Informatik II
Dynamische Programmierung

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

- Zweite Technik für 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
- Vergleich mit Divide-and-Conquer:
  - Ähnlich: DP berechnet Lsg eines Problems aus Lösungen von Teilproblemen
  - Anders: Teilprobleme werden oft nicht rekursiv gelöst sondern iterativ, beginnend mit den Lösungen der kleinsten Teilprobleme (*bottom-up*)
- Wird häufig angewandt, wenn rekursiver Ansatz immer wieder dieselben Teilprobleme lösen würde

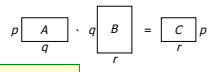
Matrix Chain Multiplication Problem (MCMP)

- Gegeben: Folge (Kette) $A_1, A_2, \ldots, A_n$ von Matrizen mit verschiedenen Dimensionen
- Gesucht: Produkt $A_1 \cdot A_2 \cdot \ldots \cdot A_n$
- Aufgabe:
  Organisiere die Multiplikationen so, daß möglichst wenig skalare Multiplikationen ausgeführt werden. (Man nutzt 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

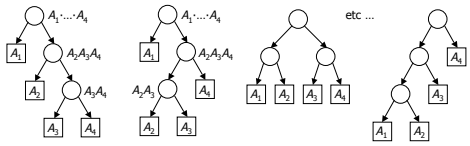
$A'' = (A_2A_3)$	25000 Mult.
A_1A''	50000 Mult.
Summe:	75000 Mult.

Problemstellung

- Gegeben: Folge von Matrizen A_1, A_2, …, A_n und die Dimensionen p_0, p_1, …, p_n, wobei Matrix A_i Dimension $p_{i-1} \times p_i$ hat
- Gesucht: 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
  - 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 Kette $\langle A_1, A_2, A_3, A_4 \rangle$ 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 (siehe 1. Semester)



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:
  $P(n+1) =: C_n$ = n-te Catalan'sche Zahl
- Es gilt:

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

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

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
  $A_{i...j} = (A_{i...k})\,(A_{k+1...j})$, $i \le k < j$
- Behauptung → Klammerung innerhalb $A_{i...k}$ muß ihrerseits optimal sein
- Ann.: die Klammerung von $A_{i...k}$ innerhalb der optimalen Lsg 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 Ann. der Optimalität der ursprünglichen Lsg zu $A_{i...j}$

Rekursive Lösung

- Auf der höchsten Stufen werden 2 Matrizen multipliziert, d.h., für jedes k, $1 \le k \le n$-1, ist $(A_{1...n}) = ((A_{1...k})\,(A_{k+1...n}))$
- 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 bestimmtes k = "Kosten links" + "Kosten rechts" + Kosten für $(A_{i...k}) \cdot (A_{k+1...j})$
- Daraus lässt sich die folgende rekursive Regel ableiten:
  - $m[i,j]$ sei minimale Anzahl von Operationen zur Berechnung des Teilprodukts $A_{i...j}$

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

Naïver rekursiver Algo

- Invariante: mcm_rek(p, i, j) liefert m[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: mcm_rek (p, 1, n)

Laufzeit der naïven rekursiven Lösung

- Sei $T(n)$ die Anzahl der Schritte zur Berechnung von rek-mat-kat für Eingabefolge der Länge n

$$T(1) = 1$$

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

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

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

- exponentielle Laufzeit!