DYNAMISCHE PROGRAMMIERUNG

1. geeignetes Teilproblem finden
2. Rekursion aufstellen
3. DP-Tabelle ausfüllen
4. Lösung auslesen

Teilsummenproblem (subset sum)

Gegeben: $A[1 \dots n]$ und b, alle in $\mathbb{N}$

Gesucht: $I \subseteq \{1, \dots, n\}$ sodass $\underbrace{\sum_{i \in I} A[i]}_{\text{Teilsumme von } A} = b$ (falls möglich)

Beispiel: A: | 5 | 3 | 10 | 3 | 1 |

$b = 12$ ja
2 nein
20 nein
>22 nein

Naiver Algorithmus: probiere alle 2^n Teilmengen → zu teuer

DP: Teilproblem: $T(i, s) :=$ „Ist s eine Teilsumme von $A[1..i]$?" $\in \{0,1\}$

Rekursion: zwei Möglichkeiten: i) benutze $A[i]$
ii) benutze $A[i]$ nicht

$$T(i,s) = T(i-1, s) \vee \underbrace{T(i-1, s-A[i])}_{0, \text{ falls } s < A[i]}$$

(∨: oder)

$$T(i,0) = 1$$
$$T(0,s) = 0 \text{ für } s \geq 1$$

Tabelle für $b = 7$

$T(i,s)$		s = 0	1	2	3	4	5	6	7 (b)
i = 0	–	1	0	0	0	0	0	0	0
1	5	1	0	0	0	0	1	0	0
2	3	1	0	0	1	0	1	0	0
3	10	1	0	0	1	0	1	0	0
4	3	1	0	0	1	0	1	1	0
5	1	1	1	0	1	1	1	1	1

Lösung (→ $T(5,7)$)

Ausfüllen: zeilenweise, von oben nach unten

Zurückverfolgen: Jeder Sprung nach links gibt einen Summanden

Laufzeit $\Theta(n\, b)$ ← neu: Laufzeit hängt von Grösse einer Zahl ab ⤳ pseudo-polynomiell

Speicher $\Theta(n\, b)$

n: Anzahl der Zahlen im Input

b: eine einzelne Zahl aus dem Input

Referenzgrösse ist in der Regel die Länge des Inputs

Hier: Inputlänge $\Theta(n + \log b)$ ← wir nehmen hier an, dass die Zahlen in A jeweils in eine RAM-Zelle passen

Für $b = 2^n$: Inputlänge $\Theta(n)$, Laufzeit $\Theta(n \cdot 2^n)$ exponentiell

Für $b = n^c$: Inputlänge $\Theta(n)$, Laufzeit $\Theta(n^{c+1})$ polynomiell

Geht es besser? Geht es polynomiell? ⤳ später

Rucksackproblem (Knapsack)

gegeben: - Rucksack mit Gewichtslimit W

- n Gegenstände mit Gewicht $w_i \in \mathbb{N}$ und Wert $v_i \in \mathbb{N}$, $i = 1 \dots n$

gesucht: Teilmenge $I \subseteq \{1, \dots, n\}$ sodass $\sum_{i \in I} w_i \leq W$

und $\sum_{i \in I} v_i$ maximal

Ansätze

- alle I ausprobieren → 2^n Möglichkeiten → zu teuer
- Greedy-Algorithmen ("gierig"): Nimm zuerst das Element: a) mit größtem Wert

oder: b) mit kleinstem Gewicht

oder: c) mit bestem Preis-Leistungs-verhältnis v_i / w_i

liefert alles kein korrektes Ergebnis, z.B.

$(v_1, w_1) = (v_2, w_2) = (2,2)$, $(v_3, w_3) = (3,3)$, $W = 4$,
→ $I = \{1,2\}$ ist optimal, aber a) nimmt $I = \{3\}$

$(v_1, w_1) = (1,1)$, $(v_2, w_2) = (9,10)$, $W = 10$
→ $I = \{2\}$ ist optimal, aber b) und c) nehmen $I = \{1\}$

DP: Teilproblem: $MW(i,w) :=$ maximaler Wert, den man aus $1 \ldots i$ mit Höchstgewicht w erreichen kann

Rekursion: $MW(i,w) = \max\{MW(i-1,w), v_i + MW(i-1, w-w_i)\}$

- $MW(i-1,w)$: verwende item i nicht
- $v_i + MW(i-1, w-w_i)$: verwende item i. Nur möglich, falls $w_i \le w$.

Tabelle:

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

Alternatives DP: Teilproblem: $MG(i,v) :=$ minimales Gewicht, das man braucht, um aus $1 \ldots i$ einen Wert $\ge v$ zu erzeugen

$MG(i,v) = \infty$, wenn v gar nicht erreicht werden kann

Rekursion: $MG(i,v) = \min\{MG(i-1,v), w_i + MG(i-1, v-v_i)\}$

- $MG(i-1,v)$: verwende i nicht
- $w_i + MG(i-1, v-v_i)$: verwende i; $MG(i-1, v-v_i) = 0$, falls $v_i \ge v$

Tabelle:

$V = v_1 + \ldots + v_n$

Laufzeit $O(nV)$

Geht es besser? Geht es polynomiell?

Vermutung: NEIN

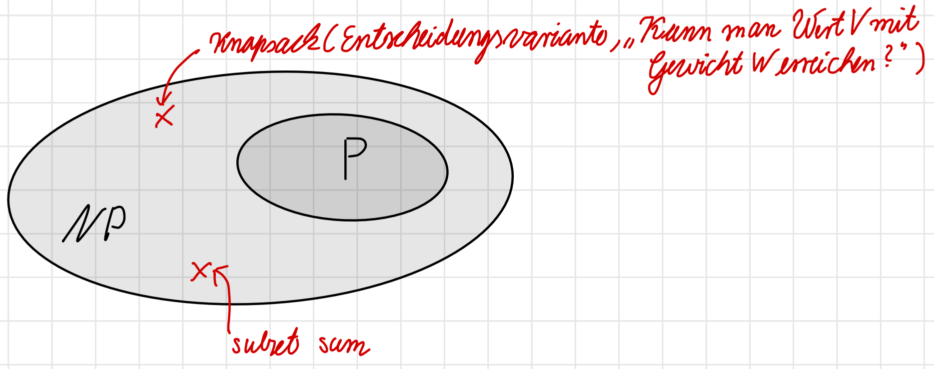
Subset sum ist ein Entscheidungsproblem: Antwort ja oder nein

P := Menge aller Entscheidungsprobleme, die in polynomieller Zeit lösbar sind

NP := Menge aller Entscheidungsprobleme, bei denen man im JA-Fall eine Lösung in polynomieller Zeit überprüfen kann

↑ nicht-deterministisch, „mit Glück" lösbar

Vermutung: $P \neq NP$ (größtes ungelöstes Problem der Informatik)



Beweis, dass ~~subset sum~~ Knapsack nicht in polynomieller Zeit gelöst werden kann, würde $P \neq NP$ zeigen.

<u>Tiefer mathematischer Satz:</u> polynomieller Algorithmus für subset sum $\Longrightarrow P = NP$

↑ „subset sum ist NP-vollständig"

polynomieller Algorithmus für Knapsack $\Longrightarrow P = NP$

„Knapsack ist NP-vollständig"

Approximationsalgorithmus für das Rucksackproblem

Input: w_i, v_i, W

Annahme: alle $w_i \le W$ ← sonst entferne alle Gegenstände mit $w_i > W$, passen eh nicht

$\leadsto V_{opt} \geq v_{max}$

(V_{opt}: Wert der optimalen Lösung; v_{max}: maximales v_i)

<u>Ziel</u>: Lösung, die fast optimal ist (in polynomieller Zeit)

<u>Idee</u>: Runden

Ersetze v_i durch $\tilde{v}_i := k \cdot \lfloor \frac{v_i}{k} \rfloor$, $k \in \mathbb{N}$ ← runde ab auf nächstes Vielfache von k

w_i und W bleiben unverändert $\leadsto$ die selben Teilmengen wie zuvor passen in den Rucksack

<u>Beispiel</u>: $k = 10$, Werte v_i: 112, 78, 1001, 17, 237, ...

gerundete Werte $\tilde{v}_i$: 110, 70, 1000, 10, 230, ...

Beobachtung: $v_i - k \overset{(1)}{\le} \tilde{v}_i \overset{(2)}{\le} v_i \le v_{max}$

$v_i \xrightarrow{DP} OPT \subseteq \{1, .., n\}$, Wert V_{opt}, Laufzeit: $O(n\,V)$ zu langsam

$\tilde{v}_i \xrightarrow{DP} \widetilde{OPT} \subseteq \{1, .., n\}$, Wert ??, Laufzeit: $O(\frac{n \tilde{V}}{k}) \le O(\frac{n^2 v_{max}}{k})$

da nur Vielfache von k auftreten, müssen wir nur jede k-te Spalte auffüllen

denn $\tilde{V} = \sum_{i=1}^{n} \tilde{v}_i \le \sum_{i=1}^{n} v_i \le n \cdot v_{max}$

Wie gut ist $\widetilde{OPT}$ für Ursprungsproblem? $\leadsto$ müssen $\sum_{i \in \widetilde{OPT}} v_i$ untersuchen.

Qualität des Outputs $\widetilde{OPT}$ für Ursprungsproblem

<u>Wissen</u>: $\widetilde{OPT}$ ist optimal für $\tilde{v}_i$.

$\leadsto$ Für jede Teilmenge I, die in den Rucksack passt, gilt:

$$\sum_{i \in \widetilde{OPT}} \tilde{v}_i \geq \sum_{i \in I} \tilde{v}_i$$

Das gilt auch für $I = OPT$: $\sum_{i \in \widetilde{OPT}} \tilde{v}_i \geq \sum_{i \in OPT} \tilde{v}_i \quad (*)$

Wert unserer Lösung (für Ursprungsproblem)

$$\leadsto \sum_{i \in \widetilde{OPT}} v_i \overset{(2)}{\geq} \sum_{i \in \widetilde{OPT}} \tilde{v}_i \overset{(*)}{\geq} \sum_{i \in OPT} \tilde{v}_i \overset{(1)}{\geq} \sum_{i \in OPT} (v_i - k) \geq \Big(\sum_{i \in OPT} v_i\Big) - nk = V_{opt} - nk \geq V_{opt} - nk \cdot \frac{V_{opt}}{v_{max}} = V_{opt} \cdot \Big(1 - \frac{nk}{v_{max}}\Big)$$

($\sum_{i \in OPT} v_i = V_{opt}$; $\frac{V_{opt}}{v_{max}} \geq 1$, da $V_{opt} \geq v_{max}$)

Idee: Wähle ein (kleines) $\varepsilon > 0$ aus und setze $k := \frac{\varepsilon \cdot V_{max}}{n}$.

eigentlich $k = \lceil \frac{\varepsilon V_{max}}{n} \rceil$, damit $k \in \mathbb{N}$

$\leadsto$ $\mathrm{WERT}(\widetilde{OPT}) = \sum_{i \in \widetilde{OPT}} v_i \geq V_{opt} \cdot (1 - \varepsilon)$ ← unsere Lösung ist fast so gut wie V_{opt}

Laufzeit: $O\Big(\frac{n^2 V_{max}}{k}\Big) = O\Big(\frac{n^2 V_{max}}{\varepsilon \cdot V_{max}/n}\Big) = O(n^3/\varepsilon)$ polynomiell in n und $1/\varepsilon$

"Fully polynomial-time approximation scheme" FPTAS

Längste aufsteigende Teilfolge

gegeben: Array $A[1...n]$ von Integers (Annahme: keine doppelt)

gesucht: Länge der / einer längsten aufsteigenden Teilfolge LAT

Teilfolge → Reihenfolge muss gleich bleiben, aber Lücken erlaubt

z.B. 2 13 17 9 11 4 78 28 15 25 99 ($A[1]$ = 2, $A[n]$ = 99; unterstrichen: 2, 9, 11, 15, 25, 99)

LAT = 2 9 11 15 25 99, Länge 6

Lösung mit DP:

Teilproblem, Versuch 1: $LAT(i) :=$ längste AT (aufsteigende Teilfolge) von $A[1...i]$

Beispiel: 1 2 5 3 4

$LAT(1) = 1$
$LAT(2) = 12$
$LAT(3) = 125$
$LAT(4) = 125$
$LAT(5) = 1234$?? Rekursion nicht (einfach) möglich

Versuch 2: merke alle Endungen einer LAT in $A[1...i]$

Beispiel: 1 5 6 2 3 4

$LAT(3) = 156$ (eindeutig)
$LAT(4) = 156$ (eindeutig)
$LAT(5) = 156, 123$?? Rekursion wieder nicht (einfach) möglich

Idee: merke auch kürzere AT in $A[1...i]$

Versuch 3: $E(i,l) :=$ „Es gibt AT der Länge l, die in i endet" $\in \{0,1\}$

Rekursion für $l \geq 2$: $E(i,l) = \begin{cases} 1, & \text{falls es } j<i \text{ gibt mit } E(j,l-1)=1 \text{ und } A[j]<A[i] \\ 0, & \text{sonst} \end{cases}$

könnten überflüssige Berechnungen sparen, da $E(l,i)=0$ für $l>i$, würde Laufzeit von $O(n^3)$ aber nicht verbessern

$E(i,1) = 1$ für alle $i \in \{1,...,n\}$

Laufzeit: $O(\sum_{l=1}^{n} \sum_{i=1}^{n} (i-1)) = O(\sum_{l=1}^{n} \frac{n(n-1)}{2}) = O(n^3)$

Beobachtung: Die Folge 123 ist immer besser als 156, denn alle Fortsetzungen von 156 passen auch an 123.
→ müssen uns für jede Länge nur die kleinstmögliche Endung merken

Versuch 4: $M(i,l) :=$ kleinstmögliche Endung einer AT der Länge l in $A[1..i]$ (∞, falls keine solche AT existiert)

A[i] passt an Folge der Länge $l-1$ — neue Folge verbessert bisheriges Minimum

Rekursion für $i \geq 2$: $M(i,l) = \begin{cases} A[i], & \text{falls } M(i-1,l-1) < A[i] < M(i-1,l) \\ M(i-1,l), & \text{sonst} \end{cases}$

$i=1$: $M(1,l) = \begin{cases} A[1], & l=1 \\ \infty, & \text{sonst} \end{cases}$

Laufzeit: $O(n^2)$

Beispiel: A = 3 7 8 4 5

i \ l	1	2	3	4	5
1	3	∞	∞	∞	∞
2	3	7	∞	∞	∞
3	3	7	8	∞	∞
4	3	4	8	∞	∞
5	3	4	5	∞	∞

Beobachtung 2:
- Zeilen sind sortiert
- Wir ändern pro Zeile nur einen Eintrag
- Diesen können wir mit binärer Suche in Zeit $O(\log n)$ finden

⇝ Wenn wir dasselbe Array für jede Zeile wiederverwenden, dauert das Update pro Zeile nur $O(\log n)$

⇝ insgesamt Laufzeit $O(n \log n)$

Beispiel: $A = $ | 2 | 13 | 17 | 9 | 11 | 4 | 78 | 28 | 15 | 25 | 99 |

ℓ	1	2	3	4	5	6
$M(*,\ell)$	2	13	17	78	25	99
		9	11	28		
		4		15		

↑ Zusatzinfo für Rückverfolgen der LAT:

$99 \to 25 \to 15 \to 11 \to 9 \to 2$

Speicher: $O(n)$

Laufzeit: $O(n \log n)$

Rückverfolgen benötigt Zusatzinformation: speichere für jedes Array-Element den Vorgänger beim Eintragen

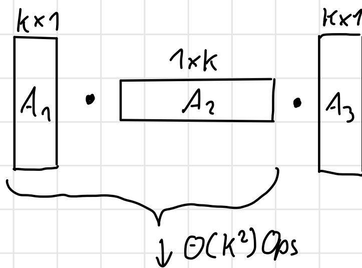
Matrixkettenmultiplikation (optional)

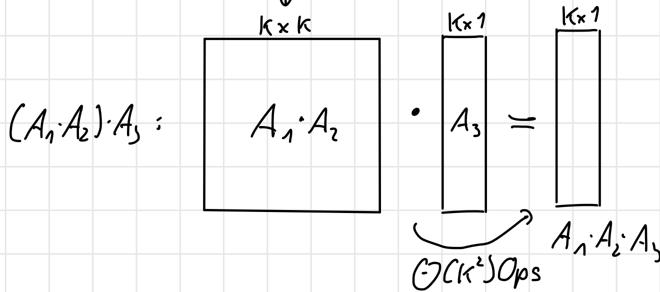
Problem: Berechne Matrixprodukt $A_1 \cdot A_2 \cdot \ldots \cdot A_n$ so günstig wie möglich

A_i ist eine $k_i \times l_i$-Matrix, $k_i, l_i \in \mathbb{N}$, $k_{i-1} = l_i$ ← damit Produkt sinnvoll ist

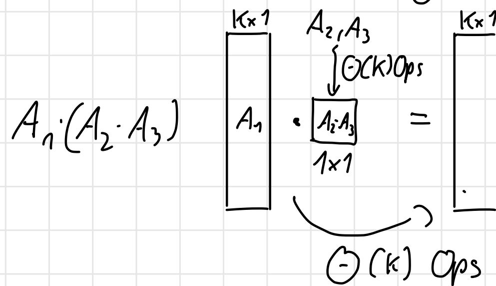
Freiheitsgrad: Assoziativität (Klammerung)

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





$\Theta(k^2)$ Ops insgesamt



$\Theta(k)$ Ops insgesamt

Teilproblem: $M(i,j)$:= minimale Zahl Ops zur Berechnung von $A_i \cdot \ldots \cdot A_j$, $i \leq j$

Rekursion: $(A_i \cdot A_{i+1} \cdot \ldots \cdot A_s) \cdot (A_{s+1} \cdot \ldots \cdot A_j)$

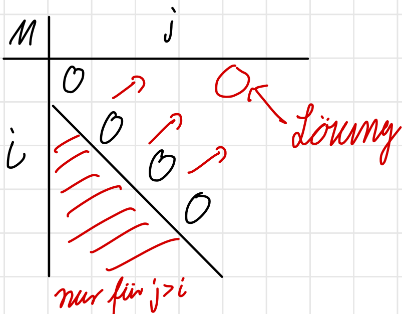
$k_i \times l_s$ — letzte Multiplikation — $k_{s+1} \times l_j = l_s \times l_j$

$$j>i: \; M(i,j) := \min_{i \leq s < j} \{ M(i,s) + M(s+1,j) + \text{Zahl Ops zur Multiplikation einer } k_i \times l_s \text{ mit einer } l_s \times l_j\text{-Matrix} \}$$

$$j=i: \; M(i,i) = 0$$

naiv: $\Theta(k_i \cdot l_s \cdot l_j)$ Operationen
Es geht aber besser mit ähnlichen Ideen wie bei Karatsubas Algorithmus.
Siehe Skript für quadratische Matrizen

Berechnungsreihenfolge: von kurzen (kleines $j-i$) zu langen (großes $j-i$) Produkten, also von Diagonale weg:



Laufzeit: $\Theta\left(\sum_{i=1}^{n} \sum_{j=i}^{n} (j-i)\right) = \Theta(n^3)$ (bekannt aus Übung)

Speicher: $\Theta(n^2)$