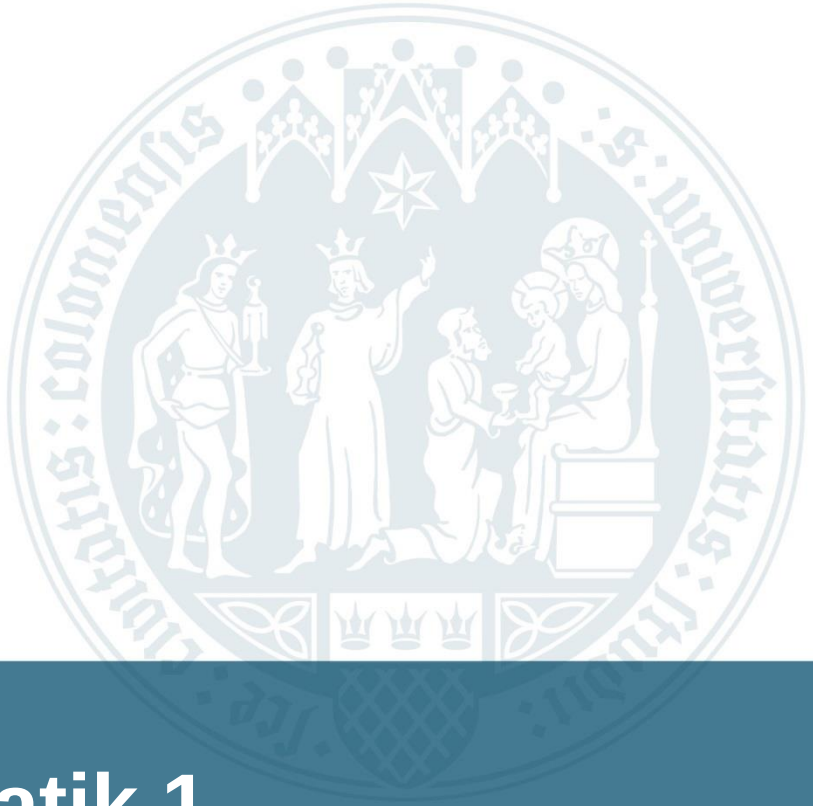
Universität zu Köln

Universität zu Köln



Grundzüge der Informatik 1

Vorlesung 11

Überblick

Überblick

- Wiederholung
  - Problemdefinition (Partition und SubsetSum)
  - Erstellen der Rekursionsgleichung (SubsetSum)
  - Entwicklung des Algorithmus
- Rucksack Problem
  - Problemdefinition
  - Erstellen der Rekursionsgleichung für den Lösungswert
  - Entwicklung des Algorithmus
  - Berechnung einer Lösung

Dynamische Programmierung - SubsetSum

Allgemeinere Fragestellung (SubsetSum)

- Gibt es L⊆M mit $\sum_{x \in L} x = U$?

Herangehensweise

- Sei M ={ $x_1,\ldots,x_n$ } (wir definieren eine Reihenfolge der Elemente, z.B. Reihenfolge im Eingabefeld)
- Definiere Indikatorfunktion Ind(U,m) mit
- $$Ind(U,m) = \begin{cases} true\,, & \text{wenn es } L \subseteq \{x_1,..,x_m\} \text{ gibt mit } \sum_{y \in L} y = U \\ false, & \text{sonst} \end{cases}$$
- Gesucht ist Rekursion für Ind(U,m)

Dynamische Programmierung - SubsetSum

Die Rekursion

- Wenn n>1, dann gilt:
- $Ind(U,n) = \begin{cases} true, & \text{wenn } U \geq x_n \text{ und } Ind(U - x_n, n-1) = true \\ & \text{oder } Ind(U, n-1) = true \\ false, & \text{sonst} \end{cases}$

Rekursionsabbruch:

- Wenn U>0 und n=1, dann gilt:
- $Ind(U,1) = \begin{cases} true, & \text{wenn } x_1 = U \\ false, & \text{sonst} \end{cases}$
- Wenn U=0 und n=1 dann gilt:
- $Ind(0,1) = true$

Dynamische Programmierung - SubsetSum

SubsetSum(A, U, n)

```
1.  Ind = new array [0..U] [1..n]
2.  for j=1 to n do
3.      Ind[j,1] = false
4.  Ind[0,1] = true                    \* leere Menge
5.  Ind[A[1],1] = true                 \* Menge {A[1]}
6.  for i=2 to n do
7.      for u=0 to U do
8.          Ind[u,i] = false
9.          if Ind[u,i-1] = true then Ind[u,i] = true
10.         if u≥A[i] und Ind[u-A[i], i-1] = true then Ind[u,i] = true
11. return Ind[U,n]
```

Dynamische Programmierung

Optimierungsprobleme

- Für eine Eingabe I sei S(I) die Menge der möglichen Lösungen
- Für $L \in S(I)$ bezeichne cost(L) eine Zielfunktion
- Aufgabe: Finde eine Lösung $L \in S(I)$ mit minimalen Kosten cost(L)

Varianten

- In der Beschreibung oben soll eine Kostenfunktion minimiert werden (*Minimierungsproblem*)
- Man kann auch eine Wertfunktion maximieren (*Maximierungsproblem*)

Dynamische Programmierung

Dynamische Programmierung für Optimierungsprobleme

1. Bestimme rekursive Struktur einer optimalen Lösung durch Zurückführen auf optimale Teillösungen

2. Entwerfe rekursive Methode zur Bestimmung des *Wertes* einer optimalen Lösung.

3. Transformiere rekursive Methode in eine iterative (bottom-up) Methode zur Bestimmung des Wertes einer optimalen Lösung.

4. Bestimmen aus dem Wert einer optimalen Lösung und in 3. ebenfalls berechneten Zusatzinformationen eine optimale Lösung.

Dynamische Programmierung

Das Rucksackproblem

- Rucksack mit begrenzter Kapazität
- Objekte mit unterschiedlichem Wert und unterschiedlicher Größe
- Wir wollen Objekte von möglichst großem Gesamtwert mitnehmen

Motivation

- Optimierung von Prozessen/Entscheidungen unter Kapazitätsbeschränkungen

Dynamische Programmierung

Das Rucksackproblem

- Rucksack mit begrenzter Kapazität
- Objekte mit unterschiedlichem Wert und unterschiedlicher Größe
- Wir wollen Objekte von möglichst großem Gesamtwert mitnehmen

Beispiel

- Rucksackgröße 6

Größe	5	2	1	3	7	4
Wert	11	5	2	8	14	9

Dynamische Programmierung

Das Rucksackproblem

- Rucksack mit begrenzter Kapazität
- Objekte mit unterschiedlichem Wert und unterschiedlicher Größe
- Wir wollen Objekte von möglichst großem Gesamtwert mitnehmen

Beispiel

- Rucksackgröße 6

Größe	5	2	1	3	7	4
Wert	11	5	2	8	14	9

- Objekt 1 und 3 passen in den Rucksack und haben einen Gesamtwert von 13

Dynamische Programmierung

Das Rucksackproblem

- Rucksack mit begrenzter Kapazität
- Objekte mit unterschiedlichem Wert und unterschiedlicher Größe
- Wir wollen Objekte von möglichst großem Gesamtwert mitnehmen

Beispiel

- Rucksackgröße 6

Größe	5	2	1	3	7	4
Wert	11	5	2	8	14	9

- Objekt 1 und 3 passen in den Rucksack und haben einen Gesamtwert von 13
- Objekte 2,3 und 4 passen und haben Gesamtwert von 15

Dynamische Programmierung

Das Rucksackproblem

- Eingabe:
  - Anzahl Objekte n
  - ObdA. sei {1,…,n} die Menge der Objekte
  - Feld der Objektgrößen g[1..n] mit g[i]$\in \mathbb{N}$
  - Feld der Objektwerte w[1..n] mit w[i]$\in \mathbb{N}$
  - Ganzzahlige Rucksackgröße G
- Ausgabe:
  - S$\subseteq${1,..,n}, so dass
  - $w(S) \coloneqq \sum_{x \in S} w(x)$ maximiert wird unter der Bedingung
  - $g(S) \coloneqq \sum_{x \in S} g(x) \leq G$

Dynamische Programmierung - Rucksackproblem

Lösungsansatz

- Bestimme zunächst den Wert einer optimalen Lösung
- Verfolge dazu Ansatz wie bei Maximumssuche und SubsetSum
- Verwende Eingabeordnung der Objekte
- Rekursion: Führe Kosten der Lösung mit n Objekten auf Lösungen mit n-1 Objekten zurück
- Leite dann die Lösung selbst aus der Tabelle des dynamischen Programms her

Dynamische Programmierung - Rucksackproblem

Zulässige und optimale Lösungen

- Wir nennen eine Lösung $S \subseteq \{1,..i\}$ *zulässig* für einen Rucksack der Größe j, wenn $g(S) \leq j$
- Eine zulässige Lösung S heißt *optimal* für Rucksackgröße j, wenn sie $w(S)$ unter allen zulässigen Lösungen maximiert

Wert optimaler Teillösungen

- Sei $O \subseteq \{1,..,i\}$ eine optimale Lösung für das Rucksackproblem mit Objekten aus {1,..,i} und Rucksackgröße j
- Sei Opt(i,j):=w(O) der *Wert* einer solchen optimalen Lösung
- Gesucht: Opt(n,G)

Dynamische Programmierung - Rucksackproblem

Aufgabe:

- Bestimmen Sie die Rekursionsgleichung für das Rucksackproblem

Dynamische Programmierung - Rucksackproblem

Opt(i,j)

Lemma 11.1 (Rekursionsabbruch: optimale Lösung für ein Objekt)

- Sei i=1 und sei {1,...,i}={1} die Eingabemenge und sei j die Rucksackgröße
- Ist j≥g[1], dann ist O={1} eine optimale Lösung mit Wert Opt(1,j) = w[1]
- Ist j<g[1], dann ist O=∅ eine optimale Lösung mit Wert Opt(1,j) = 0

Beweis

- Da es nur ein Objekt gibt, kann eine optimale Lösung nur {1} oder ∅ sein.
- Da die Objektwerte natürliche Zahlen sind, ist {1} eine optimale Lösung, wenn {1} eine zulässige Lösung ist
- Letzteres ist der Fall, wenn j≥g[1] ist
- Ansonsten ist nur ∅ eine zulässige und damit optimale Lösung

Dynamische Programmierung - Rucksackproblem

Lemma 11.2 (Zulässigkeit von Lösungen)

(a) Ist $S \subseteq \{1,..,i-1\}$ eine zulässige Lösung für Rucksackgröße j-g[i] mit Wert w(S), dann ist $S \cup \{i\}$ eine zulässige Lösung für Rucksackgröße j mit Wert $w(S \cup \{i\})$

(b) Ist $S \subseteq \{1,..,i-1\}$ eine zulässige Lösung für Rucksackgröße j, dann ist S auch eine zulässige Lösung für die ersten i Objekte und Rucksackgröße j

(c) $S = \emptyset$ ist eine zulässige Lösung für jede Rucksackgröße $j \geq 0$

Beweis (Teil 1)

(a) Wir verifizieren, dass $g(S)+g[i] \leq j$. Damit ist $S \cup \{i\}$ zulässig und hat Wert $w(S \cup \{i\})$

Dynamische Programmierung - Rucksackproblem

Lemma 11.2 (Zulässigkeit von Lösungen)

(a) Ist $S \subseteq \{1,..,i-1\}$ eine zulässige Lösung für Rucksackgröße j-g[i] mit Wert w(S), dann ist $S \cup \{i\}$ eine zulässige Lösung für Rucksackgröße j mit Wert $w(S \cup \{i\})$

(b) Ist $S \subseteq \{1,..,i-1\}$ eine zulässige Lösung für Rucksackgröße j, dann ist S auch eine zulässige Lösung für die ersten i Objekte und Rucksackgröße j

(c) $S = \emptyset$ ist eine zulässige Lösung für jede Rucksackgröße $j \geq 0$

Beweis (Teil 2)

- Offensichtlich erfüllt S auch weiterhin die Größenbedingung und ist damit auch als Teilmenge von {1,…,i} eine zulässige Lösung.

Dynamische Programmierung - Rucksackproblem

Lemma 11.2 (Zulässigkeit von Lösungen)

(a) Ist $S \subseteq \{1,..,i-1\}$ eine zulässige Lösung für Rucksackgröße j-g[i] mit Wert w(S), dann ist $S \cup \{i\}$ eine zulässige Lösung für Rucksackgröße j mit Wert $w(S \cup \{i\})$

(b) Ist $S \subseteq \{1,..,i-1\}$ eine zulässige Lösung für Rucksackgröße j, dann ist S auch eine zulässige Lösung für die ersten i Objekte und Rucksackgröße j

(c) $S = \emptyset$ ist eine zulässige Lösung für jede Rucksackgröße $j \geq 0$

Beweis (Teil 3)

- Offensichtlich erfüllt $S = \emptyset$ die Größenbeschränkung.

Dynamische Programmierung - Rucksackproblem

Lemma 11.3 (Rekursive Struktur einer optimalen Lösung)

- Sei $O \subseteq \{1,..,i\}$ eine optimale Lösung für das Rucksackproblem mit Objekten aus $\{1,\ldots,i\}$ und Rucksackgröße j. Es bezeichne Opt(i,j) den Wert dieser optimalen Lösung. Dann gilt:

(a) Ist Objekt i in O enthalten, so ist $O \setminus \{i\}$ eine optimale Lösung für das Rucksackproblem mit Objekten aus $\{1,\ldots,i-1\}$ und Rucksackgröße j-g[i]. Insbesondere gilt Opt(i,j)=w[i] + Opt(i-1, j-g[i]).

(b) Ist Objekt i nicht in O enthalten, so ist O eine optimale Lösung für das Rucksackproblem mit Objekten aus $\{1,\ldots,i-1\}$ und Rucksackgröße j. Insbesondere gilt Opt(i,j) = Opt(i-1, j).

Dynamische Programmierung - Rucksackproblem

Beweis (Teil 1)

(a) Z.z.: Ist Objekt i in O enthalten, so ist O \ {i} eine optimale Lösung für das Rucksackproblem mit Objekten aus {1,…,i-1} und Rucksackgröße j-g[i]. Insbesondere gilt Opt(i,j)=w[i] + Opt(i-1, j-g[i]).

- Sei O eine optimale Lösung mit Wert Opt(i,j), die Objekt i enthält. Da Objekt i Größe g[i] hat, gilt sicher, dass O \ {i} eine Gesamtgröße von höchstens j-g[i] hat. Damit ist O \ {i} eine zulässige Lösung für das Rucksackproblem mit Objekten aus {1,…,i-1} und Rucksackgröße j-g[i].
- Annahme: O \ {i} hat Wert R = Opt(i,j) – w[i] und ist keine optimale Lösung für das Rucksackproblem mit Objekten aus {1,..,i-1} und Rucksackgröße j-g[i].
- Dann gibt es eine bessere Lösung O* für dieses Problem mit Wert R*>R. Weiterhin ist O*$\cup$ {i} eine zul. Lösung für das Rucksackproblem mit Objekten aus {1,…,i} und Rucksackgröße j. Der Wert dieser Lösung ist R*+w[i] > R + w[i] = Opt(i,j). Widerspruch zur Optimalität von O.
- Damit ergibt sich sofort Opt(i,j)= w[i] + Opt(i-1,j-g[i]).

Dynamische Programmierung - Rucksackproblem

Beweis (Teil 2)

(b) Z.z.: Ist Objekt i nicht in O enthalten, so ist O eine optimale Lösung für das Rucksackproblem mit Objekten aus {1,…,i-1} und Rucksackgröße j. Insbesondere gilt Opt(i,j) = Opt(i-1, j).

- Sei O eine optimale Lösung mit Wert Opt(i,j), die Objekt i nicht enthält. Da Objekt i nicht in O ist, ist O \ {i} = O eine zulässige Lösung für das Rucksackproblem mit Objekten aus {1,…,i-1} und Rucksackgröße j.
- Da jede zulässige Lösung für das Rucksackproblem mit Objekten aus {1,…,i-1} auch eine Lösung für das Rucksackproblem mit Objekten aus {1,…,i} ist, folgt die Optimalität
- Damit ergibt sich sofort Opt(i,j)= Opt(i-1,j).

Dynamische Programmierung - Rucksackproblem

Korollar 11.4 (Rekursion zu den Kosten einer opt. Lösung)

- Es gilt

(a) Opt(1,j)= w[1] für j≥g[1]

(b) Opt(1,j)= 0 für j<g[1]

(c) Opt(i,j) = max{Opt(i-1,j), w[i] + Opt(i-1,j-g[i])}, falls i>1 und g[i]≤j, und

(d) Opt(i,j) = Opt(i-1,j), falls i>1 und g[i]>j.

Beweis (Teil 1)

- (a) und (b) folgen aus Lemma 11.1.

Dynamische Programmierung - Rucksackproblem

Korollar 11.4 (Rekursion zu den Kosten einer opt. Lösung)

- Es gilt

(a) Opt(1,j)= w[1] für j≥g[1]

(b) Opt(1,j)= 0 für j<g[1]

(c) Opt(i,j) = max{Opt(i-1,j), w[i] + Opt(i-1,j-g[i])}, falls i>1 und g[i]≤j, und

(d) Opt(i,j) = Opt(i-1,j), falls i>1 und g[i]>j.

Beweis (Teil 2)

- Sei nun i>1 und g[i]≤j: Aufgrund von Lemma 11.3 wissen wir, dass entweder Opt(i,j) = Opt(i-1,j) oder Opt(i,j) = w[i] + Opt(i-1,j-g[i]) ist
- Wegen Lemma 11.2 entsprechen Opt(i-1,j) und w[i] + Opt(i-1,j-g[i]) dem Wert von zulässigen Lösungen
- Damit gilt (c)

Dynamische Programmierung - Rucksackproblem

Korollar 11.4 (Rekursion zu den Kosten einer opt. Lösung)

- Es gilt

(a) Opt(1,j)= w[1] für j≥g[1]

(b) Opt(1,j)= 0 für j<g[1]

(c) Opt(i,j) = max{Opt(i-1,j), w[i] + Opt(i-1,j-g[i])}, falls i>1 und g[i]≤j, und

(d) Opt(i,j) = Opt(i-1,j), falls i>1 und g[i]>j.

Beweis (Teil 3)

- Sei nun i>1 und g[i]>j: Da g[i]>j, kann Objekt i nicht zu einer zulässigen Lösung gehören
- Aufgrund von Lemma 11.3 wissen wir, dass dann Opt(i,j) = Opt(i-1,j) gilt
- Damit gilt (d)

Dynamische Programmierung - Rucksackproblem

Rucksack(n,g,w,G)

```
1.  Opt = new array [1,..,n][0,..,G]
2.  for j = 0 to G do                          \* Rekursionsabbruch
3.      if j<g[1] then Opt[1,j]  = 0
4.      else Opt[1,j] = w[1]
5.  for i = 2 to n do
6.      for j = 0 to G do
7.          if g[i] ≤j then Opt[i,j] = max{Opt[i-1,j], w[i] + Opt[i-1,j-g[i]]}
8.      else Opt[i,j] = Opt[i-1,j]
9.  return Opt[n,G]
```

Dynamische Programmierung - Rucksackproblem

Rucksack(n,g,w,G)

1. Opt = **new array** [1,..,n][0,..,G]
2. **for** j = 0 **to** G **do** ***** Rekursionsabbruch
3. **if** j<g[1] **then** Opt[1,j] = 0
4. **else** Opt[1,j] = w[1]
5. **for** i = 2 **to** n **do**
6. **for** j = 0 **to** G **do**
7. **if** g[i] ≤j **then** Opt[i,j] = max{Opt[i-1,j], w[i] + Opt[i-1,j-g[i]]}
8. **else** Opt[i,j] = Opt[i-1,j]
9. **return** Opt[n,G]

Laufzeit

- O(nG)

Dynamische Programmierung - Rucksackproblem

i/j	0	1	2	3	4	5	6	7	8
8	0	2	3	5	7	9	10	12	13
7	0	2	3	5	7	9	10	12	13
6	0	2	3	5	6	7	9	10	10
5	0	2	3	5	6	7	9	10	10
4	0	1	3	4	5	7	8	8	8
3	0	1	1	4	5	5	5	5	6
2	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2

g	w
5	2
3	4
1	1
2	3
1	2
7	3
4	7
3	3

G=8

Dynamische Programmierung - Rucksackproblem

Beobachtung

- Sei R der Wert einer optimalen Lösung für Objekte aus {1,..,i}
- Falls g[i]≤j und Opt(i-1,j-g[i]) +w[i]= R, so ist Objekt i in mindestens einer optimalen Lösung enthalten

Dynamische Programmierung - Rucksackproblem

Wie kann man eine optimale Lösung berechnen?

- Idee: Verwende Tabelle der dynamischen Programmierung
- Fallunterscheidung + Rekursion:
  - Falls das i-te Objekt in einer optimalen Lösung für Objekte 1 bis i und Rucksackgröße j ist, so gib es aus und fahre rekursiv mit Objekt i-1 und Rucksackgröße j-g[i] fort
  - Ansonsten fahre mit Objekt i-1 und Rucksackgröße j fort

Dynamische Programmierung - Rucksackproblem

RucksackLösung(Opt,g,w,i,j)

1. **if** i=0 **return** ∅
2. **else if** g[i]>j **then return** RucksackLösung(Opt,g,w,i-1,j)
3. **else if** Opt[i,j]=w[i] + Opt[i-1,j-g[i]] **then**
   return {i} ∪ RucksackLösung(Opt,g,w,i-1,j-g[i])
4. **else return** RucksackLösung(Opt,g,w,i-1,j)

Aufruf

- Nach der Berechnung der Tabelle Opt von Rucksack wird RucksackLösung mit Opt, g,w, i=n und j=G aufgerufen.

Dynamische Programmierung - Rucksackproblem

Lemma 11.5

- Hat die optimale Lösung für Objekte aus {1,..,i} und Rucksackgröße j den Wert Opt(i,j), so berechnet Algorithmus RucksackLösung eine Teilmenge S von {1,..,i}, so dass g(S)≤j und w(S)=Opt(i,j) ist.

Beweis

- Aufgrund von Korollar 11.4 enthält Opt[i,j] jeweils den Wert Opt(i,j) einer optimalen Lösung für Objekte {1,..,i} und Rucksackgröße j. Wir zeigen das Lemma per Induktion.
- Beweis per Induktion über i.
- Induktionsanfang: Ist i=0, so gibt der Algorithmus die leere Menge zurück. Dies ist korrekt, da kein Objekt in den Rucksack gepackt werden kann.
- Induktionsannahme: Die Aussage stimmt für i-1.

Dynamische Programmierung - Rucksackproblem

Lemma 11.5

- Hat die optimale Lösung für Objekte aus {1,..,i} und Rucksackgröße j den Wert Opt(i,j), so berechnet Algorithmus RucksackLösung eine Teilmenge S von {1,..,i}, so dass g(S)≤j und w(S)=Opt(i,j) ist.

Beweis

- Induktionsschluss: Ist g[i] > j, so kann Objekt i nicht Teil einer zulässigen Lösung sein. Der Algorithmus gibt in diesem Fall RucksackLösung(Opt,g,w,i-1,j) zurück. Dies ist nach Induktionsannahme korrekt.
- Ist g[i] ≤ j und Opt[i,j]=w[i] + Opt[i-1,j-g[i]], so gibt es eine optimale Lösung, die Objekt i enthält. In diesem Fall gibt der Algorithmus {i} ∪ RucksackLösung(Opt,g,w,i-1,j-g[i]) zurück. Dies ist nach Induktionsannahme korrekt.

Dynamische Programmierung - Rucksackproblem

Lemma 11.5

- Hat die optimale Lösung für Objekte aus {1,..,i} und Rucksackgröße j den Wert Opt(i,j), so berechnet Algorithmus RucksackLösung eine Teilmenge S von {1,..,i}, so dass g(S)≤j und w(S)=Opt(i,j) ist.

Beweis

- Ist g[i] ≤ j und Opt[i,j]>w[i] + Opt[i-1,j-g[i]], so kann Objekt i nicht zu einer optimalen Lösung gehören. Der Algorithmus gibt in diesem Fall RucksackLösung(Opt,g,w,i-1,j) zurück. Dies ist nach Induktionsannahme korrekt.

Dynamische Programmierung - Rucksackproblem

8	0	2	3	5	7	9	10	12	13
7	0	2	3	5	7	9	10	12	13
6	0	2	3	5	6	7	9	10	10
5	0	2	3	5	6	7	9	10	10
4	0	1	3	4	5	7	8	8	8
3	0	1	1	4	5	5	5	5	6
2	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
i/j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**

g	w
5	2
3	4
1	1
2	3
1	2
7	3
4	7
3	3

G=8

Dynamische Programmierung - Rucksackproblem

Satz 11.6

Mit Hilfe von Algorithmus Rucksack und RucksackLösung kann man in O(nG) Zeit eine optimale Lösung für das Rucksackproblem berechnen, wobei n die Anzahl der Objekte ist und G die Größe des Rucksacks.

Beweis

- Die Laufzeit von Algorithmus RucksackLösung ist O(n), da sich bei jedem rekursiven Aufruf der erste Parameter um 1 reduziert, es nur jeweils einen rekursiven Aufruf gibt und jeder Aufruf konstante Zeit benötigt.
- Die Laufzeit wird durch Algorithmus Rucksack dominiert und ist somit O(nG). Die Korrektheit folgt aus dem Korollar zur Rekursionsgleichung und dem Lemma zur Korrektheit von RucksackLösung.

Dynamische Programmierung - Rucksackproblem

Zusammenfassung

- Rucksack Problem
  - Problemdefinition
  - Erstellen der Rekursionsgleichung für den Lösungswert
  - Entwicklung des Algorithmus
  - Finden einer Lösung

Referenzen

- T. Cormen, C. Leisserson, R. Rivest, C. Stein. Introduction to Algorithms. The MIT press. Second edition, 2001.