

# Proseminar

## Effiziente Algorithmen

### Kapitel 9: Divide & Conquer und Dynamische Programmierung

Prof. Dr. Christian Scheideler  
WS 2017

## Generische Optimierungsverfahren:

- Systematische Suche
  - lass nichts aus
- Divide and Conquer
  - löse das Ganze in Teilen
- Dynamische Programmierung
  - mache nie etwas zweimal
- Greedy Verfahren
  - schau niemals zurück
- Lokale Suche
  - denke global, handle lokal

# Systematische Suche

---

**Prinzip:** durchsuche **gesamten** Lösungsraum

Auch bekannt als „Brute Force“

**Vorteil:** sehr einfach zu implementieren

**Nachteil:** sehr zeitaufwendig und sollte daher nur für kleine Instanzen verwendet werden

# Systematische Suche

---

## Beispiele:

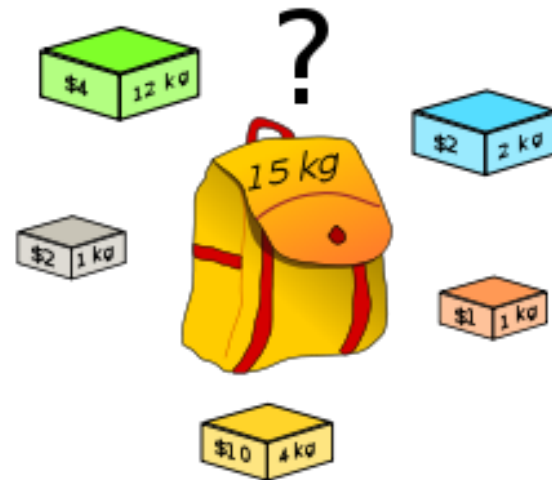
- Suche in unsortierter Liste
- Suche über Broadcasting in unstrukturierten verteilten Systemen (Peer-to-Peer Systeme)
- Rucksackproblem (siehe nächste Folie)

# Systematische Suche

---

## Rucksackproblem:

- **Eingabe:**  $n$  Objekte mit Gewichten  $w_1, \dots, w_n$  und Werten  $v_1, \dots, v_n$  und Rucksack mit Kapazität  $W$
- **Ausgabe:** Objektmenge  $M$  maximalen Wertes, die in Rucksack passt



# Systematische Suche

---

Lösung zum Rucksackproblem:

Probiere **alle Teilmengen** von Objekten aus und merke die Menge **M** von Objekten mit  $\sum_{i \in M} w_i \leq W$ , die bisher den maximalen Wert hatte

**Aufwand:**  $O(2^n)$ , da es  $2^n$  Möglichkeiten gibt, Teilmengen aus einer **n**-elementigen Menge zu bilden.

## Generische Optimierungsverfahren:

- Systematische Suche
    - lass nichts aus
  - Divide and Conquer
  - Dynamische Programmierung
- } DuA,  
Kap. 18 und 20
- Greedy Verfahren
    - schau niemals zurück
  - Lokale Suche
    - denke global, handle lokal

# 18. Divide & Conquer

---

## Teile & Herrsche:

- Problem in Teilprobleme aufteilen
- Teilprobleme rekursiv lösen
- Lösung aus Teillösungen zusammensetzen

## Probleme:

- Wie setzt man zusammen?  
[erfordert algorithmisches Geschick und Übung]
- Laufzeitanalyse (Auflösen der Rekursion)  
[ist normalerweise nach Standardschema; erfordert ebenfalls Übung]



# Divide & Conquer

---

## Beispiele:

- Mergesort
- Quicksort
- Binary Search
- Arithmische Operationen wie Multiplikation großer Zahlen oder **Matrixmultiplikation**
- **Selektion**
- **Nächstes-Paar-Problem**

# Matrix Multiplikation

---

$$\begin{pmatrix} 3 & 7 & 5 & 4 \\ 0 & 3 & 2 & 4 \\ 10 & 2 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 2 & 1 & 1 & 0 \\ 0 & 0 & 1 & 3 \\ 3 & 1 & 1 & 0 \\ 2 & 3 & 2 & 2 \end{pmatrix} = \begin{pmatrix} & & & \\ & & & \\ & & & \\ & & & \end{pmatrix}$$

# Matrix Multiplikation

---

$$A=(a_{ij})_{1 \leq i,j \leq n}$$

$$B=(b_{ij})_{1 \leq i,j \leq n}$$

$$C=(c_{ij})_{1 \leq i,j \leq n}$$

$$\begin{pmatrix} 3 & 7 & 5 & 4 \\ 0 & 3 & 2 & 4 \\ 10 & 2 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 2 & 1 & 1 & 0 \\ 0 & 0 & 1 & 3 \\ 3 & 1 & 1 & 0 \\ 2 & 3 & 2 & 2 \end{pmatrix} = \begin{pmatrix} 29 & 20 & 23 & 29 \\ 14 & 14 & \dots & \\ \dots & & & \\ \dots & & & \end{pmatrix}$$

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

# Matrix Multiplikation

---

## Teile und Herrsche:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \times \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

## Aufwand:

- 8 Multiplikationen von  $n/2 \times n/2$  Matrizen
- 4 Additionen von  $n/2 \times n/2$  Matrizen

# Matrix Multiplikation

---

## Teile und Herrsche:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \times \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

## Aufwand:

- 8 Multiplikationen von  $n/2 \times n/2$  Matrizen
- 4 Additionen von  $n/2 \times n/2$  Matrizen

## Laufzeit:

- $T(n) = 8 \cdot T(n/2) + \Theta(n^2)$

# Matrix Multiplikation

---

## Laufzeit:

- $T(n) = 8 \cdot T(n/2) + k \cdot n^2$   
           $\uparrow$            $\uparrow$            $\uparrow$   
          a          b          f(n)

## Master Theorem:

- $f(n) = k \cdot n^2$
- $a=8, b=2$

# Matrix Multiplikation

---

## Laufzeit:

- $T(n) = 8 \cdot T(n/2) + k \cdot n^2$   
           $\uparrow$            $\uparrow$            $\uparrow$   
          a          b          f(n)

## Master Theorem:

- $f(n) = k \cdot n^2$
- $a=8, b=2$
- **Fall 1:** Laufzeit  $\Theta(n^{\log_b a}) = \Theta(n^3)$

# Matrix Multiplikation

---

## Teile und Herrsche (Algorithmus von Strassen):

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \times \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

### Trick:

$$\begin{aligned} P_1 &= A \cdot (F - H) & P_5 &= (A + D) \cdot (E + H) \\ P_2 &= (A + B) \cdot H & P_6 &= (B - D) \cdot (G + H) \\ P_3 &= (C + D) \cdot E & P_7 &= (A - C) \cdot (E + F) \\ P_4 &= D \cdot (G - E) \end{aligned}$$



# Matrix Multiplikation

---

## Teile und Herrsche (Algorithmus von Strassen):

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \times \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

### Trick:

$$P_1 = A \cdot (F - H)$$

$$P_5 = (A + D) \cdot (E + H)$$

$$P_2 = (A + B) \cdot H$$

$$P_6 = (B - D) \cdot (G + H)$$

$$P_3 = (C + D) \cdot E$$

$$P_7 = (A - C) \cdot (E + F)$$

$$P_4 = D \cdot (G - E)$$

$$AE + BG = P_5 + P_4 - P_2 + P_6$$

$$AF + BH = P_1 + P_2$$

$$CE + DG = P_3 + P_4$$

$$CF + DH = P_5 + P_1 - P_3 - P_7$$

# Matrix Multiplikation

## Teile und Herrsche:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \times \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

## Trick:

$$P_1 = A \cdot (F - H)$$

$$P_5 = (A + D) \cdot (E + H)$$

$$P_2 = (A + B) \cdot H$$

$$P_6 = (B - D) \cdot (G + H)$$

$$P_3 = (C + D) \cdot E$$

$$P_7 = (A - C) \cdot (E + F)$$

$$P_4 = D \cdot (G - E)$$

$$AE + BG = P_5 + P_4 - P_2 + P_6$$

$$AF + BH = P_1 + P_2$$

$$CE + DG = P_3 + P_4$$

$$CF + DH = P_5 + P_1 - P_3 - P_7$$

7 Multiplikationen!

## Laufzeit:

- $T(n) = 7 \cdot T(n/2) + k \cdot n^2$   

The diagram shows the recurrence relation  $T(n) = 7 \cdot T(n/2) + k \cdot n^2$ . Below the equation, three labels are placed: 'a' under the coefficient 7, 'b' under the recursive term  $T(n/2)$ , and 'f(n)' under the non-recursive term  $k \cdot n^2$ . Arrows point from each label to its corresponding part of the equation.

## Master Theorem:

- $f(n) = k \cdot n^2$

# Matrix Multiplikation

---

## Laufzeit:

- $T(n) = 7 \cdot T(n/2) + k \cdot n^2$   
           $\uparrow$            $\uparrow$            $\uparrow$   
          a          b          f(n)

## Master Theorem:

- $f(n) = k \cdot n^2$
- $a=7, b=2$
- Laufzeit  $\Theta(n^{\log_b a}) = \Theta(n^{\log_2 7})$

# Divide & Conquer

---

## Beispiele:

- Mergesort
- Quicksort
- Binary Search
- Arithmische Operationen wie Multiplikation großer Zahlen oder Matrixmultiplikation
- **Selektion**
- Nächstes-Paar-Problem

# Selektion

---

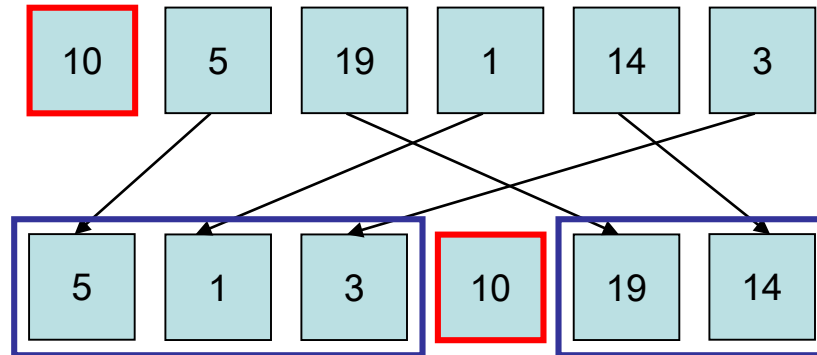
**Problem:** finde  $k$ -kleinstes Element in einer Folge von  $n$  Elementen

**Lösung:** sortiere Elemente (z.B. Mergesort), gib  $k$ -tes Element aus  $\rightarrow$  Zeit  $O(n \log n)$

Geht das auch schneller??

# Selektion

Ansatz: verfahren ähnlich zu Quicksort



- $j$ : Position des Pivotelements
- $k < j$ : mach mit linker Teilfolge weiter
- $k > j$ : mach mit rechter Teilfolge weiter

# Selektion

---

Quickselect( $A, l, r, k$ )

▷  $A[l..r]$ : Restfeld,  $k$ :  $k$ -kleinstes Element,  $l \leq k \leq r$

if  $r=l$  then return  $a[l]$

$i \leftarrow$  Randomized-Partition( $A, l, r$ ) ▷ siehe Kapitel 6

if  $k < i$  then  $x \leftarrow$  Quickselect( $A, l, i-1, k$ )

if  $k > i$  then  $x \leftarrow$  Quickselect( $A, i+1, r, k$ )

if  $k = i$  then  $x \leftarrow a[k]$

return  $x$

Zum Vergleich Quicksort( $A, l, r$ ):

if  $l < r$  then

$i \leftarrow$  Partition( $A, l, r$ )

Quicksort( $A, l, i-1$ )

Quicksort( $A, i+1, r$ )



# Quickselect

---

- $C(n)$ : erwartete Anzahl Vergleiche

Satz 9.1:  $C(n) = O(n)$

Beweis:

- Pivot ist **gut**: keine der Teilfolgen länger als  $2n/3$
- Sei  $p = \Pr[\text{Pivot ist gut}]$



- $p = 1/3$

# Quickselect

---

- Pivot **gut**: Restaufwand  $\leq C(2n/3)$
- Pivot **schlecht**: Restaufwand  $\leq C(n)$

$$C(n) \leq n + p \cdot C(2n/3) + (1-p) \cdot C(n)$$

$$\Rightarrow C(n) \leq n/p + C(2n/3)$$

$$\leq 3n + C(2n/3) \leq 3(n + 2n/3 + 4n/9 + \dots)$$

$$\leq 3n \sum_{i \geq 0} (2/3)^i$$

$$\leq 3n / (1 - 2/3) = 9n$$

# Divide & Conquer

---

## Beispiele:

- Mergesort
- Quicksort
- Binary Search
- Arithmische Operationen wie Multiplikation großer Zahlen oder Matrixmultiplikation
- Selektion
- **Nächstes-Paar-Problem**

# Nächstes-Paar-Problem

---

## Nächstes-Paar-Problem:

- **Eingabe:** Menge  $S$  von  $n$  Punkten  $P_1=(x_1,y_1), \dots, P_n=(x_n,y_n)$  im 2-dimensionalen Euklidischen Raum
- **Ausgabe:** Punktpaar mit kürzester Distanz

Annahme:  $n$  ist Zweierpotenz

# Nächstes-Paar-Problem

---

## Algo für Nächstes-Paar-Problem:

- Sortiere Punkte gemäß  $x$ -Koordinate (z.B. Mergesort, Zeit  $O(n \log n)$  )
- Löse danach Nächstes-Paar-Problem rekursiv durch Algo **ClosestPair**

# Nächstes-Paar-Problem

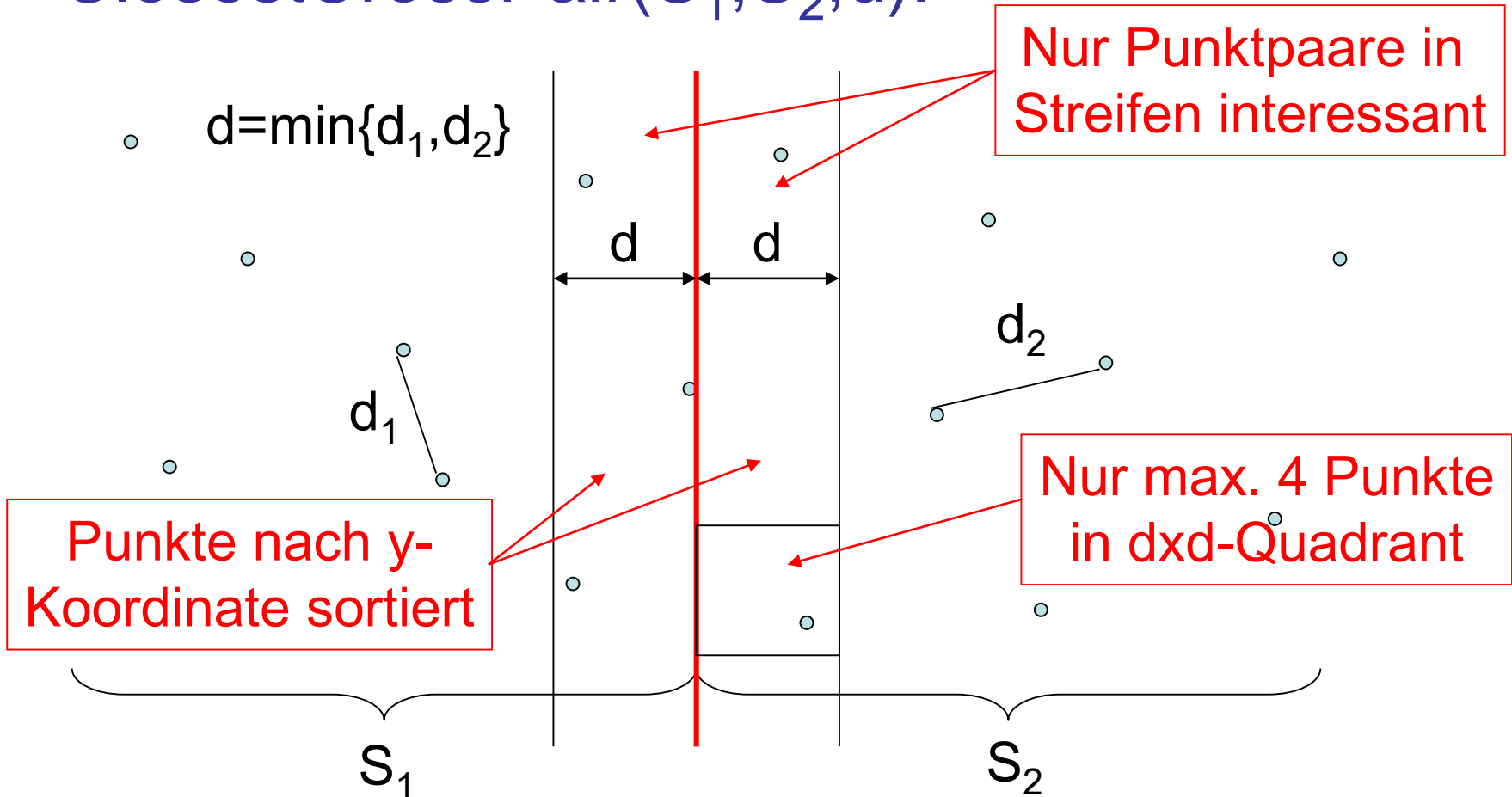
---

## Algo ClosestPair( $S$ ):

- Eingabe: nach  $x$ -Koordinate sortierte Punktmenge  $S$
- $|S|=2$ : sortiere  $S$  gemäß  $y$ -Koordinate und gib Distanz zwischen den Punkten in  $S$  zurück
- $|S|>2$ :
  - teile  $S$  in der Mitte (Position  $n/2$ ) in  $S_1$  und  $S_2$
  - $d_1 := \text{ClosestPair}(S_1)$ ;  $d_2 := \text{ClosestPair}(S_2)$
  - $d := \min\{\text{ClosestCrossPair}(S_1, S_2, \min(d_1, d_2)), d_1, d_2\}$
  - Führe  $\text{Merge}(S_1, S_2)$  durch, so dass  $S$  am Ende nach  $y$ -Koordinate sortiert ist ( $S_1, S_2$  bereits nach  $y$  sortiert)
  - gib  $d$  zurück

# Nächstes-Paar-Problem

ClosestCrossPair( $S_1, S_2, d$ ):



# Nächstes-Paar-Problem

---

## ClosestCrossPair:

- Durchlaufe die (nach der  $y$ -Koordinate sortierten) Punkte in  $S_1$  und  $S_2$  von oben nach unten (wie in  $\text{Merge}(S_1, S_2)$ ) und merke die Mengen  $M_1$  und  $M_2$  der 8 zuletzt gesehenen Punkte in  $S_1$  und  $S_2$  im  $d$ -Streifen
- Bei jedem neuen Knoten in  $M_1$ , berechne Distanzen zu Knoten in  $M_2$ , und bei jedem neuen Knoten in  $M_2$ , berechne Distanzen zu Knoten in  $M_1$
- Gib am Ende minimal gefundene Distanz zurück



# Nächstes-Paar-Problem

---

Laufzeit für Nächstes-Paar-Algo:

- Mergesort am Anfang:  $O(n \log n)$
- Laufzeit  $T(n)$  von ClosestPair Algo:

$$T(1)=O(1), T(n)=2T(n/2)+O(n)$$

Gesamtlaufzeit:  $O(n \log n)$

Brute force:  $O(n^2)$

## Generische Optimierungsverfahren:

- Systematische Suche
    - lass nichts aus
  - Divide and Conquer
  - Dynamische Programmierung
- } DuA,  
Kap. 18 und 20
- Greedy Verfahren
    - schau niemals zurück
  - Lokale Suche
    - denke global, handle lokal

# Dynamische Programmierung

---

## Gierige Algorithmen:

- Berechne Lösung schrittweise
- In jedem Schritt mache lokal optimale Wahl

## Anwendbar:

- Wenn optimale Lösung eines Problems optimale Lösung von Teilproblemen enthält

## Algorithmen:

- Scheduling Probleme
- Optimale Präfix-Kodierung (Huffman Codes)

# Dynamische Programmierung

---

## Rekursiver Ansatz:

- Lösen eines Problems durch Lösen mehrerer kleinerer Teilprobleme, aus denen sich die Lösung für das Ausgangsproblem zusammensetzt

## Phänomen:

- Mehrfachberechnungen von Lösungen

## Methode:

- Lösungen zu Teilproblemen werden iterativ beginnend mit den Lösungen der kleinsten Teilprobleme berechnet (*bottom-up*).
- Speichern einmal berechneter Lösungen in einer Tabelle

# Dynamische Programmierung

---

Typische Anwendung für dynamisches Programmieren: *Optimierungsprobleme*

Eine optimale Lösung für das Ausgangsproblem setzt sich aus *optimalen* Lösungen für kleinere Probleme zusammen.

## Mit Greedy-Algorithmen:

- Algorithmenmethode, um Optimierungsprobleme zu lösen

## Mit Divide-&-Conquer:

- Lösung eines Problems aus Lösungen zu Teilproblemen
- Aber: Lösungen zu Teilproblemen werden *nicht* rekursiv gelöst.

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

### Definition:

- Seien  $X=(x_1,\dots,x_m)$  und  $Y=(y_1,\dots,y_n)$  zwei Folgen, wobei  $x_i, y_j \in A$  für ein endliches Alphabet  $A$ .
- Dann heißt  $Y$  **Teilfolge** von  $X$ , wenn es aufsteigend sortierte Indizes  $i_1,\dots,i_n$  gibt mit  $x_{i_j} = y_j$  für  $j = 1,\dots,n$ .

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

### Definition:

- Seien  $X=(x_1,\dots,x_m)$  und  $Y=(y_1,\dots,y_n)$  zwei Folgen, wobei  $x_i, y_j \in A$  für ein endliches Alphabet  $A$ .
- Dann heißt  $Y$  **Teilfolge** von  $X$ , wenn es aufsteigend sortierte Indizes  $i_1,\dots,i_n$  gibt mit  $x_{i_j} = y_j$  für  $j = 1,\dots,n$ .

### Beispiel:

Folge Y	B	C	A	C			
Folge X	A	B	A	C	A	B	C



# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

### Definition:

- Seien  $X=(x_1,\dots,x_m)$  und  $Y=(y_1,\dots,y_n)$  zwei Folgen, wobei  $x_i, y_j \in A$  für ein endliches Alphabet  $A$ .
- Dann heißt  $Y$  **Teilfolge** von  $X$ , wenn es aufsteigend sortierte Indizes  $i_1,\dots,i_n$  gibt mit  $x_{i_j} = y_j$  für  $j = 1,\dots,n$ .

### Beispiel:

Folge Y	B	C	A	C			
Folge X	A	B	A	C	A	B	C

# Dynamische Programmierung

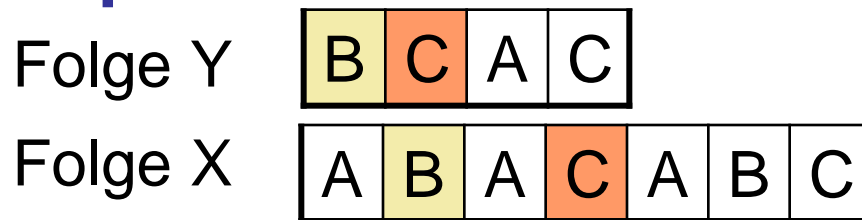
## Längste gemeinsame Teilfolge

---

### Definition:

- Seien  $X=(x_1,\dots,x_m)$  und  $Y=(y_1,\dots,y_n)$  zwei Folgen, wobei  $x_i, y_j \in A$  für ein endliches Alphabet  $A$ .
- Dann heißt  $Y$  **Teilfolge** von  $X$ , wenn es aufsteigend sortierte Indizes  $i_1,\dots,i_n$  gibt mit  $x_{i_j} = y_j$  für  $j = 1,\dots,n$ .

### Beispiel:



# Dynamische Programmierung

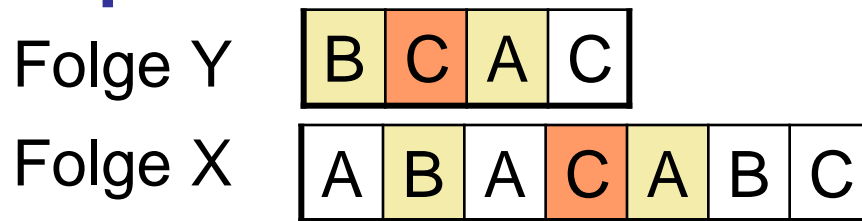
## Längste gemeinsame Teilfolge

---

### Definition:

- Seien  $X=(x_1,\dots,x_m)$  und  $Y=(y_1,\dots,y_n)$  zwei Folgen, wobei  $x_i, y_j \in A$  für ein endliches Alphabet  $A$ .
- Dann heißt  $Y$  **Teilfolge** von  $X$ , wenn es aufsteigend sortierte Indizes  $i_1,\dots,i_n$  gibt mit  $x_{i_j} = y_j$  für  $j = 1,\dots,n$ .

### Beispiel:



# Dynamische Programmierung

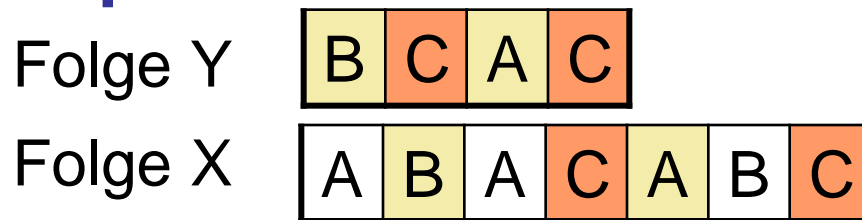
## Längste gemeinsame Teilfolge

---

### Definition:

- Seien  $X=(x_1,\dots,x_m)$  und  $Y=(y_1,\dots,y_n)$  zwei Folgen, wobei  $x_i, y_j \in A$  für ein endliches Alphabet  $A$ .
- Dann heißt  $Y$  **Teilfolge** von  $X$ , wenn es aufsteigend sortierte Indizes  $i_1,\dots,i_n$  gibt mit  $x_{i_j} = y_j$  für  $j = 1,\dots,n$ .

### Beispiel:



- $Y$  ist Teilfolge von  $X$ , Wähle  $(i_1, i_2, i_3, i_4) = (2, 4, 5, 7)$

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

### Definition:

- Seien  $X, Y, Z$  Folgen über  $A$ .
- Dann heißt  $Z$  **gemeinsame Teilfolge** von  $X$  und  $Y$ , wenn  $Z$  Teilfolge sowohl von  $X$  als auch von  $Y$  ist.

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

### Definition:

- Seien  $X, Y, Z$  Folgen über  $A$ .
- Dann heißt  $Z$  **gemeinsame Teilfolge** von  $X$  und  $Y$ , wenn  $Z$  Teilfolge sowohl von  $X$  als auch von  $Y$  ist.

### Beispiel:

Folge  $Z$ 

B	C	A	C
---	---	---	---

Folge  $X$ 

A	B	A	C	A	B	C
---	---	---	---	---	---	---

Folge  $Y$ 

B	A	C	C	A	B	B	C
---	---	---	---	---	---	---	---

- $Z$  ist gemeinsame Teilfolge von  $X$  und  $Y$

# Dynamische Programmierung

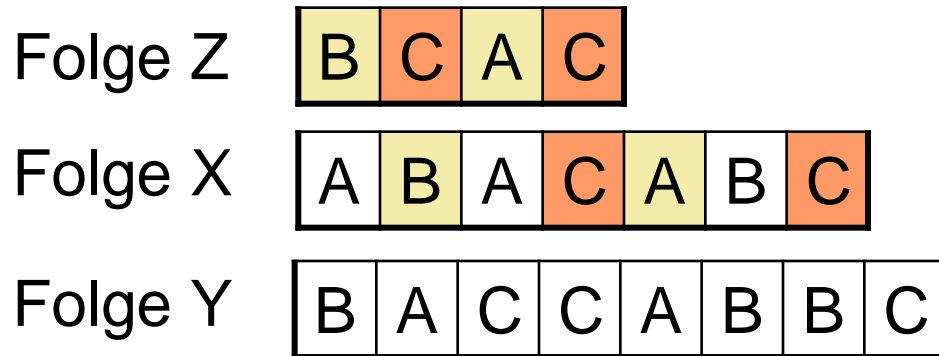
## Längste gemeinsame Teilfolge

---

### Definition:

- Seien  $X, Y, Z$  Folgen über  $A$ .
- Dann heißt  $Z$  **gemeinsame Teilfolge** von  $X$  und  $Y$ , wenn  $Z$  Teilfolge sowohl von  $X$  als auch von  $Y$  ist.

### Beispiel:



- $Z$  ist gemeinsame Teilfolge von  $X$  und  $Y$

# Dynamische Programmierung

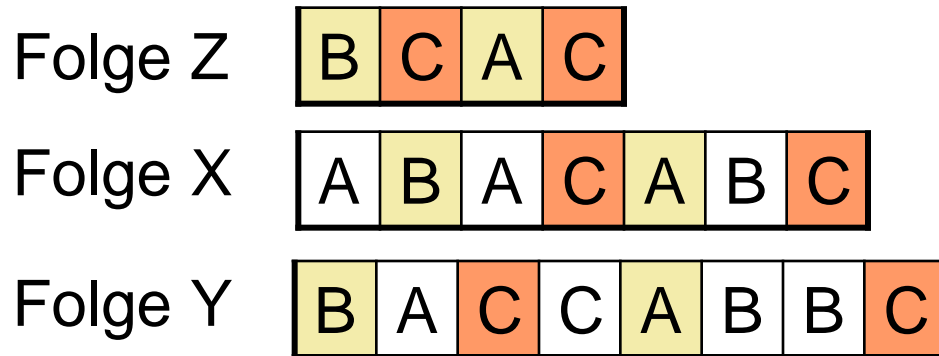
## Längste gemeinsame Teilfolge

---

### Definition:

- Seien  $X, Y, Z$  Folgen über  $A$ .
- Dann heißt  $Z$  **gemeinsame Teilfolge** von  $X$  und  $Y$ , wenn  $Z$  Teilfolge sowohl von  $X$  als auch von  $Y$  ist.

### Beispiel:



- $Z$  ist gemeinsame Teilfolge von  $X$  und  $Y$



# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

### Definition:

- Seien  $X, Y, Z$  Folgen über  $A$ .
- Dann heißt  $Z$  **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 größere Länge als  $Z$  besitzt.

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

### Definition:

- Seien  $X, Y, Z$  Folgen über  $A$ .
- Dann heißt  $Z$  **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 größere Länge als  $Z$  besitzt.

### Beispiel:

Folge X 

A	B	A	C	A	B	C
---	---	---	---	---	---	---

Folge Y 

B	A	C	C	A	B	B	C
---	---	---	---	---	---	---	---

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

### Definition:

- Seien  $X, Y, Z$  Folgen über  $A$ .
- Dann heißt  $Z$  **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 größere Länge als  $Z$  besitzt.

### Beispiel:

Folge X 

A	B	A	C	A	B	C
---	---	---	---	---	---	---

Folge Y 

B	A	C	C	A	B	B	C
---	---	---	---	---	---	---	---

Folge  $Z_1$ 

B	C	A	C
---	---	---	---

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

### Definition:

- Seien  $X, Y, Z$  Folgen über  $A$ .
- Dann heißt  $Z$  **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 größere Länge als  $Z$  besitzt.

### Beispiel:

Folge X 

A	B	A	C	A	B	C
---	---	---	---	---	---	---

Folge Y 

B	A	C	C	A	B	B	C
---	---	---	---	---	---	---	---

Folge  $Z_1$ 

B	C	A	C
---	---	---	---

Folge  $Z_2$ 

B	A	C	A	C
---	---	---	---	---

Längste Teilfolge  
hat Länge 6

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

### Eingabe:

- Folge  $X=(x_1,\dots,x_m)$
- Folge  $Y=(y_1,\dots,y_n)$

### Ausgabe:

- Längste gemeinsame Teilfolge  $Z$   
(**L**ongest **C**ommon **S**ubsequenz)

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

### Eingabe:

- Folge  $X=(x_1,\dots,x_m)$
- Folge  $Y=(y_1,\dots,y_n)$

### Ausgabe:

- Längste gemeinsame Teilfolge Z  
(**L**ongest **C**ommon **S**ubsequenz)

### Beispiel:

Folge X 

A	B	C	B	D	A	B
---	---	---	---	---	---	---

Folge Y 

B	D	C	A	B	A
---	---	---	---	---	---

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

### Algorithmus:

- Erzeuge alle möglichen Teilfolgen von  $X$
- Teste für jede Teilfolge von  $X$ , ob auch Teilfolge von  $Y$
- Merke zu jedem Zeitpunkt bisher längste gemeinsame Teilfolge

### Laufzeit:

- $2^m$  mögliche Teilfolgen
- Exponentielle Laufzeit!

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

### Satz 9.2:

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

1. Ist  $x_m = y_n$ , dann ist  $z_k = x_m = y_n$  und  $(z_1,\dots,z_{k-1})$  ist eine längste gemeinsame Teilfolge von  $(x_1,\dots,x_{m-1})$  und  $(y_1,\dots,y_{n-1})$ .



# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

### Satz 9.2:

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

1. Ist  $x_m = y_n$ , dann ist  $z_k = x_m = y_n$  und  $(z_1,\dots,z_{k-1})$  ist eine längste gemeinsame Teilfolge von  $(x_1,\dots,x_{m-1})$  und  $(y_1,\dots,y_{n-1})$ .
2. Ist  $x_m \neq y_n$  und  $z_k \neq x_m$ , dann ist  $Z$  eine längste gemeinsame Teilfolge von  $(x_1,\dots,x_{m-1})$  und  $Y$ .

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

### Satz 9.2:

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

1. Ist  $x_m = y_n$ , dann ist  $z_k = x_m = y_n$  und  $(z_1,\dots,z_{k-1})$  ist eine längste gemeinsame Teilfolge von  $(x_1,\dots,x_{m-1})$  und  $(y_1,\dots,y_{n-1})$ .
2. Ist  $x_m \neq y_n$  und  $z_k \neq x_m$ , dann ist  $Z$  eine längste gemeinsame Teilfolge von  $(x_1,\dots,x_{m-1})$  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  $(y_1,\dots,y_{n-1})$ .

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

### Satz 9.2:

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

1. Ist  $x_m = y_n$ , dann ist  $z_k = x_m = y_n$  und  $(z_1,\dots,z_{k-1})$  ist eine längste gemeinsame Teilfolge von  $(x_1,\dots,x_{m-1})$  und  $(y_1,\dots,y_{n-1})$ .
2. Ist  $x_m \neq y_n$  und  $z_k \neq x_m$ , dann ist  $Z$  eine längste gemeinsame Teilfolge von  $(x_1,\dots,x_{m-1})$  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  $(y_1,\dots,y_{n-1})$ .

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

### *Lemma 9.3:*

Sei  $C[i][j]$  die Länge einer längsten gemeinsamen Teilfolge von  $(x_1, \dots, x_i)$  und  $(y_1, \dots, y_j)$ . Dann gilt:

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

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

### *Lemma 9.3:*

Sei  $C[i][j]$  die Länge einer längsten gemeinsamen Teilfolge von  $(x_1, \dots, x_i)$  und  $(y_1, \dots, y_j)$ . Dann gilt:

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

### **Beobachtung:**

Rekursive Berechnung der  $C[i][j]$  würde zu Berechnung immer wieder derselben Werte führen. Dieses ist ineffizient. Berechnen daher die Werte  $C[i][j]$  iterativ, nämlich zeilenweise.

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

LCS-Länge(Array X, Y)

1.  $m \leftarrow \text{length}[X]$
2.  $n \leftarrow \text{length}[Y]$
3. **new** array C[0,...,m][0,...,n]
4. **for** i  $\leftarrow$  0 **to** m **do** C[i][0]  $\leftarrow$  0
5. **for** j  $\leftarrow$  0 **to** n **do** C[0][j]  $\leftarrow$  0
6. **for** i  $\leftarrow$  1 **to** m **do**
7.     **for** j  $\leftarrow$  1 **to** n **do**
8.         ➤ Längenberechnung(X, Y, C, i, j)
9. **return** C

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

LCS-Länge(Array X, Y)

1.  $m \leftarrow \text{length}[X]$
2.  $n \leftarrow \text{length}[Y]$
3. **new** array C[0,...,m][0,...,n]
4. **for** i  $\leftarrow$  0 **to** m **do** C[i][0]  $\leftarrow$  0
5. **for** j  $\leftarrow$  0 **to** n **do** C[0][j]  $\leftarrow$  0
6. **for** i  $\leftarrow$  1 **to** m **do**
7.     **for** j  $\leftarrow$  1 **to** n **do**
8.         ➤ Längenberechnung(X, Y, C, i, j)
9. **return** C

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

LCS-Länge(Array X, Y)

1.  $m \leftarrow \text{length}[X]$
2.  $n \leftarrow \text{length}[Y]$
3. **new array C[0,...,m][0,...,n]**
4. **for**  $i \leftarrow 0$  **to**  $m$  **do**  $C[i][0] \leftarrow 0$
5. **for**  $j \leftarrow 0$  **to**  $n$  **do**  $C[0][j] \leftarrow 0$
6. **for**  $i \leftarrow 1$  **to**  $m$  **do**
7.     **for**  $j \leftarrow 1$  **to**  $n$  **do**
8.         ➤ Längenberechnung(X, Y, C, i, j)
9. **return** C

Tabelle für die  
C[i][j] Werte  
anlegen.



# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

LCS-Länge(Array X, Y)

1.  $m \leftarrow \text{length}[X]$
2.  $n \leftarrow \text{length}[Y]$
3. **new** array C[0,...,m][0,...,n]
4. **for** i  $\leftarrow$  0 **to** m **do** C[i][0]  $\leftarrow$  0
5. **for** j  $\leftarrow$  0 **to** n **do** C[0][j]  $\leftarrow$  0
6. **for** i  $\leftarrow$  1 **to** m **do**
7.     **for** j  $\leftarrow$  1 **to** n **do**
8.         ➤ Längenberechnung(X, Y, C, i, j)
9. **return** C

Erste Spalte  
der Tabelle  
auf 0 setzen.

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

LCS-Länge(Array X, Y)

1.  $m \leftarrow \text{length}[X]$
2.  $n \leftarrow \text{length}[Y]$
3. **new** array C[0,...,m][0,...,n]
4. **for**  $i \leftarrow 0$  **to**  $m$  **do**  $C[i][0] \leftarrow 0$
5. **for**  $j \leftarrow 0$  **to**  $n$  **do**  $C[0][j] \leftarrow 0$
6. **for**  $i \leftarrow 1$  **to**  $m$  **do**
7.     **for**  $j \leftarrow 1$  **to**  $n$  **do**
8.         ➤ Längenberechnung(X, Y, C, i, j)
9. **return** C

Erste Reihe  
der Tabelle  
auf 0 setzen.

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

LCS-Länge(Array X, Y)

1.  $m \leftarrow \text{length}[X]$
2.  $n \leftarrow \text{length}[Y]$
3. **new** array C[0,...,m][0,...,n]
4. **for** i  $\leftarrow$  0 **to** m **do** C[i][0]  $\leftarrow$  0
5. **for** j  $\leftarrow$  0 **to** n **do** C[0][j]  $\leftarrow$  0
6. **for** i  $\leftarrow$  1 **to** m **do**
7.     **for** j  $\leftarrow$  1 **to** n **do**
8.         ➤ Längenberechnung(X, Y, C, i, j)
9. **return** C

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

Längenberechnung(Array X, Y, C, i, j)

1. **if**  $x_i = y_j$  **then**  $C[i][j] \leftarrow C[i-1][j-1] + 1$
2. **else**
3.     **if**  $C[i-1][j] \geq C[i][j-1]$  **then**  $C[i][j] \leftarrow C[i-1][j]$
4.     **else**  $C[i][j] \leftarrow C[i][j-1]$

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

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

Längenberechnung(Array X, Y, C, i, j)

1. **if**  $x_i = y_j$  **then**  $C[i][j] \leftarrow C[i-1][j-1] + 1$
2. **else**
3. **if**  $C[i-1][j] \geq C[i][j-1]$  **then**  $C[i][j] \leftarrow C[i-1][j]$
4. **else**  $C[i][j] \leftarrow C[i][j-1]$

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

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

LCS-Länge(Array X, Y)

1.  $m \leftarrow \text{length}[X]$
2.  $n \leftarrow \text{length}[Y]$
3. **new** array C[0,...,m][0,...,n]
4. **for** i  $\leftarrow$  0 **to** m **do** C[i][0]  $\leftarrow$  0
5. **for** j  $\leftarrow$  0 **to** n **do** C[0][j]  $\leftarrow$  0
6. **for** i  $\leftarrow$  1 **to** m **do**
7.     **for** j  $\leftarrow$  1 **to** n **do**
8.         ➤ Längenberechnung(X, Y, C, i, j)
9. **return** C

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

		j	0	1	2	3	4	5	6
			$y_j$	B	D	C	A	B	A
i	$x_i$								
0									
1	A								
2	B								
3	C								
4	B								
5	D								
6	A								
7	B								

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

		j	0	1	2	3	4	5	6
			$y_j$	B	D	C	A	B	A
i	$x_i$								
0			0						
1	A		0						
2	B		0						
3	C		0						
4	B		0						
5	D		0						
6	A		0						
7	B		0						



# Dynamische Programmierung

## Längste gemeinsame Teilfolge

		j	0	1	2	3	4	5	6
			0	B	D	C	A	B	A
i	$x_i$	$y_j$	0	0	0	0	0	0	0
0	$x_0$		0	0	0	0	0	0	0
1	A		0						
2	B		0						
3	C		0						
4	B		0						
5	D		0						
6	A		0						
7	B		0						

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

		j	0	1	2	3	4	5	6
			0	1	2	3	4	5	6
i		$y_j$		B	D	C	A	B	A
0	$x_i$		0	0	0	0	0	0	0
1	A		0						
2	B		0						
3	C		0						
4	B		0						
5	D		0						
6	A		0						
7	B		0						

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

		j						
		0	1	2	3	4	5	6
		$y_j$	B	D	C	A	B	A
i	$x_i$	0	0	0	0	0	0	0
0		0	0	0	0	0	0	0
1	A	0						
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

		j	0	1	2	3	4	5	6
			$y_j$	B	D	C	A	B	A
i	$x_i$	0	0	0	0	0	0	0	0
1	A	0	0	↑ 0					
2	B	0							
3	C	0							
4	B	0							
5	D	0							
6	A	0							
7	B	0							

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

		j	0	1	2	3	4	5	6
			0	1	2	3	4	5	6
i		$y_j$	B	D	C	A	B	A	
0	$x_i$	0	0	0	0	0	0	0	0
1	A	0	↑ 0						
2	B	0							
3	C	0							
4	B	0							
5	D	0							
6	A	0							
7	B	0							

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

		j	0	1	2	3	4	5	6
			$y_j$	B	D	C	A	B	A
i	$x_i$	0	0	0	0	0	0	0	0
1	A	0	↑	0					
2	B	0							
3	C	0							
4	B	0							
5	D	0							
6	A	0							
7	B	0							

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

		j	0	1	2	3	4	5	6
			$y_j$	B	D	C	A	B	A
i	$x_i$	0	0	0	0	0	0	0	0
1	A	0	↑	0	↑	0			
2	B	0							
3	C	0							
4	B	0							
5	D	0							
6	A	0							
7	B	0							

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

j		0	1	2	3	4	5	6
i		$y_j$	B	D	C	A	B	A
0	$x_i$	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0			
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						



# Dynamische Programmierung

## Längste gemeinsame Teilfolge

		j	0	1	2	3	4	5	6
			$y_j$	B	D	C	A	B	A
i	$x_i$	0	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0				
2	B	0							
3	C	0							
4	B	0							
5	D	0							
6	A	0							
7	B	0							

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

		j	0	1	2	3	4	5	6
			0	B	D	C	A	B	A
i	$x_i$	$y_j$							
0	$x_0$		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0			
2	B		0						
3	C		0						
4	B		0						
5	D		0						
6	A		0						
7	B		0						

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

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

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

		j	0	1	2	3	4	5	6
			0	B	D	C	A	<b>B</b>	A
i	$x_i$	$y_j$							
0			0	0	0	0	0	0	0
1	<b>A</b>		0	↑ 0	↑ 0	↑ 0	↖ 1		
2	B		0						
3	C		0						
4	B		0						
5	D		0						
6	A		0						
7	B		0						

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

		j	0	1	2	3	4	5	6
			$y_j$	B	D	C	A	<b>B</b>	A
i	$x_i$		0	0	0	0	0	0	0
0	$x_0$		0	0	0	0	0	0	0
1	<b>A</b>		0	↑ 0	↑ 0	↑ 0	↖ 1		
2	B		0						
3	C		0						
4	B		0						
5	D		0						
6	A		0						
7	B		0						

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

		j	0	1	2	3	4	5	6
		$y_j$	B	D	C	A	<b>B</b>	A	
i	$x_i$	0	0	0	0	0	0	0	0
1	<b>A</b>	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1		
2	B	0							
3	C	0							
4	B	0							
5	D	0							
6	A	0							
7	B	0							

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

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

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

		j	0	1	2	3	4	5	6
		$y_j$	B	D	C	A	B	<b>A</b>	
i	$x_i$								
0		0	0	0	0	0	0	0	0
1	<b>A</b>	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1		
2	B	0							
3	C	0							
4	B	0							
5	D	0							
6	A	0							
7	B	0							



# Dynamische Programmierung

## Längste gemeinsame Teilfolge

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

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

j		0	1	2	3	4	5	6
i		$y_j$	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					
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

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

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

		j	0	1	2	3	4	5	6
		$y_j$		B	D	<b>C</b>	A	B	A
i	$x_i$		0	0	0	0	0	0	0
0	$x_0$		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	<b>B</b>		0	↖ 1	← 1	← 1			
3	C		0						
4	B		0						
5	D		0						
6	A		0						
7	B		0						

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

		j	0	1	2	3	4	5	6
			0	B	D	C	A	B	A
i	$x_i$	$y_j$							
0	$x_0$		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						
4	B		0						
5	D		0						
6	A		0						
7	B		0						

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

		j	0	1	2	3	4	5	6
			0	B	D	C	A	B	A
i	$x_i$	$y_j$							
0	$x_0$		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						
5	D		0						
6	A		0						
7	B		0						

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

		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	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D		0						
6	A		0						
7	B		0						

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

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



# Dynamische Programmierung

## Längste gemeinsame Teilfolge

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

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

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

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

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

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

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

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

		j	0	1	2	3	4	5	6
		$y_j$	B	D	C	A	B	<b>A</b>	
i	$x_i$	0	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	↑ 1	↑ 2	↑ 2	<b>↖ 3</b>	← 3	
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	<b>↑ 3</b>	↑ 3	
6	<b>A</b>	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	<b>↖ 4</b>	
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	<b>↑ 4</b>	

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

		j	0	1	2	3	4	5	6
		$y_j$		B	D	C	A	<b>B</b>	<b>A</b>
i	$x_i$		0	0	0	0	0	0	0
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>B</b>		0	↖ 1	↑ 1	↑ 2	↑ 2	<b>↖ 3</b>	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	<b>↑ 3</b>	↑ 3
6	<b>A</b>		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	<b>↖ 4</b>
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	<b>↑ 4</b>

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

		j	0	1	2	3	4	5	6
		$y_j$		B	D	C	A	<b>B</b>	<b>A</b>
i	$x_i$		0	0	0	0	0	0	0
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	<b>← 2</b>	↑ 2	↑ 2
4	<b>B</b>		0	↖ 1	↑ 1	↑ 2	↑ 2	<b>↖ 3</b>	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	<b>↑ 3</b>	↑ 3
6	<b>A</b>		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	<b>↖ 4</b>
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	<b>↑ 4</b>

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

		j	0	1	2	3	4	5	6
			$y_j$	B	D	C	A	<b>B</b>	<b>A</b>
i	$x_i$	0	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>B</b>	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3	
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3	
6	<b>A</b>	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4	
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4	



# Dynamische Programmierung

## Längste gemeinsame Teilfolge

		j	0	1	2	3	4	5	6
		$y_j$		B	D	<b>C</b>	A	<b>B</b>	<b>A</b>
i	$x_i$		0	0	0	0	0	0	0
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	<b>C</b>		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	<b>B</b>		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	<b>A</b>		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

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

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

		j	0	1	2	3	4	5	6
		$y_j$		B	D	<b>C</b>	A	<b>B</b>	<b>A</b>
i	$x_i$		0	0	0	0	0	0	0
0	$x_0$		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	<b>C</b>		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	<b>B</b>		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	<b>A</b>		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

j		0	1	2	3	4	5	6
i		$y_j$	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	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

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

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

### *Lemma 9.4:*

Der Algorithmus LCS-Länge hat Laufzeit  $O(nm)$ , wenn die Folgen  $X, Y$  Länge  $n$  und  $m$  haben.

### *Lemma 9.5:*

Die Ausgabe der längsten gemeinsamen Teilfolge anhand der Tabelle hat Laufzeit  $O(n+m)$ , wenn die Folgen  $X, Y$  Länge  $n$  und  $m$  haben.

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

1. Bestimme rekursive Struktur einer optimalen Lösung.
2. Entwerfe rekursive Methode zur Bestimmung des Wertes einer optimalen Lösung.
3. Transformiere rekursive Methode in eine iterative Methode zur Bestimmung des Wertes einer optimalen Lösung.
4. Bestimme aus dem Wert einer optimalen Lösung und den in 3. berechneten Zusatzinformationen eine optimale Lösung.

# Dynamische Programmierung

## Längste gemeinsame Teilfolge

---

### Algorithmenentwurfstechnik:

- Oft bei Optimierungsproblemen angewandt

### Einsatz:

- Bei rekursiven Problemlösungen, wenn Teillösungen mehrfach benötigt werden

### Lösungsansatz:

- Tabellieren von Teilergebnissen

### Vorteil:

- Laufzeitverbesserungen, oft polynomiell statt exponentiell



# Fraktionales Rucksack-Problem

---

➤ Gegeben sind  $n$  Gegenstände. Der  $i$ -te Gegenstand besitzt Wert  $v_i$  und Gewicht  $g_i$ . Ausserdem ist eine Gewichtsschranke  $W$  gegeben.

➤ Zulässige Lösungen sind Zahlen  $a_i \in [0,1]$  mit

$$\sum_{i=1}^n a_i g_i \leq W.$$

➤ Gesucht ist eine zulässige Lösung  $a_1, \dots, a_n$  mit möglichst großem Gesamtwert

$$\sum_{i=1}^n a_i v_i.$$

# Gierige Lösung des fraktionalen Rucksack-Problems

---

## Algorithmus Gieriges-Einpacken:

1. Sortiere die Verhältnisse  $v_i/g_i$  absteigend. Sei

$$v_{\pi(1)}/g_{\pi(1)} \geq v_{\pi(2)}/g_{\pi(2)} \geq \dots v_{\pi(n)}/g_{\pi(n)}$$

für Permutation  $\pi$  auf  $(1, \dots, n)$ .

2. Bestimme maximales  $k$ , so dass noch gilt  $\sum_{i=1}^k g_{\pi(i)} \leq W$ .

3. Setze  $a_{\pi(1)} = a_{\pi(2)} = \dots = a_{\pi(k)} = 1$  und setze

$$a_{\pi(k+1)} = \frac{W - \sum_{i=1}^k g_{\pi(i)}}{g_{\pi(k+1)}}.$$

Alle anderen  $a_i$  setze auf 0.

# Gieriges Einpacken ist optimal

---

**Satz 9.6:** Gieriges Einpacken löst das fraktionale Rucksackproblem optimal.

**Beweis:**

- zerlege Gegenstand  $i$  in  $g_i$  Stücke mit Wert  $v_i/g_i$ .
- betrachte das Tupel  $(M, T)$ , wobei  $M$  die Menge aller Stücke  
der Gegenstände ist und  $T$  die Menge aller Teilmengen von  
 $M$  der Größe maximal  $W$  ist
- $(M, T)$  ist offensichtlich ein Matroid
- Ergebnis des generischen Greedy Algorithmus stimmt mit Lösung auf voriger Folie überein, diese Lösung ist also optimal

## Das Rucksackproblem:

- Rucksack mit begrenzter Kapazität
- Objekte mit unterschiedlichem Wert und unterschiedlicher Größe
- Wir wollen Objekte von möglichst großem Gesamtwert mitnehmen

# Dynamisches Programmieren – Rucksack

---

## Beispiel:

- Rucksackgröße 6

Größe	5	2	1	3	7	4
Wert	11	5	2	8	14	9

# Dynamisches Programmieren – Rucksack

---

## Beispiel:

- Rucksackgröße 6

Größe	5	2	1	3	7	4
Wert	11	5	2	8	14	9

- Objekt 1 und 3 passen und haben Gesamtwert 13
- Optimal?

# Dynamisches Programmieren – Rucksack

---

## Beispiel:

- Rucksackgröße 6

Größe	5	2	1	3	7	4
Wert	11	5	2	8	14	9

- Objekt 1 und 3 passen und haben Gesamtwert 13
- Optimal?
- Objekt 2, 3 und 4 passen und haben Gesamtwert 15 !

## Das Rucksackproblem (Optimierungsversion):

- Eingabe:  $n$  Objekte  $\{1, \dots, n\}$ ;  
Objekt  $i$  hat ganzz. pos. Größe  $g[i]$  und Wert  $v[i]$ ;  
Rucksackkapazität  $W$
- Ausgabe: Menge  $S \subseteq \{1, \dots, n\}$  mit  $\sum_{i \in S} g[i] \leq W$  und  
maximalem Wert  $\sum_{i \in S} v[i]$



## Herleiten einer Rekursion:

- Sei  $O$  optimale Lösung
- Bezeichne  $\text{Opt}(i,w)$  den Wert einer optimalen Lösung aus Objekten 1 bis  $i$  bei Rucksackgröße  $w$

## Unterscheide, ob Objekt $n$ in $O$ ist:

- Fall 1 ( $n$  nicht in  $O$ ):  
 $\text{Opt}(n,W) = \text{Opt}(n-1,W)$
- Fall 2 ( $n$  in  $O$ ):  
 $\text{Opt}(n,W) = v[n] + \text{Opt}(n-1,W-g[n])$

## Rekursion:

- $\text{Opt}(i,0) = 0$  für  $0 \leq i \leq n$
- $\text{Opt}(0,i) = 0$  für  $0 \leq i \leq W$
- Wenn  $w < g[i]$  dann  $\text{Opt}(i,w) = \text{Opt}(i-1,w)$
- Sonst,  
 $\text{Opt}(i,w) = \max\{\text{Opt}(i-1,w), v[i] + \text{Opt}(i-1,w-g[i])\}$

# Dynamisches Programmieren – Rucksack

---

## Rekursion:

- $\text{Opt}(i,0) = 0$  für  $0 \leq i \leq n$
- $\text{Opt}(0,i) = 0$  für  $0 \leq i \leq W$
- Wenn  $w < g[i]$  dann  $\text{Opt}(i,w) = \text{Opt}(i-1,w)$
- Sonst,  
 $\text{Opt}(i,w) = \max\{\text{Opt}(i-1,w), v[i] + \text{Opt}(i-1,w-g[i])\}$



Kein Objekt passt in den Rucksack

# Dynamisches Programmieren – Rucksack

---

## Rekursion:

- $\text{Opt}(i,0) = 0$  für  $0 \leq i \leq n$
- $\text{Opt}(0,i) = 0$  für  $0 \leq i \leq W$
- Wenn  $w < g[i]$  dann  $\text{Opt}(i,w) = \text{Opt}(i-1,w)$
- Sonst,  
 $\text{Opt}(i,w) = \max\{\text{Opt}(i-1,w), v[i] + \text{Opt}(i-1,w-g[i])\}$



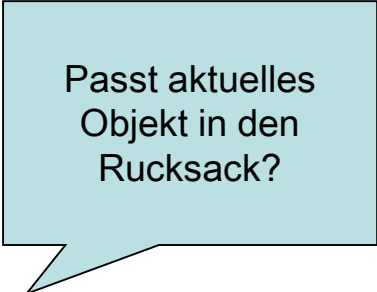
Kein Objekt steht zur Auswahl

# Dynamisches Programmieren – Rucksack

---

## Rekursion:

- $\text{Opt}(i,0) = 0$  für  $0 \leq i \leq n$
- $\text{Opt}(0,i) = 0$  für  $0 \leq i \leq W$
- Wenn  $w < g[i]$  dann  $\text{Opt}(i,w) = \text{Opt}(i-1,w)$
- Sonst,  
 $\text{Opt}(i,w) = \max\{\text{Opt}(i-1,w), v[i] + \text{Opt}(i-1,w-g[i])\}$



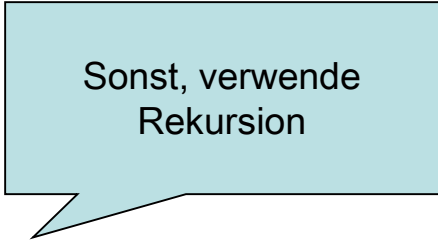
Passt aktuelles  
Objekt in den  
Rucksack?

# Dynamisches Programmieren – Rucksack

---

## Rekursion:

- $\text{Opt}(i,0) = 0$  für  $0 \leq i \leq n$
- $\text{Opt}(0,i) = 0$  für  $0 \leq i \leq W$
- Wenn  $w < g[i]$  dann  $\text{Opt}(i,w) = \text{Opt}(i-1,w)$



Sonst, verwende  
Rekursion

- Sonst,  
 $\text{Opt}(i,w) = \max\{\text{Opt}(i-1,w), v[i] + \text{Opt}(i-1,w-g[i])\}$

## Rucksack( $n, W$ )

1. Initialisiere Feld  $A[0, \dots, n][0, \dots, W]$  mit  $A[0, i] = 0$  für alle  $0 \leq i \leq n$  und  $A[j, 0] = 0$  für alle  $0 \leq j \leq W$
2. **for**  $i \leftarrow 1$  **to**  $n$  **do**
3.     **for**  $j \leftarrow 1$  **to**  $W$  **do**
4.         Berechne  $A[i, j]$  nach Rekursion
5. **return**  $A[n, W]$

# Dynamisches Programmieren – Rucksack

**Beispiel:**

n	0								
	0								
	0								
	0								
	0								
	0								
	0								
1	0								
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	↙	↙
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3



# Dynamisches Programmieren – Rucksack

## Beispiel:

n	0								
0	0								
0	0								
0	0								
0	0								
0	0								
0	0								
0	0								
1	0	0	0	0	0				
0	0	0	0	0	0	0	0	0	0
	0	1							W

g[1] > W:  
Also  $\text{Opt}(i, w) = \text{Opt}(i-1, w)$

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

# Dynamisches Programmieren – Rucksack

## Beispiel:

n	0								
	0								
	0								
	0								
	0								
	0								
	0								
1	0	0	0	0	0	2			
0	0	0	0	0	0	0	0	0	0
	0	1							W

$$\text{Opt}(i,w) = \max\{\text{Opt}(i-1,w), v[i] + \text{Opt}(i-1,w-g[i])\}$$

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

# Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
	0								
	0								
	0								
	0								
	0								
	0								
1	0	0	0	0	0	2	2		
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

# Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
	0								
	0								
	0								
	0								
	0								
	0								
1	0	0	0	0	0	2	2	2	
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

# Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
	0								
	0								
	0								
	0								
	0								
	0								
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

# Dynamisches Programmieren – Rucksack

## Beispiel:

n	0								
	0								
	0								
	0								
	0								
	0								
	0								
	0	0	0						
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	↙	↙
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

# Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
	0								
	0								
	0								
	0								
	0								
	0	0	0	4	4				
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

# Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
	0								
	0								
	0								
	0								
	0								
	0	0	0	4	4	4	4	4	
1	0	0	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3



# Dynamisches Programmieren – Rucksack

## Beispiel:

n	0								
	0								
	0								
	0								
	0								
	0								
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	↙	↙
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

# Dynamisches Programmieren – Rucksack

## Beispiel:

n	0								
	0								
	0								
	0								
	0								
	0	1	1						
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

# Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
	0								
	0								
	0								
	0								
	0	1	1	4					
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	↙	↙
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

# Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
	0								
	0								
	0								
	0								
	0	1	1	4	5	5	5	5	
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

# Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
	0								
	0								
	0								
	0								
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	↙	↙
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

# Dynamisches Programmieren – Rucksack

## Beispiel:

n	0								
	0								
	0								
	0								
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	<b>2</b>	<b>3</b>
	1	2
	7	3
	4	7
n	3	3

# Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
	0								
	0								
	0	2	3	5	6	7	9	10	10
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	↙	↙
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

# Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
	0								
	0	2	3	5	6	7	9	10	10
	0	2	3	5	6	7	9	10	10
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

		Größe	Wert
		↙	↙
		g	v
1		5	2
2		3	4
		1	1
		2	3
		1	2
		7	3
		4	7
n		3	3



# Dynamisches Programmieren – Rucksack

## Beispiel:

n	0								
	0	2	3	5	7	9	10	12	13
	0	2	3	5	6	7	9	10	10
	0	2	3	5	6	7	9	10	10
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	↙	↙
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

# Dynamisches Programmieren – Rucksack

## Beispiel:

n	0	2	3	5	7	9	10	12	13
	0	2	3	5	7	9	10	12	13
	0	2	3	5	6	7	9	10	10
	0	2	3	5	6	7	9	10	10
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	↙	↙
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

# Dynamisches Programmieren – Rucksack

## Beispiel:

n	0	2	3	5	7	9	10	12	13
	0	2	3	5	7	9	10	12	13
	0	2	3	5	6	7	9	10	10
	0	2	3	5	6	7	9	10	10
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

Optimaler Lösungswert für W=8

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

# Dynamisches Programmieren – Rucksack

## Beispiel:

n	0	2	3	5	7	9	10	12	13
	0	2	3	5	7	9	10	12	13
	0	2	3	5	6	7	9	10	10
	0	2	3	5	6	7	9	10	10
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

Optimaler Lösungswert für W=8

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

## *Satz 9.7*

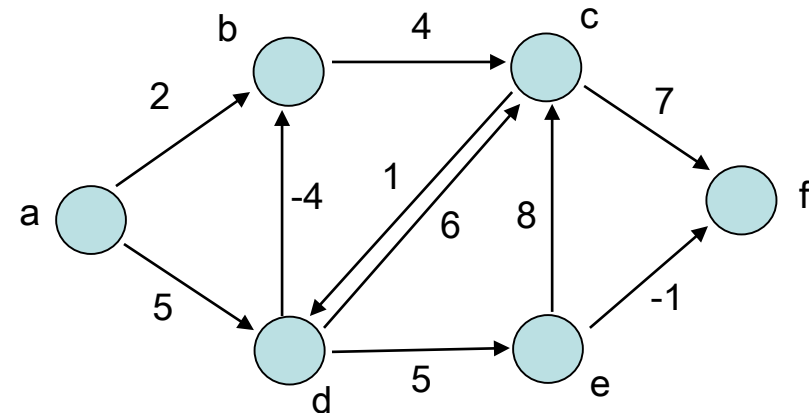
Algorithmus Rucksack berechnet in  $\Theta(nW)$  Zeit den Wert einer optimalen Lösung, wobei  $n$  die Anzahl der Objekte ist und  $W$  die Größe des Rucksacks.

# Dynamische Programmierung - APSP

## All Pairs Shortest Path (APSP):

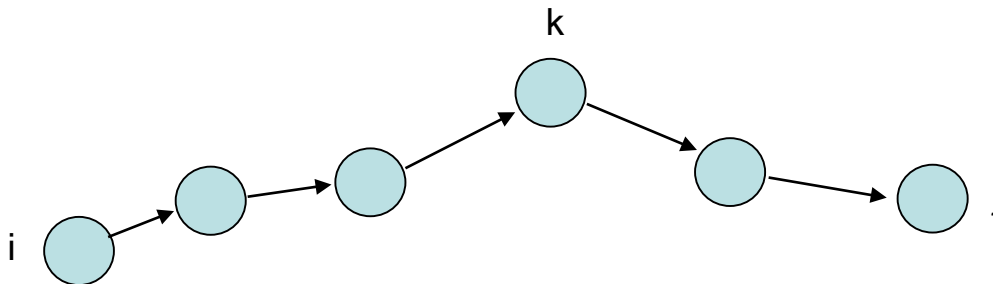
- Eingabe: Gewichteter Graph  $G=(V,E)$
- Ausgabe: Für jedes Paar von Knoten  $u,v \in V$  die Distanz von  $u$  nach  $v$  sowie einen kürzesten Weg

	a	b	c	d	e	f
a	0	1	5	5	10	9
b	$\infty$	0	4	5	10	9
c	$\infty$	-3	0	1	6	5
d	$\infty$	-4	0	0	5	4
e	$\infty$	5	8	9	0	-1
f	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	0



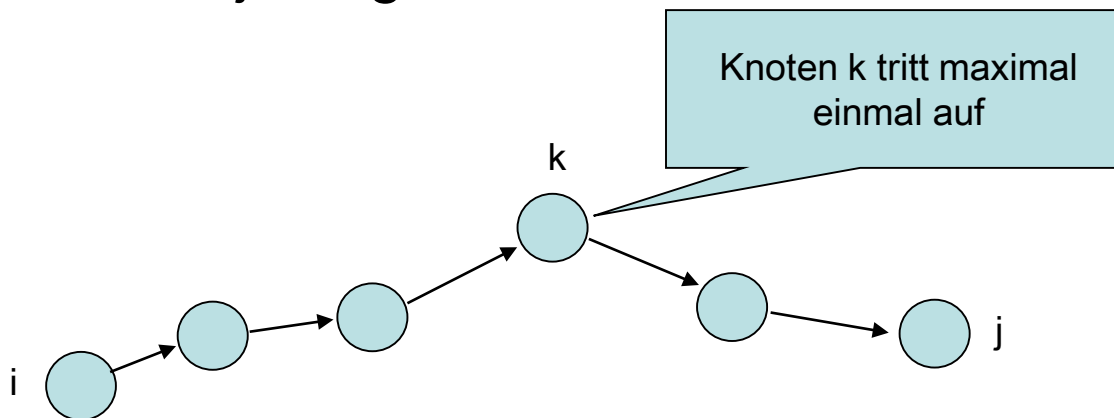
## Zur Erinnerung:

- Sei  $G$  ein Graph ohne negative Zyklen und sei  $j$  von  $i$  aus erreichbar. Dann gibt es einen kürzesten  $i$ - $j$ -Weg, der keinen Knoten doppelt benutzt.
- Wir können also annehmen, dass jeder Knoten in jedem Weg maximal einmal vorkommt
- Betrachte  $i$ - $j$ -Weg, der nur über Knoten aus  $\{1, \dots, k\}$  läuft:



## Zur Erinnerung:

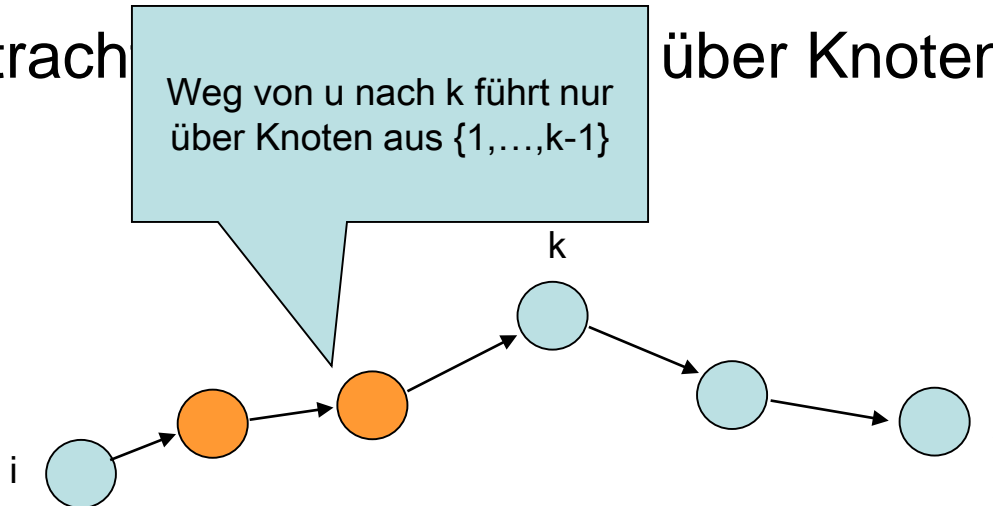
- Sei  $G$  ein Graph ohne negative Zyklen und sei  $j$  von  $i$  aus erreichbar. Dann gibt es einen kürzesten  $i$ - $j$ -Weg, der keinen Knoten doppelt benutzt.
- Wir können also annehmen, dass jeder Knoten in jedem Weg maximal einmal vorkommt
- Betrachte  $i$ - $j$ -Weg, der nur über Knoten aus  $\{1, \dots, k\}$  läuft:





## Zur Erinnerung:

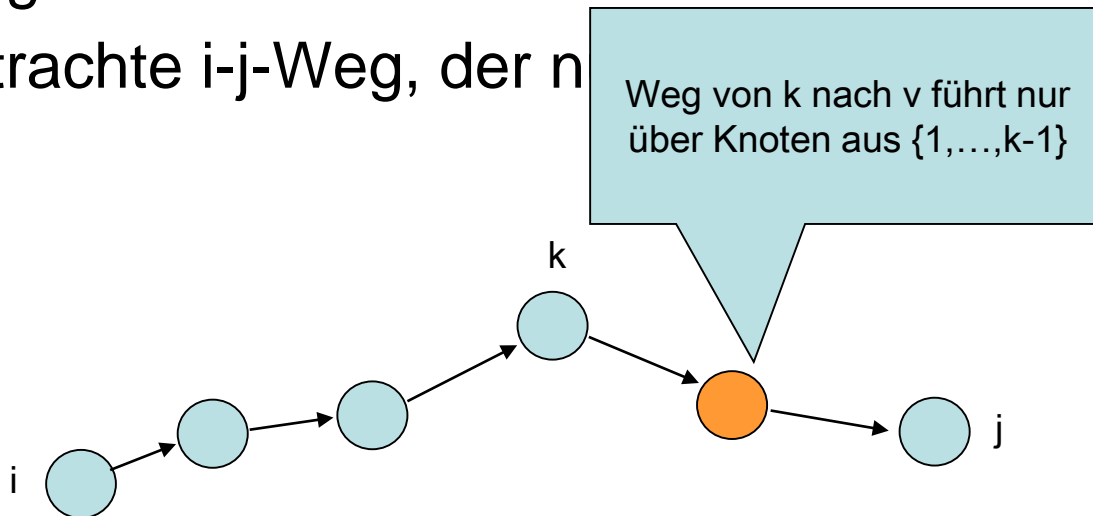
- Sei  $G$  ein Graph ohne negative Zyklen und sei  $j$  von  $i$  aus erreichbar. Dann gibt es einen kürzesten  $i$ - $j$ -Weg, der keinen Knoten doppelt benutzt.
- Wir können also annehmen, dass jeder Knoten in jedem Weg maximal einmal vorkommt
- Betrachte einen kürzesten Weg  $P$  über Knoten aus  $\{1, \dots, k\}$  läuft:



# Dynamische Programmierung - APSP

## Zur Erinnerung:

- Sei  $G$  ein Graph ohne negative Zyklen und sei  $j$  von  $i$  aus erreichbar. Dann gibt es einen kürzesten  $i$ - $j$ -Weg, der keinen Knoten doppelt benutzt.
- Wir können also annehmen, dass jeder Knoten in jedem Weg maximal einmal vorkommt
- Betrachte  $i$ - $j$ -Weg, der  $n$  Knoten  $v_1, \dots, v_k$  enthält. Ein kürzester  $i$ - $j$ -Weg  $w$  mit  $v_k$  als  $k$ -ten Knoten  $w_k$  läuft:

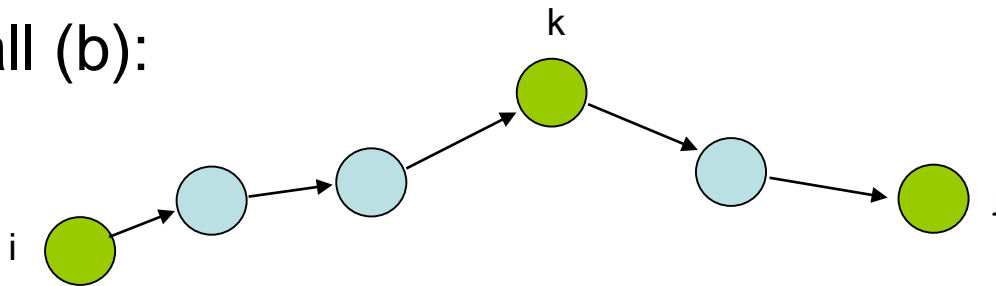


# Dynamische Programmierung - APSP

---

- Kürzester  $i$ - $j$ -Weg über Knoten aus  $\{1, \dots, k\}$  ist
- (a) kürzester  $i$ - $j$ -Weg über Knoten aus  $\{1, \dots, k-1\}$  oder
- (b) kürzester  $i$ - $k$ -Weg über Knoten aus  $\{1, \dots, k-1\}$  gefolgt von kürzestem  $k$ - $j$ -Weg über Knoten aus  $\{1, \dots, k-1\}$

Fall (b):



## Die Rekursion:

- Sei  $d_{ij}^{(k)}$  die Länge eines kürzesten  $i$ - $j$ -Wegs mit Knoten aus  $\{1, \dots, k\}$

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & , \text{ falls } k=0 \\ \min ( d_{ij}^{(k-1)} , d_{ik}^{(k-1)} + d_{kj}^{(k-1)} ) & , \text{ falls } k \geq 1 \end{cases}$$

- Matrix  $D^{(n)} = (d_{ij}^{(n)})$  enthält die gesuchte Lösung

# Dynamische Programmierung - APSP

---

Floyd-Warshall( $W, n$ )

1.  $D^{(0)} \leftarrow W$

2. **for**  $k \leftarrow 1$  **to**  $n$  **do**

3.   **for**  $i \leftarrow 1$  **to**  $n$  **do**

4.       **for**  $j \leftarrow 1$  **to**  $n$  **do**

5.            $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

6. **return**  $D^{(n)}$

# Dynamische Programmierung

---

## Generelle Vorgehensweise:

### 1. Aufstellung der Rekursionsgleichung

- Initialfall (z.B.  $\text{OPT}(i,0)=\text{OPT}(0,j)=0$  )
- Rekursion (z.B.  $\text{OPT}(i,j)=\dots$ )
- Optimaler Wert (z.B.  $\text{OPT} = \text{OPT}(n,W)$  )

### 2. Formulierung des dynamischen Programms

**Wichtig:** Berechnung so durchführen, dass in der Rekursion auf bereits berechnete Werte zurückgegriffen werden kann!

# Dynamische Programmierung

---

## Beispiel 1: Matrixketten-Multiplikation

Gegeben: Matrizen  $A_1, \dots, A_n$ , und Werte  $p_0, \dots, p_n \in \mathbb{N}$ , wobei Matrix  $A_i$  eine  $p_{i-1} \times p_i$ -Matrix ist.

Gesucht: Minimale Anzahl an Multiplikationen, um  $A_1 \cdot \dots \cdot A_n$  zu berechnen.

Beobachtung:

- Sei  $A_{i..j} = A_i \cdot \dots \cdot A_j$
- Dann kostet das Matrixprodukt  $A_{i..k} \cdot A_{k+1..j}$   $p_{i-1} \cdot p_k \cdot p_j$  viele Multiplikationen (mit der naiven Methode)

# Dynamische Programmierung

---

- $m[i,j]$ : minimale Anzahl an Multiplikationen, um  $A_{i..j}$  zu berechnen

Initialfall:

$$m[i,i]=0 \text{ für alle } i \in \{1, \dots, n\}.$$

Rekursion:

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

Optimaler Wert:

$$m[1,n]$$

Berechnung:

- Führe erst Initialfall aus
- Berechne  $m[i,j]$  für alle  $i < j$  mit  $|j-i|=d$ , angefangen mit  $d=1$



# Dynamische Programmierung

---

## Beispiel 2: Optimaler binärer Suchbaum

**Gegeben:** Schlüssel  $k_1, \dots, k_n$ , mit Zugriffswahrscheinlichkeiten  $p_1, \dots, p_n \in [0, 1]$ , so dass  $\sum_{i=1}^n p_i = 1$ .

**Gesucht:** Binärer Suchbaum  $T$  mit minimaler erwarteter Suchzeit, d.h.  $\sum_{i=1}^n p_i \cdot (\text{Tiefe}_T(k_i) + 1)$  ist minimal.

( $\text{Tiefe}_T(k)$ : Tiefe des Knotens mit Schlüssel  $k$  in  $T$   
(Tiefe der Wurzel ist 0) )

# Dynamische Programmierung

---

- $m[i,j]$ : minimale erwartete Suchzeit für einen Binärbaum mit den Schlüsseln  $k_i$  bis  $k_j$

Initialfall:

- $m[i,i-1]=0$  für alle  $i \in \{1, \dots, n\}$
- $m[i,i]=p_i$  für alle  $i \in \{1, \dots, n\}$

Rekursion:

$$m[i,j] = \min_{i \leq k \leq j} m[i,k-1] + m[k+1,j] + \sum_{l=i}^j p_l \quad \text{für alle } i < j$$

Optimaler Wert:

$$m[1,n]$$

Berechnung:

- Führe erst Initialfall aus
- Berechne  $m[i,j]$  für alle  $i < j$  mit  $|j-i|=d$ , angefangen mit  $d=1$
- Gib am Ende  $m[1,n]$  aus

# Dynamische Programmierung

---

## Beispiel 3: Längster einfacher Weg in einem gerichteten azyklischen Graph

**Gegeben:** Gerichteter azyklischer Graph  $G=(V,E)$  mit  $V=\{1,\dots,n\}$ , wobei die Knoten gemäß ihrer Nummern topologisch sortiert sind, d.h. für alle  $(v,w)\in E$  gilt  $v<w$ .

**Gesucht:** Länge des längsten einfachen (d.h. kreisfreien) Weges in  $G$ .

# Dynamische Programmierung

- $L[i,j]$ : Länge des längsten einfachen Weges von Knoten  $i$  nach Knoten  $j$  in  $G$
- $A[i,j] \in \{0,1\}$ : ist 1 genau dann wenn  $(i,j) \in E$

Initialfall:

- $L[i,i]=0$  für alle  $i \in \{1, \dots, n\}$

Falls die Menge leer ist, ist  $\max\{\dots\}=0$ .

Rekursion:

$$L[i,j] = \max \{ \max\{L[i,k] + L[k,j] \mid i < k < j \text{ und } L[i,k] > 0 \text{ und } L[k,j] > 0\}, A[i,j] \} \text{ für alle } i < j$$

Optimaler Wert:

$$\max_{i < j} L[i,j] \quad (=0: \text{ es gibt keine Kanten in } G)$$

Berechnung:

- Führe erst Initialfall aus
- Berechne  $L[i,j]$  für alle  $i < j$  mit  $|j-i|=d$ , angefangen mit  $d=1$
- Gib am Ende  $\max_{i < j} L[i,j]$  aus

# Dynamische Programmierung

---

## Beispiel 4: Fahrstuhloptimierung

**Gegeben:** Flure  $f_1, \dots, f_n \in \mathbb{N}$ , bei denen die  $n$  Kunden aussteigen wollen ( $f_1 \leq \dots \leq f_n$ ), und maximale Anzahl Haltepunkte  $k \in \mathbb{N}$ .

**Gesucht:** Minimale Anzahl an Fluren, die die Kunden bei maximal  $k$  Haltepunkten zu Fuß gehen müssen, um ihren Zielflur zu erreichen, wenn der Fahrstuhl bei Flur 1 startet.

# Dynamische Programmierung

---

- $m[i,j]$ : minimale Anzahl an Fluren, die die Kunden zu Fuß gehen müssen, wenn der Fahrstuhl genau  $j$ -mal hält und der höchste Halt Flur  $i$  ist.
- $flure[i,j]$ : Minimale Anzahl an Fluren, die Kunden mit Aussteigewunsch  $f$  mit  $i \leq f < j$  laufen müssen, wenn der Fahrstuhl nur bei Flur  $i$  und  $j$  aber nicht dazwischen hält. (Kann direkt berechnet werden.)

Initialfall:

$$m[i,1] = flure[1,i] + flure[i,\infty] \text{ für alle } i \in \{2, \dots, f_n\}$$

Rekursion:

$$m[i,j+1] = \min_{j+1 \leq l < i} m[l,j] - flure[l,\infty] + flure[l,i] + flure[i,\infty] \text{ für alle } i > j+1, j \geq 1$$

Optimaler Wert:

$$\min_{k+1 \leq i \leq f_n} m[i,k]$$

# Dynamische Programmierung

---

## Teile & Herrsche:

- Aufteilen der Eingabe in mehrere Unterprobleme
- Rekursives lösen der Unterprobleme
- Zusammenfügen

## Gierige Algorithmen:

- Konstruiere Lösung Schritt für Schritt
- In jedem Schritt optimiere einfaches, lokales Kriterium

## Dynamische Programmierung:

- Formuliere Problem rekursiv
- Vermeide mehrfache Berechnung von Teilergebnissen
- Verwende „bottom-up“ Implementierung

# Probleme

- 10131: Is Bigger Smarter?
- 103: Stacking Boxes
- 108: Maximum Sum
- 116: Unidirectional TSP
- 222: Budget Travel
- 10069: Distinct Subsequences

Hausaufgabe:

- 10003: Cutting Sticks