Kapitel 6: Dynamische Programmierung

Prinzip der Dynamischen Programmierung

Beispiel 1: Montagebänder

Beispiel 2: Matrix-Kettenmultiplikation

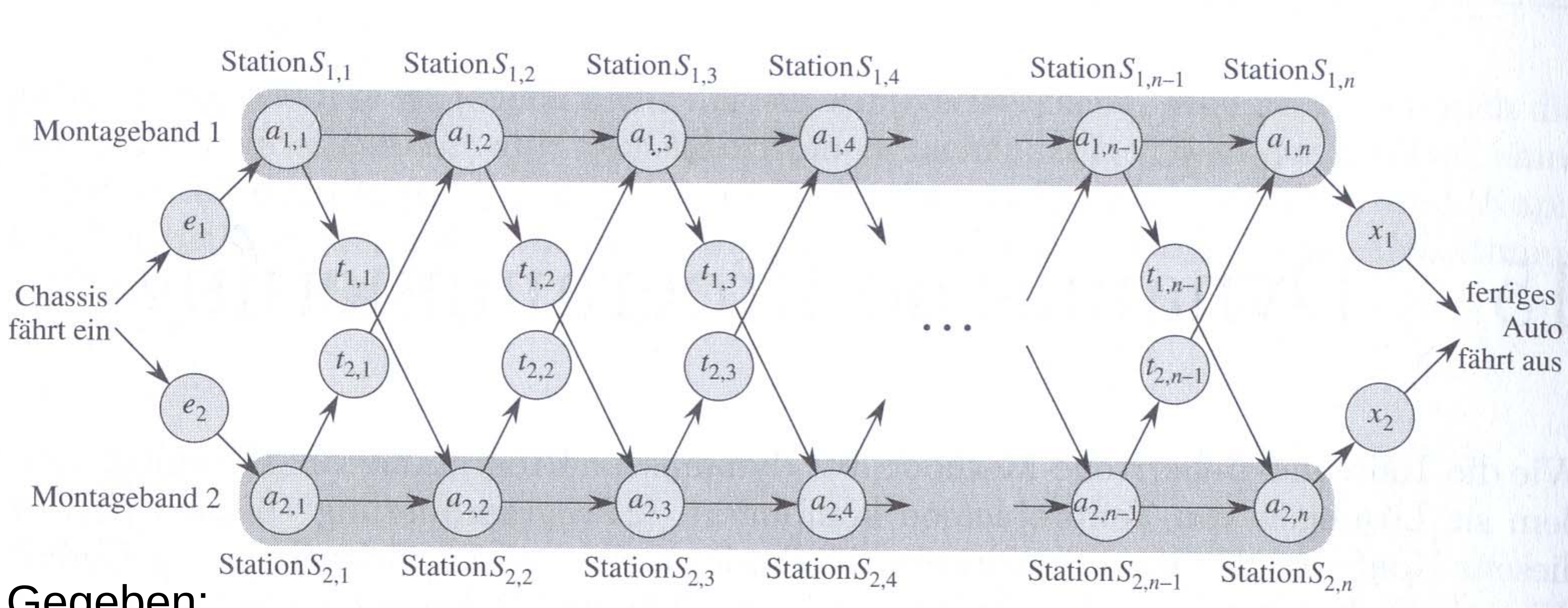
- Häufig verwendete Lösungsstrategie für komplexe Probleme:
  - Zerlege die Eingabe des Problems in ein/mehrere Teilproblem(e)
  - Wende dieses Prinzip rekursiv an
  - Unterschreitet die Eingabegröße einen Grenzwert, kann die Lösung einfach berechnet werden
  - Konstruiere die Lösung des Problems aus der Lösung des Teilproblems / den Lösungen der Teilprobleme
- ➔ Rekursive Algorithmen
- Beispiel: Divide&Conquer-Prinzip (z.B. Mergesort, Quicksort)
  - Teilung in zwei voneinander unabhängige zu lösende Teilprobleme
  - Rekursive Lösung der Teilprobleme
  - Zusammenfügen der Teillösungen zur Gesamtlösung
- Komplexe rekursive Schema können dazu führen, dass Teilprobleme mehrfach im Rekursionsbaum auftreten.
  - Rekursive Implementierung führt zu hoher Laufzeit
  - Berechne die Lösung von Teilproblemen nur einmal und speichere sie in einer Tabelle:

Dynamische Programmierung

Prinzip der Dynamischen Programmierung

- ‚Programmierung' in Dynamischer Programmierung hat historische Gründe und steht für das ‚systematische Füllen von Tabellen', nicht für das Schreiben von Computerprogrammen.
- Entwicklungsschritte in der Dynamischen Programmierung:
  1. **Charakterisiere** die Struktur einer optimalen Lösung
  2. **Definiere** den Wert einer optimalen Lösung rekursiv (Die Umsetzung in einen rekursiven Algorithmus würde zu einem top-down-Ansatz führen)
  3. **Berechne** den Wert einer optimalen Lösung mit einem bottom-up-Ansatz (Speichere dabei die bereits berechneten Teillösungen)
  4. **Konstruiere** eine zugehörige optimale Lösung
- Dynamische Programmierung
  - wird häufig zur Lösung von Optimierungsproblemen eingesetzt.
  - ist ein sehr mächtiges Paradigma im Algorithmenentwurf.
  - sollte bzgl. seiner Anwendbarkeit für ein neues Optimierungsproblem (mittels Durchführung von Schritt 1) getestet werden.

6.2 Beispiel 1: Ablaufkoordination von Montagebändern



Gegeben:

- Zwei Montagebänder mit n Stationen $S_{1,1},\ldots,S_{1,n}$ und $S_{2,1},\ldots,S_{2,n}$
- Montagezeiten für alle Stationen: $a_{1,1},\ldots,a_{1,n}$ und $a_{2,1},\ldots,a_{2,n}$
- Ein- und Ausfahrzeiten e_1, x_1 und e_2, x_2
- Transferzeiten bei Montagebandwechsel $t_{1,1},\ldots t_{1,n-1}$ und $t_{2,1},\ldots,t_{2,n-1}$

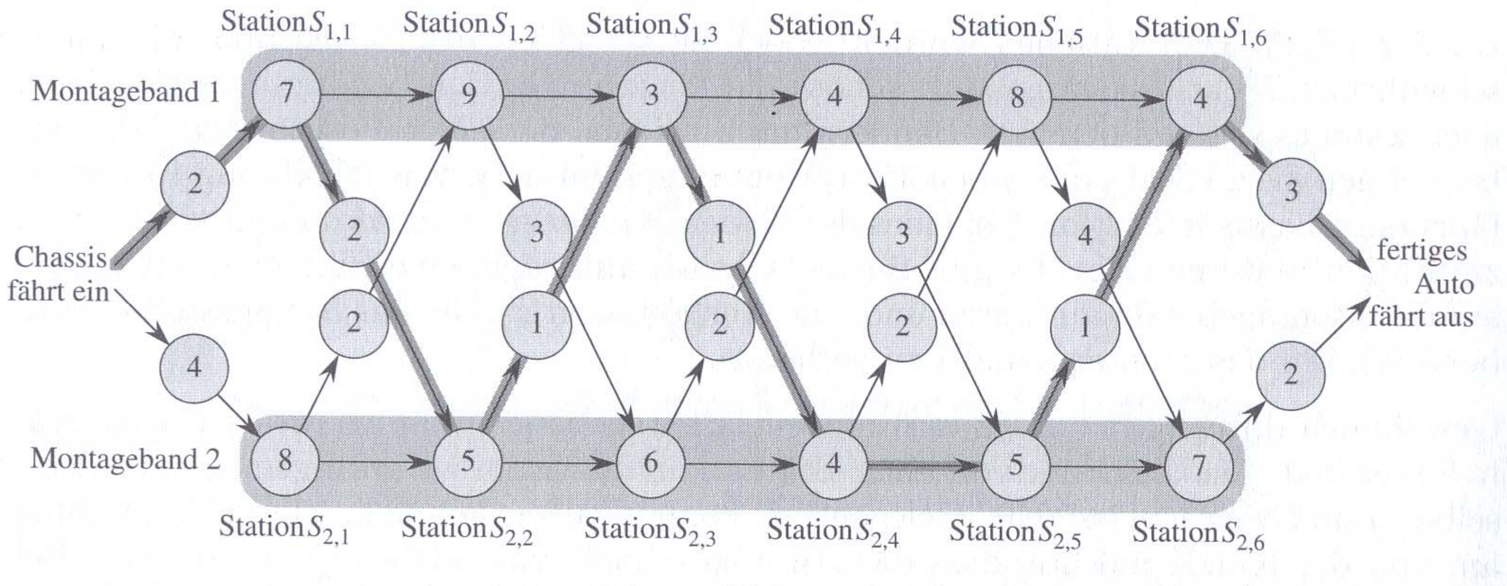
Gesucht: Schnellst mögliche Montage (unter Verwendung beider Bänder)

- **Größe des Lösungsraums:**
  - Für jeden der n Montageschritte können wir uns für eine der beiden Stationen entscheiden

    => es gibt $\Omega(2^N)$ verschiedene Montagefahrten
- **Struktur einer optimalen Lösung:**
  - Lässt sich eine optimale Lösung aus einer optimalen Lösung eines Teilproblems konstruieren/ableiten?
  - Betrachte eine schnellste Montagefahrt bis zur Station $S_{1,j}$:
    - j=1: Chassis fährt über zu Band 1, Zeit: e_1, keine Alternative
    - j>1: Möglichkeit 1: Chassis fährt von $S_{1,j-1}$ direkt zu $S_{1,j}$

      Möglichkeit 2: Chassis wechselt das Band und kommt von $S_{2,j-1}$ zu $S_{1,j}$ und nimmt die Transferzeit $t_{2,j-1}$ in Kauf
  - Schnellste Montagefahrt setzt sich aus optimalen Teilfahrten zusammen:
    - Führt eine schnellste Montagefahrt von $S_{1,j-1}$ zu $S_{1,j}$ (Möglichkeit 1), so ist die Teilfahrt zu $S_{1,j-1}$ ebenfalls eine schnellste Montagefahrt (sonst könnte die schnellste Montagefahrt verkürzt werden).
    - Analog folgt, dass auch die Teilfahrt zu $S_{2,j-1}$ optimal sein muss.

Schritt 1: Charakterisierung der Struktur der schnellsten Montagefahrt



(a)

- Eigenschaft der **optimalen Teilstruktur**:
  - Eine optimale Lösung des Problems beinhaltet optimale Lösungen von Teilproblemen.

  konkret:
  - Eine schnellste Montagefahrt bis zur Station $S_{i,j}$ besteht aus einer der schnellsten Montagefahrten bis zu den Stationen $S_{i,j-1}$.

=> Voraussetzung für die Anwendbarkeit der Dynamischen Programmierung.

Schritt 2: Rekursive Lösung des Problems

- Eigenschaft der optimalen Teilstruktur erlaubt eine rekursive Lösung des Problems:
  - f*: Zeit einer optimalen Montagefahrt
  - $f_i[j]$: optimale Zeit für eine Montagefahrt zur Station $S_{i,j}$ (inkl. Montagezeit $a_{i,j}$)

$$f^* = \min\left(f_1[n] + x_1,\ f_2[n] + x_2\right)$$

$$f_1[1] = e_1 + a_{1,1}$$

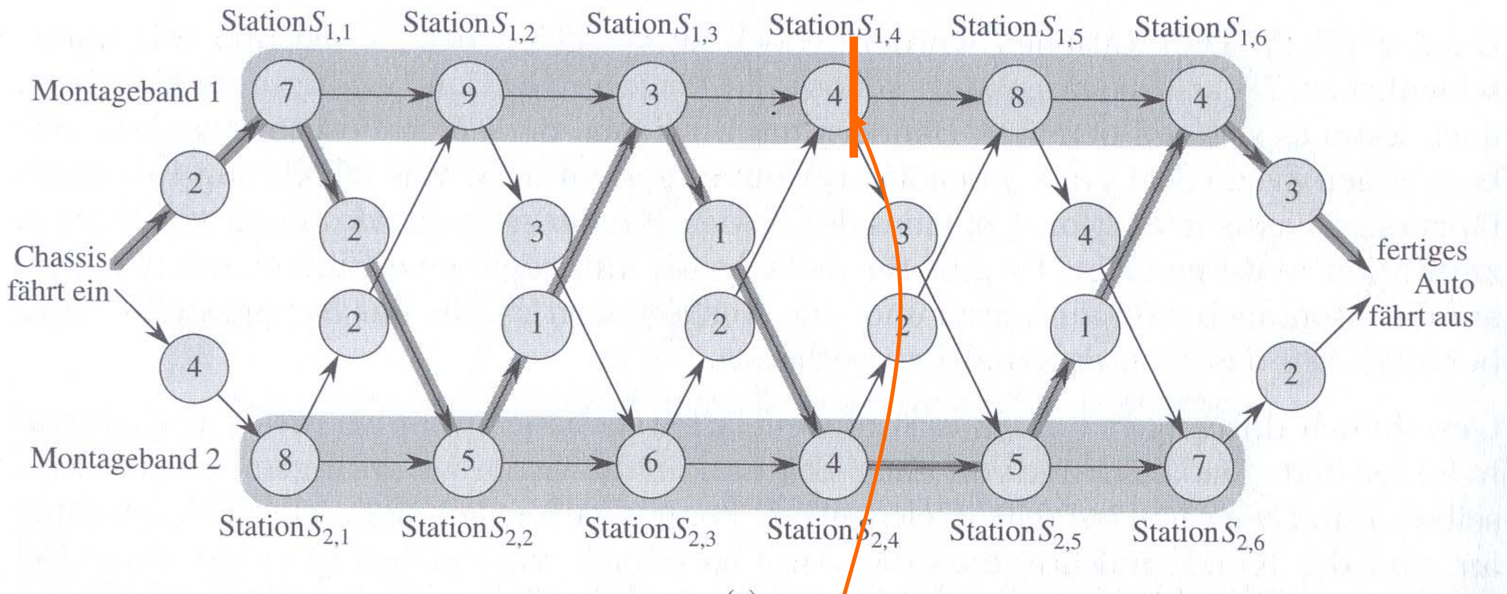
$$f_2[1] = e_2 + a_{2,1}$$

  - für j>1 gibt es die Alternativen, über Station $S_{1,j-1}$ oder $S_{2,j-1}$ zu laufen:

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{falls } j = 1 \\ \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{falls } j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{falls } j = 1 \\ \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{falls } j \geq 2 \end{cases}$$

Schritt 2: Rekursive Lösung des Problems



(a)

j	1	2	3	4	5	6	
$f_1[j]$	9	18	20	24	32	35	$f^* = 38$
$f_2[j]$	12	16	22	25	30	37	

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{falls } j = 1 \\ \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{falls } j \geq 2 \end{cases}$$

Schritt 2: Rekursive Lösung des Problems

- CALL-FASTEST-WAY(a,t,e,x,n)

  return min(RECURSIVE-FASTEST-WAY(a,t,e,1,n) + x_1 ,
  RECURSIVE-FASTEST-WAY(a,t,e,2,n) + x_2)

- RECURSIVE-FASTEST-WAY(a,t,e,i,j)

  // berechnet Zeit $f_i[j]$ einer optimalen Montagefahrt zur Station $S_{i,j}$

  if(j == 1) **return** (e_i + $a_{i,1}$)

  else

  // Möglichkeit 1: kein Bandwechsel

  f_i = RECURSIVE-FASTEST-WAY(a,t,e,i,j-1) +$a_{i,j}$

  // Möglichkeit 2: Bandwechsel von Band (3-i) auf Band i

  f_{3-i} = RECURSIVE-FASTEST-WAY(a,t,e,3-i,j-1) +$t_{3-i,j-1}$ +$a_{i,j}$

  return min(f_i, f_{3-i})

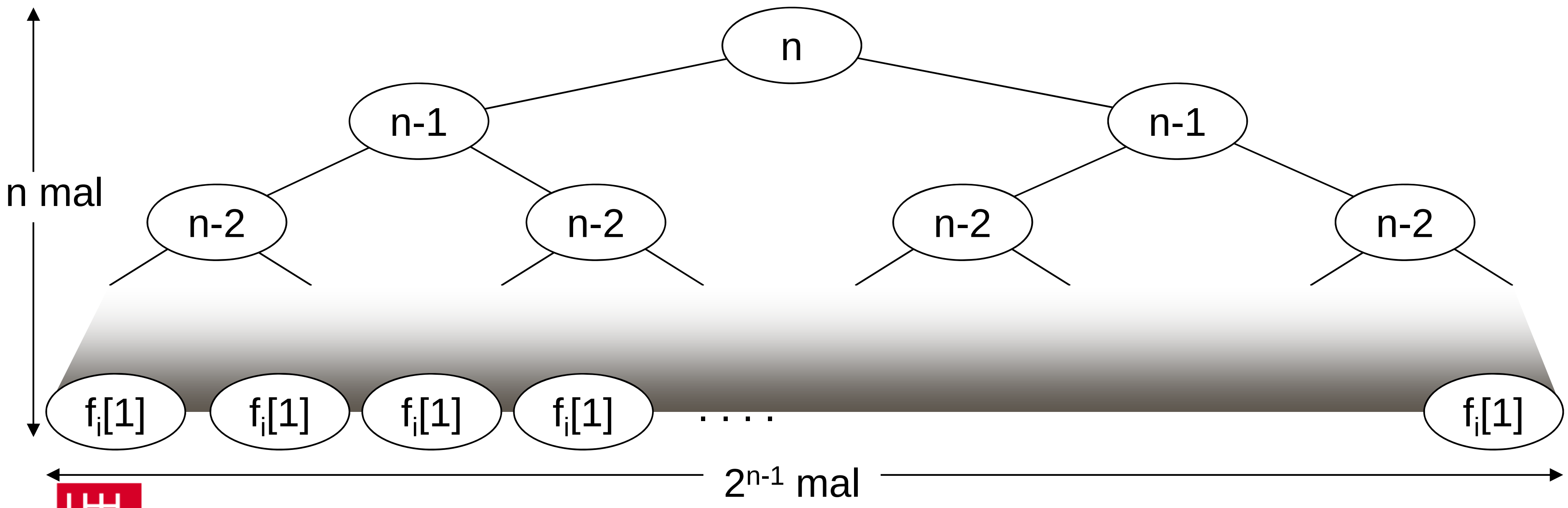
$$T_{RFW}(n) = \begin{cases} c & : \quad n = 1 \\ 2T_{RFW}(n-1) + c & : \quad n > 1 \end{cases}$$

Schritt 2: Rekursive Lösung des Problems

$$T_{RFW}(n) = \begin{cases} c & : \quad n = 1 \\ 2T_{RFW}(n-1) + c & : \quad n > 1 \end{cases}$$

$$T_{RFW}(n) = \underbrace{2(2(\cdots 2(2}_{n-1\,\text{mal}} T_{RFW}(1) + \underbrace{c) + \cdots c) + c}_{n-1\,\text{mal}}$$

$$= 2^{n-1}c + \sum_{i=0}^{n-1} 2^i c = (2^{n-1} + 2^n - 1)c = \left(\frac{3}{2}2^n - 1\right)c = \Theta\left(2^n\right)$$



Schritt 3: Berechne den Wert der optimalen Lösung

- Lösung:
  - Berechnung der Funktionswerte $f_i[j]$ bottom-up (Reihenfolge j=1,2, …, n)
  - Speicherung der Resultate in den Arrays $f_1[]$ und $f_2[]$

- TIME-OF-FASTEST-WAY(a,t,e,x,n)

$f_1[1] = e_1 + a_{1,1}$

$f_2[1] = e_2 + a_{2,1}$

$\textbf{for } j = 2 \textbf{ to } n$

$\quad f_1[j] = \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j})$

$\quad f_2[j] = \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j})$

$\textbf{return } \min(f_1[n] + x_1,\ f_2[n] + x_2)$

- Laufzeit: $T_{\text{TIME-OF-FASTEST-WAY}}(n) = \Theta(n)$
- Speicherbedarf: $S_{\text{TIME-OF-FASTEST-WAY}}(n) = \Theta(n)$

Schritt 3: Berechne den Wert der optimalen Lösung

- FASTEST-WAY(a,t,e,x,n)

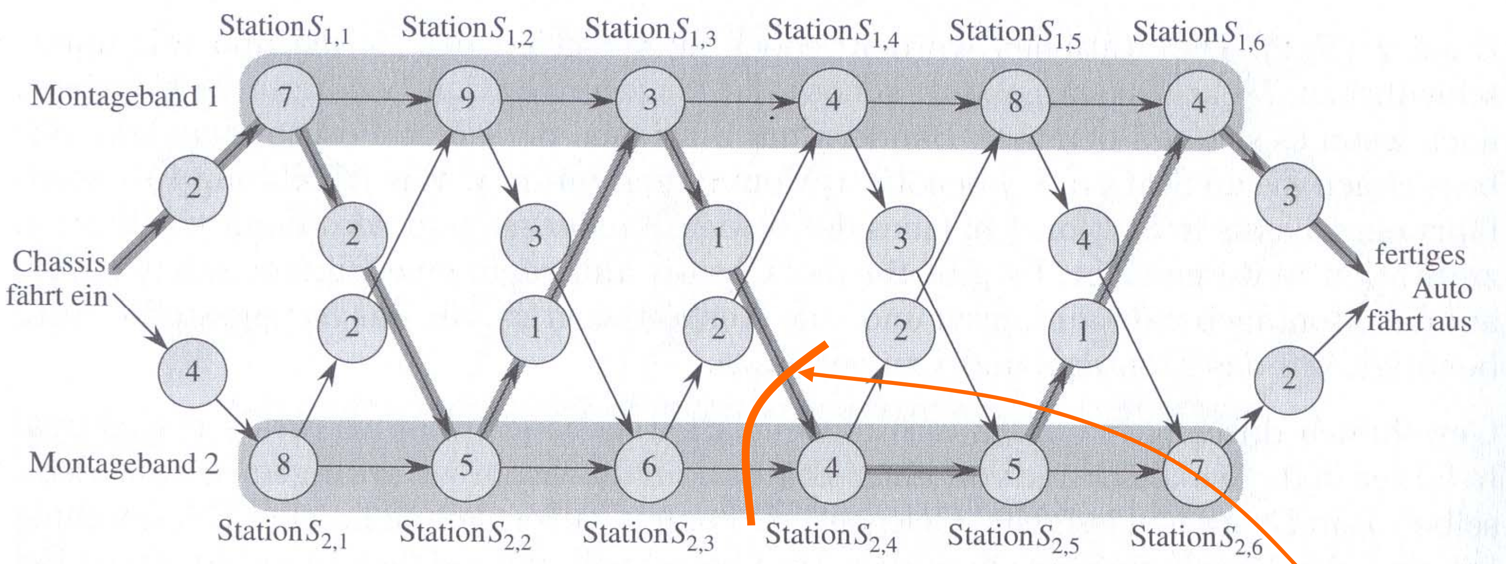
```
f_1[1] = e_1 + a_1,1; f_2[1] = e_2 + a_2,1
for( j = 2 to n )
  if( f_1[j-1] +a_1,j ≤ f_2[j-1] +t_2,j-1 +a_1,j )
       f_1[j] = f_1[j-1] + a_1,j
       l_1[j] = 1
  else f_1[j] = f_2[j-1] + t_2,j-1 + a_1,j
       l_1[j]´= 2
  if( f_2[j-1] +a_2,j ≤ f_1[j-1] +t_1,j-1 +a_2,j )
       f_2[j] = f_2[j-1] + a_2,j
       l_2[j] = 2
  else f_2[j] = f_1[j-1] + t_1,j-1 + a_2,j
       l_2[j] = 1
if( f_1[n] + x_1 ≤ f_2[n] + x_2 )
  f* = f_1[n] + x_1
  l* = 1
else f* = f_2[n] + x_2
  l* = 2
return( f*, l* )
```

Berechnung von $f_1[]$ und $l_1[]$

Berechnung von $f_2[]$ und $l_2[]$

Berechnung von f* und l*

Schritt 4: Konstruktion einer schnellsten Montagefahrt



(a)

- Funktion TIME-OF-FASTEST-WAY() liefert bereits die Montagezeit, allerdings nicht die zugehörige Fahrt (d.h. Auswahl der Stationen).

j	2	3	4	5	6	
$l_1[j]$	1	2	1	1	2	$l^* = 1$
$l_2[j]$	1	2	1	2	2	

➔ Speicherung der Montageband-Nummer (1 oder 2), die zur kürzesten Fahrt geführt hat: $l_i[j]$ und l^*:

$l_i[j]$: Nummer des Bandes, dessen Station (j-1) auf der Fahrt mit Zeit $f_i[j]$ zu Station $S_{i,j}$ verwendet wurde, j=2,…,n

l^* : Nummer des Bandes, dessen Station n verwendet wurde

Schritt 4: Konstruktion einer schnellsten Montagefahrt

- FASTEST-WAY(a,t,e,x,n)

```
f_1[1] = e_1 + a_{1,1}; f_2[1] = e_2 + a_{2,1}
for( j = 2 to n )
  if( f_1[j-1] +a_{1,j} ≤ f_2[j-1] +t_{2,j-1} +a_{1,j} )
        f_1[j] = f_1[j-1] + a_{1,j}
        l_1[j] = 1
  else  f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}
        l_1[j] = 2
  if( f_2[j-1] +a_{2,j} ≤ f_1[j-1] +t_{1,j-1} +a_{2,j} )
        f_2[j] = f_2[j-1] + a_{2,j}
        l_2[j] = 2
  else  f_2[j] = f_1[j-1] + t_{1,j-1} + a_{2,j}
        l_2[j] = 1
if( f_1[n] + x_1 ≤ f_2[n] + x_2 )
  f* = f_1[n] + x_1
  l* = 1
else f* = f_2[n] + x_2
  l* = 2
return( f*, l* )
```

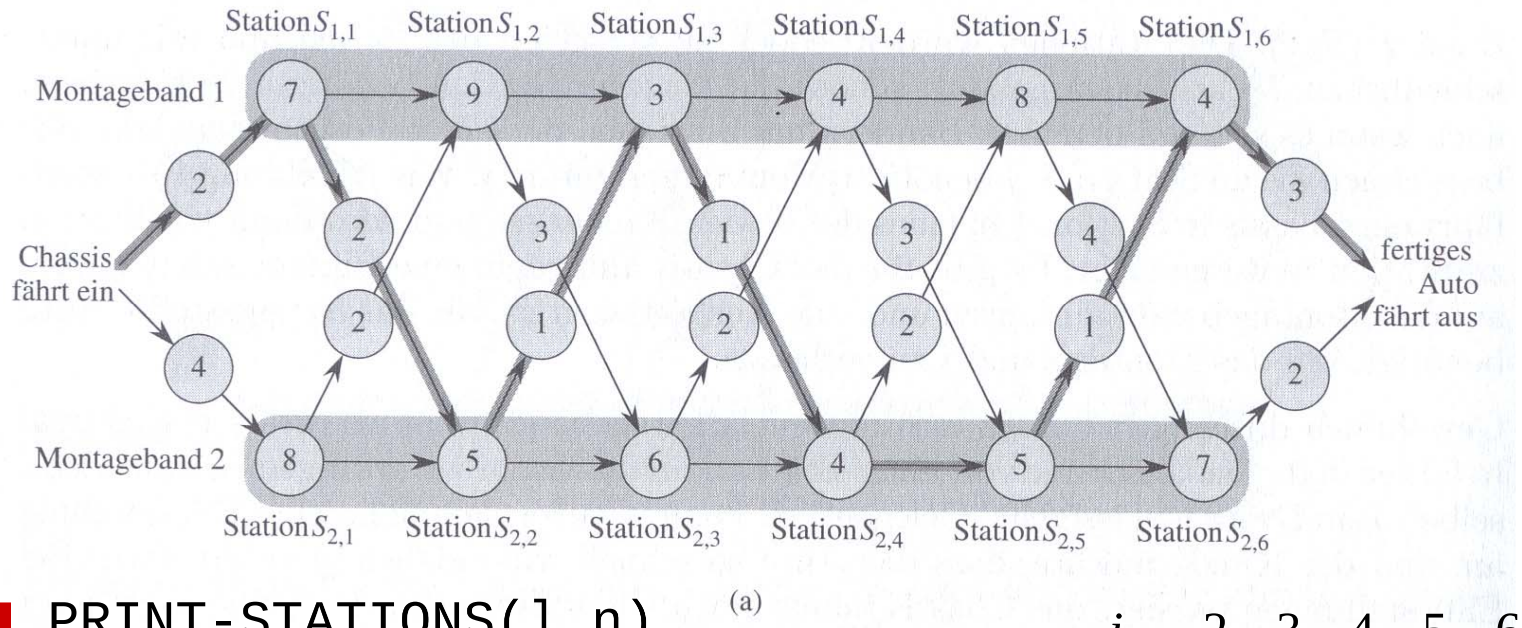
Berechnung von f_1[] und l_1[]

Berechnung von f_2[] und l_2[]

Berechnung von f* und l*

Schritt 4: Konstruktion einer schnellsten Montagefahrt

- Beginnend mit l* lässt sich die schnellste Montagefahrt rekonstruieren:



(a)

- PRINT-STATIONS(l,n)

```
i = l*
write "Band" i ", Station" n
for( j = n downto 2 )
  i = li[j]
  write "Band" i ", Station" j-1
```

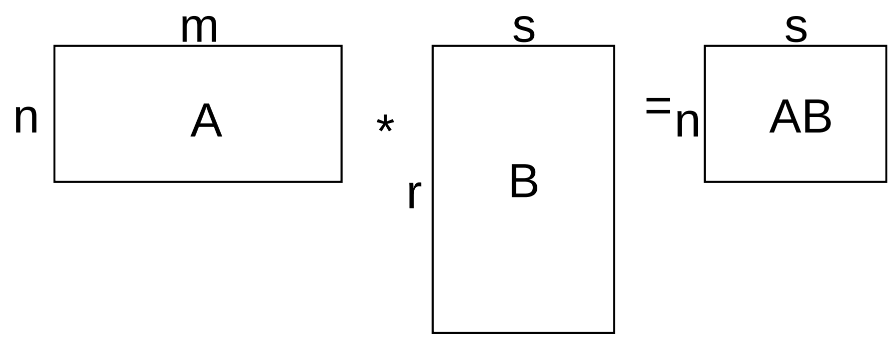
j	2	3	4	5	6	
$l_1[j]$	1	2	1	1	2	$l^* = 1$
$l_2[j]$	1	2	1	2	2	

6.3 Beispiel 2: Matrix-Kettenmultiplikation

$$A = [a_{ij}]_{1 \le i \le n, 1 \le j \le m}$$

$$B = [b_{ij}]_{1 \le i \le r, 1 \le j \le s}$$

$$AB = \begin{cases} \left[\sum_{k=1}^{m} a_{ik} b_{kj} \right]_{1 \le i \le n, 1 \le j \le s} & : \text{falls } m = r \\ \text{n.d.} & : \text{sonst} \end{cases}$$



- **Assoziativität** der Matrixmultiplikation:
  A (B C) = (A B) C

- A.s : Anzahl Spalten der Matrix A
  A.z : Anzahl Zeilen der Matrix A

  Laufzeit hängt von den Dimensionen der Matrizen ab:
  $T_{\text{MATRIX-MULTIPLY}}(A,B) = O(A.z\ A.s\ B.s)$
  $= O(A.z\ B.z\ B.s)$

- MATRIX-MULTIPLY(A,B)

```
if( A.s ≠ B.z )
  error "inkompatible Matrizen"
else
  for( i = 1 to A.z )
    for( j = 1 to B.s )
      C[i,j] = 0
      for( k = 1 to A.s )
        C[i,j] += A[i,k]*B[k,j]
  return C
```

Beispiel 2: Matrix-Kettenmultiplikation

- Problem der **Matrix-Kettenmultiplikation:**

  geg: Sequenz von Matrizen <A_1, A_2, …, A_n>

  ges: Reihenfolge der Matrixmultiplikation (vollständige Klammerung), die die Anzahl skalarer Multiplikationen minimiert. (Die Berechnung des Produkts A_1A_2*…*A_n wird nicht als Teil des Problems betrachtet.)

- Beispiel:
  - A_1: 10 x 100 Matrix A_2: 100 x 5 Matrix A_3: 5 x 50 Matrix
  - A_1A_2 ist eine 10 x 5 Matrix

    A_2A_3 ist eine 100 x 50 Matrix
  - ((A_1 A_2) A_3) : # Multiplikationen 10*100*5 + 10*5*50 = 7.500

    (A_1 (A_2 A_3)) : # Multiplikationen 100*5*50 + 10*100*50 = 75.000

  ((A_1 A_2) A_3) kann 10* schneller berechnet werden als (A_1 (A_2 A_3))

Schritt 1: Struktur der optimalen Klammerung

- Größe des Lösungsraums P(n)

  Wie viele verschiedene Klammerungen P(n) gibt es bei der Multiplikation von n Matrizen?

  n=1 : P(n) = 1

  n=2 : P(n) = 1

  n=3 : P(n) = 2

  n=4 : P(n) = 5

  (A (B (C D)))
  (A ((B C) D))
  ((A B) (C D))
  ((A (B C)) D)
  (((A B) C) D)

  n>1 : Es gibt n-1 Möglichkeiten für die letzte auszuführende Multiplikation
  Liegt diese zwischen k und k+1, so gibt es P(k) Möglichkeiten für die Klammerung des ersten, P(n-k) Möglichkeiten für den zweiten Faktor.

$$P(n) = \begin{cases} 1 & \text{falls } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{falls } n \geq 2 \end{cases} \qquad P(n) = \Omega\left(\frac{4^n}{n^{2/3}}\right)$$

  Catalan-Zahlen, wachsen exponentiell mit n

Schritt 1: Struktur der optimalen Klammerung

- Wie kann eine optimale Lösung aus optimalen Teillösungen konstruiert werden?
  - Sei $A_{i..j} = A_i A_{i+1} \ldots A_j$ das Produkt der Matrizen A_i bis A_j
  - Die Matrix A_i sei eine p_{i-1} x p_i - Matrix
  - Für i<j gibt es eine Position k der zuletzt ausgeführten Matrix-Multiplikation:

    $A_{i..j} = (A_i \ldots A_k) (A_{k+1} \ldots A_j)$
  - Für die zuletzt ausgeführte Matrix-Multiplikation (an Position k) werden $p_{i-1}\, p_k\, p_j$ skalare Multiplikationen benötigt. Diese Zahl ist unabhängig davon, wie $A_{i..k}$ und $A_{k+1..j}$ berechnet werden.
  - Die optimale Anzahl skalarer Multiplikationen ist die Summe über die jeweils optimale Anzahl zur Berechnung von $A_{i..k}$ und $A_{k+1..j}$ und $p_{i-1}\, p_k\, p_j$

  ➔ Die Lösung des Matrix-Kettenmultiplikationsproblems erfüllt die Eigenschaft der optimalen Teilstruktur.

Schritt 2: Rekursive Lösung des Matrix-Kettenmultiplikationsproblems

- Rekursive Beschreibung:
  - Sei m[i,j] die minimale Anzahl skalarer Multiplikationen zur Berechnung von $A_{i..j}$. Liegt die letzte auszuführende Multiplikation zwischen k und k+1, gilt:

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

  - m[i,j] lässt sich durch Minimierung über alle möglichen Werte k bestimmen:

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

- `RECURSIVE-MATRIX-CHAIN(p, i, j)` // Erster Aufruf mit i=1, j=n

```
if( i == j ) return 0
else
  m = ∞
  for( k = i to j-1 )
    m = min( m, RECURSIVE-MATRIX-CHAIN(p, i, k) +
            RECURSIVE-MATRIX-CHAIN(p,k+1,j) + p[i-1]*p[k]*p[j])
    return m
```

Schritt 3: Berechnung der minimalen Anzahl Multiplikationen

- Die Laufzeit von RECURSIVE-MATRIX-CHAIN ist exponentiell:

$$T_{RMCO}(n) = \begin{cases} c & : \quad n = 1 \\ \sum_{k=1}^{n-1} \left(T(k) + T(n-k) + c \right) + c & : \quad n > 1 \end{cases} \qquad T_{RMCO}(n) = \Omega(2^n)$$

  (Beweis: Zeige $T_{RMCO}(n) \geq 2^{n-1}$)
- Überlappende Teilprobleme:
  - Es gilt $1 \leq i \leq j \leq n$, somit gibt es n(n+1)/2 verschiedene Teilprobleme

    => in der Berechnung von RECURSIVE-MATRIX-CHAIN gibt es überlappende Teilprobleme
  - Speichere das Resultat für die Eingabe (p, i, j) in einer n x n - Matrix m an Position m[i,j]
- Bottom-up Berechnung
  - Zur Berechnung von m[i,j] werden nur Matrixwerte m[u,v] verwendet mit kleinerer Kettenlänge, also v - u +1 < j - i +1
  - Berechne die Matrix mit steigenden (j - i +1)-Werten

Schritt 3: Berechnung der minimalen Anzahl Multiplikationen

- MATRIX-CHAIN-ORDER(p)

```
n = p.length-1
for( i = 1 to n ) m[i,i] = 0   // Initialisierung trivialer m[]-Werte
for( l = 2 to n )              // Berechnung erfolgt in der Reihenfolge
                               //   wachsender Kettenlängen l:
  for( i = 1 to n-l+1 )        // Iteriere durch alle Paare (i,j) mit Länge
    j = i+l-1
    m[i,j] = ∞
    for( k = i to j-1 )        // Minimiere über alle möglichen k-Werte
      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
return m, s                    // der gesuchte Wert steht in m[1,n]
```

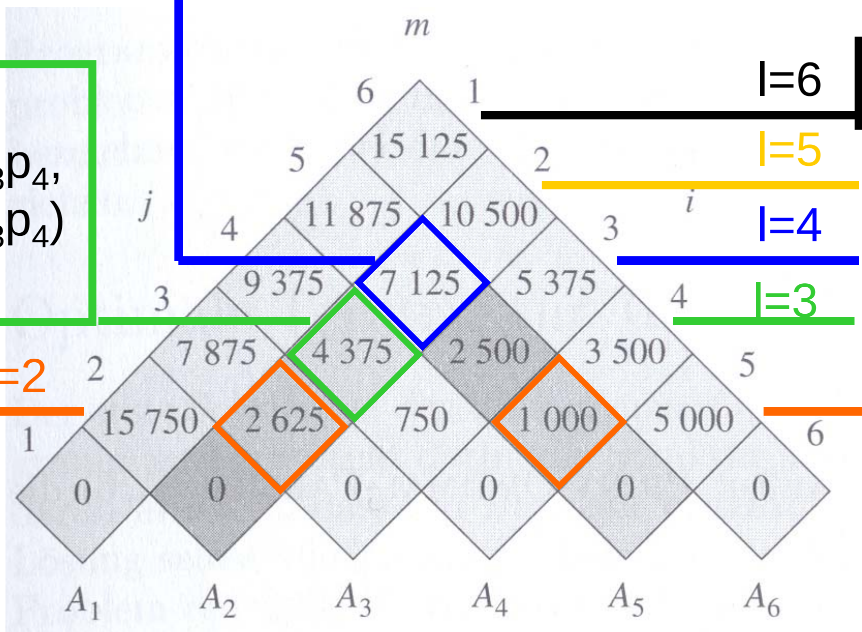
- Laufzeit: $T_{\text{MATRIX-CHAIN-ORDER}}(n) = O(n^3)$
- Speicherbedarf: $S_{\text{MATRIX-CHAIN-ORDER}}(n) = \Theta(n^2)$

Ein Beispiel für n=6 (Teil 1)

$$m[2,5] = \min\begin{cases} m[2,2]+m[3,5]+p_1p_2p_5 = 0+2500+35\cdot 15\cdot 20 = 13000 \\ m[2,3]+m[4,5]+p_1p_3p_5 = 2625+1000+35\cdot 5\cdot 20 = 7125 \\ m[2,4]+m[5,5]+p_1p_4p_5 = 4375+0+35\cdot 10\cdot 20 = 11375 \end{cases} = 7125$$

m[2,4] = min (
m[2,2] + m[3,4] + $p_1p_3p_4$,
m[2,3] + m[4,4] + $p_2p_3p_4$)
= 4375

m[2,3] = p_1 * p_2 * p_3
= 35 * 15 * 5
= 2625



l=6 m[1,6] = 15125

l=5

l=4

l=3

l=2

m[4,5] = p_3 * p_4 * p_5
= 5 * 10 * 20
= 1000

p[i]: 30 x 35, 35 x 15, 15 x 5, 5 x 10, 10 x 20, 20 x 25

Schritt 4: Konstruktion einer optimalen vollständigen Klammerung

- Sei s[i,j] der k-Wert, der die optimale Lösung für das Teilprodukt $A_{i..j}$ liefert, d.h. $A_{i..j}$ soll durch $(A_{i..k})(A_{k+1..j})$ berechnet werden

- MATRIX-CHAIN-ORDER(p)

```
n = p.length-1
for( i = 1 to n ) m[i,i] = 0   // Initialisierung trivialer m[]-Werte
for( l = 2 to n )              // Berechnung erfolgt in der Reihenfolge
                               //   wachsender Kettenlängen l:
  for( i = 1 to n-l+1 )        // Iteriere durch alle Paare (i,j) mit Länge
    j = i+l-1
    m[i,j] = ∞
    for( k = i to j-1 )        // Minimiere über alle möglichen k-Werte
      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
return m, s                    // der gesuchte Wert steht in m[1,n]
```

Schritt 4: Konstruktion einer optimalen vollständigen Klammerung

- Bestimmung einer optimalen vollständigen Klammerung mittels s[i,j]:
  - Berechnung von $A_{1..n} = (A_{1..s[1,n]})\,(A_{s[1,n]+1..n})$

$A_{1..n}$

k=s[1,n]

$A_{1..s[1,n]}$ $A_{s[1,n]+1..n}$

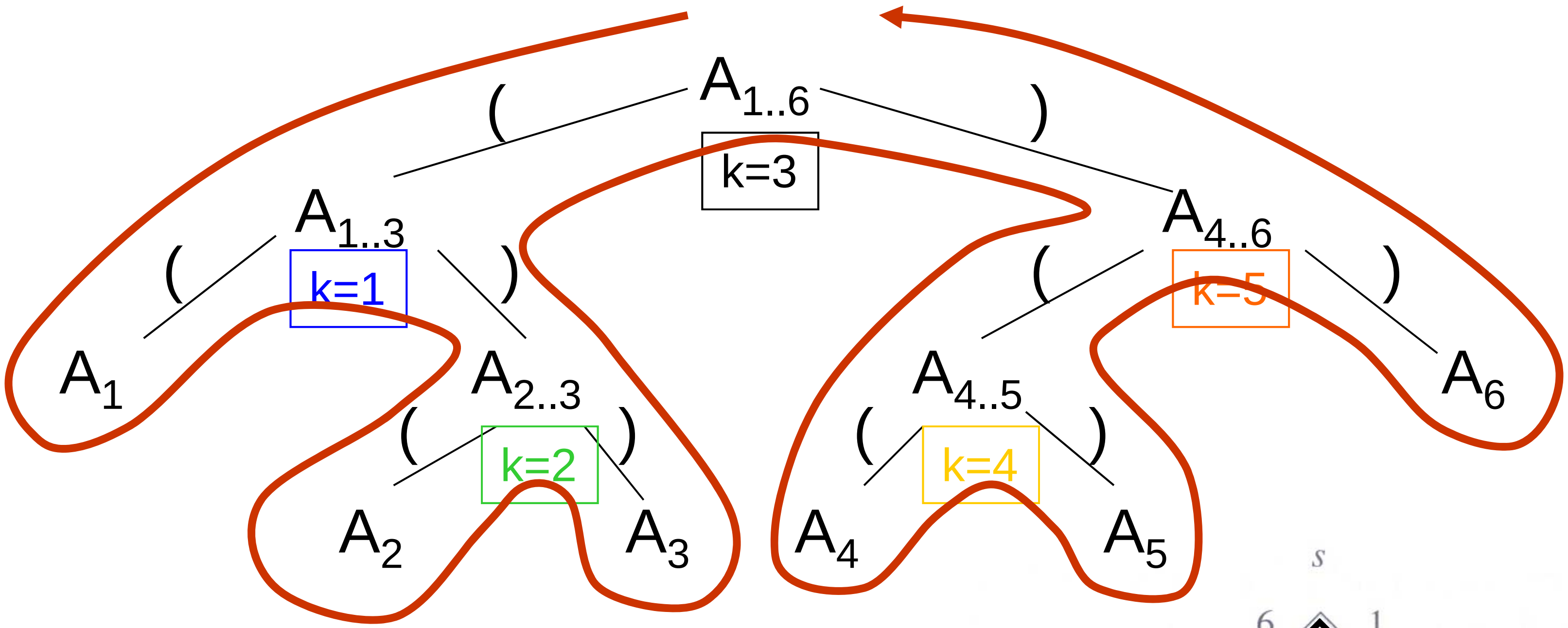
k=s[1,s[1,n]] k=s[s[1,n]+1,n]

$A_{1..s[1,s[1,n]]}$ $A_{s[1,s[1,n]]+1..s[1,n]}$ $A_{s[1,n]+1..s[s[1,n]+1,n]}$ $A_{s[s[1,n]+1,n]+1..n}$

- Inorder-Traversal des durch s[i,j] definierten Baumes:
- PRINT-OPTIMAL-PARENTHESIS(s,i,j)

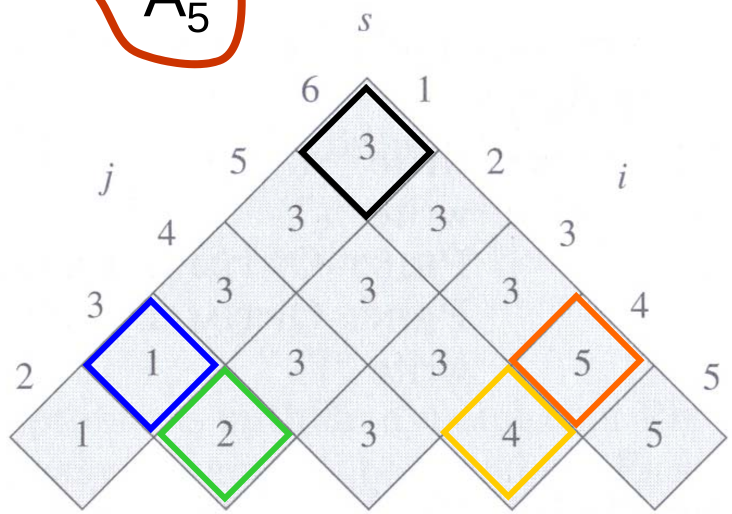
```
if( i == j )
  write "A"_i
else write "("
      PRINT-OPTIMAL-PARENTHESIS(s,i,s[i,j])
      PRINT-OPTIMAL-PARENTHESIS(s,s[i,j]+1,j)
      write ")"
```

Ein Beispiel für n=6 (Teil 2)



Optimale vollständige Klammerung:

$((A_1(A_2A_3))((A_4A_5)A_6))$



Abschließende Bemerkungen zur Dynamischen Programmierung

- Dynamische Programmierung erlaubt die effiziente Lösung von Optimierungsproblemen mit zwei Eigenschaften:
  1. **Optimale Teilstruktur**: Optimale Lösung kann aus optimalen Teillösungen konstruiert werden. Die optimalen Teillösungen können **unabhängig** voneinander bestimmt werden.
  2. **Überlappende Teilprobleme**: Es gibt eine (polynomielle) Anzahl von Teilproblemen, deren Lösungen immer wieder zur Lösung größerer Teilprobleme herangezogen werden.
- Entwicklung von Algorithmen nach dem Prinzip der Dynamischen Programmierung:
  1. **Struktur der optimalen Lösung** bestimmen (optimale Teilstruktur nachweisen)
  2. Eine **rekursive Lösung** entwickeln (Top-Down-Berechnung)
  3. Berechnung der optimalen Kosten durch Umkehr der Berechnungsreihenfolge (**Bottom-Up-Berechnung** + Memoisation)
  4. Über die optimalen Kosten die **optimale Lösung rekonstruieren**