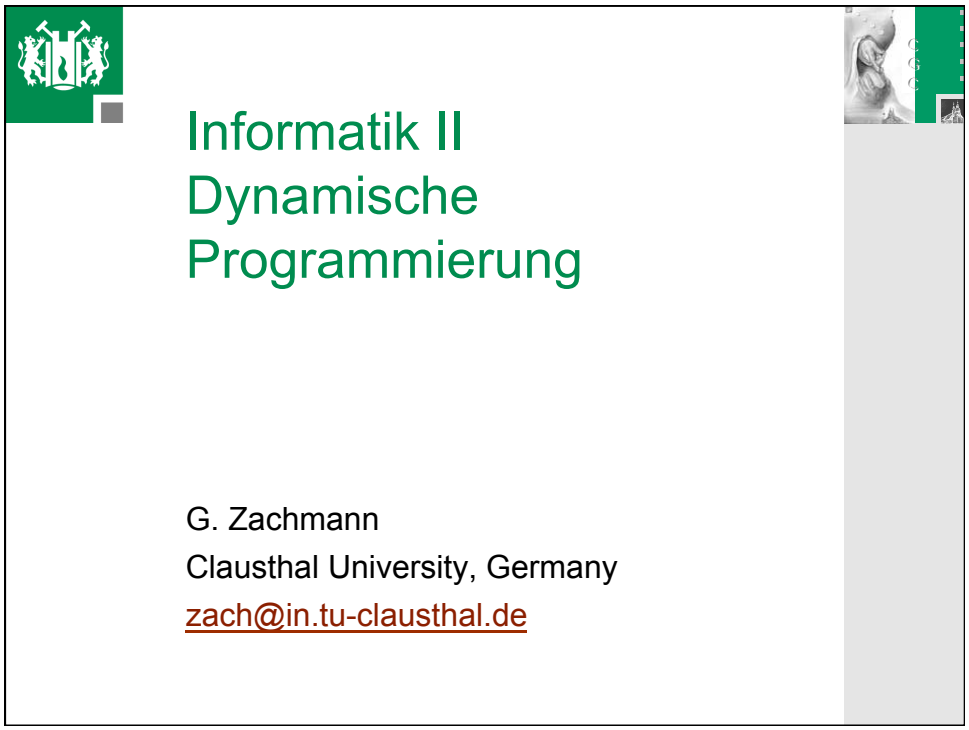
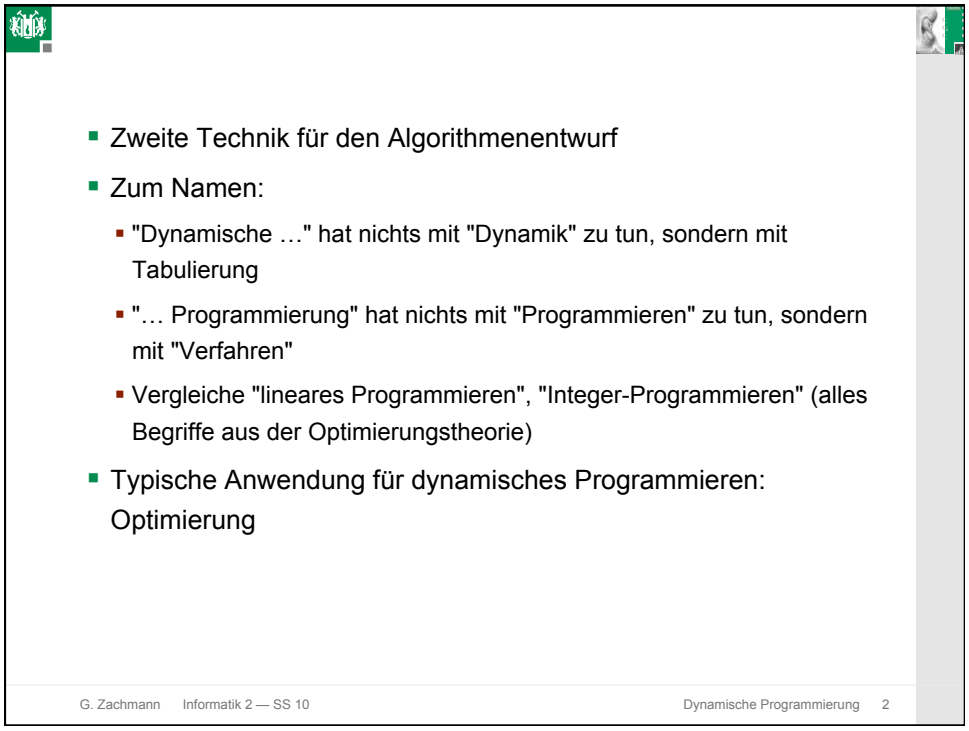


Informatik II
Dynamische Programmierung

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



- Zweite Technik für den Algorithmenentwurf
- Zum Namen:
  - "Dynamische …" hat nichts mit "Dynamik" zu tun, sondern mit Tabulierung
  - "… Programmierung" hat nichts mit "Programmieren" zu tun, sondern mit "Verfahren"
  - Vergleiche "lineares Programmieren", "Integer-Programmieren" (alles Begriffe aus der Optimierungstheorie)
- Typische Anwendung für dynamisches Programmieren: Optimierung

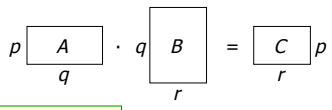
Matrix Chain Multiplication Problem (MCMP)

- Gegeben: eine Folge (Kette) $A_1, A_2, \ldots, A_n$ von Matrizen mit verschiedenen Dimensionen
- Gesucht: das Produkt $A_1 \cdot A_2 \cdot \ldots \cdot A_n$
- Aufgabe:
  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

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

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




```
# 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 = $p \cdot q \cdot r$
- Bem: für 2 $n \times n$-Matrizen werden hier n^3 Multiplikationen benötigt, es geht auch mit $O(n^{2.378})$

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!

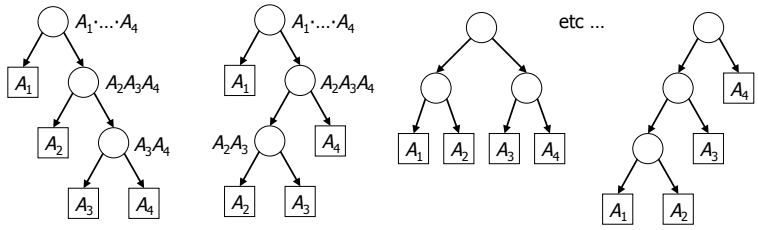
- Verallgemeinerung: ermittle die optimale Ausführungsreihenfolge für eine Menge von Operationen
  - Z.B. im Compilerbau: Code-Optimierung
  - Bei Datenbanken: Anfrageoptimierung

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

- Sei $A_{i..j}$ 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
  $$A_{i..j} = (A_{i...k})\,(A_{k+1..j}) \quad , \quad i \le k < j$$
- 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 → Widerspruch zur Annahme der Optimalität der ursprünglichen Lösung zu $A_{i..j}$

Eine rekursive Lösung

- Auf der höchsten Stufen werden 2 Matrizen multipliziert, d.h., für jedes k, $1 \le k \le n-1$, ist $(A_{1\ldots n}) = ((A_{1\ldots k})\,(A_{k+1\ldots n}))$
- Die optimalen Kosten können beschrieben werden als
  - $i = j$ → Folge enthält nur eine Matrix, keine Kosten
  - $i < j$ → kann geteilt werden, indem jedes k, $i \le k < j$ betrachtet wird:
    Kosten für ein bestimmtes k = "Kosten links" + "Kosten rechts"
    Kosten für die Matrix-Multiplikation $(A_{i\ldots k}) \cdot (A_{k+1\ldots j})$
- Daraus lässt sich die folgende rekursive Regel ableiten:
  $m[i,j]$ sei die minimale Anzahl von Operationen zur Berechnung des Teilprodukts $A_{i\ldots j}$

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

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 Schritte zur Berechnung von **`mcm_rek`** für Eingabefolge der Länge n

$$T(1) = 1$$

$$T(n) = 1 + \sum_{k=1}^{n-1} \big(T(k) + T(n-k) + 2\big)$$

$$\approx 2n + 2 \sum_{k=1}^{n-1} T(k) \quad \Rightarrow$$

$$T(n) \geq 2^{n-1} \qquad \text{(vollständige Induktion)}$$

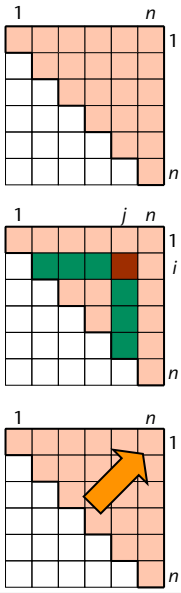
- Exponentielle Laufzeit! ☹

Formulierung mittels Dynamischer Programmierung

- Beobachtung: die Anzahl der Teilprobleme $A_{i\ldots j}$ mit $1 \leq i \leq j \leq n$ ist nur $\frac{n(n+1)}{2} \in \Theta\big(n^2\big)$
- 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?

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

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



Berechnungsbeispiel

A_1: 30 × 35
A_2: 35 × 15
A_3: 15 × 5
A_4: 5 × 10
A_5: 10 × 20
A_6: 20 × 25

p = (30, 35, 15, 5, 10, 20, 25)

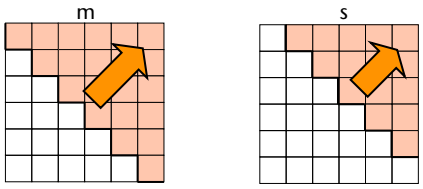
i \ *j*	1	2	3	4	5	6
1	0	15750	7875	9375		
2		0	2625	4375		
3			0	750	2000	
4				0	1000	3500
5					0	5000
6						0

m

$$m[2,5] = \min_{2 \le k < 5} \{m[2,k] + m[k+1,5] + p_1 p_k p_5\}$$
$$= \min \left\{ \begin{array}{l} m[2,2] + m[3,5] + p_1 p_2 p_5, \\ m[2,3] + m[4,5] + p_1 p_3 p_5, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 \end{array} \right\}$$
$$= \min \left\{ \begin{array}{l} 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000 \\ 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125 \\ 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{array} \right\}$$
$$= 7125$$

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 $A_{i\ldots j}$ ist es optimal, zuerst $A_{i\ldots k}$, danach $A_{k+1\ldots j}$ und zum Schluss die beiden Teilergebnisse zu multiplizieren:

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

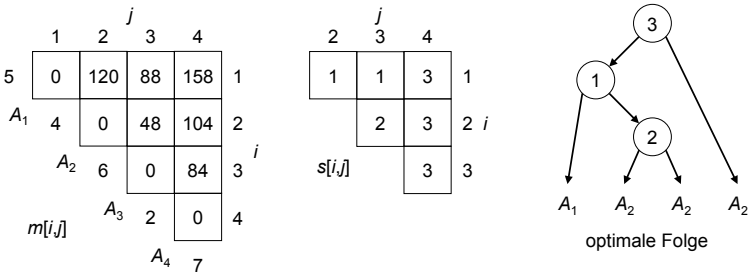
Algorithmus mittels dynamischer Programmierung

```
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 $((A_1(A_2A_3))A_4)$



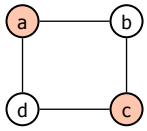
Die Technik der dynamischen Programmierung

- 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!
     - 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. Teillsg zu $A_1 \ldots A_k$ enthält) kann nicht optimal sein

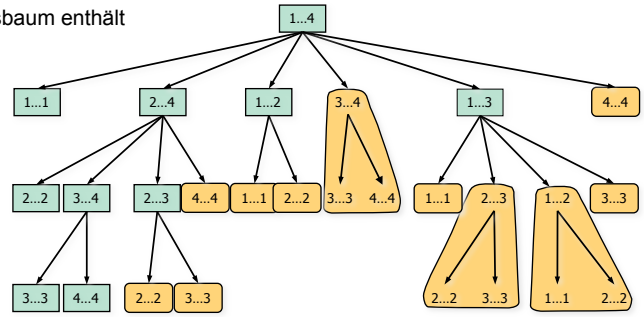
- Achtung: die zweite Bedingung (Teillösungen müssen optimal sein) ist manchmal nicht erfüllt:
  - Beispiel: längster Pfad durch einen Graphen
  - Aufgabe hier: bestimme längsten Pfad von a nach c



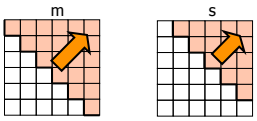
  - Im Beispiel rechts: Lösung besteht aus Teilpfaden a→b und b→c
  - Aber diese sind nicht optimale(!) Lösungen der entspr. Teilprobleme
    - Optimale (d.h., längste) Lösung für a→b = a→d→c→b

- 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



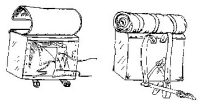
- Rekonstruktion der optimalen Lösung:
  - Optimale Lösung für Gesamtproblem beinhaltet 3 Schritte:
    1. Entscheidung treffen zur Zerlegung des Problems in Teile
    2. Optimalen Wert für Teilprobleme berechnen
    3. Optimalen Wert für Gesamtproblem "zusammensetzen"
  - Dynamische Programmierung berechnet zunächst oft nur den "Weg" zur optimalen Lösung, aber
    - im zweiten Schritt wird dann die optimale Lösung mittels diese Weges berechnet;
    - dazu Entscheidungen einfach in Phase 1 speichern und in Phase 2 dann "abspielen"
    - Beispiel: MCMP
      Speichere Index k, der zum optimalen Wert führt in zweitem Array s
      $A_{i..j} = (A_i \cdot \cdots \cdot A_{s[i,j]})(A_{s[i,j]+1} \cdot \cdots \cdot A_j)$



Schritte bei der dynamischen Programmierung

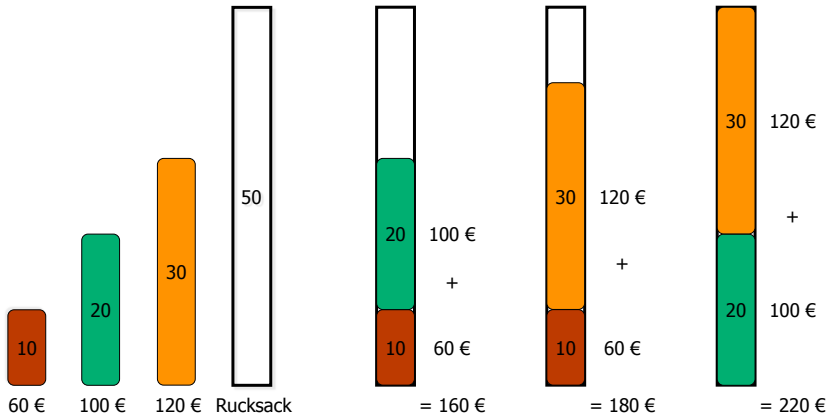
1. Charakterisiere die (rekursive) Struktur der optimalen Lösung (Prinzip der optimalen Substruktur)
2. Definiere den Wert einer optimalen Lösung rekursiv
3. Transformiere die rekursive Methode in eine iterative bottom-up Methode, bei der alle Zwischenergebnisse in einer Tabelle gespeichert werden

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



- Fazit: es ist keine gute Strategie, das Objekt mit dem besten Verhältnis Profit:Gewicht als erstes zu wählen

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

$$\text{maximiere} \quad \sum_{i=1}^{n} v_i x_i \qquad \text{(gesamter Profit)}$$

$$\text{unter der Bedinung} \quad \sum_{i=1}^{n} w_i x_i \leq c \qquad \text{(Gewichtsbedingung)}$$

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