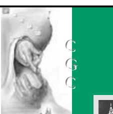


Informatik II
Dynamische Programmierung

G. Zachmann
Clausthal University, Germany
zach@in.tu-clausthal.de

- Zweite Technik für den Algorithmenentwurf
- Zur Herkunft des Begriffes:
  - "… Programmierung" hat nichts mit "Programmieren" zu tun, sondern mit "Verfahren"
    - Vergleiche "lineares Programmieren", "Integer-Programmieren" (alles Begriffe aus der Optimierungstheorie)
  - Dynamic programming = planning over time
  - Secretary of Defense was hostile to mathematical research
  - Bellman sought an impressive name to avoid confrontation
    - "it's impossible to use dynamic in a pejorative sense"
    - "something not even a Congressman could object to"
- Typische Anwendung für dynamisches Programmieren: Optimierung

Matrix Chain Multiplication Problem (MCMP)

- Gegeben: eine Folge (Kette) $A_1, A_2, \ldots, A_n$ von Matrizen mit verschiedenen Dimensionen
- Gesucht: das Produkt $A_1 \cdot A_2 \cdot \ldots \cdot A_n$
- Optimierungsaufgabe:
  Organisiere die Multiplikationen so, daß möglichst wenig skalare Multiplikationen ausgeführt werden
  - Generelle Idee hier: nutze Assoziativität aus
- Definition:
  Ein Matrizenprodukt heißt vollständig geklammert, wenn es entweder eine einzelne Matrix oder das geklammerte Produkt zweier vollständig geklammerter Matrizenprodukte ist

Multiplikation zweier Matrizen

$$A = (a_{ij})_{p \times q}, \; B = (b_{ij})_{q \times r}, \; A \cdot B = C = (c_{ij})_{p \times r}$$

$$c_{ij} = \sum_{k=1}^{q} a_{ik} b_{kj}$$

$p\,[A]_q \cdot q\,[B]_r = [C]_r\,p$

```
# Eingabe: pxq Matrix A, qxr Matrix B
# Ausgabe: pxr Matrix C = A·B
for i in range( 0,p ):
    for j in range( 0,r ):
        C[i,j] = 0
        for k in range( 0,q ):
            C[i,j] += A[i,k] * B[k,j]
```

- Anzahl der Muliplikationen und Additionen = $p \cdot q \cdot r$
- Erinnerung: für 2 $n \times n$-Matrizen werden hier n^3 Multiplikationen benötigt, aber es geht auch mit $O(n^{2.\mathrm{xxx}})$

Beispiel

- Berechnung des Produkts von A_1, A_2, A_3 mit
  A_1 : 10 × 100 Matrix
  A_2 : 100 × 5 Matrix
  A_3 : 5 × 50 Matrix

1. Klammerung $(A_1A_2)A_3$ erfordert

$A' = (A_1A_2)$	5000 Mult.
$A'A_3$	2500 Mult.
Summe:	7500 Mult.

2. Klammerung $A_1(A_2A_3)$ erfordert

$A'' = (A_2A_3)$	25000 Mult.
A_1A''	50000 Mult.
Summe:	75000 Mult.

Problemstellung

- Gegeben: Folge von Matrizen $A_1, A_2, \ldots, A_n$ und die Dimensionen $p_0, p_1, \ldots, p_n$, wobei Matrix A_i Dimension $p_{i-1} \times p_i$ hat
- Gesucht: eine Multiplikationsreihenfolge, die die Anzahl der Multiplikationen minimiert
- Beachte: der Algorithmus führt die Multiplikationen nicht aus, er bestimmt nur die optimale Reihenfolge!

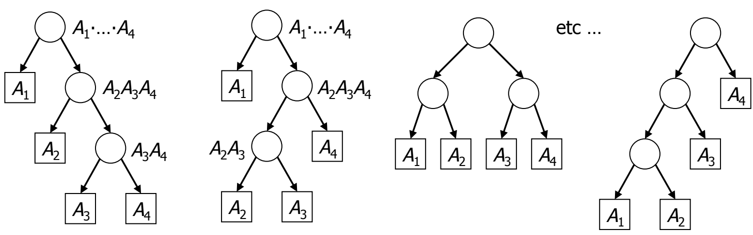
- Anwendungen: ermittle die optimale Ausführungsreihenfolge für eine Menge von Operationen
  - Z.B. im Compilerbau: Code-Optimierung
  - Bei Datenbanken: Anfrageoptimierung

Beispiel für $\langle A_1 \cdot A_2 \cdot \ldots \cdot A_n \rangle$

- Alle vollständig geklammerten Matrizenprodukte der Folge $\langle A_1, A_2, A_3, A_4 \rangle$ sind:

$$\big(A_1(A_2(A_3A_4))\big), \big(A_1((A_2A_3)A4)\big), \big((A_1A_2)(A_3A_4)\big),$$
$$\big((A_1(A_2A_3))A_4\big), \big(((A_1A_2)A_3)A_4\big)$$

- Klammerungen entsprechen strukturell verschiedenen Auswertungsbäumen:



Anzahl der verschiedenen Klammerungen

- $P(n)$ sei die Anzahl der verschiedenen Klammerungen von $A_1 \cdot \ldots \cdot A_k \cdot A_{k+1} \cdot \ldots \cdot A_n$:

$$P(1) = 1$$
$$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k) \quad \text{für } n \geq 2$$

- Definition:

$$P(n+1) =: C_n \quad = \text{n-te Catalan'sche Zahl}$$

- Es gilt (o. Bew.):

$$P(n+1) = \frac{1}{n+1}\binom{2n}{n} \approx \frac{4^n}{n\sqrt{\pi n}} + O\left(\frac{4^n}{\sqrt{n^5}}\right)$$

- Folge: Finden der optimalen Klammerung durch Ausprobieren aller Möglichkeiten ist sinnlos

Die Struktur der optimalen Klammerung

- Bezeichne mit $A_{i..j}$ das Produkt der Matrizen i bis j
  - $A_{i..j}$ ist eine $p_{i-1} \times p_j$ -Matrix
- Behauptung:
  Jede optimale Lösung des MCMP enthält optimale Lösungen von Teilproblemen
- Anders gesagt:
  Jede optimale Lösung des MCMP setzt sich zusammen aus optimalen Lösungen von bestimmten Teilproblemen

Beweis (durch Widerspruch)

- Sei eine optimale Lösung so geklammert
  $A_{i..j} = (A_{i...k})\ (A_{k+1...j})\ ,\quad i \leq k < j$
- Behauptung: die Klammerung innerhalb $A_{i...k}$ muß ihrerseits optimal sein
- Ann.: die Klammerung von $A_{i...k}$ innerhalb der optimalen Lösung zu $A_{i...j}$ sei nicht optimal

→ Es gibt bessere Klammerung von $A_{i...k}$ mit geringeren Kosten

- Setze diese Teillösung in Lösung zu $A_{i...j} = (A_{i...k})\ (A_{k+1...j})$ ein
- Ergibt eine bessere Lösung für $A_{i...j}$ → Widerspruch zur Annahme der Optimalität der ursprünglichen Lösung zu $A_{i...j}$

Eine rekursive Lösung

- Die optimalen Kosten können beschrieben werden als
  - $i = j$ → Folge enthält nur eine Matrix, keine Kosten
  - $i < j$ → kann geteilt werden, indem jedes k, $i \leq k < j$ betrachtet wird:
    Kosten für ein bestimmtes k = "Kosten links" + "Kosten rechts"
    + Kosten für die Matrix-Multiplikation $(A_{i...k}) \cdot (A_{k+1...j})$
- Daraus lässt sich die folgende rekursive Regel ableiten:
  $m[i,j]$ sei die minimale Anzahl von Operationen zur Berechnung des Teilprodukts $A_{i...j}$

$$m[i,j] = \begin{cases} 0, & \text{falls } i = j \\ \min\limits_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} \end{cases}$$

Ein naiver rekursiver Algorithmus

```
# Input p = Vektor der Array-Größen
# Output m[i,j] = optimale Kosten für die
# Multiplikation der Arrays i, .., j
def mcm_rek( p, i, j ):
    if i = j:
        return 0
    m = ∞
    for k in range( i,j ):
        q = p[i-1]*p[k]*p[j]      + \
            mcm_rek( p, i, k )    + \
            mcm_rek( p, k+1, j )
        if q < m:
            m = q
    return m
```

- Aufruf für das gesamte Problem: **`mcm_rek( p, 1, n )`**

Laufzeit des naiven rekursiven Algorithmus'

- Sei $T(n)$ die Anzahl der Operationen zur Berechnung von **`mcm_rek`** für Eingabefolge der Länge n

$$T(1) = 1$$
$$T(n) = 1 + \sum_{k=1}^{n-1} \big(T(k) + T(n-k) + 2\big)$$
$$\approx 2n + 2 \sum_{k=1}^{n-1} T(k) \quad \Rightarrow$$
$$T(n) \geq 2^{n-1} \qquad \text{(vollständige Induktion)}$$

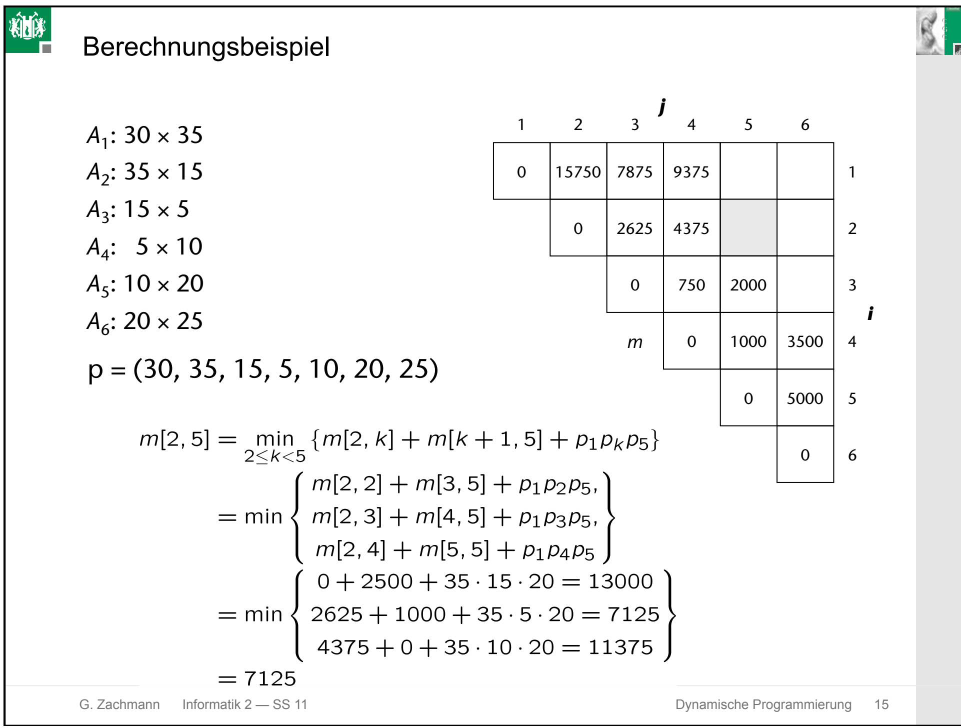
- Exponentielle Laufzeit! ☹

Formulierung mittels Dynamischer Programmierung

- Beobachtung: die Anzahl der Teilprobleme $A_{i..j}$ mit $1 \leq i \leq j \leq n$ ist nur $\frac{n(n+1)}{2} \in \Theta(n^2)$
- Folgerung: der naive rekursive Algo berechnet viele Teilprobleme mehrfach!
- Idee: Bottom-up-Berechnung der optimalen Lösung:
  - Speichere Teillösungen in einer Tabelle
  - Daher "dynamische Programmierung"
- Welche Tabelleneinträge werden für m[i,j] benötigt?

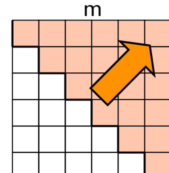
$$m[i,j] = \min_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\}$$

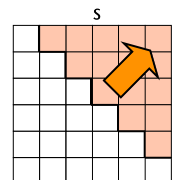
- Hier: bottom-up = von der Diagonale nach "rechts oben"



Gewinnung der optimalen Reihenfolge

- Speichere die Position für die beste Trennung, d.h., denjenigen Wert k, der zum minimalen Wert von $m[i,j]$ führt
- Speichere dazu in einem zweiten Array $s[i,j]$ dieses optimale k:
  - $s[i,j]$ wird nur für Folgen mit mindestens 2 Matrizen und $j > i$ benötigt





  - $s[i,j]$ gibt an, welche Multiplikation zuletzt ausgeführt werden soll
  - Für $s[i,j] = k$ und die Teilfolge $A_{i...j}$ ist es optimal, zuerst $A_{i...k}$, danach $A_{k+1...j}$ und zum Schluss die beiden Teilergebnisse zu multiplizieren:

$$A_{i..j} = A_i \cdots A_j = (A_i \cdots A_{s[i,j]})(A_{s[i,j]+1} \cdots A_j)$$

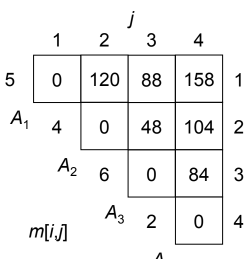
Algorithmus mittels dynamischer Programmierung

```
n = len(p) - 1
for i in range( 0, n+1 ): # assume m has dim (n+1)·(n+1)
    m[i,i] = 0
for L in range( 2,n+1 ):  # consider chains of length L
    for i in range( 1,n-L ):
        j = i+L-1         # len = L → j-i = L-1
        m[i,j] = ∞
        for k in range( i,j ):
            q = m[i,k] + m[k+1,j] + p[i-1]*p[k]*p[j]
            if q < m[i,j]:
                m[i,j] = q
                s[i,j] = k
```

- Komplexität: es gibt 3 geschachtelte Schleifen, die jeweils höchstens n-mal durchlaufen werden, die Laufzeit beträgt also $O(n^3)$

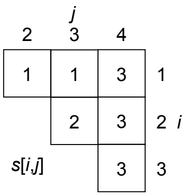
Beispiel

- Gegeben: die Folge von Dimensionen (5, 4, 6, 2, 7)
- Multiplikation von A_1 (5×4), A_2 (4×6), A_3 (6×2) und A_4 (2×7)
- Optimale Folge ist $((A_1(A_2A_3))A_4)$



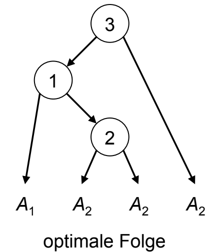
m[i,j] (Dimensionen: A_1: 5, 4; A_2: 6; A_3: 2; A_4: 7)

i \ j	1	2	3	4
1	0	120	88	158
2		0	48	104
3			0	84
4				0

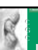


s[i,j]

i \ j	2	3	4
1	1	1	3
2		2	3
3			3



optimale Folge



Die Technik der dynamischen Programmierung

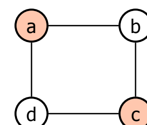
- Rekursiver Ansatz: Lösen eines Problems durch Lösen mehrerer kleinerer Teilprobleme, aus denen sich die Lösung für das Ausgangsproblem zusammensetzt
  - Häufiger Effekt: Mehrfachberechnungen von Lösungen
- Bottom-up-Berechnung: fügt Lösungen kleinerer Unterprobleme zusammen, um größere Unterprobleme zu lösen und liefert so eine Lösung für das gesamte Problem
  - Methode: iterative Erstellung einer Tabelle

Wichtige Begriffe

- Optimale Unterstruktur (*optimal substructure*):
  - Ein Problem besitzt die (Eigenschaft der) optimalen Substruktur, bzw. gehorcht dem Prinzip der Optimalität :⇔:

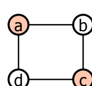
1. Die Lösung eines Problems setzt sich aus den Lösungen von Teilproblemen zusammen
   - Bsp. MCMP: gesuchte Klammerung von $A_1 \ldots A_n$ setzt sich zusammen aus der Klammerung einer (bestimmten) Teilkette $A_1 \ldots A_k$ und einer Teilkette $A_{k+1} \ldots A_n$
2. Wenn die Lösung optimal ist, dann müssen auch die Teillösungen optimal sein! (meistens durch Widerspruchsbeweis)
   - Bsp. MCMP: wir haben folgende Behauptung bewiesen:
     Falls Klammerung zu $A_1 \ldots A_k$ nicht optimal ⇒
     Klammerung zu $A_1 \ldots A_n$ (die gemäß Ann. die Teillsg zu $A_1 \ldots A_k$ enthält) kann nicht optimal sein

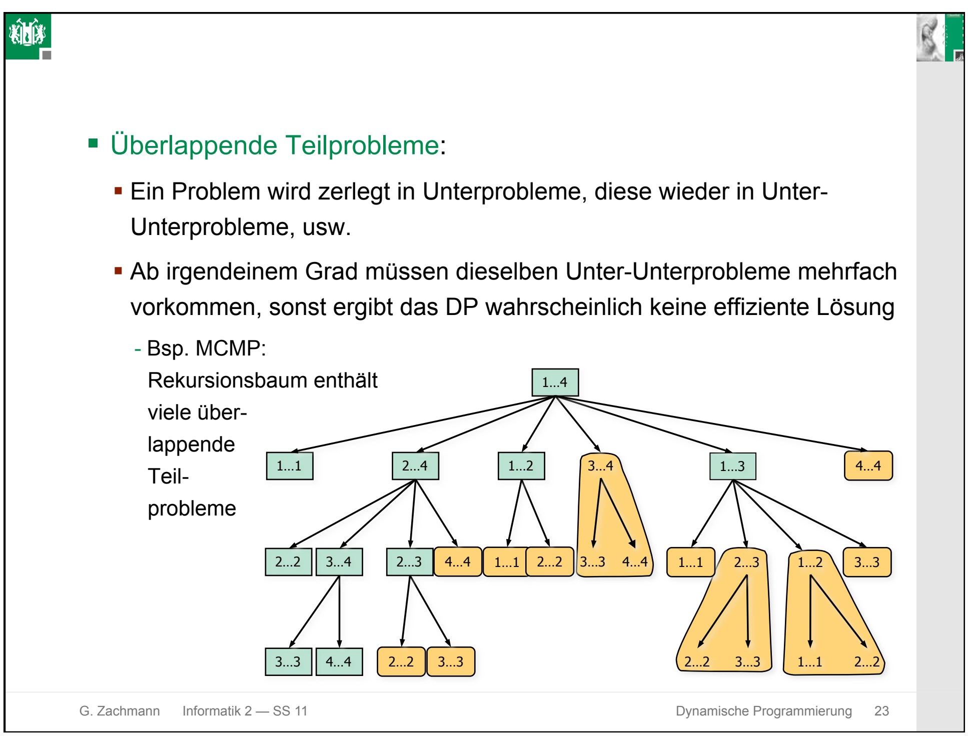
- Achtung: die zweite Bedingung (Teillösungen müssen optimal sein) ist manchmal nicht erfüllt:
  - Beispiel: längster Pfad durch einen Graphen
  - Aufgabe hier: bestimme längsten Pfad von a nach c



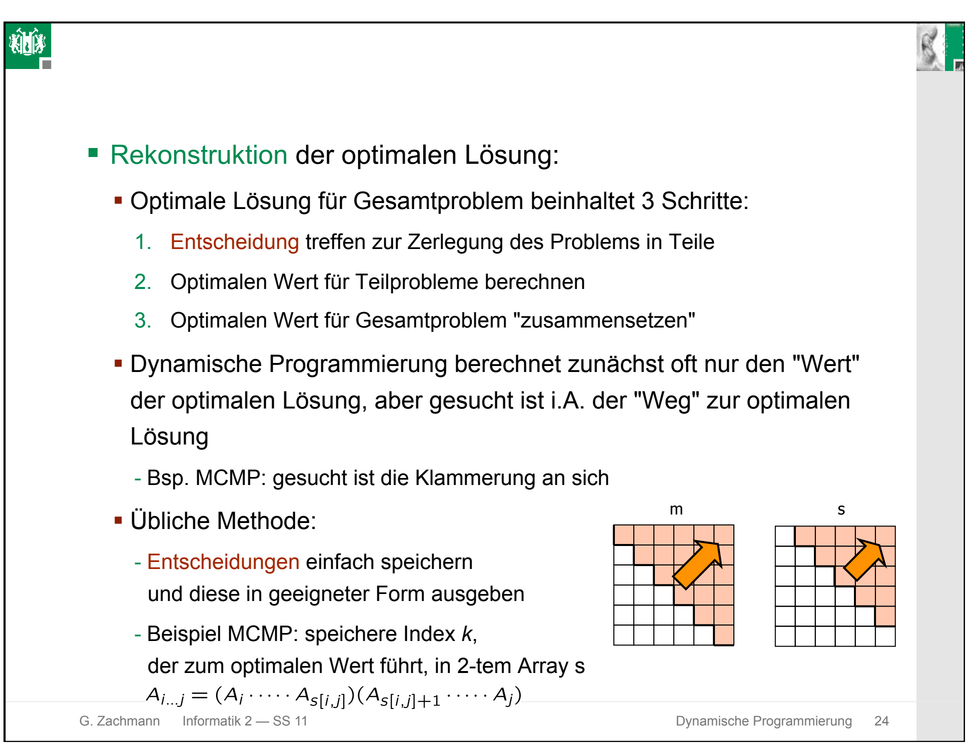
  - Im Beispiel rechts: Lösung besteht aus Teilpfaden a→b und b→c
  - Aber diese sind nicht optimale(!) Lösungen der entspr. Teilprobleme
    - Optimale (d.h., längste) Lösung für a→b = a→d→c→b

- Unabhängigkeit der Teillösungen:
  - Die Teilprobleme heißen (im Sinne der Dyn. Progr.) unabhängig :⇔: die Optimierung des einen Teilproblems beeinflußt nicht die Optimierung des anderen (z.B. bei der Wahl der Unterteilung)
    - Bsp. MCMP: die Wahl der Klammerung für $A_1 \ldots A_k$ ist völlig unabhängig von der Klammerung für $A_{k+1} \ldots A_n$
    - Gegenbsp. "längster Pfad": die optimale Lsg für a→b (nämlich a→d→c→b) nimmt der optimalen Lsg für b→c Elemente weg

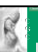




Überlappende Teilprobleme:
Ein Problem wird zerlegt in Unterprobleme, diese wieder in Unter-Unterprobleme, usw.
Ab irgendeinem Grad müssen dieselben Unter-Unterprobleme mehrfach vorkommen, sonst ergibt das DP wahrscheinlich keine effiziente Lösung
Bsp. MCMP: Rekursionsbaum enthält viele überlappende Teilprobleme
1...4
1...1
2...4
1...2
3...4
1...3
4...4
2...2
3...4
2...3
4...4
1...1
2...2
3...3
4...4
1...1
2...3
1...2
3...3
3...3
4...4
2...2
3...3
2...2
3...3
1...1
2...2
G. Zachmann Informatik 2 — SS 11
Dynamische Programmierung 23



Rekonstruktion der optimalen Lösung:
Optimale Lösung für Gesamtproblem beinhaltet 3 Schritte:
1. Entscheidung treffen zur Zerlegung des Problems in Teile
2. Optimalen Wert für Teilprobleme berechnen
3. Optimalen Wert für Gesamtproblem "zusammensetzen"
Dynamische Programmierung berechnet zunächst oft nur den "Wert" der optimalen Lösung, aber gesucht ist i.A. der "Weg" zur optimalen Lösung
Bsp. MCMP: gesucht ist die Klammerung an sich
Übliche Methode:
Entscheidungen einfach speichern und diese in geeigneter Form ausgeben
Beispiel MCMP: speichere Index k, der zum optimalen Wert führt, in 2-tem Array s
$A_{i..j} = (A_i \cdot \ldots \cdot A_{s[i,j]})(A_{s[i,j]+1} \cdot \ldots \cdot A_j)$
m
s
G. Zachmann Informatik 2 — SS 11
Dynamische Programmierung 24

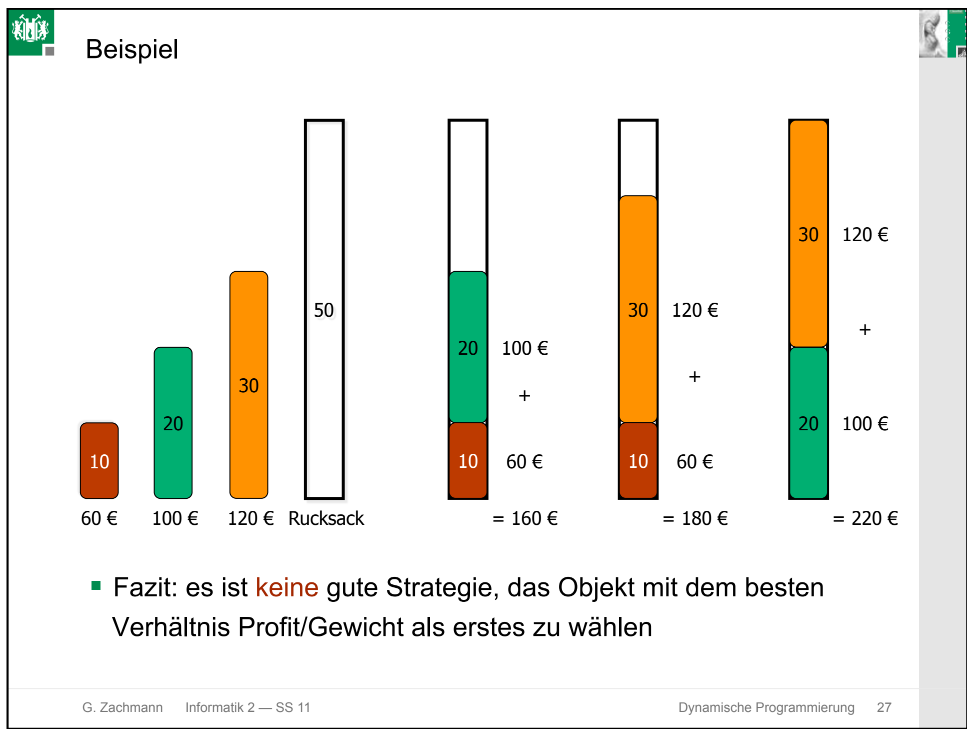


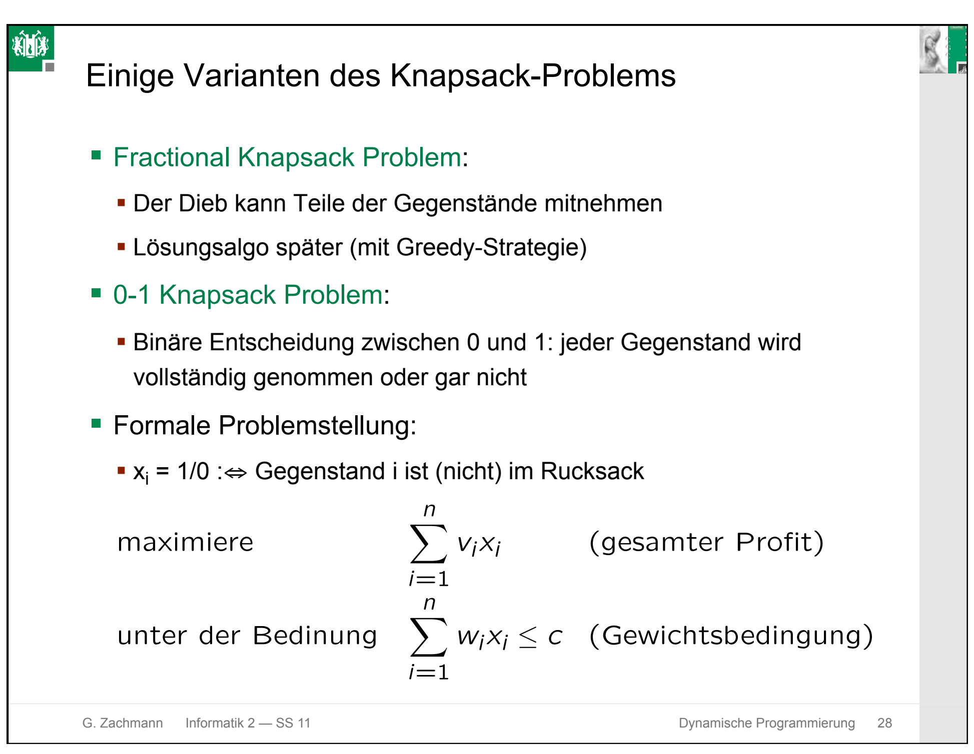
Schritte zur dynamischen Programmierung

1. Charakterisiere die (rekursive) Struktur der optimalen Lösung (Prinzip der optimalen Substruktur)
2. Definiere den Wert einer optimalen Lösung rekursiv
3. Transformiere die rekursive Methode in eine iterative bottom-up Methode, bei der alle Zwischenergebnisse in einer Tabelle gespeichert werden

Das Rucksack-Problem (*Knapsack Problem*)

- Das Problem:
  - "Die Qual der Wahl"
  - Beispiel: ein Dieb raubt einen Laden aus; um möglichst flexibel zu sein, hat er für die Beute nur einen Rucksack dabei
  - Im Laden findet er *n* Gegenstände; der *i*-te Gegenstand hat den Wert v_i und das Gewicht w_i
  - Sein Rucksack kann höchstens das Gewicht *c* tragen
  - w_i und *c* sind ganze Zahlen(!) (die v_i können aus $\mathbb{R}^+$ sein)
- Aufgabe: welche Gegenstände sollten für den maximalen Profit gewählt werden?

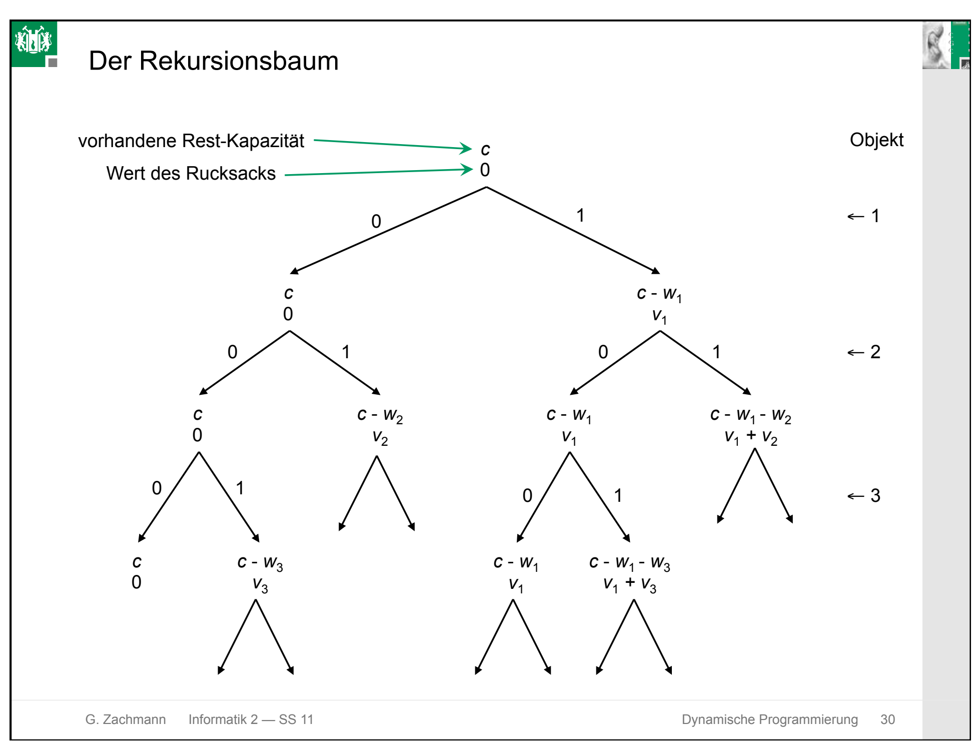

Beispiel
10
20
30
50
60 €
100 €
120 €
Rucksack
20
100 €
+
10
60 €
= 160 €
30
120 €
+
10
60 €
= 180 €
30
120 €
+
20
100 €
= 220 €
Fazit: es ist keine gute Strategie, das Objekt mit dem besten Verhältnis Profit/Gewicht als erstes zu wählen
G. Zachmann Informatik 2 — SS 11 Dynamische Programmierung 27


Einige Varianten des Knapsack-Problems
Fractional Knapsack Problem:
Der Dieb kann Teile der Gegenstände mitnehmen
Lösungsalgo später (mit Greedy-Strategie)
0-1 Knapsack Problem:
Binäre Entscheidung zwischen 0 und 1: jeder Gegenstand wird vollständig genommen oder gar nicht
Formale Problemstellung:
x_i = 1/0 :⇔ Gegenstand i ist (nicht) im Rucksack
maximiere $\sum_{i=1}^{n} v_i x_i$ (gesamter Profit)
unter der Bedingung $\sum_{i=1}^{n} w_i x_i \leq c$ (Gewichtsbedingung)
G. Zachmann Informatik 2 — SS 11 Dynamische Programmierung 28

Die rekursive Lösung

- Betrachte den ersten Gegenstand i=1;
  es gibt zwei Möglichkeiten:
  1. Der Gegenstand wird in den Rucksack gepackt (x_1=1)
     → Rest-Problem:
     $$\text{maximiere} \sum_{i=2}^{n} v_i x_i \quad \text{wobei} \sum_{i=2}^{n} w_i x_i \leq c - w_1$$
  2. Der Gegenstand wird nicht in den Rucksack gepackt (x_1=0)
     → Rest-Problem:
     $$\text{maximiere} \sum_{i=2}^{n} v_i x_i \quad \text{wobei} \sum_{i=2}^{n} w_i x_i \leq c$$
- Berechne beide Fälle, wähle den besseren

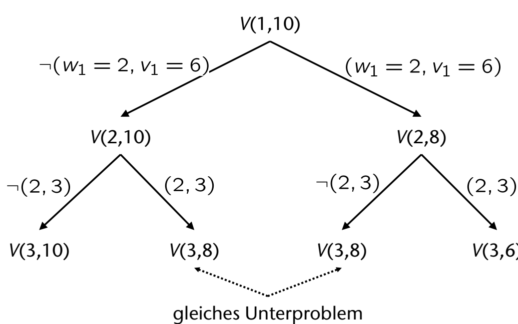
Der Rekursionsbaum



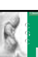
- Sei $V(i,k)$ = der maximal mögliche Wert für die Gegenstände i, i+1, …, n bei gegebener max. Kapazität k
- $V(i,k)$ kann dann für $i \le n$ geschrieben werden als

$$V(i,k) = \begin{cases} 0 & i = n \wedge w_n > k \\ v_n & i = n \wedge w_n \le k \\ V(i+1,k) & i < n \wedge w_i > k \\ \max\left\{ \begin{array}{l} V(i+1,k), \\ v_i + V(i+1,k-w_i) \end{array} \right\} & i < n \wedge w_i \le k \end{cases}$$

- Beispiel: n = 5, c = 10, w = (2, 2, 6, 5, 4), v = (6, 3, 5, 4, 6)



- Algorithmus, basierend auf diesen 4 Fällen, hat Laufzeit von $O(2^n)$
- Ist ineffizient, denn $V(i,k)$ wird für die gleichen i und k mehrmals berechnet → dynamic programming



Lösung mittels Dynamischer Programmierung

- Ineffizienz kann vermieden werden, indem alle $V(i,k)$, einmal berechnet, in einer Tabelle gespeichert werden
- Die Tabelle wird in der Reihenfolge i = n, n-1, …, 2, 1 für 1 ≤ k ≤ c gefüllt

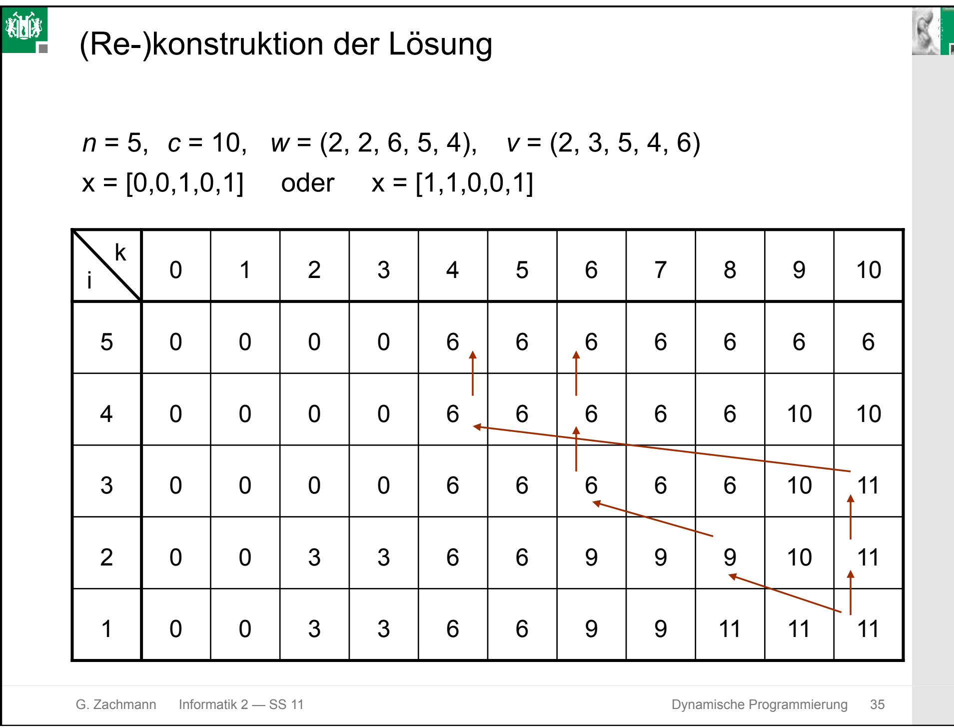
k	1	2	…	j-1	j	j+1	…	c
$V(n, k)$	0	0	…	0	v_n	v_n	…	v_n

j ist das erste k mit $w_n \le k$

Beispiel

$n = 5, \quad c = 10, \quad w = (2, 2, 6, 5, 4), \quad v = (2, 3, 5, 4, 6)$

i \ k	0	1	2	3	4	5	6	7	8	9	10
5	0	0	0	0	6	6	6	6	6	6	6
4	0	0	0	0	6	6	6	6	6	10	10
3	0	0	0	0	6	6	6	6	6	10	11
2	0	0	3	3	6	6	9	9	9	10	11
1	0	0	3	3	6	6	9	9	11	11	11



i \ k	0	1	2	3	4	5	6	7	8	9	10
5	0	0	0	0	6	6	6	6	6	6	6
4	0	0	0	0	6	6	6	6	6	10	10
3	0	0	0	0	6	6	6	6	6	10	11
2	0	0	3	3	6	6	9	9	9	10	11
1	0	0	3	3	6	6	9	9	11	11	11

Bemerkungen

- Aufwand: $O(n \cdot c)$, c = Kapazität des Rucksacks
- Solche Komplexitäten nennt man auch pseudo-polynomial time
  - Dabei erscheint nicht (nur) die Länge des Inputs, sondern (auch) dessen *Wert* in der Komplexitätsfunktion!
- Achtung: dieser Algorithmus klappt nur, wenn c und die w_i Integers sind!
- Falls c oder die w_i keine Integers sind, dann ist das Problem "NP-vollständig", und es gibt (wahrscheinlich) keinen polynomiellen Algorithmus

G. Zachmann Informatik 2 — SS 11 Dynamische Programmierung 36

Die längste gemeinsame Teilfolge

- Seien $X = (x_1, \ldots, x_m)$ und $Y = (y_1, \ldots, y_n)$ zwei Folgen, wobei $x_i, y_i \in A$ für ein endliches Alphabet A.
  Dann heißt Y Teilfolge von X, wenn es aufsteigend sortierte Indizes $i_1, \ldots, i_n$ gibt, mit $x_{i_j} = y_j$ für $j = 1, \ldots, n$
- Beispiel: Y = BCAC ist Teilfolge von X = ABACABC;
  wähle $(i_1, i_2, i_3, i_4) = (2,4,5,7)$
- Sind X, Y, Z Folgen über A, so heißt Z gemeinsame Teilfolge von X und Y, wenn Z Teilfolge sowohl von X als auch Y ist
- Beispiel: Z = BCAC ist gemeinsame Teilfolge von X = ABACABC und Y = BACCABBC

- Z heißt längste gemeinsame Teilfolge von X und Y, wenn Z gemeinsame Teilfolge von X und Y ist und es keine andere gemeinsame Teilfolge von X und Y gibt, die länger ist
- Beispiel: Z = BCAC ist *nicht* längste gemeinsame Teilfolge von X =ABACABC und Y = BACCABBC, denn BACAC ist eine längere gemeinsame Teilfolge von X und Y
- Beim Problem Längste-Gemeinsame-Teilfolge (*longest-common-subsequence problem*, LCSP) sind als Eingabe zwei Folgen $X = (x_1, \ldots, x_m)$ und $Y = (y_1, \ldots, y_n)$ gegeben, gesucht ist eine längste gemeinsame Teilfolge X und Y
- Anwendung: "Distanz" zwischen Strings messen (Edit-Distanz)

Ein naiver Algorithmus

- Für jede mögliche Unterfolge von X:
  prüfe ob es eine Unterfolge von Y ist
- Laufzeit: $\Theta(n\ 2^m)$
  - Es gibt 2^m mögliche Unterfolgen von X
  - Für jede Unterfolge wird Zeit $\Theta(n)$ benötigt, um zu überprüfen, ob diese Unterfolge von Y ist:
    - Scanne Y, "verbrauche" jeweils den nächsten Buchstaben von X, falls er passt
    - X ist Unterfolge von Y, wenn am Ende von Y kein Zeichen von X mehr übrig ist

Die Struktur des LCSP

- Definition: sei $X = (x_1, \ldots, x_m)$ eine beliebige Folge.
  Für $i = 0, 1, \ldots, m$ ist der i-te Präfix von X definiert als
  $$X_i = (x_1, \ldots, x_i).$$
  Der i-te Präfix von X besteht also aus den ersten i Symbolen von X, der 0-te Präfix ist die leere Folge.

- Satz:

  Seien $X = (x_1, \ldots, x_m)$ und $Y = (y_1, \ldots, y_n)$ beliebige Folgen und sei $Z = (z_1, \ldots, z_k)$ eine längste gemeinsame Teilfolge von X und Y.

  Dann gilt eine der folgenden drei Aussagen:

  1. Ist $x_m = y_n$, dann ist $z_k = x_m = y_n$, und Z_{k-1} ist eine längste gemeinsame Teilfolge von X_{m-1} und Y_{n-1}
  2. Ist $x_m \neq y_n$ und $z_k \neq x_m$, dann ist Z eine längste gemeinsame Teilfolge von X_{m-1} und Y
  3. Ist $x_m \neq y_n$ und $z_k \neq y_n$, dann ist Z eine längste gemeinsame Teilfolge von X und Y_{n-1}

Beweis

- Fall 1 ($x_m = y_n$):
  1. Jede gemeinsame Teilfolge Z', die nicht mit $z'_l = x_m = y_n$ endet, kann verlängert werden, indem $x_m = y_n$ angefügt wird ⇒
     - die LCS Z muß mit $x_m = y_n$ enden;
  2. Z_{k-1} ist längste gemeinsame Teilfolge von X_{m-1} und Y_{n-1}, denn
     - es gibt keine längere gemeinsame Teilfolge von X_{m-1} und Y_{n-1}, oder Z wäre keine längste gemeinsame Teilfolge.
- Fall 2 ($x_m \neq y_n$ und $z_k \neq x_m$):
  - Da Z nicht mit x_m endet ⇒
  - Z ist gemeinsame Teilfolge von X_{m-1} und Y;
  - daher gibt es keine längere gemeinsame Teilfolge von X_{m-1} und Y, oder Z wäre keine längste gemeinsame Teilfolge

Eine Rekursion für die Länge der LCS

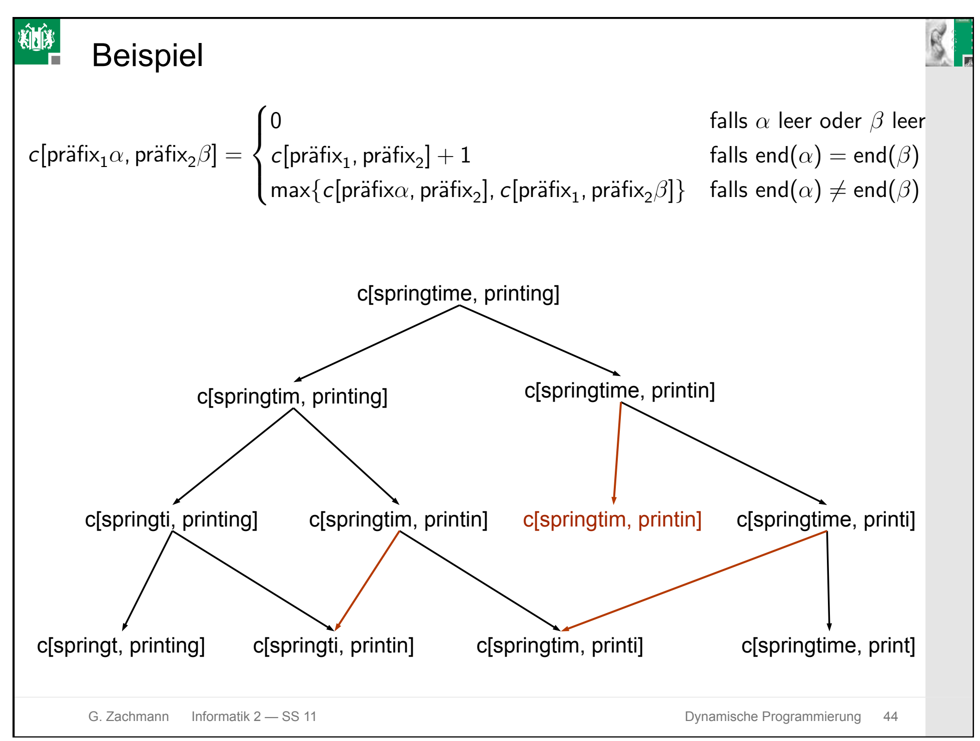
- Lemma:
  Sei $c[i,j]$ die Länge einer längsten gemeinsamen Teilfolge des i-ten Präfix X_i von X und des j-ten Präfix Y_j von Y, dann gilt

$$c[i,j] = \begin{cases} 0 & \text{falls } i = 0 \vee j = 0 \\ c[i-1,j-1]+1 & \text{falls } i,j > 0 \wedge x_i = y_j \\ \max\{c[i-1,j], c[i,j-1]\} & \text{falls } i,j > 0 \wedge x_i \neq y_j \end{cases}$$

- Beobachtung:
  - Die rekursive Berechnung der $c[m,n]$ würde immer wieder zur Berechnung derselben Werte führen!
  - Also dynamische Programmierung: berechne die Werte $c[i,j]$ iterativ bottom-up, z.B. zeilenweise
  - Zusätzlich: speichere in $b[i,j]$ Informationen zur späteren (Re-) Konstruktion einer längsten gemeinsamen Teilfolge

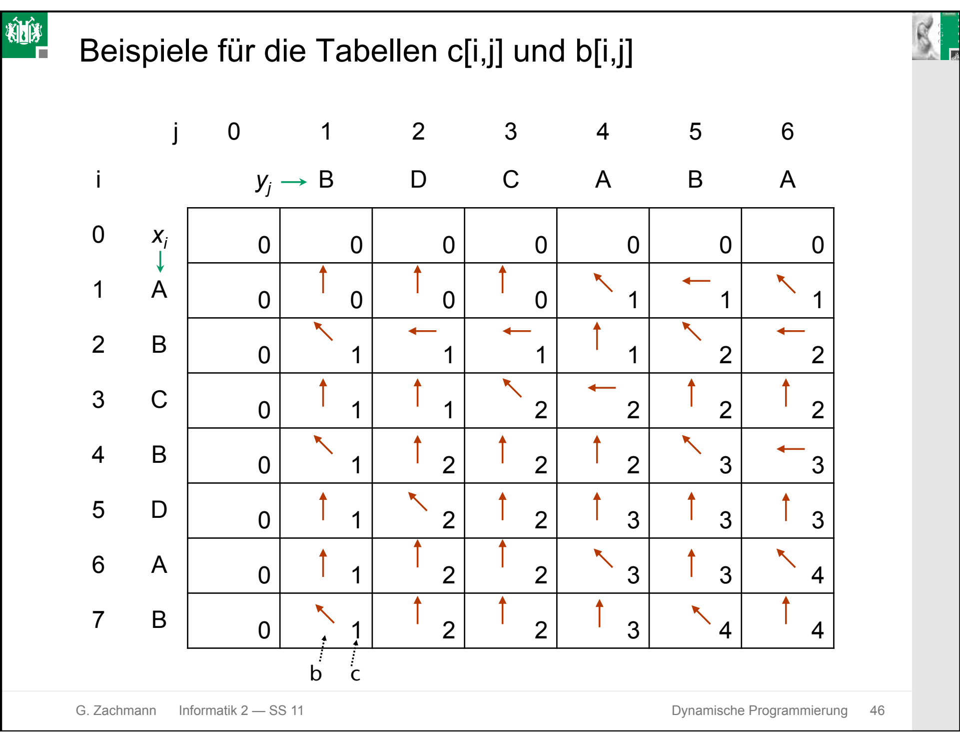
Beispiel

$$c[\text{präfix}_1\alpha, \text{präfix}_2\beta] = \begin{cases} 0 & \text{falls } \alpha \text{ leer oder } \beta \text{ leer} \\ c[\text{präfix}_1, \text{präfix}_2]+1 & \text{falls end}(\alpha) = \text{end}(\beta) \\ \max\{c[\text{präfix}\alpha, \text{präfix}_2], c[\text{präfix}_1, \text{präfix}_2\beta]\} & \text{falls end}(\alpha) \neq \text{end}(\beta) \end{cases}$$



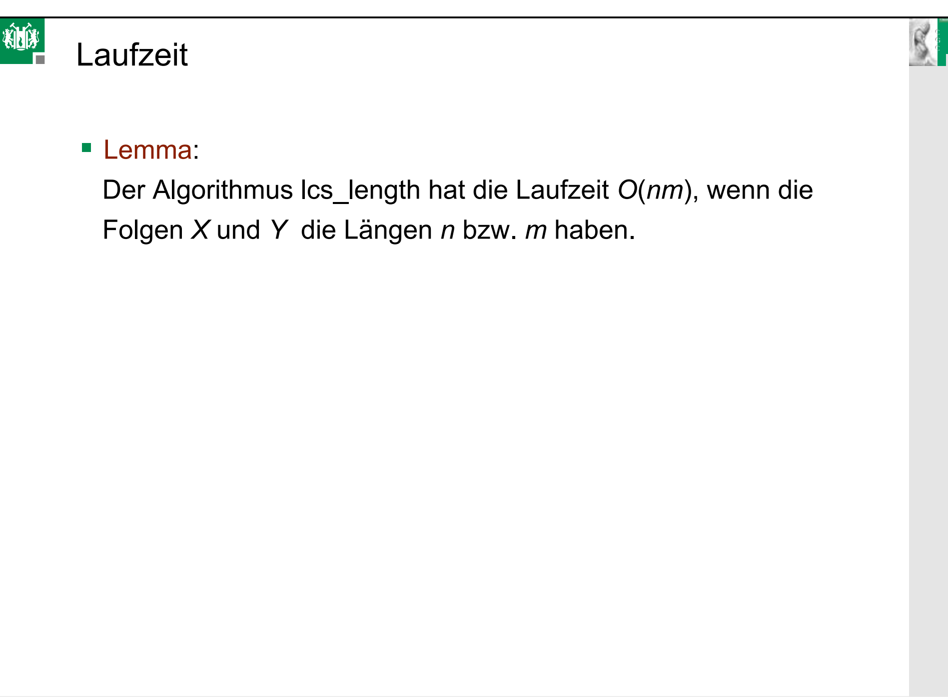
Berechnung der Werte $c[i,j]$

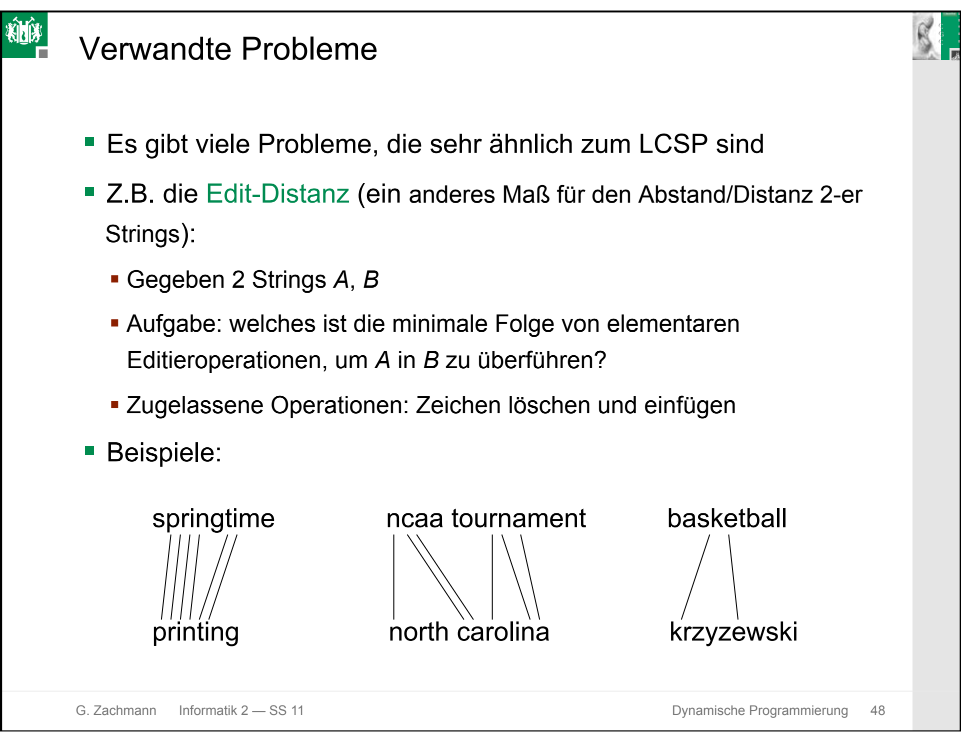
```
def lcs_length( x, y ):
    for i in range( 0, len(x) ):
        c[i,0] = 0
    for j in range( 0, len(y) ):
        c[0,j] = 0
    for i in range( 1, len(x) ):
        for j in range( 1, len(y) ):
            if x[i] == y[j]:
                c[i,j] = c[i-1,j-1] + 1
                b[i,j] = "NW"
            else:
                if c[i-1,j] >= c[i,j-1]:
                    c[i,j] = c[i-1,j]
                    b[i,j] = "N"
                else:
                    c[i,j] = c[i,j-1]
                    b[i,j] = "W"
    return b, c
```

Beispiele für die Tabellen c[i,j] und b[i,j]

i	j	0	1	2	3	4	5	6
	y_j →		B	D	C	A	B	A
0	x_i ↓	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 2	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 3	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4


Laufzeit
Lemma:
Der Algorithmus lcs_length hat die Laufzeit O(nm), wenn die Folgen X und Y die Längen n bzw. m haben.
G. Zachmann Informatik 2 — SS 11 Dynamische Programmierung 47


Verwandte Probleme
Es gibt viele Probleme, die sehr ähnlich zum LCSP sind
Z.B. die Edit-Distanz (ein anderes Maß für den Abstand/Distanz 2-er Strings):
Gegeben 2 Strings A, B
Aufgabe: welches ist die minimale Folge von elementaren Editieroperationen, um A in B zu überführen?
Zugelassene Operationen: Zeichen löschen und einfügen
Beispiele:
springtime
printing
ncaa tournament
north carolina
basketball
krzyzewski
G. Zachmann Informatik 2 — SS 11 Dynamische Programmierung 48

- Approximative Stringsuche:
  - Gegeben Text *T* und String *S*
  - Finde dasjenige Teilstück $T[i:j]$, das am ähnlichsten zu *S* ist (von allen anderen Teilstücken $T[i':j']$)
- Anwendungen: DNA-Sequence-Alignment, Google, Spell-Checker

Memoisierung im Top-down-Ansatz

- "Memo" = Gedächtnis
- Üblicherweise ist Formulierung der optimalen Lösung rekursiv, aber Algorithmus geht bottom-up vor
- *Memoization* [sic] = Technik in der dynamischen Programmierung, falls ein Bottom-up-Algorithmus nicht klar ist
- Notizblock-Methode zur Beschleunigung einer rekursiven Problemlösung:
  - Algo bleibt rekursiv
  - Ein Teilproblem wird nur beim ersten Auftreten gelöst
  - Die Lösung wird in einer Tabelle (z.B. Hash Table) gespeichert und bei jedem späteren Auftreten desselben Teilproblems (d.h., rekursiver Aufruf mit denselben Parametern) wird die Lösung (ohne erneute Rechnung!) in der Tabelle nachgeschlagen

Beispiel: MCMP mittels Memoisierung

```
def mcm_mem_rek( p, i, j ):
    if i == j:
        return 0
    if m[i,j] < ∞ :               # check first,
        return m[i,j]             # if already computed
    for k in range( i,j ):
        q = p[i-1]*p[k]*p[j] + \
            mcm_rek(p,i,k) + \
            mcm_rek(p, k+1, j)
        if q < m[i,j]:
            m[i,j] = q
    return m[i,j]

def mcm_mem( p ):
    for i in range( 1, len(p)+1 ):
        for j in range( 1, len(p)+1 ):
          m[i,j] = ∞                   # z.B. maxint
    return mcm_mem_rek( p, 1, len(p)-1 )
```


Aufwand

- Behauptung: Zur Berechnung aller Einträge $m[i,j]$ mit Hilfe von mcm_mem_rek genügen insgesamt $O(n^3)$ Schritte
- Beweis:
  - $O(n^2)$ Einträge
  - jedes Element $m[i,j]$ wird einmal eingetragen
  - jeder Eintrag $m[i,j]$ wird zur Berechnung von weniger als $2n$ weiteren Einträgen $m[i',j']$ herangezogen, wobei $i = i' \wedge j < j'$ oder $i > i' \wedge j = j'$

- Bemerkungen zum MCMP
  - Es gibt einen Algorithmus mit linearer Laufzeit $O(n)$, der eine Klammerung mit Multiplikationsaufwand $\leq 1.155 \cdot M_{opt}$ findet
  - Es gibt einen Algorithmus mit Laufzeit $O(n \log n)$, der eine optimale Klammerung findet

Zusammenfassung

- Dynamische Programmierung = Algorithmenentwurfstechnik, die oft bei Optimierungsproblemen angewandt wird
  - Man muß eine Menge von Entscheidungen treffen, die Bedingungen unterliegen, um eine optimale (min. / max.) Lösung zu erlangen
  - Es kann verschiedene Lösungswege geben
- Allgemein einsetzbar bei rekursiven Verfahren, wenn Teillösungen (von Unterproblemen) mehrfach benötigt werden
- Lösungsansatz: Tabellieren von Teilergebnissen
- Vergleich mit Divide-and-Conquer:
  - Ähnlich: DP berechnet Lösung eines Problems aus Lösungen von Teilproblemen
  - Anders: Teilprobleme werden oft nicht rekursiv gelöst sondern iterativ, beginnend mit den Lösungen der kleinsten Teilprobleme (bottom-up)

Zwei verschiedene Ansätze

- Bottom-up
  - \+ kontrollierte effiziente Tabellenverwaltung, spart Zeit
  - \+ spezielle optimierte Berechnungsreihenfolge, spart Platz
  - \- weitgehende Umcodierung des Originalprogramms erforderlich
  - \- möglicherweise Berechnung nicht benötigter Werte
- Top-down (Memoisierung, Notizblockmethode)
  - \+ Originalprogramm wird nur gering oder nicht verändert
  - \+ nur tatsächlich benötigte Werte werden berechnet
  - \- eventuell unnötige rekursive Aufrufe
  - \- Tabellengröße oft nicht optimal