

1.3.2 DP für Rucksackprobleme

Algorithmus 1.11 liefert eine Lösung für Subset Sum  
- also für einen Spezialfall von Knapsack, bei dem der Unterschied zwischen Kosten  $z_i$  und Wert  $p_i$

keine Rolle spielt.

Das Prinzip lässt sich aber verallgemeinern!

Idee: Betrachte nicht nur  $P(x, i)$  (Wert  $x$  ist mit ersten  $i$  Objekten erreichbar)  
sondern  $P(x, i)$ : Höchster Wert, der sich mit den ersten  $i$  Objekten erreichen lässt, wenn Kosten  $x$  erlaubt sind

Nehmen wir an,  $P(x, i-1)$  ist für alle  $x \in \{0, \dots, Z\}$  bekannt. Kommt dann noch Objekt  $i$  mit Kosten  $z_i$ , Wert  $p_i$

dazu, gibt es folgende Möglichkeiten:

- (1) Objekt  $i$  ist zu teuer für eine Kostenschranke  $x$ :  $z_i > x$
- (2) Objekt  $i$  ist nicht zu teuer, bringt aber keine Verbesserung.
- (3) Objekt  $i$  ist nicht zu teuer und bringt eine Verbesserung.

Also:

$$P(x, i) = \begin{cases} P(x, i-1) & \text{falls } z_i > x \\ \max\{P(x, i-1), P(x-z_i, i-1) + p_i\} & \text{falls } z_i \leq x \end{cases}$$

Das ist eine Rekursionsgleichung - die sogenannte "Bellman-Rekursion", nach Richard Bellman (1957).

Noch einmal auf den Punkt gebracht:

ENTWEDER verwendet man  $i$  nicht  $\Rightarrow P(x, i) = P(x, i-1)$

ODER man verwendet  $i \Rightarrow P(x, i) = P(x-z_i, i-1) + p_i$

- d.h. man löst das kleinere Teilproblem so gut wie möglich, und zwar per Vorwärtsrekursion.

Beispiel 1.14 (Knapsackproblem)

$i$	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

Mit  $z = 9, n = 7$

Tabelle für  $P(x, i)$ :

$i \backslash x$	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

Erste Zeile: keine Objekte

Zweite Zeile: Objekt 1 mit  $z_1=2$ ,  $p_2=6$

Dritte Zeile: Objekte 1, 2; Objekt 1 dominiert Objekt 2,  
deshalb wird 2 erst ab  $x=5$  verwendet

Vierte Zeile: (etc.)

Beachte: • Dominierte Teillösungen werden ignoriert  
- und auch nicht mehr benötigt.

• Nur Verbesserungen werden berücksichtigt.

ALGORITHMUS 1.15 (DP für Knapsack)

Input:  $z_1, \dots, z_n, Z, p_1, \dots, p_n$

Output: Funktion

$$P: \{1, \dots, Z\} \times \{1, \dots, n\} \rightarrow \mathbb{R}$$

$$(x, i) \mapsto P(x, i)$$

mit  $F(x, i) =$  größtmöglicher Wert einer Teilmenge von  $\{1, \dots, i\}$   
 mit Gesamtkosten höchstens  $x$ ,

d.h.  $\max \sum_{j=1}^i p_j y_j$   
 mit  $\sum_{j=1}^i z_j y_j \leq x$   
 und  $y_j \in \{0, 1\}$

und Optimalwert  $P^*$

- ① FOR ( $x=0$ ) TO  $Z$  DO {  
 $P(x, 0) := 0$  // Initialisierung  
 }
- ② FOR ( $i=1$ ) TO  $n$  DO {  
 ②.1 FOR ( $x=0$ ) TO  $(z_i-1)$  DO { // Objekt  $i$  zu groß  
 $P(x, i) := P(x, i-1)$   
 }  
 ②.2 FOR ( $x=z_i$ ) TO  $Z$  DO { // Objekt  $i$  könnte verwendet werden  
 ②.2.1 IF ( $P(x-z_i, i-1) + p_i > P(x, i-1)$ ) THEN // Verwende  $i$   
 $P(x, i) := P(x-z_i, i-1) + p_i$   
 ②.2.2 ELSE // Ignoriere  $i$   
 $P(x, i) := P(x, i-1)$   
 }  
 }  
 }  
 ③ RETURN  $P^* := P(Z, n)$  // Bester Wert für alle;  
 Rest der Funktion ist berechnet

Satz 1.16

Algorithmus 1.15 berechnet einen besten Lösungswert für das Rucksackproblem, in  $O(nZ)$ .

Beweis: Selbst!

↖ Pseudopolynomiell:  
Input hat Größe  $n \cdot \log Z$

Ein Nachteil des Ansatzes:

Am Ende hat man eine ganze Tabelle berechnet, d.h. viel Speicherplatz verbraucht! U.U. interessiert einen aber nur der Optimalwert - und man benötigt jeweils nur die unmittelbar vorangehende Zeile.

Also: Speichere jeweils nur eine Zeile! (Überschreibe bei der Berechnung die vorangehende - bzw. behalte sie)

Hier kann man nicht nur Speicherplatz sparen sondern auch Codezeilen: (2.1) und (2.2) ersparen sich.

Damit erhält man: