Universität Konstanz

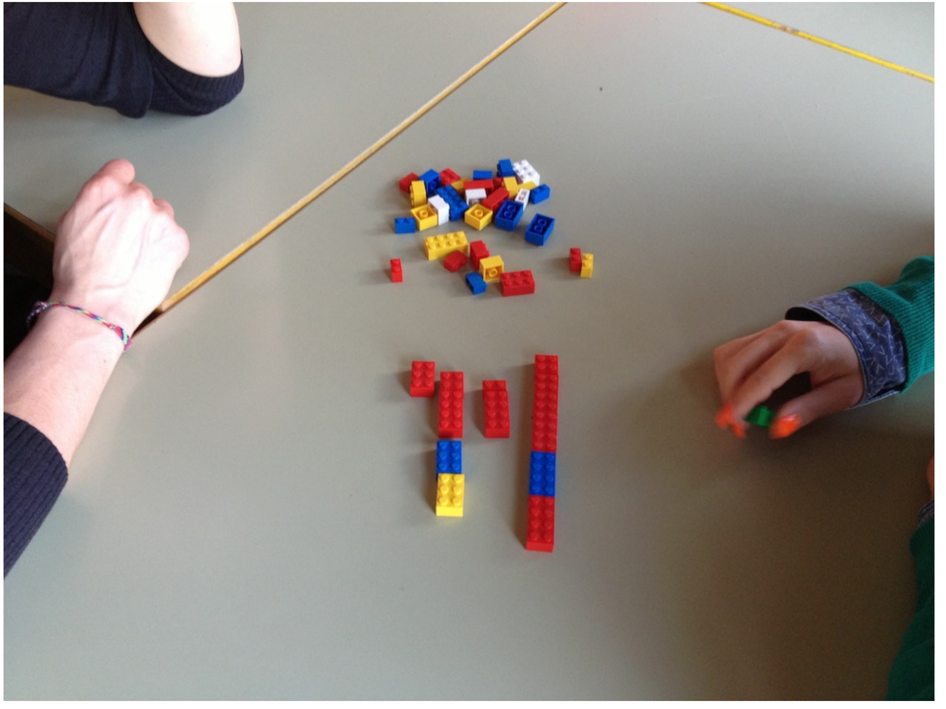
Konzepte der Informatik

Algorithmik
Dynamische Programmierung

Barbara Pampel

Universität Konstanz, WiSe 2023/2024

Beispiel



Optimierungsprobleme

- es gibt zu jeder Eingabe verschiedene zulässige Lösungen
- diese Lösungen haben Werte
- gesucht ist eine möglichst gute Lösung
  d.h. mit möglichst großem bzw. kleinem Wert

Abgrenzung

Greedy

- sukzessives Erweitern von Teillösungen
- lokal optimale Wahl
- aber, nur korrekt, wenn lokal optimale Wahl auch zu global optimalen Lösung fürt

Abgrenzung

Greedy

- sukzessives Erweitern von Teillösungen
- lokal optimale Wahl
- aber, nur korrekt, wenn lokal optimale Wahl auch zu global optimalen Lösung fürt

Divide and Conquer

- Eingabe wird in entkoppelte Teillösungen zerlegt
- getrennt gelöst
- ggf. dann wieder zusammengesetzt

Abgrenzung

Greedy

- sukzessives Erweitern von Teillösungen
- lokal optimale Wahl
- aber, nur korrekt, wenn lokal optimale Wahl
  auch zu global optimalen Lösung fürt

Divide and Conquer

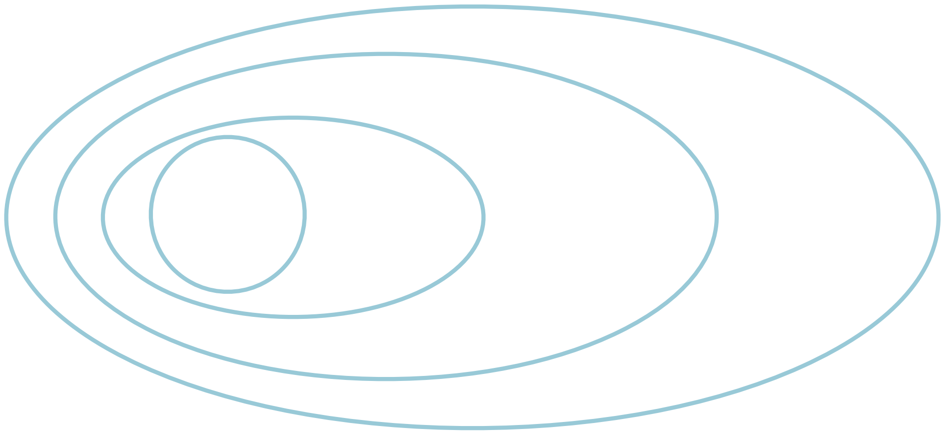
- Eingabe wird in entkoppelte Teillösungen zerlegt
- getrennt gelöst
- ggf. dann wieder zusammengesetzt

Rekursive Aufrufe

- Berechnung erfolgt bei Durchführung einer Methode

Dynamische Programmierung - Idee

- Rekursionsformel, aber Speichern und Wiederverwendung von Teillösungen
- schrittweise die überlappenden Teillösungen zur Gesamtlösung zusammensetzen
- aber, globale Werte im Auge behalten ⇒ Teilllösungen speichern
- typisch ist suksessives Füllen einer Matrix



Zur Verfügung stehen Steine der Längen $l_1 \dots l_n$. Ziel ist es, eine Mauer der geforderten Länge l, mit möglichst wenig Steinen zu bauen.

- Greedy nicht immer möglich
- Durchtesten aller möglichen Kombinationen sehr aufwändig
- Dynamische Programmierung: Füllen der Tabelle

Anzahle / Länge	<0	0	1	2	3	4	5	6	…	l
Anzahl Steine l_1	∞	0							…	
Anzahl Steine l_2	∞	0							…	
Anzahl Steine l_3	∞	0							…	
⋮										
Anzahl Steine l_n	∞	0							…	
Anzahl gesamt	∞	0							…	

Das Lego-Mauern-Problem

- Rekursionsformel

$$L[i] := \begin{cases} 1 + min_{j=1\dots n}(L[i - l_j]) & \text{if } i > 0 \\ 0 & \text{if } i = 0 \\ \infty & \text{else} \end{cases}$$

- Beispiel

Anzahlen / Länge	< 0	0	1	2	3	4	5	6
# Steine 1	∞	0	1	2	0	0	1	0
# Steine 3	∞	0	0	0	1	0	0	2
# Steine 4	∞	0	0	0	0	1	1	0
# gesamt	∞	0	1	2	1	1	2	2

Beispiel 2

lange Mauern mit begrenzten Kosten

- Steine haben Werte: s_i hat Länge l_i und Kosten k_i.
- Problemstellung: Baue eine möglist lange Mauer,
  welche eine gesetzte Kostengrenze k nicht überschreitet

Beispiel 2

lange Mauern mit begrenzten Kosten

- Steine haben Werte: s_i hat Länge l_i und Kosten k_i.
- Problemstellung: Baue eine möglist lange Mauer, welche eine gesetzte Kostengrenze k nicht überschreitet

Stein / Kosten	0	1	2	3	4	5	6	…	k
s_1									
s_2									
s_3									
⋮									
s_n									
keiner	0	0	0	0	0	0	0	…	0

Das Rucksackproblem - Knapsack

- Rekursionsformel: $R[i,j] := max(\underbrace{v(i) + R[i+1, j-w(i)]}_{\text{Element } i \text{ wird verwendet}}, \quad \underbrace{R[i+1, j]}_{i \text{ wird nicht verwendet}} \quad)$

Das Rucksackproblem - Knapsack

- Rekursionsformel: $R[i,j] := max(\underbrace{v(i) + R[i+1, j-w(i)]}_{\text{Element } i \text{ wird verwendet}}, \underbrace{R[i+1,j]}_{i \text{ wird nicht verwendet}})$

Algorithm 2: Knapsack-Algorithmus

Input: Grenze k, Gewichte-Array $w[]$ und Nutzwert-Array $v[]$
Data: Matrix $R := [1 \ldots (n+1), 0 \ldots k]$ mit Einträgen 0
begin
    for $i = n \ldots 1$ **do**
        for $j = 1 \ldots k$ **do**
            if $w(i) \leq j$ **then**
                $R[i,j] := max(v(i) + R[i+1, j-w(i)], R[i+1,j])$
            else $R[i,j] := R[i+1,j]$
    return $R[1,k]$