

21. Dynamic Programming II

Subset Sum Problem, Rucksackproblem, Greedy Algorithmus vs dynamische Programmierung [Ottman/Widmayer, Kap. 7.2, 7.3, 5.7, Cormen et al, Kap. 15,35.5]

Aufgabe



Teile obige “Gegenstände” so auf zwei Mengen auf, dass beide Mengen den gleichen Wert haben.

Aufgabe



Teile obige "Gegenstände" so auf zwei Mengen auf, dass beide Mengen den gleichen Wert haben.

Eine Lösung:



Subset Sum Problem

Seien $n \in \mathbb{N}$ Zahlen $a_1, \dots, a_n \in \mathbb{N}$ gegeben.

Ziel: Entscheide, ob eine Auswahl $I \subseteq \{1, \dots, n\}$ existiert mit

$$\sum_{i \in I} a_i = \sum_{i \in \{1, \dots, n\} \setminus I} a_i.$$

Naiver Algorithmus

Prüfe für jeden Bitvektor $b = (b_1, \dots, b_n) \in \{0, 1\}^n$, ob

$$\sum_{i=1}^n b_i a_i \stackrel{?}{=} \sum_{i=1}^n (1 - b_i) a_i$$

Naiver Algorithmus

Prüfe für jeden Bitvektor $b = (b_1, \dots, b_n) \in \{0, 1\}^n$, ob

$$\sum_{i=1}^n b_i a_i \stackrel{?}{=} \sum_{i=1}^n (1 - b_i) a_i$$

Schlechtester Fall: n Schritte für jeden der 2^n Bitvektoren b . Anzahl Schritte:
 $\mathcal{O}(n \cdot 2^n)$.

Algorithmus mit Aufteilung

- Zerlege Eingabe in zwei gleich grosse Teile: $a_1, \dots, a_{n/2}$ und $a_{n/2+1}, \dots, a_n$.

Algorithmus mit Aufteilung

- Zerlege Eingabe in zwei gleich grosse Teile: $a_1, \dots, a_{n/2}$ und $a_{n/2+1}, \dots, a_n$.
- Iteriere über alle Teilmengen der beiden Teile und berechne Teilsummen $S_1^k, \dots, S_{2^{n/2}}^k$ ($k = 1, 2$).

Algorithmus mit Aufteilung

- Zerlege Eingabe in zwei gleich grosse Teile: $a_1, \dots, a_{n/2}$ und $a_{n/2+1}, \dots, a_n$.
- Iteriere über alle Teilmengen der beiden Teile und berechne Teilsummen $S_1^k, \dots, S_{2^{n/2}}^k$ ($k = 1, 2$).
- Sortiere die Teilsummen: $S_1^k \leq S_2^k \leq \dots \leq S_{2^{n/2}}^k$.

Algorithmus mit Aufteilung

- Zerlege Eingabe in zwei gleich grosse Teile: $a_1, \dots, a_{n/2}$ und $a_{n/2+1}, \dots, a_n$.
- Iteriere über alle Teilmengen der beiden Teile und berechne Teilsummen $S_1^k, \dots, S_{2^{n/2}}^k$ ($k = 1, 2$).
- Sortiere die Teilsummen: $S_1^k \leq S_2^k \leq \dots \leq S_{2^{n/2}}^k$.
- Prüfe ob es Teilsummen gibt, so dass $S_i^1 + S_j^2 = \frac{1}{2} \sum_{i=1}^n a_i =: h$

Algorithmus mit Aufteilung

- Zerlege Eingabe in zwei gleich grosse Teile: $a_1, \dots, a_{n/2}$ und $a_{n/2+1}, \dots, a_n$.
- Iteriere über alle Teilmengen der beiden Teile und berechne Teilsummen $S_1^k, \dots, S_{2^{n/2}}^k$ ($k = 1, 2$).
- Sortiere die Teilsummen: $S_1^k \leq S_2^k \leq \dots \leq S_{2^{n/2}}^k$.
- Prüfe ob es Teilsummen gibt, so dass $S_i^1 + S_j^2 = \frac{1}{2} \sum_{i=1}^n a_i =: h$
 - Beginne mit $i = 1, j = 2^{n/2}$.

Algorithmus mit Aufteilung

- Zerlege Eingabe in zwei gleich grosse Teile: $a_1, \dots, a_{n/2}$ und $a_{n/2+1}, \dots, a_n$.
- Iteriere über alle Teilmengen der beiden Teile und berechne Teilsummen $S_1^k, \dots, S_{2^{n/2}}^k$ ($k = 1, 2$).
- Sortiere die Teilsummen: $S_1^k \leq S_2^k \leq \dots \leq S_{2^{n/2}}^k$.
- Prüfe ob es Teilsummen gibt, so dass $S_i^1 + S_j^2 = \frac{1}{2} \sum_{i=1}^n a_i =: h$
 - Beginne mit $i = 1, j = 2^{n/2}$.
 - Gilt $S_i^1 + S_j^2 = h$ dann fertig

Algorithmus mit Aufteilung

- Zerlege Eingabe in zwei gleich grosse Teile: $a_1, \dots, a_{n/2}$ und $a_{n/2+1}, \dots, a_n$.
- Iteriere über alle Teilmengen der beiden Teile und berechne Teilsummen $S_1^k, \dots, S_{2^{n/2}}^k$ ($k = 1, 2$).
- Sortiere die Teilsummen: $S_1^k \leq S_2^k \leq \dots \leq S_{2^{n/2}}^k$.
- Prüfe ob es Teilsummen gibt, so dass $S_i^1 + S_j^2 = \frac{1}{2} \sum_{i=1}^n a_i =: h$
 - Beginne mit $i = 1, j = 2^{n/2}$.
 - Gilt $S_i^1 + S_j^2 = h$ dann fertig
 - Gilt $S_i^1 + S_j^2 > h$ dann $j \leftarrow j - 1$

Algorithmus mit Aufteilung

- Zerlege Eingabe in zwei gleich grosse Teile: $a_1, \dots, a_{n/2}$ und $a_{n/2+1}, \dots, a_n$.
- Iteriere über alle Teilmengen der beiden Teile und berechne Teilsummen $S_1^k, \dots, S_{2^{n/2}}^k$ ($k = 1, 2$).
- Sortiere die Teilsummen: $S_1^k \leq S_2^k \leq \dots \leq S_{2^{n/2}}^k$.
- Prüfe ob es Teilsummen gibt, so dass $S_i^1 + S_j^2 = \frac{1}{2} \sum_{i=1}^n a_i =: h$
 - Beginne mit $i = 1, j = 2^{n/2}$.
 - Gilt $S_i^1 + S_j^2 = h$ dann fertig
 - Gilt $S_i^1 + S_j^2 > h$ dann $j \leftarrow j - 1$
 - Gilt $S_i^1 + S_j^2 < h$ dann $i \leftarrow i + 1$

Beispiel

Menge $\{1, 6, 2, 3, 4\}$ mit Wertesumme 16 hat 32 Teilmengen.

Beispiel

Menge $\{1, 6, 2, 3, 4\}$ mit Wertesumme 16 hat 32 Teilmengen.
Aufteilung in $\{1, 6\}$, $\{2, 3, 4\}$ ergibt folgende 12 Teilmengen mit Wertesummen:

Beispiel

Menge $\{1, 6, 2, 3, 4\}$ mit Wertesumme 16 hat 32 Teilmengen.

Aufteilung in $\{1, 6\}$, $\{2, 3, 4\}$ ergibt folgende 12 Teilmengen mit Wertesummen:

$\{1, 6\}$				$\{2, 3, 4\}$							
$\{\}$	$\{1\}$	$\{6\}$	$\{1, 6\}$	$\{\}$	$\{2\}$	$\{3\}$	$\{4\}$	$\{2, 3\}$	$\{2, 4\}$	$\{3, 4\}$	$\{2, 3, 4\}$
0	1	6	7	0	2	3	4	5	6	7	9

Beispiel

Menge $\{1, 6, 2, 3, 4\}$ mit Wertesumme 16 hat 32 Teilmengen.

Aufteilung in $\{1, 6\}$, $\{2, 3, 4\}$ ergibt folgende 12 Teilmengen mit Wertesummen:

$\{1, 6\}$				$\{2, 3, 4\}$							
$\{\}$	$\{1\}$	$\{6\}$	$\{1, 6\}$	$\{\}$	$\{2\}$	$\{3\}$	$\{4\}$	$\{2, 3\}$	$\{2, 4\}$	$\{3, 4\}$	$\{2, 3, 4\}$
0	1	6	7	0	2	3	4	5	6	7	9

Beispiel

Menge $\{1, 6, 2, 3, 4\}$ mit Wertesumme 16 hat 32 Teilmengen.
Aufteilung in $\{1, 6\}$, $\{2, 3, 4\}$ ergibt folgende 12 Teilmengen mit Wertesummen:

$\{1, 6\}$				$\{2, 3, 4\}$							
$\{\}$	$\{1\}$	$\{6\}$	$\{1, 6\}$	$\{\}$	$\{2\}$	$\{3\}$	$\{4\}$	$\{2, 3\}$	$\{2, 4\}$	$\{3, 4\}$	$\{2, 3, 4\}$
0	1	6	7	0	2	3	4	5	6	7	9

Beispiel

Menge $\{1, 6, 2, 3, 4\}$ mit Wertesumme 16 hat 32 Teilmengen.
Aufteilung in $\{1, 6\}$, $\{2, 3, 4\}$ ergibt folgende 12 Teilmengen mit Wertesummen:

$\{1, 6\}$				$\{2, 3, 4\}$							
$\{\}$	$\{1\}$	$\{6\}$	$\{1, 6\}$	$\{\}$	$\{2\}$	$\{3\}$	$\{4\}$	$\{2, 3\}$	$\{2, 4\}$	$\{3, 4\}$	$\{2, 3, 4\}$
0	1	6	7	0	2	3	4	5	6	7	9

Beispiel

Menge $\{1, 6, 2, 3, 4\}$ mit Wertesumme 16 hat 32 Teilmengen.
Aufteilung in $\{1, 6\}$, $\{2, 3, 4\}$ ergibt folgende 12 Teilmengen mit Wertesummen:

$\{1, 6\}$				$\{2, 3, 4\}$							
$\{\}$	$\{1\}$	$\{6\}$	$\{1, 6\}$	$\{\}$	$\{2\}$	$\{3\}$	$\{4\}$	$\{2, 3\}$	$\{2, 4\}$	$\{3, 4\}$	$\{2, 3, 4\}$
0	1	6	7	0	2	3	4	5	6	7	9

⇔ Eine Lösung: $\{1, 3, 4\}$

- Teilsummegenerierung in jedem Teil: $\mathcal{O}(2^{n/2} \cdot n)$.
- Sortieren jeweils: $\mathcal{O}(2^{n/2} \log(2^{n/2})) = \mathcal{O}(n2^{n/2})$.
- Zusammenführen: $\mathcal{O}(2^{n/2})$

Gesamtlaufzeit

$$\mathcal{O}(n \cdot 2^{n/2}) = \mathcal{O}(n(\sqrt{2})^n).$$

Wesentliche Verbesserung gegenüber ganz naivem Verfahren –
aber immer noch exponentiell!

Dynamische Programmierung

Aufgabe: sei $z = \frac{1}{2} \sum_{i=1}^n a_i$. Suche Auswahl $I \subset \{1, \dots, n\}$, so dass $\sum_{i \in I} a_i = z$.

Dynamische Programmierung

Aufgabe: sei $z = \frac{1}{2} \sum_{i=1}^n a_i$. Suche Auswahl $I \subset \{1, \dots, n\}$, so dass $\sum_{i \in I} a_i = z$.

DP-Tabelle: $[0, \dots, n] \times [0, \dots, z]$ -Tabelle T mit Wahrheitseinträgen. $T[k, s]$ gibt an, ob es eine Auswahl $I_k \subset \{1, \dots, k\}$ gibt, so dass $\sum_{i \in I_k} a_i = s$.

Dynamische Programmierung

Aufgabe: sei $z = \frac{1}{2} \sum_{i=1}^n a_i$. Suche Auswahl $I \subset \{1, \dots, n\}$, so dass $\sum_{i \in I} a_i = z$.

DP-Tabelle: $[0, \dots, n] \times [0, \dots, z]$ -Tabelle T mit Wahrheitseinträgen. $T[k, s]$ gibt an, ob es eine Auswahl $I_k \subset \{1, \dots, k\}$ gibt, so dass $\sum_{i \in I_k} a_i = s$.

Initialisierung: $T[0, 0] = \text{true}$. $T[0, s] = \text{false}$ für $s > 0$.

Dynamische Programmierung

Aufgabe: sei $z = \frac{1}{2} \sum_{i=1}^n a_i$. Suche Auswahl $I \subset \{1, \dots, n\}$, so dass $\sum_{i \in I} a_i = z$.

DP-Tabelle: $[0, \dots, n] \times [0, \dots, z]$ -Tabelle T mit Wahrheitseinträgen. $T[k, s]$ gibt an, ob es eine Auswahl $I_k \subset \{1, \dots, k\}$ gibt, so dass $\sum_{i \in I_k} a_i = s$.

Initialisierung: $T[0, 0] = \text{true}$. $T[0, s] = \text{false}$ für $s > 0$.

Berechnung:

$$T[k, s] \leftarrow \begin{cases} T[k-1, s] & \text{falls } s < a_k \\ T[k-1, s] \vee T[k-1, s-a_k] & \text{falls } s \geq a_k \end{cases}$$

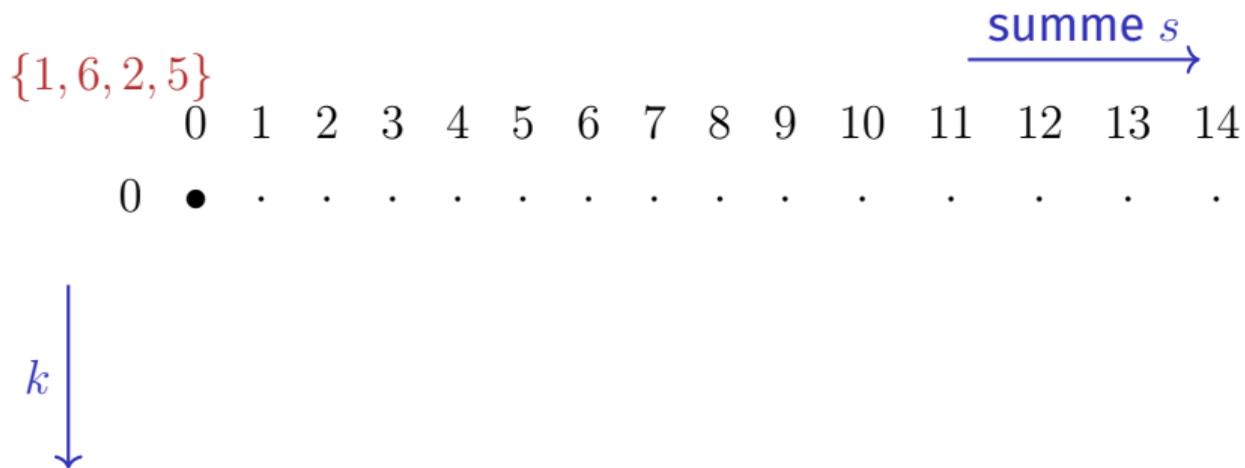
für aufsteigende k und innerhalb k dann s .

Beispiel

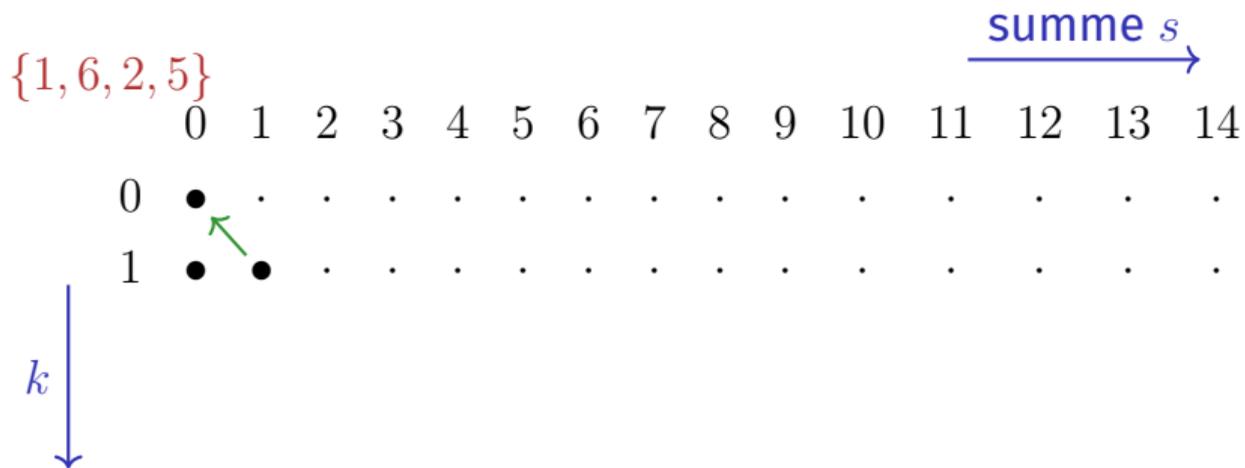
$\{1, 6, 2, 5\}$ summe s \rightarrow
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

k \downarrow

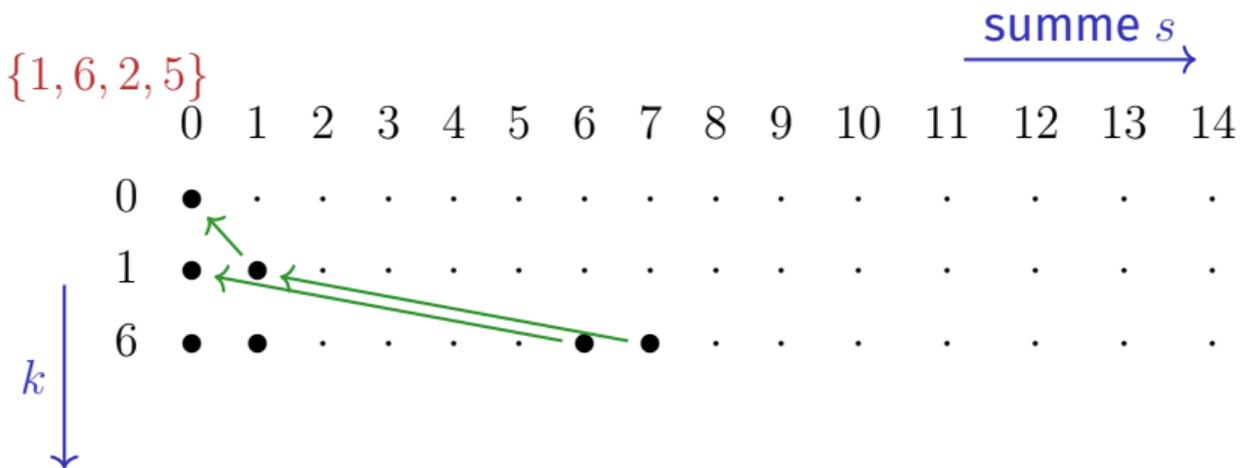
Beispiel



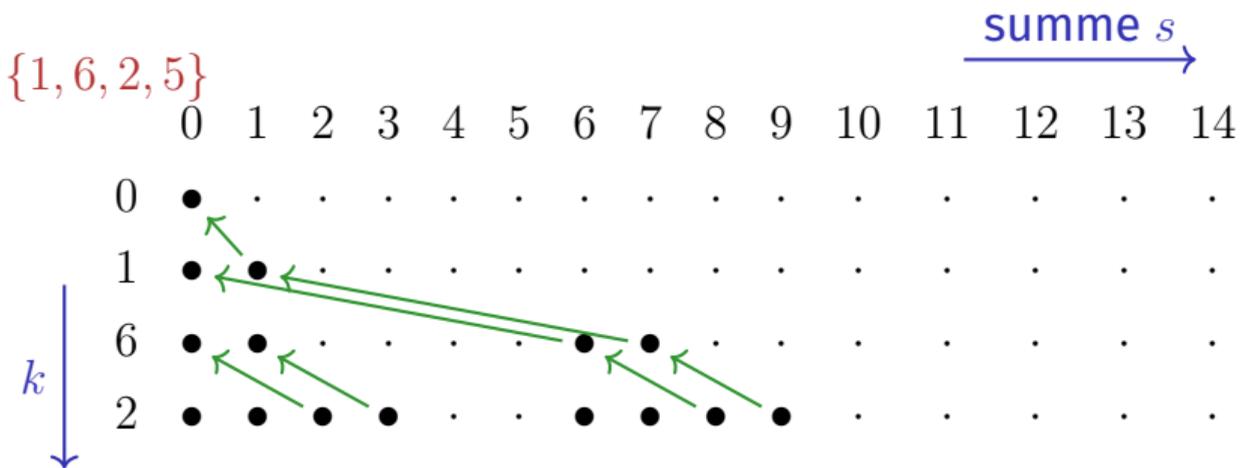
Beispiel



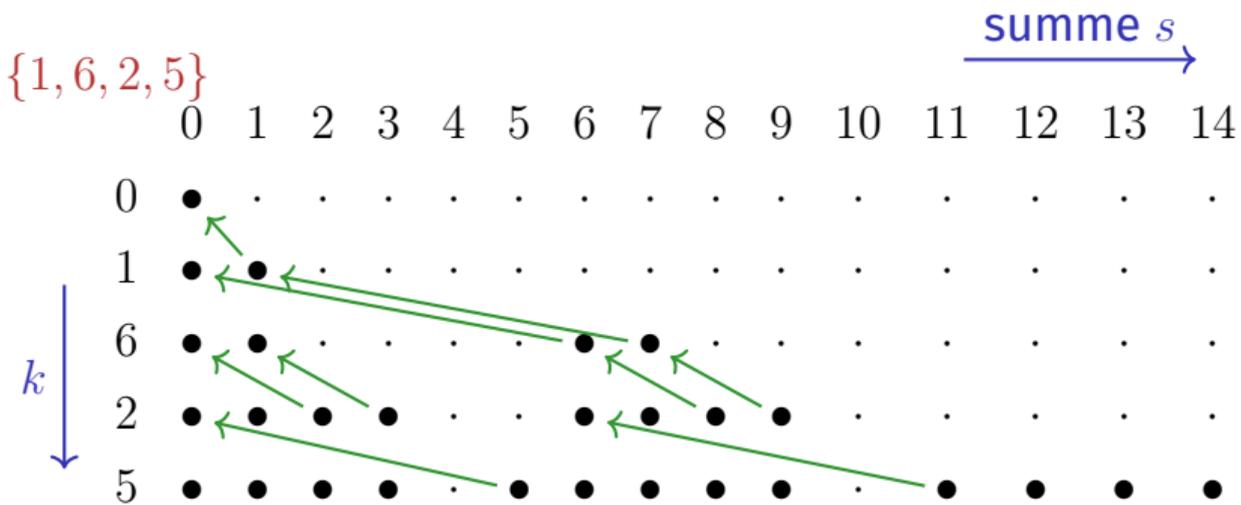
Beispiel



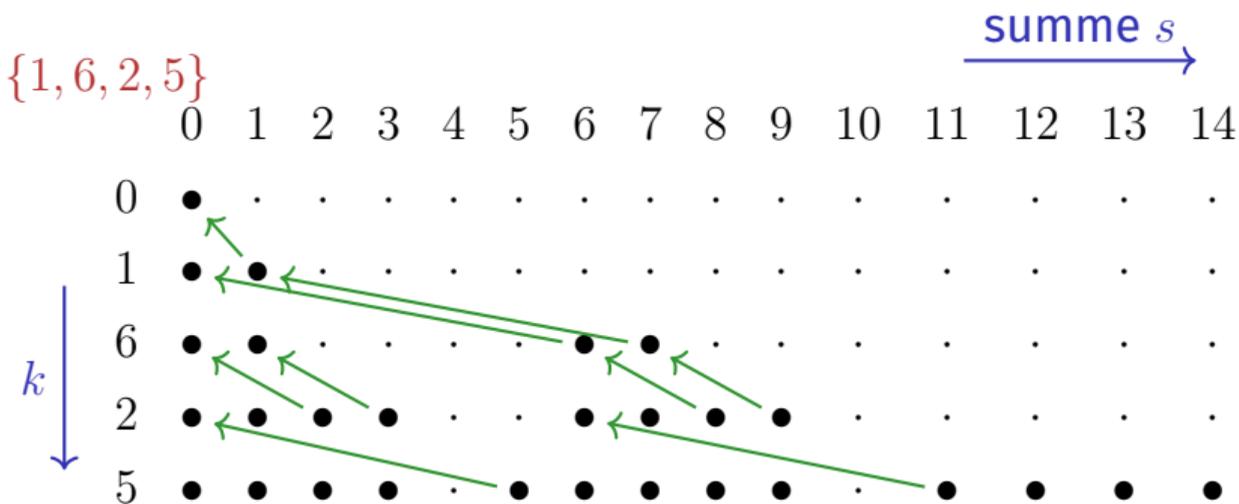
Beispiel



Beispiel



Beispiel



Auslesen der Lösung: wenn $T[k, s] = T[k - 1, s]$ dann a_k nicht benutzt und bei $T[k - 1, s]$ weiterfahren, andernfalls a_k benutzt und bei $T[k - 1, s - a_k]$ weiterfahren.

Der Algorithmus benötigt $\mathcal{O}(n \cdot z)$ Elementaroperationen.

Der Algorithmus benötigt $\mathcal{O}(n \cdot z)$ Elementaroperationen.

Was ist denn jetzt los? Hat der Algorithmus plötzlich polynomielle Laufzeit?

Der Algorithmus benötigt $\mathcal{O}(n \cdot z)$ Elementaroperationen.

Was ist denn jetzt los? Hat der Algorithmus plötzlich polynomielle Laufzeit?

Aufgelöst

Der Algorithmus hat nicht unbedingt eine polynomielle Laufzeit. z ist eine Zahl und keine Anzahl!

Aufgelöst

Der Algorithmus hat nicht unbedingt eine polynomielle Laufzeit. z ist eine Zahl und keine Anzahl!

Eingabelänge des Algorithmus \cong Anzahl Bits zur *vernünftigen* Repräsentation der Daten. Bei der Zahl z wäre das $\zeta = \log z$.

Aufgelöst

Der Algorithmus hat nicht unbedingt eine polynomielle Laufzeit. z ist eine Zahl und keine Anzahl!

Eingabelänge des Algorithmus \cong Anzahl Bits zur *vernünftigen* Repräsentation der Daten. Bei der Zahl z wäre das $\zeta = \log z$.

Also: Algorithmus benötigt $\mathcal{O}(n \cdot 2^\zeta)$ Elementaroperationen und hat exponentielle Laufzeit in ζ .

Aufgelöst

Der Algorithmus hat nicht unbedingt eine polynomielle Laufzeit. z ist eine Zahl und keine Anzahl!

Eingabelänge des Algorithmus \cong Anzahl Bits zur *vernünftigen* Repräsentation der Daten. Bei der Zahl z wäre das $\zeta = \log z$.

Also: Algorithmus benötigt $\mathcal{O}(n \cdot 2^\zeta)$ Elementaroperationen und hat exponentielle Laufzeit in ζ .

Sollte z allerdings polynomiell sein in n , dann hat der Algorithmus polynomielle Laufzeit in n . Das nennt man pseudopolynomiell.

Man weiss, dass der Subset-Sum Algorithmus zur Klasse der NP-vollständigen Probleme gehört (und somit *NP-schwer* ist).

³⁵Die bedeutenste ungelöste Frage der theoretischen Informatik!

Man weiss, dass der Subset-Sum Algorithmus zur Klasse der NP-vollständigen Probleme gehört (und somit *NP-schwer* ist).

P: Menge aller in Polynomialzeit lösbarer Probleme.

NP: Menge aller Nichtdeterministisch in Polynomialzeit lösbarer Probleme.

³⁵Die bedeutenste ungelöste Frage der theoretischen Informatik!

Man weiss, dass der Subset-Sum Algorithmus zur Klasse der NP-vollständigen Probleme gehört (und somit *NP-schwer* ist).

P: Menge aller in Polynomialzeit lösbarer Probleme.

NP: Menge aller Nichtdeterministisch in Polynomialzeit lösbarer Probleme.

Implikationen:

- NP enthält P.
- Probleme in Polynomialzeit verifizierbar.
- Unter der (noch?) unbewiesenen³⁵ Annahme, dass $NP \neq P$, gibt es für das Problem keinen Algorithmus mit polynomieller Laufzeit.

³⁵Die bedeutendste ungelöste Frage der theoretischen Informatik!

Das Rucksackproblem

Wir packen unseren Koffer und nehmen mit ...

- Zahnbürste
- Hantelset
- Kaffeemaschine
- Oh jeh – zu schwer.

Das Rucksackproblem

Wir packen unseren Koffer und nehmen mit ...

- Zahnbürste
 - Hantelset
 - Kaffemaschine
 - Oh jeh – zu schwer.
- Zahnbürste
 - Luftballon
 - Taschenmesser
 - Ausweis
 - Hantelset
 - Oh jeh – zu schwer.

Das Rucksackproblem

Wir packen unseren Koffer und nehmen mit ...

- Zahnbürste
 - Hantelset
 - Kaffemaschine
 - Oh jeh – zu schwer.
- Zahnbürste
 - Luftballon
 - Taschenmesser
 - Ausweis
 - Hantelset
 - Oh jeh – zu schwer.
- Zahnbürste
 - Kaffemaschine
 - Taschenmesser
 - Ausweis
 - Oh jeh – zu schwer.

Das Rucksackproblem

Wir packen unseren Koffer und nehmen mit ...

- | | | |
|-----------------------|-----------------------|-----------------------|
| ■ Zahnbürste | ■ Zahnbürste | ■ Zahnbürste |
| ■ Hantelset | ■ Luftballon | ■ Kaffemaschine |
| ■ Kaffemaschine | ■ Taschenmesser | ■ Taschenmesser |
| ■ Oh jeh – zu schwer. | ■ Ausweis | ■ Ausweis |
| | ■ Hantelset | ■ Oh jeh – zu schwer. |
| | ■ Oh jeh – zu schwer. | |

Wollen möglichst viel mitnehmen. Manche Dinge sind uns aber wichtiger als andere.

Rucksackproblem (engl. Knapsack problem)

Gegeben:

- Menge von $n \in \mathbb{N}$ Gegenständen $\{1, \dots, n\}$.
- Jeder Gegenstand i hat Nutzwert $v_i \in \mathbb{N}$ und Gewicht $w_i \in \mathbb{N}$.
- Maximalgewicht $W \in \mathbb{N}$.
- Bezeichnen die Eingabe mit $E = (v_i, w_i)_{i=1, \dots, n}$.

Rucksackproblem (engl. Knapsack problem)

Gegeben:

- Menge von $n \in \mathbb{N}$ Gegenständen $\{1, \dots, n\}$.
- Jeder Gegenstand i hat Nutzwert $v_i \in \mathbb{N}$ und Gewicht $w_i \in \mathbb{N}$.
- Maximalgewicht $W \in \mathbb{N}$.
- Bezeichnen die Eingabe mit $E = (v_i, w_i)_{i=1, \dots, n}$.

Gesucht:

eine Auswahl $I \subseteq \{1, \dots, n\}$ die $\sum_{i \in I} v_i$ maximiert unter $\sum_{i \in I} w_i \leq W$.

Gierige (engl. greedy) Heuristik

Sortiere die Gegenstände absteigend nach Nutzen pro Gewicht v_i/w_i :
Permutation p mit $v_{p_i}/w_{p_i} \geq v_{p_{i+1}}/w_{p_{i+1}}$

Gierige (engl. greedy) Heuristik

Sortiere die Gegenstände absteigend nach Nutzen pro Gewicht v_i/w_i :

Permutation p mit $v_{p_i}/w_{p_i} \geq v_{p_{i+1}}/w_{p_{i+1}}$

Füge Gegenstände in dieser Reihenfolge hinzu ($I \leftarrow I \cup \{p_i\}$), sofern das zulässige Gesamtgewicht dadurch nicht überschritten wird.

Gierige (engl. greedy) Heuristik

Sortiere die Gegenstände absteigend nach Nutzen pro Gewicht v_i/w_i :

Permutation p mit $v_{p_i}/w_{p_i} \geq v_{p_{i+1}}/w_{p_{i+1}}$

Füge Gegenstände in dieser Reihenfolge hinzu ($I \leftarrow I \cup \{p_i\}$), sofern das zulässige Gesamtgewicht dadurch nicht überschritten wird.

Das ist schnell: $\Theta(n \log n)$ für Sortieren und $\Theta(n)$ für die Auswahl. Aber ist es auch gut?

Gegenbeispiel zur greedy strategy

$$v_1 = 1 \quad w_1 = 1 \quad v_1/w_1 = 1$$

$$v_2 = W - 1 \quad w_2 = W \quad v_2/w_2 = \frac{W-1}{W}$$

Gegenbeispiel zur greedy strategy

$$v_1 = 1 \quad w_1 = 1 \quad v_1/w_1 = 1$$

$$v_2 = W - 1 \quad w_2 = W \quad v_2/w_2 = \frac{W-1}{W}$$

Greedy Algorithmus wählt $\{v_1\}$ mit Nutzwert 1.

Beste Auswahl: $\{v_2\}$ mit Nutzwert $W - 1$ und Gewicht W .

Greedy *kann* also beliebig schlecht sein.

Dynamic Programming

Unterteile das Maximalgewicht.

Dynamic Programming

Unterteile das Maximalgewicht.

Dreidimensionale Tabelle $m[i, w, v]$ (“machbar”) aus Wahrheitswerten.

Dynamic Programming

Unterteile das Maximalgewicht.

Dreidimensionale Tabelle $m[i, w, v]$ (“machbar”) aus Wahrheitswerten.

$m[i, w, v] = \text{true}$ genau dann wenn

- Auswahl der ersten i Teile existiert ($0 \leq i \leq n$)
- deren Gesamtgewicht höchstens w ($0 \leq w \leq W$) und
- Nutzen mindestens v ($0 \leq v \leq \sum_{i=1}^n v_i$) ist.

Berechnung der DP Tabelle

Initial

- $m[i, w, 0] \leftarrow \text{true}$ für alle $i \geq 0$ und alle $w \geq 0$.
- $m[0, w, v] \leftarrow \text{false}$ für alle $w \geq 0$ und alle $v > 0$.

Berechnung der DP Tabelle

Initial

- $m[i, w, 0] \leftarrow \text{true}$ für alle $i \geq 0$ und alle $w \geq 0$.
- $m[0, w, v] \leftarrow \text{false}$ für alle $w \geq 0$ und alle $v > 0$.

Berechnung

$$m[i, w, v] \leftarrow \begin{cases} m[i-1, w, v] \vee m[i-1, w-w_i, v-v_i] & \text{falls } w \geq w_i \text{ und } v \geq v_i \\ m[i-1, w, v] & \text{sonst.} \end{cases}$$

aufsteigend nach i und für festes i aufsteigend nach w und für festes i und w aufsteigend nach v .

Lösung: Grösstes v , so dass $m[i, w, v] = \text{true}$ für ein i und w .

Nach der Definition des Problems gilt offensichtlich, dass

■ für $m[i, w, v] = \text{true}$ gilt:

$$m[i', w, v] = \text{true} \quad \forall i' \geq i ,$$

$$m[i, w', v] = \text{true} \quad \forall w' \geq w ,$$

$$m[i, w, v'] = \text{true} \quad \forall v' \leq v .$$

■ für $m[i, w, v] = \text{false}$ gilt:

$$m[i', w, v] = \text{false} \quad \forall i' \leq i ,$$

$$m[i, w', v] = \text{false} \quad \forall w' \leq w ,$$

$$m[i, w, v'] = \text{false} \quad \forall v' \geq v .$$

Das ist ein starker Hinweis darauf, dass wir keine 3d-Tabelle benötigen.

DP Tabelle mit 2 Dimensionen

Tabelleneintrag $t[i, w]$ enthält statt Wahrheitswerten das jeweils grösste v , das erreichbar ist³⁶ mit

- den Gegenständen $1, \dots, i$ ($0 \leq i \leq n$)
- bei höchstem zulässigen Gewicht w ($0 \leq w \leq W$).

³⁶So etwas ähnliches hätten wir beim Subset Sum Problem auch machen können, um die dünnbesetzte Tabelle etwas zu verkleinern

Berechnung

Initial

- $t[0, w] \leftarrow 0$ für alle $w \geq 0$.

Berechnung

$$t[i, w] \leftarrow \begin{cases} t[i-1, w] & \text{falls } w < w_i \\ \max\{t[i-1, w], t[i-1, w-w_i] + v_i\} & \text{sonst.} \end{cases}$$

aufsteigend nach i und für festes i aufsteigend nach w .

Lösung steht in $t[n, w]$

Beispiel

$$E = \{(2, 3), (4, 5), (1, 1)\}$$

0 1 2 3 4 5 6 7 \xrightarrow{w}

i ↓

Beispiel

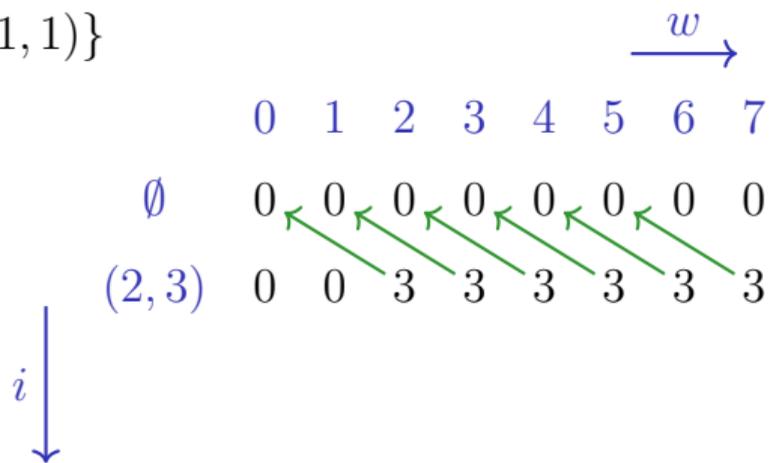
$$E = \{(2, 3), (4, 5), (1, 1)\}$$

		\xrightarrow{w}							
		0	1	2	3	4	5	6	7
\emptyset		0	0	0	0	0	0	0	0

i
 \downarrow

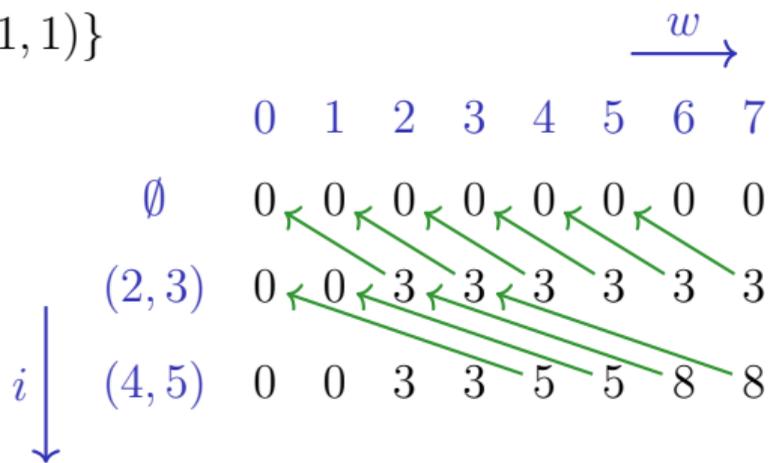
Beispiel

$$E = \{(2, 3), (4, 5), (1, 1)\}$$



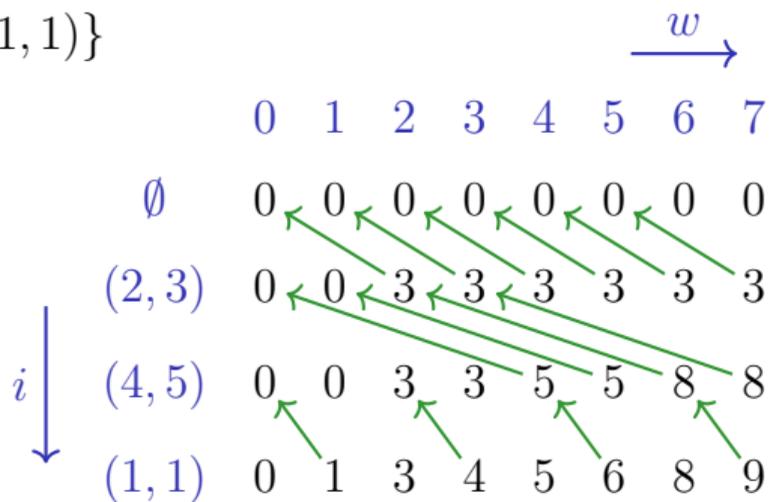
Beispiel

$$E = \{(2, 3), (4, 5), (1, 1)\}$$



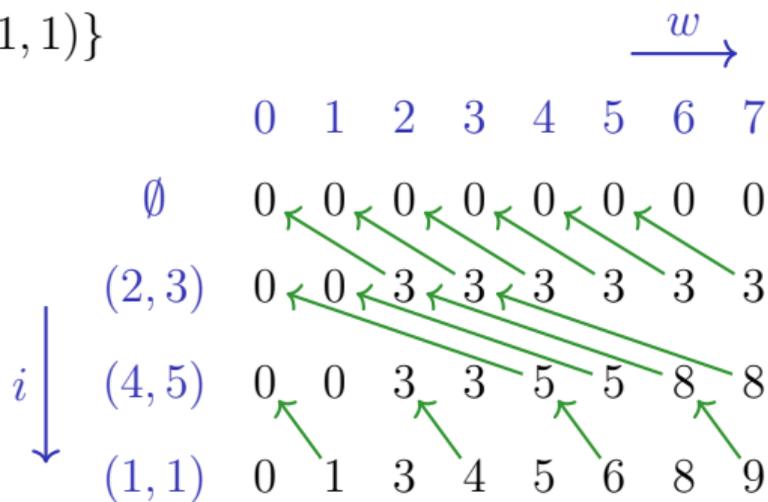
Beispiel

$$E = \{(2, 3), (4, 5), (1, 1)\}$$



Beispiel

$$E = \{(2, 3), (4, 5), (1, 1)\}$$



Auslesen der Lösung: wenn $t[i, w] = t[i - 1, w]$ dann Gegenstand i nicht benutzt und bei $t[i - 1, w]$ weiterfahren, andernfalls benutzt und bei $t[i - 1, w - w_i]$ weiterfahren.

Die beiden Algorithmen für das Rucksackproblem haben eine Laufzeit in $\Theta(n \cdot W \cdot \sum_{i=1}^n v_i)$ (3d-Tabelle) und $\Theta(n \cdot W)$ (2d-Tabelle) und sind beide damit pseudopolynomiell, liefern aber das bestmögliche Resultat.

Der greedy Algorithmus ist sehr schnell, liefert aber unter Umständen beliebig schlechte Resultate.

Im folgenden beschäftigen wir uns mit einer Lösung dazwischen.