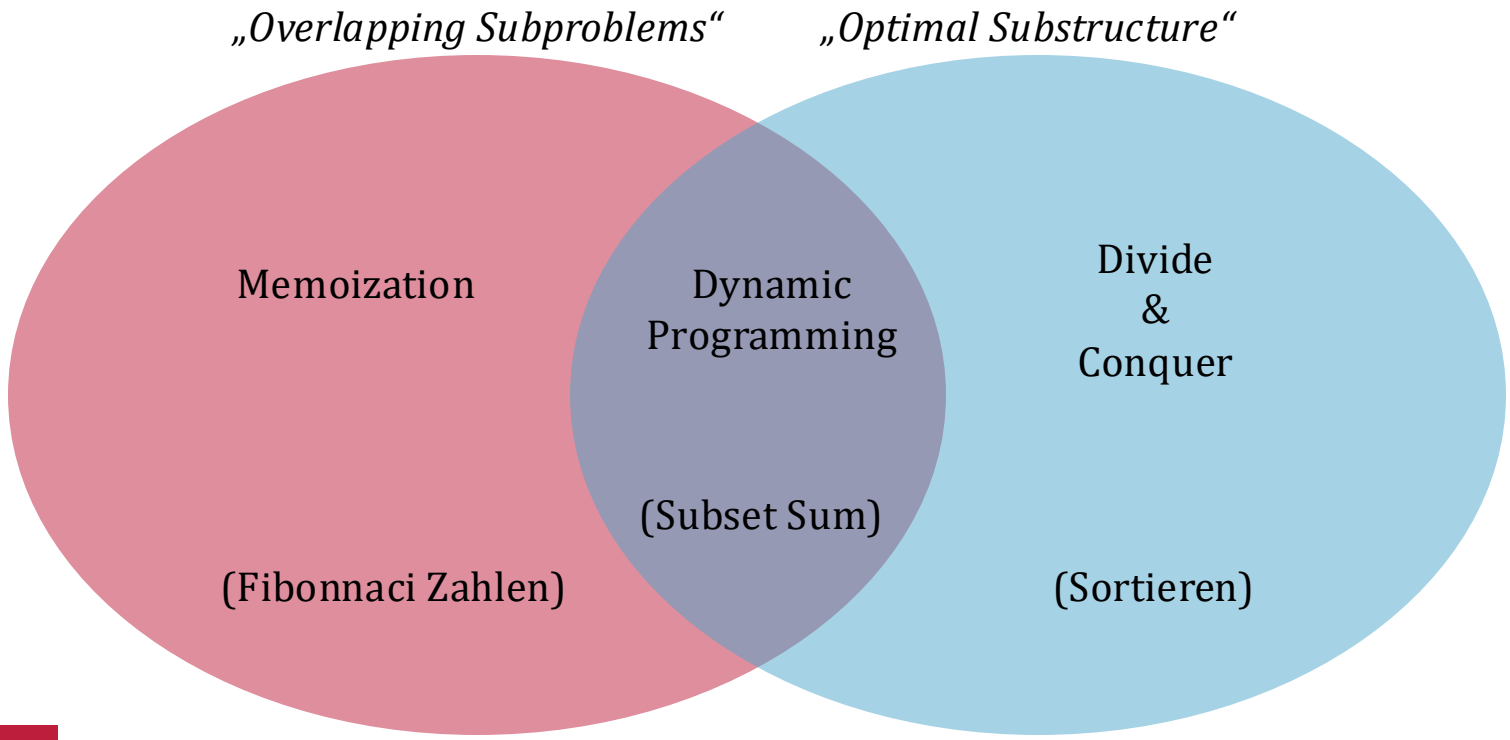
Technische Universität Braunschweig

Algorithmen und Datenstrukturen 2

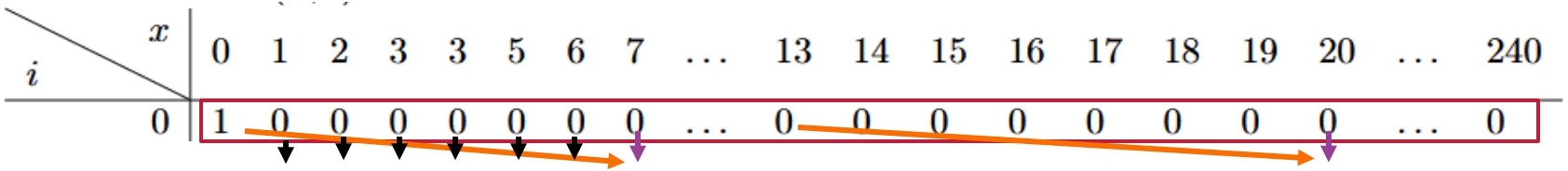
Arne Schmidt

Dynamic Programming



Wertetabelle

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



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

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

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

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

Objekt passt eh nicht.

Was passiert, wenn wir es nicht nehmen?

Was passiert, wenn wir es nehmen?

Fragen Wiederholung

Wie lässt sich Dynamic Programming für andere Knapsack Probleme anwenden?

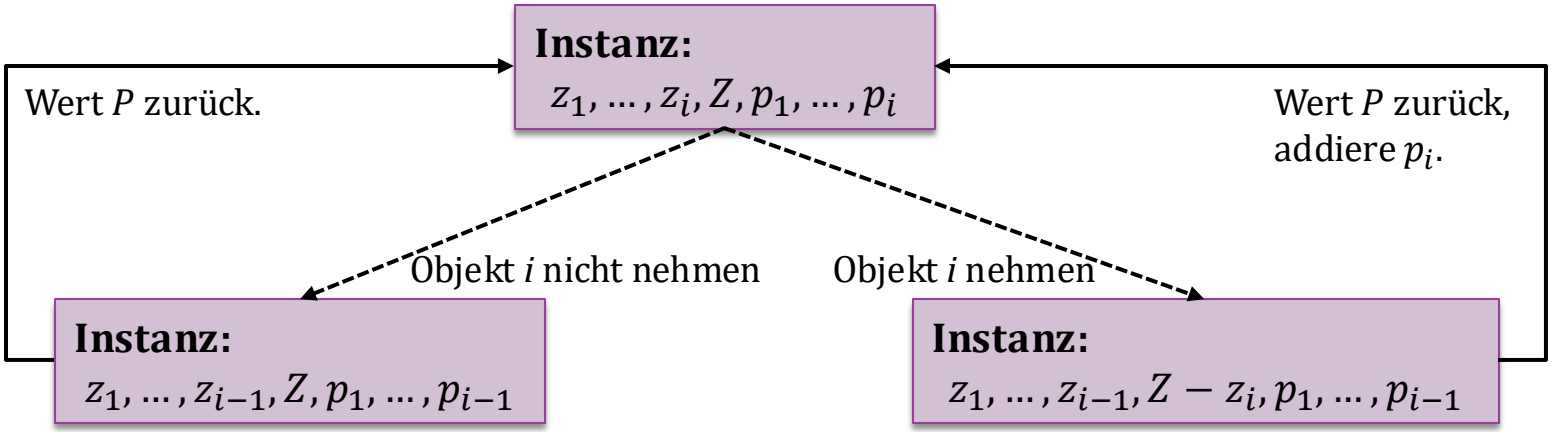
Wie sieht das für andere Probleme aus?

Kapitel 2.2 – Dynamic Programming
Maximum Knapsack

Ansätze für Knapsack

Wir haben nun nicht nur Gewichte, sondern:
z_i und p_i, sowie Z.

Frage:
Gibt es Subprobleme wie bei Subset Sum, deren Lösung uns hilft?



Rekursionsgleichung für Maximum Knapsack

Zu überlegen:
- Worüber geht die Rekursion?
- Was bedeutet die rekursive Funktion?
- Was sind (einfache) Basisfälle?
- Welche Bedingungen gibt es, ein Objekt zu nehmen?

Sei $P(x, i)$ der größtmögliche Wert, der mit den ersten i Objekten mit Kapazität x erreicht werden kann.

Basisfälle

$$P(x,i) = \begin{cases} 0, & \text{falls } i = 0 \\ P(x, i-1), & \text{falls } x < z_i \\ \max(P(x,i-1), P(x-z_i, i-1) + p_i), & \text{falls } x \geq z_i. \end{cases}$$

Objekt passt eh nicht.

Was passiert, wenn wir es nicht nehmen?

Was passiert, wenn wir es nehmen?

Beispiel 2.4

Sei $Z = 9$ und seien folgende sieben Objekte gegeben.

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$i \setminus x$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	12	14	15
6	0	0	6	6	6	11	11	12	14	15
7	0	0	6	6	6	11	11	12	14	15

Das halten wir noch einmal formal als Algorithmus fest.

Algorithmus für Maximum Knapsack

Satz 2.6:
Algorithmus 2.5 berechnet den besten Lösungswert für das Rucksackproblem in Zeit $O(nZ)$.

Korrektheit und Optimalität können wieder per Induktion gezeigt werden.

Laufzeit wie bei Subset Sum:
Wir füllen eine nZ große Tabelle aus.
Für jede Zelle konstanter Aufwand.

Algorithmus 2.5 Dynamic Programming für MAXIMUM KNAPSACK

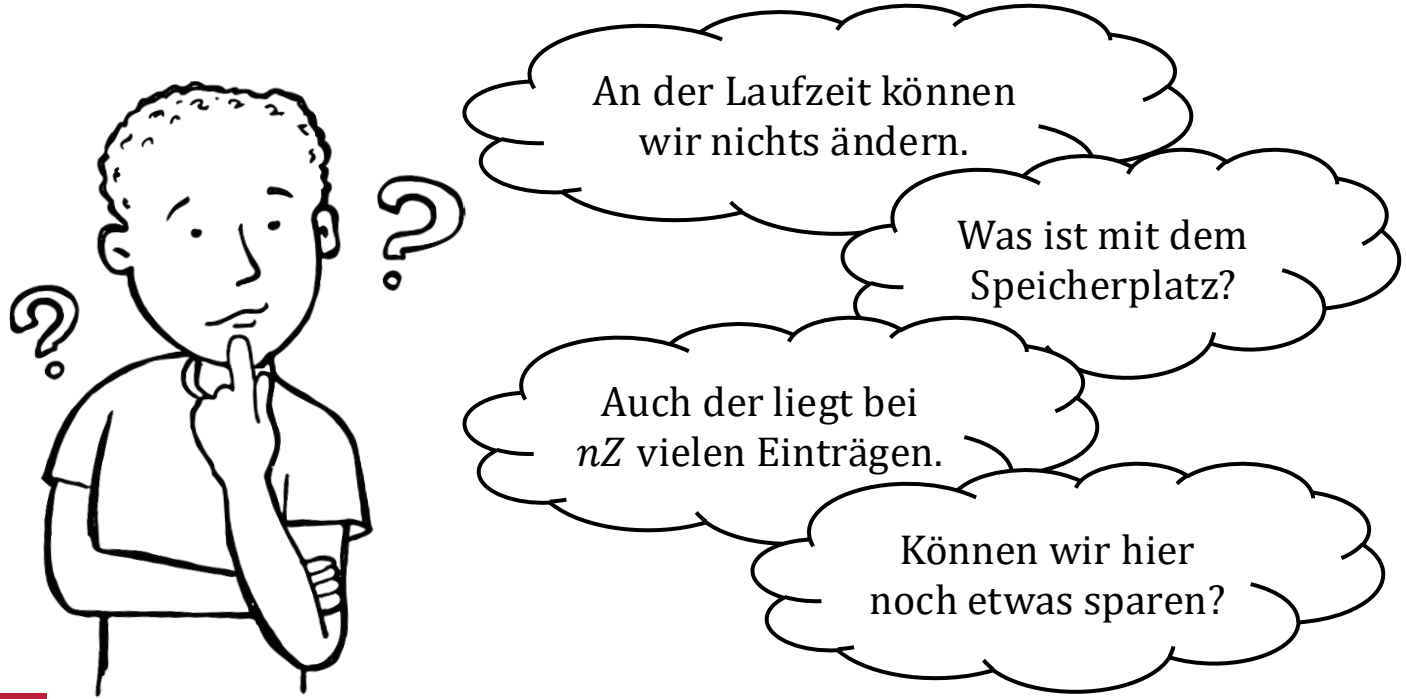
Eingabe: $z_1, ..., z_n, Z, p_1, ...p_n$

Ausgabe: Funktion $P : \{1, ..., Z\} \times \{1, ..., n\} \to \mathbb{R};\ (x, i) \mapsto P(x, i)$

mit $P(x,i) = max \sum_{j=1}^{i} p_j y_j$ mit $\sum_{j=1}^{i} z_j y_j \leq x$, für $y_j \in \{0, 1\}$

```
1:  for (x = 0) to Z do
2:      P(x, 0) := 0
3:  for (i = 1) to n do
4:      for (x = 0) to (z_i − 1) do
5:          P(x, i) := P(x, i − 1)
6:      for (x = z_i) to Z do
7:          if ((P(x − z_i, i − 1) + p_i) > P(x, i − 1)) then
8:              P(x, i) := P(x − z_i, i − 1) + p_i
9:          else
10:             P(x, i) := P(x, i − 1)
```

Weitere Überlegungen



Dynamic Programming für Knapsack in sparsam

Beobachtung:
- Wir greifen immer nur auf die letzten beiden Zeilen zu (zuletzt beschrieben und aktuell zu beschreiben).
- Ob wir die neue Zeile von links oder von rechts schreiben ist egal.

Wir müssen also maximal zwei Zeilen speichern.
Wenn wir außerdem von rechts die neue Zeile schreiben, können wir direkt die alte Zeile überschreiben (benötigte Einträge stehen noch weiter links).
→ Wir benötigen nur eine einzige Zeile!

Algorithmus für Maximum Knapsack

Satz 2.8:
Algorithmus 2.7 berechnet den besten Lösungswert für das Rucksackproblem in Zeit $O(nZ)$.

Algorithmus 2.7 Dynamic Programming für MAXIMUM KNAPSACK - sparsamer

Eingabe: $z_1, ..., z_n, Z, p_1, ...p_n$

Ausgabe: Funktion $P : \{1, ..., Z\} \to \mathbb{R};\ x \mapsto P(x)$

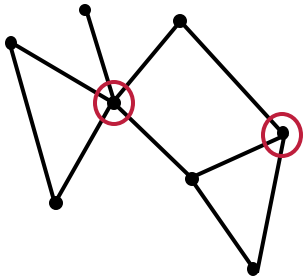
mit $P(x) = \max \sum_{j=1}^{n} p_j y_j$ mit $\sum_{j=1}^{n} z_j y_j \leq x$, für $y_j \in \{0, 1\}$

```
1: for (x = 0) to Z do
2:     P(x) := 0
3: for (i = 1) to n do
4:     for (x = Z) downto z_i do
5:         if ((P(x − z_i) + p_i) > P(x)) then
6:             P(x) := P(x − z_i) + p_i
7: return p* := P(Z)
```

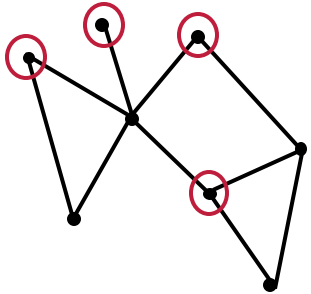
Kapitel 2.3 – Dynamic Programming
Gewichtete unabhängige Mengen

Unabhängige Mengen

Definition:
Sei $G = (V, E)$ ein Graph. Eine Knotenmenge $I \subseteq V$ heißt **unabhängige Menge**, wenn für je zwei Knoten $v_i, v_j \in I$ die Kante $\{v_i, v_j\} \notin E$.



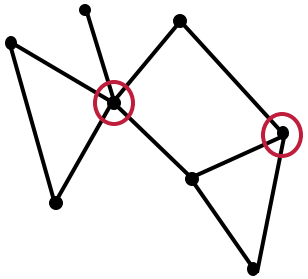
Unabhängige Menge der Größe 2



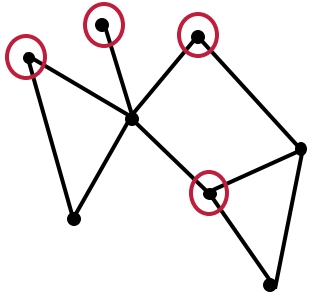
Unabhängige Menge der Größe 4

Gewichtete unabhängige Mengen

Problem:
Gegeben sei ein Graph $G = (V, E)$ und eine Knotenkostenfunktion $c: V \to \mathbb{N}$.
Gesucht ist eine unabhängige Menge I, sodass $\sum_{v \in I} c(v)$ maximal.



Unabhängige Menge der Größe 2



Unabhängige Menge der Größe 4

DP für gew. unabh. Mengen auf Bäumen

Aufgabe:
Entwirf einen DP-Algorithmus mit Laufzeit $O(n)$, welcher auf Bäumen eine gewichtete unabhängige Menge maximalen Gesamtgewichts bestimmt.

Wir nehmen dazu vereinfacht an:
Der Baum hat eine Wurzel → Wir können über Teilbäume gehen.

Wie lautet eine geeignete Rekursionsfunktion für Dynamic Programming?

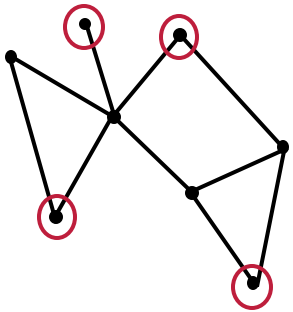
Vorschlag:
$\mathcal{I}(v)$: Der Wert einer gewichteten unabhängigen Menge maximalen Gesamtgewichts im Teilbaum mit Wurzel v.

Reicht uns das?
Schauen wir uns das an der Tafel an!

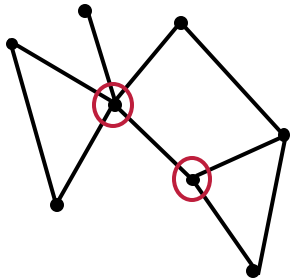
Kapitel 2.4 – Dynamic Programming
Gewichtete unabhängige, dominierende Mengen

Dominierende Mengen

Definition:
Sei $G = (V, E)$ ein Graph. Eine Knotenmenge $D \subseteq V$ heißt **dominierende Menge**, wenn für jeden Knoten $v \in V$ gilt: $v \in D$ oder es existiert $u \in N(v)$ mit $u \in D$.



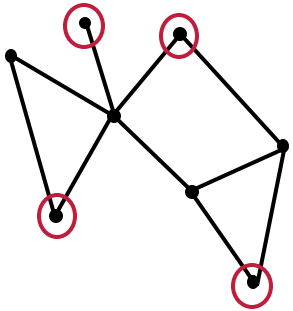
Dominierende Menge der Größe 4
(Gleichzeitig auch unabh. Menge!)



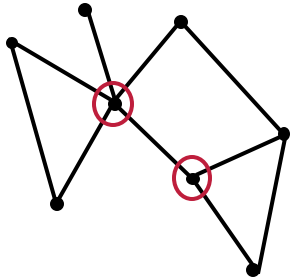
Dominierende Menge der Größe 2

Gewichtete unabhängige, dominierende Mengen

Problem:
Gegeben sei ein Graph $G = (V, E)$ und eine Knotenkostenfunktion $c: V \to \mathbb{N}$.
Gesucht ist eine dominierende, unabhängige Menge D, sodass $\sum_{v \in D} c(v)$ minimal.



Dominierende Menge der Größe 4
(Gleichzeitig auch unabh. Menge!)



Dominierende Menge der Größe 2

DP für gew. unabh., dom. Mengen auf Bäumen

Aufgabe:
Entwirf einen DP-Algorithmus mit Laufzeit $O(n)$, welcher auf Bäumen eine gewichtete unabhängige, dominierende Menge minimalen Gesamtgewichts bestimmt.

Wir nehmen dazu wieder vereinfacht an:
Der Baum hat eine Wurzel → Wir können über Teilbäume gehen.

Wie lautet eine geeignete Rekursionsfunktion für Dynamic Programming?

Vorschlag:
$\mathcal{D}(v)$: Der Wert einer gewichteten unabhängigen Menge maximalen Gesamtgewichts im Teilbaum mit Wurzel v.

Reicht das?
Schauen wir uns das an der Tafel an!

DP für gew. unabh., dom. Mengen auf Bäumen

Aufgabe:
Entwirf einen DP-Algorithmus mit Laufzeit $O(n)$, welcher auf Bäumen eine gewichtete unabhängige, dominierende Menge minimalen Gesamtgewichts bestimmt.

Wir nehmen dazu wieder vereinfacht an:
Der Baum hat eine Wurzel → Wir können über Teilbäume gehen.

Wie lautet eine geeignete Rekursionsfunktion für Dynamic Programming?

Vorschlag (ggf. etwas einfacher zu modellieren):
$\mathcal{D}(v, p)$: Der Wert einer gewichteten unabhängigen Menge maximalen Gesamtgewichts im Teilbaum mit Wurzel v, wenn der Vaterknoten in D liegt oder nicht ($p \in \{0,1\}$).

Schauen wir uns das an der Tafel an!

Nächstes Kapitel