Karlsruher Institut für Technologie

Tutorium Algorithmen 1

12 · Dynamische Programmierung · 15.7.2024

Peter Bohner Tutorium 3

Dynamische Programmierung

- Rekurrenzgleichung beschreibt Algorithmus
- Unterschied zu Teile und Herrsche:
  - Zwischenergebnisse werden oft wiederverwendet
- Berechne alle Zwischenergebnisse & speichere sie
  - Reihenfolge hängt von Abhängigkeiten ab

$\Rightarrow$ Abwägung zwischen Laufzeit & Speicherbedarf

- Wichtigster Bestandteil: **Rekursionsformel** (Wie benutze ich Teilprobleme?)
- Vorgehen: Erstelle Tabelle
  - Fülle Tabelle Zelle für Zelle unter Benutzung bereits gefüllter Zellen

Beispiel: Floyd-Warshall-Algorithmus

$$sp(u, v, 0) = \begin{cases} c(u, v), u \neq v \wedge (u, v) \in E \\ 0, u = v \\ \infty, \text{sonst} \end{cases}$$

$$sp(u, v, k) = \min(sp(u, v, k-1), sp(u, k, k-1) + sp(k, v, k-1))$$

Peter Bohner - peter.bohner@student.kit.edu

Beispiel

Szenario: Wir wollen Rückgeld mit möglichst wenigen Münzen zahlen. Die Art der Münzen ist dabei nicht festgelegt.

Gegeben:

- Menge an Münzarten C mit Elementen $\in \mathbb{N}$
- Zu bezahlender Preis $X \in \mathbb{N}$

Gesucht

- Minimale Anzahl an Münzen aus C mit Summe X

Tabelle und Rekursion

- Betrachte als Teilprobleme die Preise von 0 bis X
- Also eindimensionale Tabelle P der Größe $X + 1$
- Minimale Anzahl für 0 ist 0, für alle Elemente aus C1

Peter Bohner - peter.bohner@student.kit.edu

Beispiel

Tabelle und Rekursion

- Betrachte als Teilprobleme die Preise von 0 bis X
- Also eindimensionale Tabelle P der Größe $X + 1$
- Minimale Anzahl für 0 ist 0 (Also $P(0) = 0$)
- Für Wert $i \leq X$ und Münze $c \in C$ betrachte P(i -c) und suche das minimum aus allen c
- Also $P(i) = \min_{c \in C}\{P(i - c) + 1\}$

Beispiel $C = \{1, 2, 4, 6\}, X = 23$

X	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
P(X)	0	1	1	2	1	2	1	2	2	3	2	3	2	3	3	4	3	4	3	4	4	5	4	5

Aktuell betrachtete Lösung

Teillösungen, die in Frage kommen

Lösungen Rekonstruieren

X	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
P(X)	0	1	1	2	1	2	1	2	2	3	2	3	2	3	3	4	3	4	3	4	4	5	4	5

- Wir wissen das wir 5 Münzen brauchen
- Aber nicht welche
- Ähnlich wie bei kürzeste Wege algos können wir beim berechnen Pointer speichern
- Wenn mehrere Teillösungen in Frage kommen ist die Lösung nicht eindeutig

Beispiel: Longest common suffix

Gegeben: $w_1, w_2 \in \Sigma^*$
Gesucht: Zerlegung $p_1, p_2, s \in \Sigma^*$ mit $w_1 = p_1 s$, $w_2 = p_2 s$ mit $|s|$ maximal

- Idee: DP über Länge des Suffix
- $S(k) =$„Längster Suffix mit Länge $\leq k$“
- $S(0) = 0$ (ε ist immer ein Suffix)
- $S(k) = \begin{cases} k, S(k-1) = k-1 \wedge w_1(|w_1| - k) = w_2(|w_2| - k) \\ S(k-1), \text{sonst} \end{cases}$
- Gesucht: $S(\min(|w_1|, |w_2|))$

Fall 1: $|s| \geq k - 1$ und $k$tes Zeichen gleich $\Rightarrow |s| \geq k$
Fall 2: Längerer Suffix als $|s| = S(k-1)$ nicht möglich

- keine gemeinsamen Basisfälle
- iterative & rekursive Lösung nahezu gleich

Longest common substring

Gegeben: $w_1, w_2 \in \Sigma^*$
Gesucht: Zerlegung $p_1, p_2, s_1, s_2, c \in \Sigma^*$ mit $w_1 = p_1 c s_1 \wedge w_2 = p_2 c s_2$ mit $|c|$ maximal
Beispiel: $w_1 =$ Algorithmen, $w_2 =$ Logarithmus $\Rightarrow c =$ rithm

- Idee: Jeder Substring hört an einem Index auf

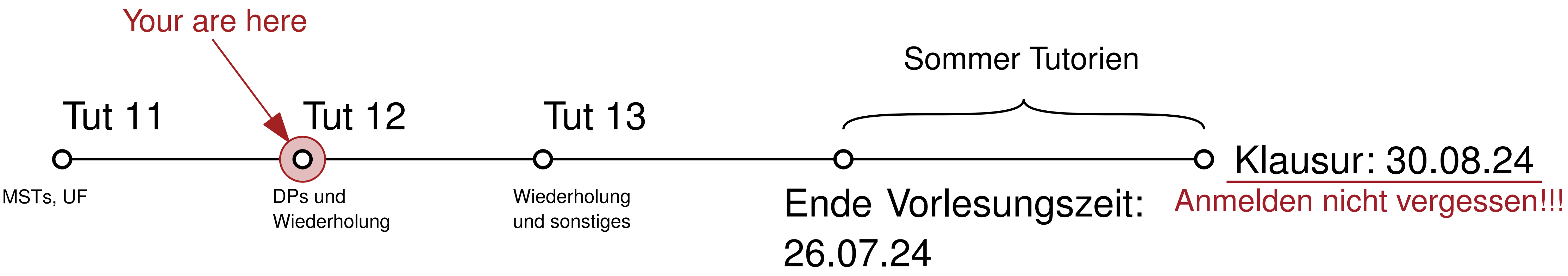
Definiere **LCS**(i, j) =„Längster Suffix von $w_1(0) \ldots w_1(i)$ und $w_2(0) \ldots w_2(j)$“

- **LCS**$(0, j) = 0$
- **LCS**$(i, 0) = 0$
- **LCS**$(i, j) = \begin{cases} LCS(i-1, j-1) + 1, w_1(i) = w_2(j) \\ 0, \text{sonst} \end{cases}$

Gesucht: $\max\limits_{0 \leq i < |w_1|, 0 \leq j < |w_2|}$ **LCS**(i, j)

- wenige gemeinsame Zwischenergebnisse

Roadmap



Falls ihr spezielle Themen nochmal wiederholen wollt, ruhig Bescheid geben (per Discord, E-Mail, auf einem Übungsblatt...)

Zusammenfassung

Was haben wir gemacht?

- Spannbäume
- Union Find

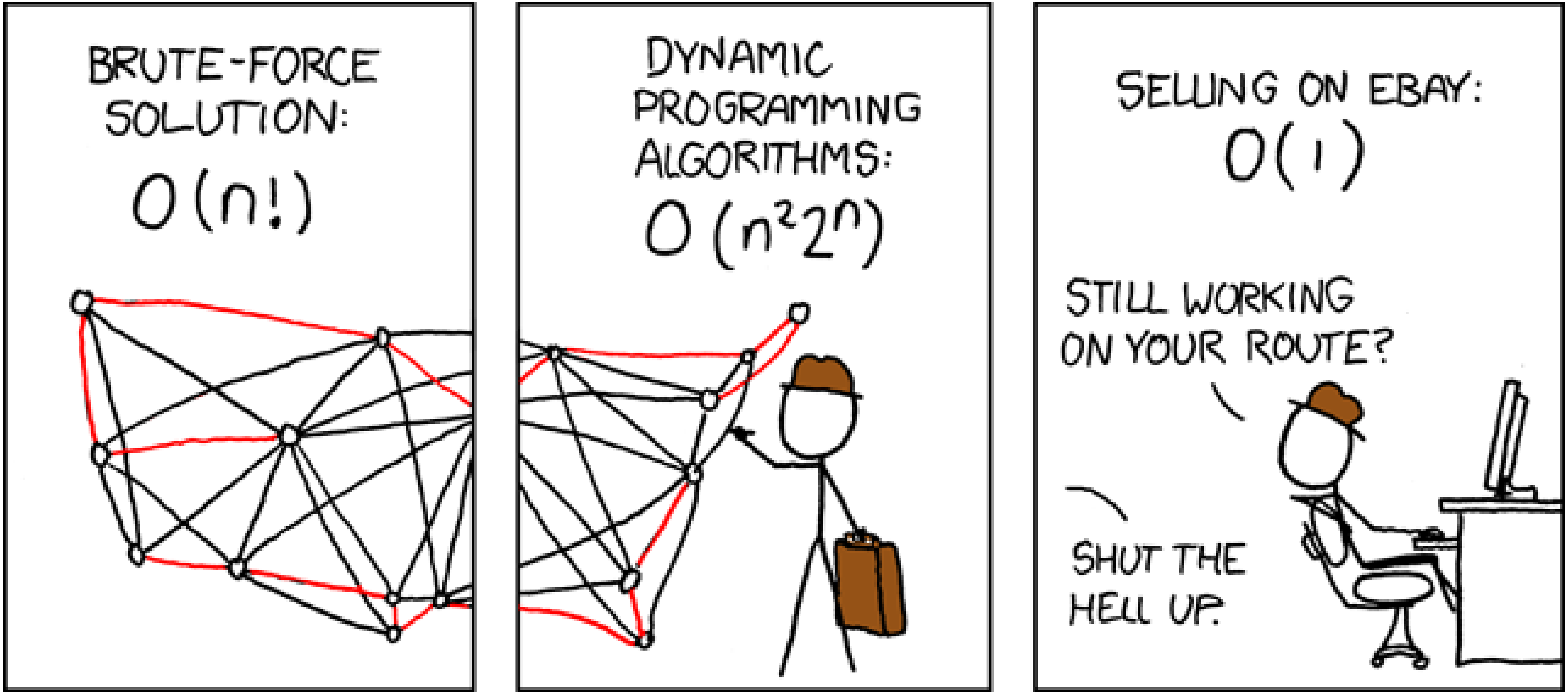
Worauf könnt ihr euch nächste Woche freuen?

- Dynamische Programmierung

Fragen?

Fragen!

Ende



https://xkcd.com/399/

Peter Bohner - peter.bohner@student.kit.edu