Algorithmen & Datenstrukturen

Herbst 2019

Vorlesung 7

Dynamisches Programmieren (DP)

Allgemeine Vorgehensweise:

1. Designe Induktion / Rekurrenz
2. Definiere Tabelle (Dimensionen, Initialwerte)
3. Fülle Bottom-up
4. Lösung durch Zurückverfolgen

Teilsummenproblem (subset sum)

gegeben: Gesucht von verschiedenen Wert

$$7, 105, 11, 19, 5, \ldots$$

Teile gerecht zwischen zwei Geschwistern, wenn es geht

Allgemeiner:

gegeben: $A[1], \ldots, A[n]$ und $b \in \mathbb{N}$

gesucht: $I \subseteq \{1, \ldots, n\}$ so daß

b ist dann Teilsumme von A $\quad \sum_{i \in I} A[i] = b$ falls möglich

Beispiel: A: 5, 3, 7, 3, 1

b:	
9	ja
2	nein
17	nein
> 19	nein

Naive Algorithmus: probiere alle Teilmengen

$O(2^n \cdot n)$ Laufzeit

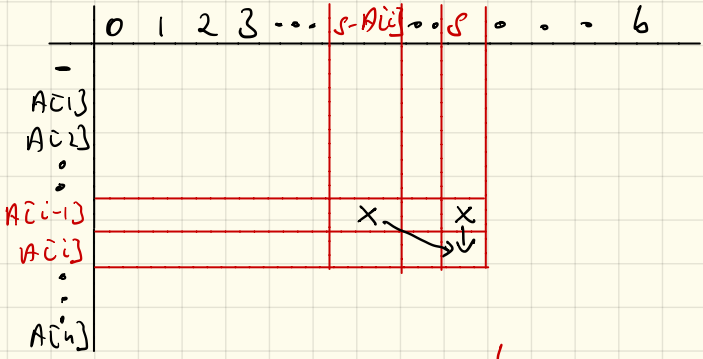
DP: Grundidee:

Lösung existiert
$\Rightarrow$ b ist Teilsumme von $A[1..n-1]$
oder $b - A[n]$ ist Teilsumme von $A[1..n-1]$

$TS(i,s)$: Wahrheitswert (also 1 oder 0) von
"s ist Teilsumme von $A[1..i]$"

Rekurrenz: $TS(i,s) = TS(i-1,s) \vee TS(i-1, s-A[i])$

Tabelle:



Beispiel:

	0	1	2	3	4	5	6	7	8	9 (b)
-	1	0	0	0	0	0	0	0	0	0
5	1	0	0	0	0	1	0	0	0	0
3	1	0	0	1	0	1	0	0	1	0
7	1	0	0	1	0	1	0	1	1	0
3	1	0	0	1	0	1	1	1	1	0
1	1	1	0	1	1	1	1	1	1	1 ✓

A

Vorgänger: jeder Sprung nach links → eine Zahl
5, 3, 1
(nicht eindeutig)

neu: Laufzeit hängt von Grösse einer Zahl ab!

Laufzeit: $O(bn)$ ← Verhalten für $n \to \infty$, $b \to \infty$

Speicher: $O(bn)$

z.B. alle n Zahlen in A sind 64 bit, aber b kann beliebig gross sein, braucht $\log_2 b$ bits

$\Rightarrow$ Eingabegrösse ist nicht mehr n, sondern $O(n \log_2 b)$

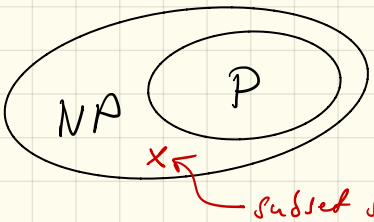
$b = 2^n \Rightarrow O(n^2)$ Eingabe $\Rightarrow O(2^n n)$ exponentiell in n^2

$b = n^c \Rightarrow O(n \log n)$ Eingabe
$O(n^{c+1})$ Laufzeit, polynomiell in $n \log n$

Man sagt: Laufzeit ist pseudopolynomiell

P: Menge aller Probleme mit polynomieller Laufzeit

NP: Menge aller Probleme bei denen man in polynomieller Zeit testen kann ob eine Lösung korrekt ist



Vermutung: $P \neq NP$

Beweis $\Rightarrow$ sehr berühmt

Bemerkung: in P, NP betrachtet man Entscheidungsprobleme (z.B. „ist b Teilsumme von A?")

Rucksackproblem (knapsack problem)

gegeben: • Rucksack mit Gewichtslimit W
• n Gegenstände mit Gewicht $w_i \in \mathbb{N}$, Wert $v_i \in \mathbb{N}$, $i = 1..n$

gesucht: $I \subseteq \{1..n\}$ so daß $\sum_{i \in I} w_i \leq W$
und $\sum_{i \in I} v_i$ maximal

Naiver Algorithmus: alle I ausprobieren, bestes nehmen. Laufzeit $O(2^n n)$

"Greedy" Algorithmus: sortiere Gegenstände nach Wertdichte v_i / w_i, wähle in dieser Reihenfolge.

Kann beliebig schlecht sein.
Beispiel: $(v_1, w_1) = (1, 1)$ $(v_2, w_2) = (W-1, W)$

DP

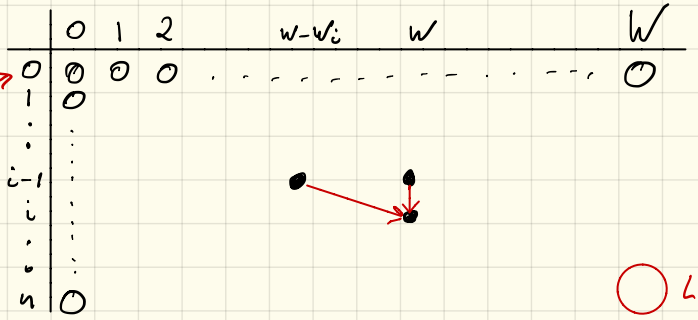
Rekurrenz: Einfach: opt. Lösung für n Gegenstände ist entweder opt. Lösung für die ersten $n-1$ mit W oder $W - w_n$

$MV(i, w)$ = Max. Wert von $I \subseteq \{1..i\}$ mit Schranke w.

↑ Max value

Rekurrenz: $MV(i,w) = \max(MV(i-1,w),\ MV(i-1,w-w_i)+v_i)$

Tabelle:

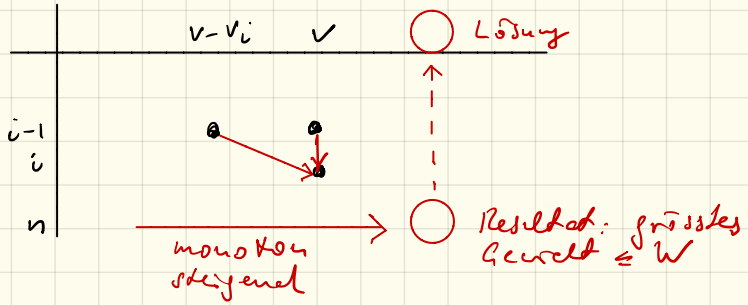


Laufzeit/Speicher: $O(nW)$ (pseudopolynomiell)

Geht auch mit $O(nV)$, $V = v_1 + \dots + v_n$:

$MinG(i,v)$ = minimales Gewicht um mit $I \subseteq \{1..i\}$ Wert $\geq v$ zu erhalten

$MinG(i,v) = \min(MinG(i-1,v),\ MinG(i-1,v-v_i)+w_i)$



Wie können wir das beschleunigen?

Neue Idee: Berechnung einer approximativen Lösung.

$K \in \mathbb{N}$

Input bisher: w_i, v_i, W — INPUT

Input approx: $w_i, \lfloor v_i/K \rfloor, W$ — $\overline{\text{INPUT}}$ ← approximiert

Beispiel: Werte $112, 78, 1001, 17, 237, \ldots$

$K=10$, $\overline{\text{Werte}}$ $11, 7, 100, 1, 23, \ldots$

$$\text{INPUT} \xrightarrow{O(nV)} \text{OPT} \quad \text{wie weit entfernt?}$$

$$\overline{\text{INPUT}} \longrightarrow \overline{\text{OPT}}$$

$$O(n\bar{V}) \leq O(n^2 v_{max}/K)$$

denn: $\bar{V} = \sum_{i=1}^{n} \lfloor \frac{v_i}{K} \rfloor \leq \sum_{i=1}^{n} \frac{v_i}{K} \leq \frac{1}{K} \sum_{i=1}^{n} v_i \leq \frac{n}{K} v_{max}$ (maximales v_i)

$$\text{Wert-OPT} = \sum_{i \in OPT} v_i$$

$$\text{Wert-}\overline{\text{OPT}} = \sum_{i \in \overline{OPT}} v_i$$

dann wählbar um beliebig gut zu approximieren

Ziel: $\text{Wert-}\overline{\text{OPT}} \geq (1-\varepsilon)\,\text{Wert-OPT}$

Es gilt: $v_i - K \leq K \cdot \lfloor \frac{v_i}{K} \rfloor \leq v_i$

$$\sum_{i \in OPT} (v_i - K) \leq \sum_{i \in OPT} K \lfloor \frac{v_i}{K} \rfloor$$

$$\leq K \sum_{i \in \overline{OPT}} \lfloor \frac{v_i}{K} \rfloor \quad \text{(Optimalität von } \overline{OPT}\text{)}$$

$$\leq \sum_{i \in \overline{OPT}} v_i$$

$$= \text{Wert-}\overline{\text{OPT}}$$

$$\sum_{i \in OPT} (v_i - K) = \text{Wert-OPT} - \sum_{i \in OPT} K$$

$$\geq \text{Wert-OPT} - nK$$

also: $\text{Wert-}\widetilde{OPT} \geq \text{Wert-OPT} - nK \overset{!}{\geq} (1-\varepsilon)\,\text{Wert-OPT}$ ← soll sein

heißt: $-nK \overset{!}{\geq} -\varepsilon\,\text{Wert-OPT}$

$\Leftrightarrow \quad K \leq \frac{\varepsilon}{n}\,\text{Wert-OPT}$

Annahme: alle $w_i \leq W$ (sonst entferne diese Gegenstände in $O(n)$)

$\Rightarrow \text{Wert-OPT} \geq v_{max}$

also wähle $K = \frac{\varepsilon}{n} v_{max}$

$\Rightarrow$ Laufzeit $O(n^3/\varepsilon)$ polynomiell in n und $1/\varepsilon$

„Fully polynomial approximation scheme"

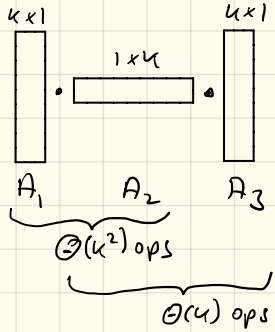
Matrix Kettenmultiplikation

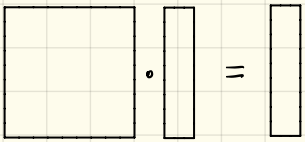
Problem: Berechne $A_1 \cdot A_2 \cdot \ldots \cdot A_n$ so günstig wie möglich. A_i sind Matrizen.

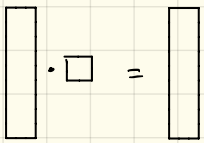
Freiheitsgrad: Assoziativität = Klammerung

z.B. $(A_1 A_2) A_3 = A_1 (A_2 A_3)$

Beispiel:



$(A_1 A_2) A_3$ =  $\Theta(k^2)$ ops insgesamt

$A_1 (A_2 A_3)$ =  $\Theta(k)$ ops insgesamt

Idee: Betrachte die letzte Multiplikation einer optimalen Lösung

$$A_1 A_2 \cdots A_n = \underbrace{(A_1 \cdots A_i)}\,\underbrace{(A_{i+1} \cdots A_n)}$$

Klammerung links/rechts ist optimal

$M(p,q)$ = min Ops zur Berechnung Produkt $A_p \cdots A_q$

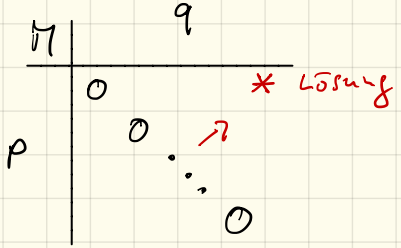
Rekurrenz:

Berechnung $O(q-p)$

$$M(p,q) = \min_{p \le i < q} \left(M(p,i) + M(i+1,q) + \text{Kosten zur Berechnung } (A_p \cdots A_i) \cdot (A_{i+1} \cdots A_q) \right)$$
$$p \le q$$

Basis: $M(p,p) = 0$, $p = 1 \ldots n$

In welcher Reihenfolge berechnen?
Von kurzen zu langen Produkten, also von der Diagonale weg:



In $M(p,q)$ gilt ja immer $p \le q$

Laufzeit: $O(n^3)$ Speicher: $O(n^2)$

Beispiel A_1, A_2, A_3 von zuvor, betrachte nur Mults

	1	2	3
1	0	4	24
2		0	4
3			0