

SS09

# Effiziente Algorithmen

## 4. Kapitel:

# Dynamische Programmierung

Martin Dietzfelbinger

**Juni 2009**

---

# Kapitel 4: Dynamische Programmierung

Typische Eigenschaften des Ansatzes der

**Dynamischen Programmierung:**

Identifizierung von (vielen) **Teilproblemen**.

Identifizierung einfacher **Basisfälle**.

Formulierung einer Version der Eigenschaft

**Substrukturen optimaler Strukturen müssen optimal sein**

Daraus: Rekursionsgleichungen für Werte optimaler Lösungen:

**Bellman'sche Optimalitätsgleichungen**

Iterative Berechnung der optimalen Werte (und Strukturen).

---

## 4.1 Das All-Pairs-Shortest-Paths-Problem

(**APSP**-Pr.) (Zentrales Beispiel)

„Kürzeste Wege zwischen allen Knoten“.

---

## 4.1 Das All-Pairs-Shortest-Paths-Problem

(**APSP**-Pr.) (Zentrales Beispiel)

„Kürzeste Wege zwischen allen Knoten“.

**Eingabe:** Gerichteter Graph  $G = (V, E, c)$

mit  $V = \{1, \dots, n\}$  und  $E \subseteq \{(v, w) \mid 1 \leq v, w \leq n, v \neq w\}$ .

---

## 4.1 Das All-Pairs-Shortest-Paths-Problem

(APSP-Pr.) (Zentrales Beispiel)

„Kürzeste Wege zwischen allen Knoten“.

**Eingabe:** Gerichteter Graph  $G = (V, E, c)$

mit  $V = \{1, \dots, n\}$  und  $E \subseteq \{(v, w) \mid 1 \leq v, w \leq n, v \neq w\}$ .

$c: E \rightarrow \mathbb{R} \cup \{+\infty\}$ :

**Gewichts-/Kosten-/Längenfunktion.**

---

## 4.1 Das All-Pairs-Shortest-Paths-Problem

(APSP-Pr.) (Zentrales Beispiel)

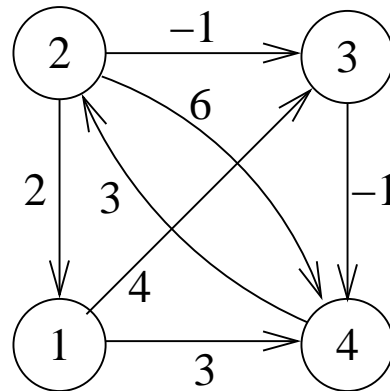
„Kürzeste Wege zwischen allen Knoten“.

**Eingabe:** Gerichteter Graph  $G = (V, E, c)$

mit  $V = \{1, \dots, n\}$  und  $E \subseteq \{(v, w) \mid 1 \leq v, w \leq n, v \neq w\}$ .

$c: E \rightarrow \mathbb{R} \cup \{+\infty\}$ :

**Gewichts-/Kosten-/Längenfunktion.**



---

**Länge** eines Wegs  $p = (v = v_0, v_1, \dots, v_r = w)$  ist

$$\sum_{1 \leq s \leq r} c(v_{s-1}, v_s).$$

---

**Länge** eines Wegs  $p = (v = v_0, v_1, \dots, v_r = w)$  ist

$$\sum_{1 \leq s \leq r} c(v_{s-1}, v_s).$$

**Gesucht:** für jedes Paar  $(v, w)$ ,  $1 \leq v, w \leq n$ :

Länge  $S(v, w)$  eines „kürzesten Weges“ von  $v$  nach  $w$ .

---

**Länge** eines Wegs  $p = (v = v_0, v_1, \dots, v_r = w)$  ist

$$\sum_{1 \leq s \leq r} c(v_{s-1}, v_s).$$

**Gesucht:** für jedes Paar  $(v, w)$ ,  $1 \leq v, w \leq n$ :

Länge  $S(v, w)$  eines „kürzesten Weges“ von  $v$  nach  $w$ .

**Algorithmus von Dijkstra** löst das

**Single-Source-Shortest-Paths-Problem (SSSP-Problem)**

**im Fall nicht negativer Kantengewichte**

---

**Länge** eines Wegs  $p = (v = v_0, v_1, \dots, v_r = w)$  ist

$$\sum_{1 \leq s \leq r} c(v_{s-1}, v_s).$$

**Gesucht:** für jedes Paar  $(v, w)$ ,  $1 \leq v, w \leq n$ :

Länge  $S(v, w)$  eines „kürzesten Weges“ von  $v$  nach  $w$ .

**Algorithmus von Dijkstra** löst das

**Single-Source-Shortest-Paths-Problem (SSSP-Problem)**

im Fall nicht negativer Kantengewichte

Hier: **Algorithmus von Floyd-Warshall**

---

**Länge** eines Wegs  $p = (v = v_0, v_1, \dots, v_r = w)$  ist

$$\sum_{1 \leq s \leq r} c(v_{s-1}, v_s).$$

**Gesucht:** für jedes Paar  $(v, w)$ ,  $1 \leq v, w \leq n$ :

Länge  $S(v, w)$  eines „kürzesten Weges“ von  $v$  nach  $w$ .

**Algorithmus von Dijkstra** löst das

**Single-Source-Shortest-Paths-Problem (SSSP-Problem)**

im Fall nicht negativer Kantengewichte

Hier: **Algorithmus von Floyd-Warshall**

Wir verlangen:

Es darf keine **Kreise** mit negativem Gesamtgewicht geben:

$$(*) \quad v = v_0, v_1, \dots, v_r = v \text{ Kreis} \Rightarrow \sum_{1 \leq s \leq r} c(v_{s-1}, v_s) \geq 0.$$

---

Grund: Wenn es einen solchen Kreis von  $v$  nach  $v$  gibt, und irgendein Weg von  $v$  nach  $w$  existiert, dann gibt es Wege von  $v$  nach  $w$  mit beliebig stark negativer Länge – die Frage nach einem kürzesten Weg ist sinnlos.

---

Grund: Wenn es einen solchen Kreis von  $v$  nach  $v$  gibt, und irgendein Weg von  $v$  nach  $w$  existiert, dann gibt es Wege von  $v$  nach  $w$  mit beliebig stark negativer Länge – die Frage nach einem kürzesten Weg ist sinnlos.

**Konsequenzen:** (1) Wenn  $p$  Weg von  $v$  nach  $w$  ist, dann existiert ein Weg  $p'$  von  $v$  nach  $w$ , der nicht länger ist als  $p$  und auf dem sich keine Knoten wiederholen.

---

Grund: Wenn es einen solchen Kreis von  $v$  nach  $v$  gibt, und irgendein Weg von  $v$  nach  $w$  existiert, dann gibt es Wege von  $v$  nach  $w$  mit beliebig stark negativer Länge – die Frage nach einem kürzesten Weg ist sinnlos.

**Konsequenzen:** (1) Wenn  $p$  Weg von  $v$  nach  $w$  ist, dann existiert ein Weg  $p'$  von  $v$  nach  $w$ , der nicht länger ist als  $p$  und auf dem sich keine Knoten wiederholen.

( $u, \dots, u$ -Segmente aus  $p$  herausschneiden, gegebenenfalls wiederholt. Hierdurch kann sich der Weg nicht verlängern.)

---

Grund: Wenn es einen solchen Kreis von  $v$  nach  $v$  gibt, und irgendein Weg von  $v$  nach  $w$  existiert, dann gibt es Wege von  $v$  nach  $w$  mit beliebig stark negativer Länge – die Frage nach einem kürzesten Weg ist sinnlos.

**Konsequenzen:** (1) Wenn  $p$  Weg von  $v$  nach  $w$  ist, dann existiert ein Weg  $p'$  von  $v$  nach  $w$ , der nicht länger ist als  $p$  und auf dem sich keine Knoten wiederholen.

( $u, \dots, u$ -Segmente aus  $p$  herausschneiden, gegebenenfalls wiederholt. Hierdurch kann sich der Weg nicht verlängern.)

(2) Wenn es einen Weg von  $v$  nach  $w$  gibt, dann auch einen mit minimaler Länge (einen „**kürzesten Weg**“).

---

Grund: Wenn es einen solchen Kreis von  $v$  nach  $v$  gibt, und irgendein Weg von  $v$  nach  $w$  existiert, dann gibt es Wege von  $v$  nach  $w$  mit beliebig stark negativer Länge – die Frage nach einem kürzesten Weg ist sinnlos.

**Konsequenzen:** (1) Wenn  $p$  Weg von  $v$  nach  $w$  ist, dann existiert ein Weg  $p'$  von  $v$  nach  $w$ , der nicht länger ist als  $p$  und auf dem sich keine Knoten wiederholen.

( $u, \dots, u$ -Segmente aus  $p$  herausschneiden, gegebenenfalls wiederholt. Hierdurch kann sich der Weg nicht verlängern.)

(2) Wenn es einen Weg von  $v$  nach  $w$  gibt, dann auch einen mit minimaler Länge (einen „**kürzesten Weg**“).

**Grund:** Wegen (1) muss man nur Wege mit nicht mehr als  $n - 1$  Kanten betrachten; davon gibt es nur endlich viele.

---

Grund: Wenn es einen solchen Kreis von  $v$  nach  $v$  gibt, und irgendein Weg von  $v$  nach  $w$  existiert, dann gibt es Wege von  $v$  nach  $w$  mit beliebig stark negativer Länge – die Frage nach einem kürzesten Weg ist sinnlos.

**Konsequenzen:** (1) Wenn  $p$  Weg von  $v$  nach  $w$  ist, dann existiert ein Weg  $p'$  von  $v$  nach  $w$ , der nicht länger ist als  $p$  und auf dem sich keine Knoten wiederholen.

( $u, \dots, u$ -Segmente aus  $p$  herausschneiden, gegebenenfalls wiederholt. Hierdurch kann sich der Weg nicht verlängern.)

(2) Wenn es einen Weg von  $v$  nach  $w$  gibt, dann auch einen mit minimaler Länge (einen „**kürzesten Weg**“).

**Grund:** Wegen (1) muss man nur Wege mit nicht mehr als  $n - 1$  Kanten betrachten; davon gibt es nur endlich viele.

Es kann aber mehrere „kürzeste Wege“ von  $v$  nach  $w$  geben!

---

**Ausgabe:** Matrix  $(S(v, w))_{1 \leq v, w \leq n}$  mit

$S(v, w)$  = Länge eines kürzesten Weges (gerichtet)  
von  $v$  nach  $w$ , für **alle**  $(v, w) \in V \times V$ .

---

**Ausgabe:** Matrix  $(S(v, w))_{1 \leq v, w \leq n}$  mit

$S(v, w)$  = Länge eines kürzesten Weges (gerichtet)  
von  $v$  nach  $w$ , für **alle**  $(v, w) \in V \times V$ .

Falls in  $G$  kein Weg von  $v$  nach  $w$  existiert, soll  $S(v, w) = \infty$  sein.

---

O.B.d.A.:  $E = V \times V - \{(v, v) \mid v \in V\}$

Nichtvorhandene Kanten  $(v, w)$  werden mit  $c(v, w) = \infty$  repräsentiert.

---

O.B.d.A.:  $E = V \times V - \{(v, v) \mid v \in V\}$

Nichtvorhandene Kanten  $(v, w)$  werden mit  $c(v, w) = \infty$  repräsentiert.

**Erster Schritt:** Identifiziere geeignete **Teilprobleme**.

---

O.B.d.A.:  $E = V \times V - \{(v, v) \mid v \in V\}$

Nichtvorhandene Kanten  $(v, w)$  werden mit  $c(v, w) = \infty$  repräsentiert.

**Erster Schritt:** Identifiziere geeignete **Teilprobleme**.

Gegeben ein  $k$ ,  $0 \leq k \leq n$ , betrachte Wege von  $v$  nach  $w$ , die **unterwegs** (also vom Start- und Endpunkt abgesehen) nur Knoten in  $\{1, \dots, k\}$  besuchen.

---

O.B.d.A.:  $E = V \times V - \{(v, v) \mid v \in V\}$

Nichtvorhandene Kanten  $(v, w)$  werden mit  $c(v, w) = \infty$  repräsentiert.

**Erster Schritt:** Identifiziere geeignete **Teilprobleme**.

Gegeben ein  $k$ ,  $0 \leq k \leq n$ , betrachte Wege von  $v$  nach  $w$ , die **unterwegs** (also vom Start- und Endpunkt abgesehen) nur Knoten in  $\{1, \dots, k\}$  besuchen.

$k = 0$ : Kein Knoten darf unterwegs besucht werden; es handelt sich nur um die Kanten  $(v, w)$  mit  $c(v, w) < \infty$ , oder um Wege  $(v, v)$  der Länge 0.

---

O.B.d.A.:  $E = V \times V - \{(v, v) \mid v \in V\}$

Nichtvorhandene Kanten  $(v, w)$  werden mit  $c(v, w) = \infty$  repräsentiert.

**Erster Schritt:** Identifiziere geeignete **Teilprobleme**.

Gegeben ein  $k$ ,  $0 \leq k \leq n$ , betrachte Wege von  $v$  nach  $w$ , die **unterwegs** (also vom Start- und Endpunkt abgesehen) nur Knoten in  $\{1, \dots, k\}$  besuchen.

$k = 0$ : Kein Knoten darf unterwegs besucht werden; es handelt sich nur um die Kanten  $(v, w)$  mit  $c(v, w) < \infty$ , oder um Wege  $(v, v)$  der Länge 0.

$k = 1$ : Es kommen nur Wege in Frage, die aus der Kante  $(v, w)$  bestehen oder aus den Kanten  $(v, 1)$  und  $(1, w)$ .

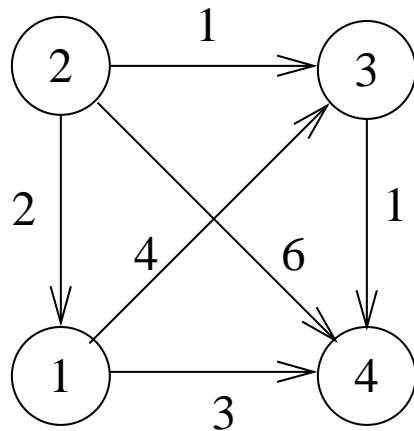
USW.

---

$S(v, w, k) :=$  Länge eines kürzesten Weges von  $v$  nach  $w$ , der nur Zwischenknoten in  $\{1, \dots, k\}$  benutzt. (Falls kein solcher Weg existiert:  $S(v, w, k) := \infty$ )

$S(v, w, k) :=$  Länge eines kürzesten Weges von  $v$  nach  $w$ , der nur Zwischenknoten in  $\{1, \dots, k\}$  benutzt. (Falls kein solcher Weg existiert:  $S(v, w, k) := \infty$ )

*Beispiel:* (Die nicht eingezeichneten Kanten haben Länge  $\infty$ .)



$$S(2,4,0) = 6$$

$$S(2,4,1) = 5$$

$$S(2,4,2) = 5$$

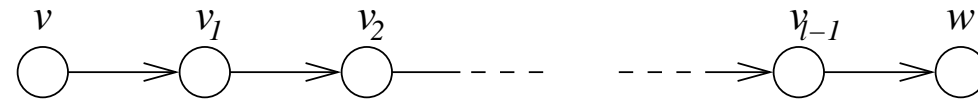
$$S(2,4,3) = 2$$

↓  
monoton  
fallend

---

„Bellman'sche Optimalitätsgleichungen“:

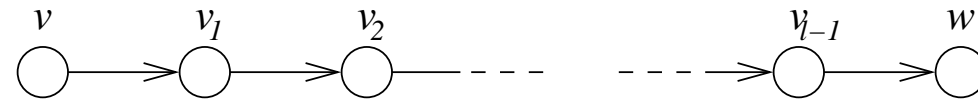
Betrachte Weg  $p$  von  $v$  nach  $w$ ; dabei seien die Zwischenknoten  $v_1, \dots, v_{l-1}$  aus  $\{1, \dots, k\}$ :



---

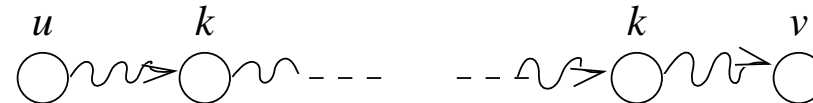
„Bellman'sche Optimalitätsgleichungen“:

Betrachte Weg  $p$  von  $v$  nach  $w$ ; dabei seien die Zwischenknoten  $v_1, \dots, v_{l-1}$  aus  $\{1, \dots, k\}$ :



Angenommen, dieser Weg ist optimal. Dann gilt:

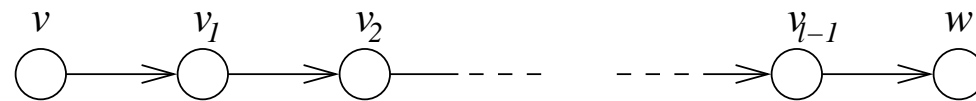
- 1) (O.B.d.A.) Der Knoten  $k$  kommt auf dem Weg höchstens einmal vor. – Sonst: ersetze ein Teilstück



---

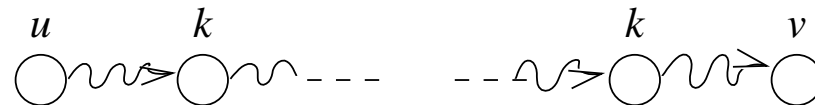
„Bellman'sche Optimalitätsgleichungen“:

Betrachte Weg  $p$  von  $v$  nach  $w$ ; dabei seien die Zwischenknoten  $v_1, \dots, v_{l-1}$  aus  $\{1, \dots, k\}$ :

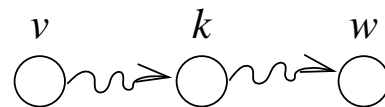


Angenommen, dieser Weg ist optimal. Dann gilt:

- 1) (O.B.d.A.) Der Knoten  $k$  kommt auf dem Weg höchstens einmal vor. – Sonst: ersetze ein Teilstück



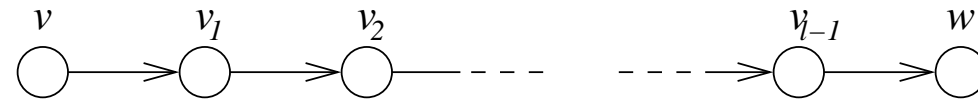
mit  $u, v \in V$  durch



---

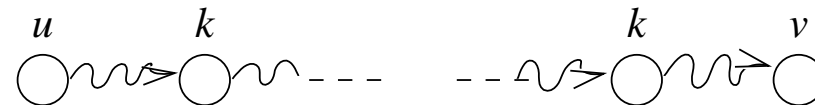
„Bellman'sche Optimalitätsgleichungen“:

Betrachte Weg  $p$  von  $v$  nach  $w$ ; dabei seien die Zwischenknoten  $v_1, \dots, v_{l-1}$  aus  $\{1, \dots, k\}$ :

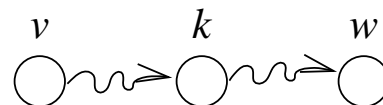


Angenommen, dieser Weg ist optimal. Dann gilt:

- 1) (O.B.d.A.) Der Knoten  $k$  kommt auf dem Weg höchstens einmal vor. – Sonst: ersetze ein Teilstück



mit  $u, v \in V$  durch



Der neue Weg ist dann nicht länger als  $p$ .

---

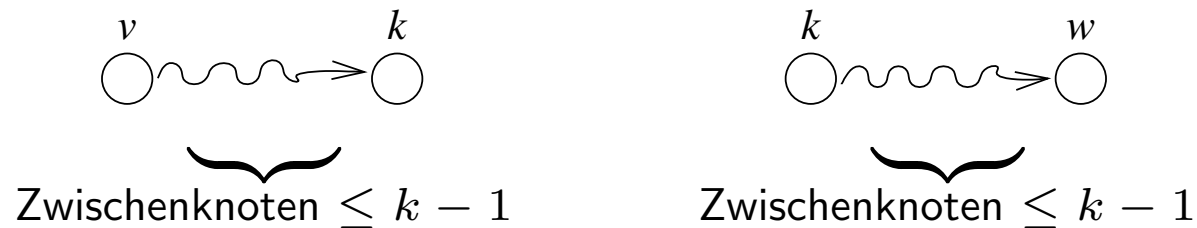
2) Falls der Knoten  $k$  auf dem Weg **nicht** vorkommt,

---

2) Falls der Knoten  $k$  auf dem Weg **nicht** vorkommt, ist  $p$  für Zwischenknoten  $1, \dots, k - 1$  optimal.

- 
- 2) Falls der Knoten  $k$  auf dem Weg **nicht** vorkommt, ist  $p$  für Zwischenknoten  $1, \dots, k - 1$  optimal.
  - 3) Falls der Knoten  $k$  auf  $p$  **vorkommt**,

- 
- 2) Falls der Knoten  $k$  auf dem Weg **nicht** vorkommt, ist  $p$  für Zwischenknoten  $1, \dots, k - 1$  optimal.
- 3) Falls der Knoten  $k$  auf  $p$  **vorkommt**, zerfällt  $p$  in zwei Teile



die beide bezüglich der Zwischenknotenmenge  $\{1, \dots, k - 1\}$  optimal sind

- 
- 2) Falls der Knoten  $k$  auf dem Weg **nicht** vorkommt, ist  $p$  für Zwischenknoten  $1, \dots, k - 1$  optimal.
- 3) Falls der Knoten  $k$  auf  $p$  **vorkommt**, zerfällt  $p$  in zwei Teile



die beide bezüglich der Zwischenknotenmenge  $\{1, \dots, k - 1\}$  optimal sind (sonst könnte ein Teilweg durch einen billigeren ersetzt werden, Widerspruch zur Optimalität von  $p$ ).

---

Punkte 2) und 3) drücken aus, dass Substrukturen einer optimalen Lösung selbst wieder optimal sind.

---

Punkte 2) und 3) drücken aus, dass Substrukturen einer optimalen Lösung selbst wieder optimal sind.

Die **Bellman'schen Optimalitätsgleichungen** für den Algorithmus von Floyd-Warshall lauten dann:

---

Punkte 2) und 3) drücken aus, dass Substrukturen einer optimalen Lösung selbst wieder optimal sind.

Die **Bellman'schen Optimalitätsgleichungen** für den Algorithmus von Floyd-Warshall lauten dann:

$$\begin{aligned} S(v, w, k) \\ &= \min\{S(v, w, k - 1), S(v, k, k - 1) + S(k, w, k - 1)\}, \\ &\text{für } 1 \leq v, w \leq n, 1 \leq k \leq n. \end{aligned}$$

---

Punkte 2) und 3) drücken aus, dass Substrukturen einer optimalen Lösung selbst wieder optimal sind.

Die **Bellman'schen Optimalitätsgleichungen** für den Algorithmus von Floyd-Warshall lauten dann:

$$\begin{aligned} S(v, w, k) \\ &= \min\{S(v, w, k - 1), S(v, k, k - 1) + S(k, w, k - 1)\}, \\ &\text{für } 1 \leq v, w \leq n, 1 \leq k \leq n. \end{aligned}$$

**Basisfälle:**  $S(v, v, 0) = 0$ .

---

Punkte 2) und 3) drücken aus, dass Substrukturen einer optimalen Lösung selbst wieder optimal sind.

Die **Bellman'schen Optimalitätsgleichungen** für den Algorithmus von Floyd-Warshall lauten dann:

$$\begin{aligned} S(v, w, k) \\ &= \min\{S(v, w, k - 1), S(v, k, k - 1) + S(k, w, k - 1)\}, \\ &\text{für } 1 \leq v, w \leq n, 1 \leq k \leq n. \end{aligned}$$

**Basisfälle:**  $S(v, v, 0) = 0$ .

$S(v, w, 0) = c(v, w)$ , für  $v \neq w$ .

---

Wir beschreiben einen **iterativen** Algorithmus.

---

Wir beschreiben einen **iterativen** Algorithmus. Zur Aufbewahrung der Zwischenergebnisse benutzen wir ein Array  $S[1..n, 1..n, 0..n]$ , das durch

$$S[v, w, 0] := c(v, w), \quad 1 \leq v, w \leq n, \quad v \neq w;$$

$$S[v, v, 0] := 0, \quad 1 \leq v \leq n$$

initialisiert wird.

---

Wir beschreiben einen **iterativen** Algorithmus. Zur Aufbewahrung der Zwischenergebnisse benutzen wir ein Array  $S[1..n, 1..n, 0..n]$ , das durch

$$S[v, w, 0] := c(v, w), \quad 1 \leq v, w \leq n, \quad v \neq w;$$

$$S[v, v, 0] := 0, \quad 1 \leq v \leq n$$

initialisiert wird. Unser Algorithmus füllt das Array  $S$  gemäß wachsendem  $k$  aus:

---

Wir beschreiben einen **iterativen** Algorithmus. Zur Aufbewahrung der Zwischenergebnisse benutzen wir ein Array  $S[1..n, 1..n, 0..n]$ , das durch

$$S[v, w, 0] := c(v, w), \quad 1 \leq v, w \leq n, \quad v \neq w;$$

$$S[v, v, 0] := 0, \quad 1 \leq v \leq n$$

initialisiert wird. Unser Algorithmus füllt das Array  $S$  gemäß wachsendem  $k$  aus:

```
for k from 1 to n do
  for v from 1 to n do
    for w from 1 to n do
       $S[v, w, k] := \min\{S[v, w, k - 1], S[v, k, k - 1] + S[k, w, k - 1]\}.$ 
```

---

**Korrektheit:** Folgt aus der Vorüberlegung.

Es werden genau die Werte  $S(v, w, k)$  gemäß den Bellman'schen Optimalitätsgleichungen ausgerechnet.

---

**Korrektheit:** Folgt aus der Vorüberlegung.

Es werden genau die Werte  $S(v, w, k)$  gemäß den Bellman'schen Optimalitätsgleichungen ausgerechnet.

**Laufzeit:** Drei geschachtelte Schleifen:  $\Theta(n^3)$ .

---

**Korrektheit:** Folgt aus der Vorüberlegung.

Es werden genau die Werte  $S(v, w, k)$  gemäß den Bellman'schen Optimalitätsgleichungen ausgerechnet.

**Laufzeit:** Drei geschachtelte Schleifen:  $\Theta(n^3)$ .

Bei dieser simplen Implementierung wird noch Platz verschwendet, da für den  $k$ -ten Schleifendurchlauf offenbar nur die  $(k - 1)$ -Komponenten des Arrays  $S$  benötigt werden.

---

**Korrektheit:** Folgt aus der Vorüberlegung.

Es werden genau die Werte  $S(v, w, k)$  gemäß den Bellman'schen Optimalitätsgleichungen ausgerechnet.

**Laufzeit:** Drei geschachtelte Schleifen:  $\Theta(n^3)$ .

Bei dieser simplen Implementierung wird noch Platz verschwendet, da für den  $k$ -ten Schleifendurchlauf offenbar nur die  $(k - 1)$ -Komponenten des Arrays  $S$  benötigt werden.

Man überlegt sich leicht, dass man mit zwei Matrizen auskommt, eine „alte“ ( $(k - 1)$ -Version) und eine „neue“ ( $k$ -Version), zwischen denen dann immer passend umzuschalten ist.

---

**Korrektheit:** Folgt aus der Vorüberlegung.

Es werden genau die Werte  $S(v, w, k)$  gemäß den Bellman'schen Optimalitätsgleichungen ausgerechnet.

**Laufzeit:** Drei geschachtelte Schleifen:  $\Theta(n^3)$ .

Bei dieser simplen Implementierung wird noch Platz verschwendet, da für den  $k$ -ten Schleifendurchlauf offenbar nur die  $(k - 1)$ -Komponenten des Arrays  $S$  benötigt werden.

Man überlegt sich leicht, dass man mit zwei Matrizen auskommt, eine „alte“ ( $(k - 1)$ -Version) und eine „neue“ ( $k$ -Version), zwischen denen dann immer passend umzuschalten ist.

Der Platzaufwand beträgt damit  $O(n^2)$ , der Zeitbedarf offensichtlich immer noch  $O(n^3)$ .

---

Dieser Spezialfall des Ansatzes „dynamische Programmierung“ lässt noch eine weitere Verbesserung zu. Es gilt

$$(*) \quad S(v, k, k) = S(v, k, k - 1) \text{ und} \\ S(k, w, k) = S(k, w, k - 1), \text{ für } 1 \leq v, w, k \leq n,$$

da wiederholtes Benutzen des Knotens  $k$  den Weg von  $v$  nach  $k$  bzw. von  $k$  nach  $w$  nicht verkürzen kann (keine Kreise mit negativer Gesamtlänge).

---

Dieser Spezialfall des Ansatzes „dynamische Programmierung“ lässt noch eine weitere Verbesserung zu. Es gilt

$$(*) \quad \begin{aligned} S(v, k, k) &= S(v, k, k - 1) \text{ und} \\ S(k, w, k) &= S(k, w, k - 1), \text{ für } 1 \leq v, w, k \leq n, \end{aligned}$$

da wiederholtes Benutzen des Knotens  $k$  den Weg von  $v$  nach  $k$  bzw. von  $k$  nach  $w$  nicht verkürzen kann (keine Kreise mit negativer Gesamtlänge).

Man kann also für die Berechnung von  $S(v, w, k)$  den „alten“ oder den „neuen“ Wert im S-Array benutzen. Damit erübrigt es sich, „altes“ und „neues“ Array zu unterscheiden, und der dritte Index  $k$  kann wegfallen:

```
for k from 1 to n do
  for v from 1 to n do
    for w from 1 to n do
      S[v,w] := min{S[v,w], S[v,k] + S[k,w]}.
```

---

Man möchte nicht nur den **Wert** des kürzesten Weges kennenlernen, sondern auch zu gegebenen  $v, w$  diesen Weg **konstruieren** können. Hierfür benutzt man ein Hilfsarray  $I[1..n, 1..n]$ , in dem für jedes Paar  $v, w$  die Information gehalten wird, welches der bis zur Runde  $k$  gefundene „größte“ Knoten ist, der für die Konstruktion eines kürzesten Weges von  $v$  nach  $w$  benötigt worden ist.

Dies liefert den Floyd-Warshall-Algorithmus.

---

### 4.1.1 Algorithmus Floyd-Warshall( $C[1..n,1..n]$ )

**Eingabe:**  $C[1..n,1..n]$ : Matrix der Kantenkosten/längen

**Ausgabe:**  $S[1..n,1..n]$ : Kosten der kürzesten Wege

$I[1..n,1..n]$ : Hilfsarray zur Konstruktion der kürzesten Wege

**Datenstruktur:** Matrizen  $S[1..n,1..n]$ ,  $I[1..n,1..n]$

Ziel: (\*  $S[v,w]$  enthält  $S(v,w)$  \*)

(\*  $I[v,w]$ : möglichst kleiner max. Knoten auf kürzestem  $(v,w)$ -Weg \*)

```
(1)  for v from 1 to n do
(2)    for w from 1 to n do
(3)      if v = w then S[v,w] ← 0; I[v,w] ← 0
(4)      else S[v,w] ← C[v,w];
(5)      if S[v,w] < ∞ then I[v,w] ← 0 else I[v,w] ← -1;
(6)  for k from 1 to n do
(7)    for v from 1 to n do
(8)      for w from 1 to n do
(9)        if S[v,k] + S[k,w] < S[v,w] then
(10)         S[v,w] ← S[v,k] + S[k,w]; I[v,w] ← k;
(11)  Ausgabe: S[1..n,1..n] und I[1..n,1..n].
```

---

**Korrektheit:** Es gilt folgende Schleifeninvariante: Nach dem Schleifendurchlauf  $k$  ist  $S[v, w] = S(v, w, k)$ .

(Beweis durch vollständige Induktion, unter Ausnutzung der Bellman-Gleichungen und von (\*).)

Zudem:  $I[v, w]$  ist das kleinste  $\ell$  mit der Eigenschaft, dass es unter den

**kürzesten Wegen von  $v$  nach  $w$  über  $1, \dots, k$**

einen gibt, dessen **maximaler** Eintrag  $\ell$  ist.

(Beweis durch Induktion über  $k$ .)

Ein kürzester Weg von  $v$  nach  $w$  kann dann mit einer einfachen rekursiven Prozedur „printPathInner“ rekonstruiert werden (Übung).

---

## 4.1.2 Satz

Der Algorithmus von Floyd-Warshall löst das „All-Pairs-Shortest-Paths- Problem“ in Zeit  $O(n^3)$  und Platz  $O(n^2)$ .

---

## 4.2 Der Bellman-Ford-Algorithmus

**Zweck:** Kürzeste Wege von einem Startknoten aus.

Im Unterschied zum Algorithmus von Dijkstra dürfen negative Kanten vorhanden sein;

Algorithmus scheitert, wenn vom Startknoten  $s$  aus ein Kreis mit negativer Gesamtlänge erreichbar ist.

Details: Tafel.

---

## 4.3 Das 0-1-Rucksackproblem

**Problemstellung:**

**Eingabe:** Ganzzahlige **Volumina**  $a_1, \dots, a_n > 0$ , **Nutzenwerte**  $c_1, \dots, c_n > 0$ , ganzzahlige Volumenschranke  $b$ .

---

## 4.3 Das 0-1-Rucksackproblem

### Problemstellung:

**Eingabe:** Ganzzahlige **Volumina**  $a_1, \dots, a_n > 0$ , **Nutzenwerte**  $c_1, \dots, c_n > 0$ , ganzzahlige Volumenschranke  $b$ .

**Ausgabe:**  $I \subseteq \{1, \dots, n\}$  derart, dass  $\sum_{i \in I} a_i \leq b$  und  $\sum_{i \in I} c_i$  möglichst groß.

---

## 4.3 Das 0-1-Rucksackproblem

### Problemstellung:

**Eingabe:** Ganzzahlige **Volumina**  $a_1, \dots, a_n > 0$ , **Nutzenwerte**  $c_1, \dots, c_n > 0$ , ganzzahlige Volumenschranke  $b$ .

**Ausgabe:**  $I \subseteq \{1, \dots, n\}$  derart, dass  $\sum_{i \in I} a_i \leq b$  und  $\sum_{i \in I} c_i$  möglichst groß.

Im Gegensatz zum **fraktionalen Rucksackproblem** (Kap. 3.1) ist es hier nicht erlaubt, Objekte zu teilen.

---

## 4.3 Das 0-1-Rucksackproblem

### Problemstellung:

**Eingabe:** Ganzzahlige **Volumina**  $a_1, \dots, a_n > 0$ , **Nutzenwerte**  $c_1, \dots, c_n > 0$ , ganzzahlige Volumenschranke  $b$ .

**Ausgabe:**  $I \subseteq \{1, \dots, n\}$  derart, dass  $\sum_{i \in I} a_i \leq b$  und  $\sum_{i \in I} c_i$  möglichst groß.

Im Gegensatz zum **fraktionalen Rucksackproblem** (Kap. 3.1) ist es hier nicht erlaubt, Objekte zu teilen.

Der Lösungsvektor  $(x_1, \dots, x_n)$  muss also **0-1-wertig** sein.

---

Ansatz: Dynamische Programmierung.

Identifizierung von nützlichen **Teilproblemen**:

---

Ansatz: Dynamische Programmierung.

Identifizierung von nützlichen **Teilproblemen**:

Für  $1 \leq k \leq n$  und  $0 \leq v \leq b$  definieren wir

$$m(k, v) := \max \left\{ \sum_{i \in I} c_i \mid I \subseteq \{1, \dots, k\} \wedge \sum_{i \in I} a_i \leq v \right\}.$$

---

Ansatz: Dynamische Programmierung.

Identifizierung von nützlichen **Teilproblemen**:

Für  $1 \leq k \leq n$  und  $0 \leq v \leq b$  definieren wir

$$m(k, v) := \max \left\{ \sum_{i \in I} c_i \mid I \subseteq \{1, \dots, k\} \wedge \sum_{i \in I} a_i \leq v \right\}.$$

Das Teilproblem **P(k, v)** besteht darin, ein  $I \subseteq \{1, \dots, k\}$  mit  $\sum_{i \in I} a_i \leq v$  zu finden, das  $\sum_{i \in I} c_i$  maximiert.

---

Ansatz: Dynamische Programmierung.

Identifizierung von nützlichen **Teilproblemen**:

Für  $1 \leq k \leq n$  und  $0 \leq v \leq b$  definieren wir

$$m(k, v) := \max \left\{ \sum_{i \in I} c_i \mid I \subseteq \{1, \dots, k\} \wedge \sum_{i \in I} a_i \leq v \right\}.$$

Das Teilproblem **P(k, v)** besteht darin, ein  $I \subseteq \{1, \dots, k\}$  mit  $\sum_{i \in I} a_i \leq v$  zu finden, das  $\sum_{i \in I} c_i$  maximiert.

Wir suchen also nach einer Auswahl, die nur **Objekte Nummer 1, ..., k** benutzt, eine modifizierte **Gewichtsschranke v** einhält, und den Nutzen maximiert.

---

Ansatz: Dynamische Programmierung.

Identifizierung von nützlichen **Teilproblemen**:

Für  $1 \leq k \leq n$  und  $0 \leq v \leq b$  definieren wir

$$m(k, v) := \max \left\{ \sum_{i \in I} c_i \mid I \subseteq \{1, \dots, k\} \wedge \sum_{i \in I} a_i \leq v \right\}.$$

Das Teilproblem **P(k, v)** besteht darin, ein  $I \subseteq \{1, \dots, k\}$  mit  $\sum_{i \in I} a_i \leq v$  zu finden, das  $\sum_{i \in I} c_i$  maximiert.

Wir suchen also nach einer Auswahl, die nur **Objekte Nummer 1, ..., k** benutzt, eine modifizierte **Gewichtsschranke v** einhält, und den Nutzen maximiert.

Trivial:  $m(0, v) = 0$  für alle  $v$ .

---

Ansatz: Dynamische Programmierung.

Identifizierung von nützlichen **Teilproblemen**:

Für  $1 \leq k \leq n$  und  $0 \leq v \leq b$  definieren wir

$$m(k, v) := \max \left\{ \sum_{i \in I} c_i \mid I \subseteq \{1, \dots, k\} \wedge \sum_{i \in I} a_i \leq v \right\}.$$

Das Teilproblem **P(k, v)** besteht darin, ein  $I \subseteq \{1, \dots, k\}$  mit  $\sum_{i \in I} a_i \leq v$  zu finden, das  $\sum_{i \in I} c_i$  maximiert.

Wir suchen also nach einer Auswahl, die nur **Objekte Nummer 1, ..., k** benutzt, eine modifizierte **Gewichtsschranke v** einhält, und den Nutzen maximiert.

Trivial:  $m(0, v) = 0$  für alle  $v$ .

$P(n, b)$  ist das Gesamtproblem.

---

Eigenschaft „**Optimale Substruktur**“:

---

Eigenschaft „**Optimale Substruktur**“:

Sei  $I$  optimale Lösung für  $P(k, v)$ , d. h.:

$$m(k, v) = \sum_{i \in I} c_i, \quad I \subseteq \{1, \dots, k\}, \quad \sum_{i \in I} a_i \leq v.$$

---

Eigenschaft „**Optimale Substruktur**“:

Sei  $I$  optimale Lösung für  $P(k, v)$ , d. h.:

$$m(k, v) = \sum_{i \in I} c_i, \quad I \subseteq \{1, \dots, k\}, \quad \sum_{i \in I} a_i \leq v.$$

Dann tritt einer der folgenden Fälle ein.

---

## Eigenschaft „**Optimale Substruktur**“:

Sei  $I$  optimale Lösung für  $P(k, v)$ , d. h.:

$$m(k, v) = \sum_{i \in I} c_i, \quad I \subseteq \{1, \dots, k\}, \quad \sum_{i \in I} a_i \leq v.$$

Dann tritt einer der folgenden Fälle ein.

**1. Fall:**  $k \in I$ . – Dann ist  $I - \{k\}$  optimale Lösung für  $P(k - 1, v - a_k)$ , d. h.:

$$m(k, v) - c_k = \sum_{i \in I - \{k\}} c_i = m(k - 1, v - a_k).$$

---

## Eigenschaft „**Optimale Substruktur**“:

Sei  $I$  optimale Lösung für  $P(k, v)$ , d. h.:

$$m(k, v) = \sum_{i \in I} c_i, \quad I \subseteq \{1, \dots, k\}, \quad \sum_{i \in I} a_i \leq v.$$

Dann tritt einer der folgenden Fälle ein.

**1. Fall:**  $k \in I$ . – Dann ist  $I - \{k\}$  optimale Lösung für  $P(k - 1, v - a_k)$ , d. h.:

$$m(k, v) - c_k = \sum_{i \in I - \{k\}} c_i = m(k - 1, v - a_k).$$

**2. Fall:**  $k \notin I$ . – Dann ist  $I$  optimale Lösung für  $P(k - 1, v)$ , d. h.:

$$m(k, v) = m(k - 1, v).$$

---

*Beweis:* **Indirekt.**

---

*Beweis:* **Indirekt.**

**1. Fall:**  $k \in I$ .

---

*Beweis:* **Indirekt.**

**1. Fall:**  $k \in I$ . – **Annahme:**  $I - \{k\}$  nicht optimal für  $P(k - 1, v - a_k)$ .

---

*Beweis:* **Indirekt.**

**1. Fall:**  $k \in I$ . – **Annahme:**  $I - \{k\}$  nicht optimal für  $P(k - 1, v - a_k)$ .

Dann gäbe es eine bessere Teilmenge  $J \subseteq \{1, \dots, k - 1\}$

---

*Beweis:* **Indirekt.**

**1. Fall:**  $k \in I$ . – **Annahme:**  $I - \{k\}$  nicht optimal für  $P(k - 1, v - a_k)$ .

Dann gäbe es eine bessere Teilmenge  $J \subseteq \{1, \dots, k - 1\}$  mit  $\sum_{i \in J} a_i \leq v - a_k$  und  $\sum_{i \in J} c_i > \sum_{i \in I - \{k\}} c_i$ .

---

*Beweis:* **Indirekt.**

**1. Fall:**  $k \in I$ . – **Annahme:**  $I - \{k\}$  nicht optimal für  $P(k - 1, v - a_k)$ .

Dann gäbe es eine bessere Teilmenge  $J \subseteq \{1, \dots, k - 1\}$  mit  $\sum_{i \in J} a_i \leq v - a_k$  und  $\sum_{i \in J} c_i > \sum_{i \in I - \{k\}} c_i$ .

Dann wäre aber  $J \cup \{k\}$  eine bessere Lösung für  $P(k, v)$  als  $I$

---

*Beweis:* **Indirekt.**

**1. Fall:**  $k \in I$ . – **Annahme:**  $I - \{k\}$  nicht optimal für  $P(k - 1, v - a_k)$ .

Dann gäbe es eine bessere Teilmenge  $J \subseteq \{1, \dots, k - 1\}$  mit  $\sum_{i \in J} a_i \leq v - a_k$  und  $\sum_{i \in J} c_i > \sum_{i \in I - \{k\}} c_i$ .

Dann wäre aber  $J \cup \{k\}$  eine bessere Lösung für  $P(k, v)$  als  $I$ , **Widerspruch.**

---

*Beweis:* **Indirekt.**

**1. Fall:**  $k \in I$ . – **Annahme:**  $I - \{k\}$  nicht optimal für  $P(k - 1, v - a_k)$ .

Dann gäbe es eine bessere Teilmenge  $J \subseteq \{1, \dots, k - 1\}$  mit  $\sum_{i \in J} a_i \leq v - a_k$  und  $\sum_{i \in J} c_i > \sum_{i \in I - \{k\}} c_i$ .

Dann wäre aber  $J \cup \{k\}$  eine bessere Lösung für  $P(k, v)$  als  $I$ , **Widerspruch.**

**2. Fall:**  $k \notin I$ .

---

*Beweis:* **Indirekt.**

**1. Fall:**  $k \in I$ . – **Annahme:**  $I - \{k\}$  nicht optimal für  $P(k - 1, v - a_k)$ .

Dann gäbe es eine bessere Teilmenge  $J \subseteq \{1, \dots, k - 1\}$  mit  $\sum_{i \in J} a_i \leq v - a_k$  und  $\sum_{i \in J} c_i > \sum_{i \in I - \{k\}} c_i$ .

Dann wäre aber  $J \cup \{k\}$  eine bessere Lösung für  $P(k, v)$  als  $I$ , **Widerspruch.**

**2. Fall:**  $k \notin I$ . – **Annahme:**  $I$  nicht optimal für  $P(k - 1, v)$ .

---

*Beweis:* **Indirekt.**

**1. Fall:**  $k \in I$ . – **Annahme:**  $I - \{k\}$  nicht optimal für  $P(k - 1, v - a_k)$ .

Dann gäbe es eine bessere Teilmenge  $J \subseteq \{1, \dots, k - 1\}$  mit  $\sum_{i \in J} a_i \leq v - a_k$  und  $\sum_{i \in J} c_i > \sum_{i \in I - \{k\}} c_i$ .

Dann wäre aber  $J \cup \{k\}$  eine bessere Lösung für  $P(k, v)$  als  $I$ , **Widerspruch.**

**2. Fall:**  $k \notin I$ . – **Annahme:**  $I$  nicht optimal für  $P(k - 1, v)$ .

Dann gäbe es eine bessere Teilmenge  $J \subseteq \{1, \dots, k - 1\}$ .

---

*Beweis:* **Indirekt.**

**1. Fall:**  $k \in I$ . – **Annahme:**  $I - \{k\}$  nicht optimal für  $P(k - 1, v - a_k)$ .

Dann gäbe es eine bessere Teilmenge  $J \subseteq \{1, \dots, k - 1\}$  mit  $\sum_{i \in J} a_i \leq v - a_k$  und  $\sum_{i \in J} c_i > \sum_{i \in I - \{k\}} c_i$ .

Dann wäre aber  $J \cup \{k\}$  eine bessere Lösung für  $P(k, v)$  als  $I$ , **Widerspruch.**

**2. Fall:**  $k \notin I$ . – **Annahme:**  $I$  nicht optimal für  $P(k - 1, v)$ .

Dann gäbe es eine bessere Teilmenge  $J \subseteq \{1, \dots, k - 1\}$ .

Dann wäre aber  $J$  auch besser als  $I$  für  $P(k, v)$

---

*Beweis:* **Indirekt.**

**1. Fall:**  $k \in I$ . – **Annahme:**  $I - \{k\}$  nicht optimal für  $P(k - 1, v - a_k)$ .

Dann gäbe es eine bessere Teilmenge  $J \subseteq \{1, \dots, k - 1\}$  mit  $\sum_{i \in J} a_i \leq v - a_k$  und  $\sum_{i \in J} c_i > \sum_{i \in I - \{k\}} c_i$ .

Dann wäre aber  $J \cup \{k\}$  eine bessere Lösung für  $P(k, v)$  als  $I$ , **Widerspruch.**

**2. Fall:**  $k \notin I$ . – **Annahme:**  $I$  nicht optimal für  $P(k - 1, v)$ .

Dann gäbe es eine bessere Teilmenge  $J \subseteq \{1, \dots, k - 1\}$ .

Dann wäre aber  $J$  auch besser als  $I$  für  $P(k, v)$ , **Wider-spruch.**

---

Die optimale Lösung für  $P(k, v)$  **enthält** also eine optimale Lösung für  $P(k - 1, v')$  für ein  $v' \leq v$ .

Eigenschaft „**Optimale Substruktur**“!

---

Die optimale Lösung für  $P(k, v)$  **enthält** also eine optimale Lösung für  $P(k - 1, v')$  für ein  $v' \leq v$ .

Eigenschaft „**Optimale Substruktur**“!

Bellman'sche Optimalitätsgleichungen:

---

Die optimale Lösung für  $P(k, v)$  **enthält** also eine optimale Lösung für  $P(k - 1, v')$  für ein  $v' \leq v$ .

Eigenschaft „**Optimale Substruktur**“!

Bellman'sche Optimalitätsgleichungen:

$$m(k, v) = \begin{cases} m(k - 1, v - a_k) + c_k, & \text{falls } v \geq a_k \text{ und diese Summe } \mathbf{größer} \text{ als } m(k - 1, v) \text{ ist;}^* \\ m(k - 1, v), & \text{sonst.} \end{cases}$$

für  $1 \leq k \leq n, 0 \leq v \leq b$ ;

$m(0, v) = 0, \quad \text{für } 0 \leq v \leq b.$

---

Die optimale Lösung für  $P(k, v)$  **enthält** also eine optimale Lösung für  $P(k - 1, v')$  für ein  $v' \leq v$ .

Eigenschaft „**Optimale Substruktur**“!

Bellman'sche Optimalitätsgleichungen:

$$m(k, v) = \begin{cases} m(k - 1, v - a_k) + c_k, & \text{falls } v \geq a_k \text{ und diese Summe } \mathbf{größer} \text{ als } m(k - 1, v) \text{ ist;}^* \\ m(k - 1, v), & \text{sonst.} \end{cases}$$

für  $1 \leq k \leq n, 0 \leq v \leq b$ ;

$m(0, v) = 0, \quad \text{für } 0 \leq v \leq b.$

\* In diesem Fall ist  $k$  Element jeder optimalen Lösung von  $P(k, v)$ .

---

Die optimale Lösung für  $P(k, v)$  **enthält** also eine optimale Lösung für  $P(k - 1, v')$  für ein  $v' \leq v$ .

Eigenschaft „**Optimale Substruktur**“!

Bellman'sche Optimalitätsgleichungen:

$$m(k, v) = \begin{cases} m(k - 1, v - a_k) + c_k, & \text{falls } v \geq a_k \text{ und diese Summe } \mathbf{größer} \text{ als } m(k - 1, v) \text{ ist;}^* \\ m(k - 1, v), & \text{sonst.} \end{cases}$$

für  $1 \leq k \leq n, 0 \leq v \leq b$ ;

$m(0, v) = 0, \quad \text{für } 0 \leq v \leq b.$

\* In diesem Fall ist  $k$  Element jeder optimalen Lösung von  $P(k, v)$ .

Wir berechnen alle  $m(k, v)$  mittels einer iterativen Prozedur.

---

### 4.3.1 Algorithmus DP-Zero-One-Knapsack( $a_1, \dots, a_n, c_1, \dots, c_n, b$ )

**Eingabe:**  $a_1, \dots, a_n$ : Volumina;  $c_1, \dots, c_n$ : Nutzenwerte;  $b$ : Schranke

**Ausgabe:**  $I \subseteq \{1, \dots, n\}$ : Optimale legale Auswahl.

**Datenstrukturen:**  $m[0..n, 0..b]$ : integer;

```
(1)   for v from 0 to b do m[0,v] ← 0;
(2)   for k from 1 to n do
(3)     for v from 0 to b do
(4)       if v - ak ≥ 0
(5)         then s ← m[k-1, v-ak] + ck else s ← 0;
(6)       if s > m[k-1, v]
(7)         then m[k, v] ← s;
(8)       else m[k, v] ← m[k-1, v];
      (* Konstruktion der optimalen Menge: *)
(9)   I ← ∅; r ← m[n, b];
(10)  for k from n downto 1 do
(11)    if m[k-1, r-ak] + ck > m[k-1, r]
(12)      then r ← r - ak; I ← I ∪ {k};
(13)  return I.
```

---

**Korrektheit** des ermittelten Wertes  $m[n, b]$  ergibt sich aus den Bellman'schen Optimalitätsgleichungen durch Induktion über  $k$ .

---

**Korrektheit** des ermittelten Wertes  $m[n, b]$  ergibt sich aus den Bellman'schen Optimalitätsgleichungen durch Induktion über  $k$ .

**Korrektheit** der ausgegebenen Menge  $I$  folgt aus der obigen Überlegung, dass die Bedingung

$$m(k - 1, v - a_k) + c_k > m(k - 1, v)$$

entscheidend ist für die Frage, ob  $k$  in der optimalen Lösung von  $P(k, v)$  enthalten ist oder nicht, und mit Induktion.

---

**Korrektheit** des ermittelten Wertes  $m[n, b]$  ergibt sich aus den Bellman'schen Optimalitätsgleichungen durch Induktion über  $k$ .

**Korrektheit** der ausgegebenen Menge  $I$  folgt aus der obigen Überlegung, dass die Bedingung

$$m(k - 1, v - a_k) + c_k > m(k - 1, v)$$

entscheidend ist für die Frage, ob  $k$  in der optimalen Lösung von  $P(k, v)$  enthalten ist oder nicht, und mit Induktion.

**Laufzeit** des Algorithmus:  $O(n \cdot b)$ .

---

**Korrektheit** des ermittelten Wertes  $m[n, b]$  ergibt sich aus den Bellman'schen Optimalitätsgleichungen durch Induktion über  $k$ .

**Korrektheit** der ausgegebenen Menge  $I$  folgt aus der obigen Überlegung, dass die Bedingung

$$m(k - 1, v - a_k) + c_k > m(k - 1, v)$$

entscheidend ist für die Frage, ob  $k$  in der optimalen Lösung von  $P(k, v)$  enthalten ist oder nicht, und mit Induktion.

**Laufzeit** des Algorithmus:  $O(n \cdot b)$ .

**Bemerkung:** Die Nutzen- und Volumenwerte können auch rationale Zahlen sein; die Volumenschranke  $b$  **muss** ganzzahlig sein.

---

**Korrektheit** des ermittelten Wertes  $m[n, b]$  ergibt sich aus den Bellman'schen Optimalitätsgleichungen durch Induktion über  $k$ .

**Korrektheit** der ausgegebenen Menge  $I$  folgt aus der obigen Überlegung, dass die Bedingung

$$m(k - 1, v - a_k) + c_k > m(k - 1, v)$$

entscheidend ist für die Frage, ob  $k$  in der optimalen Lösung von  $P(k, v)$  enthalten ist oder nicht, und mit Induktion.

**Laufzeit** des Algorithmus:  $O(n \cdot b)$ .

**Bemerkung:** Die Nutzen- und Volumenwerte können auch rationale Zahlen sein; die Volumenschranke  $b$  **muss** ganzzahlig sein. (Gegebenenfalls: Mit Nenner durchmultiplizieren.)

---

Die **Größe**, **nicht die Bitlänge** der Gewichtsschranke  $b$  geht in die Laufzeit ein.

---

Die **Größe, nicht die Bitlänge** der Gewichtsschranke  $b$  geht in die Laufzeit ein.

Wir nennen Algorithmen mit einer Laufzeit  $O(p(n, B))$ , wo  $n$  die (strukturelle) Größe der Eingabe und  $B$  eine obere Schranke für einige oder alle Zahlenkomponenten der Eingabe, und  $p$  ein Polynom in zwei Variablen ist, *pseudopolynomiell*.

---

Die **Größe**, nicht die **Bitlänge** der Gewichtsschranke  $b$  geht in die Laufzeit ein.

Wir nennen Algorithmen mit einer Laufzeit  $O(p(n, B))$ , wo  $n$  die (strukturelle) Größe der Eingabe und  $B$  eine obere Schranke für einige oder alle Zahlenkomponenten der Eingabe, und  $p$  ein Polynom in zwei Variablen ist, *pseudopolynomiell*.

Die Existenz eines pseudopolynomiellen Algorithmus für das Rucksackproblem widerspricht nicht der Tatsache, dass das 0-1-Rucksackproblem wahrscheinlich keinen Polynomialzeitalgorithmus hat (nächstes Semester: NP-Vollständigkeit!): Die *natürliche* Eingabegröße berücksichtigt  $\log b$  und nicht  $b$ .

*Beispiel:*

$b = 6:$

$i$	1	2	3	4
$a_i$	3	4	2	1
$c_i$	4	6	2	5

$m[k, v]:$

	$v: 0$	1	2	3	4	5	6
$k = 0$	0	0	0	0	0	0	0
1	0	0	0	4	4	4	4
2	0	0	0	4	6	6	6
3	0	0	2	4	6	6	8
4	0	5	5	7	9	11	11

---

Ablezen von  $I$  aus dem Array  $m[0..n, 0..b]$ :

Benutze  $m(k-1, v-a_k) + c_k > m[k-1, v]$  als Kriterium.

---

Ablezen von  $I$  aus dem Array  $m[0..n, 0..b]$ :

Benutze  $m(k-1, v-a_k) + c_k > m[k-1, v]$  als Kriterium.

(Eingerahmte Einträge der Tabelle.)

$$m[4, 6] = 11 > 8 = m[3, 6],$$

$$\text{also } I := \{4\}, v := 6 - 1 = 5$$

$$m[3, 5] = 6 \not> 6 = m[2, 5],$$

$$m[2, 5] = 6 > 4 = m[1, 5],$$

$$\text{also } I := I \cup \{2\}, v := 5 - 4 = 1$$

$$m[1, 1] = 0 \not> 0 = m[0, 1].$$

Resultat: Optimale Menge ist  $I = \{2, 4\}$ .

---

Man kann auf die Speicherung des gesamten Arrays  $m$  verzichten.

---

Man kann auf die Speicherung des gesamten Arrays  $m$  verzichten.

Man hält immer nur die letzte Zeile, und aktualisiert die Einträge **von rechts nach links**.

---

Man kann auf die Speicherung des gesamten Arrays  $m$  verzichten.

Man hält immer nur die letzte Zeile, und aktualisiert die Einträge **von rechts nach links**.

(Rechts vom Arbeitspunkt  $v$ : Einträge  $m(k, v')$ , links davon: Einträge  $m(k - 1, v')$ .)

---

Man kann auf die Speicherung des gesamten Arrays  $m$  verzichten.

Man hält immer nur die letzte Zeile, und aktualisiert die Einträge **von rechts nach links**.

(Rechts vom Arbeitspunkt  $v$ : Einträge  $m(k, v')$ , links davon: Einträge  $m(k - 1, v')$ .)

Eine Bitmatrix  $b[1..n, 0..b]$  führt mit, ob Objekt  $k$  für eine optimale Bepackung notwendig ist oder nicht.

---

### 4.3.1 Algorithmus DP-Zero-One-Knapsack( $a_1, \dots, a_n, c_1, \dots, c_n, b$ )

**Eingabe:**  $a_1, \dots, a_n$ : Volumina;  $c_1, \dots, c_n$ : Nutzenwerte;  $b$ : Schranke

**Ausgabe:**  $I \subseteq \{1, \dots, n\}$ : Optimale legale Auswahl.

**Datenstrukturen:**  $m[0..b]$ : integer;  $b[1..n, 0..b]$ : Boolean;

```
(1)   for v from 0 to b do m[v] ← 0;
(2)   for k from 1 to n do
(3)     for v from b downto 0 do
(4)       if v - ak ≥ 0 then s ← m[v-ak] + ck else s ← 0;
(5)       if s > m[v]
(6)         then m[v] ← s; b[k,v] ← 1;
(7)         else b[k,v] ← 0;
      (* Konstruktion der optimalen Menge: *)
(8)   I ← ∅; r ← m[b];
(9)   for k from n downto 1 do
(10)    if b[k,r] = 1 then
(11)      r ← r - ak; I ← I ∪ {k};
(12)  return I.
```

---

### 4.3.3. Satz

Das 0-1-Rucksackproblem (mit Ausgabe der optimalen Lösung) ist in Zeit  $O(nb)$  lösbar.

---

### 4.3.3. Satz

Das 0-1-Rucksackproblem (mit Ausgabe der optimalen Lösung) ist in Zeit  $O(nb)$  lösbar.  $\square$

---

## 4.4 Editierdistanz

**Problemstellung:** Sei  $\Sigma$  ein Alphabet.

---

## 4.4 Editierdistanz

**Problemstellung:** Sei  $\Sigma$  ein Alphabet.

*Beispiele:* Lateinisches Alphabet, ASCII-Alphabet,  $\{A,C,G,T\}$ .

Wenn  $x = a_1 \dots a_m$  und  $y = b_1 \dots b_n$  zwei Zeichenreihen über  $\Sigma$  sind, möchte man herausfinden, wie **ähnlich** (oder unähnlich) sie sind.

„Freizeit“ und „Arbeit“ sind nicht identisch, aber intuitiv einander ähnlicher als „Informatik“ und „Schwimmen“.

Wie messen?

---

## 4.4 Editierdistanz

**Problemstellung:** Sei  $\Sigma$  ein Alphabet.

*Beispiele:* Lateinisches Alphabet, ASCII-Alphabet,  $\{A,C,G,T\}$ .

Wenn  $x = a_1 \dots a_m$  und  $y = b_1 \dots b_n$  zwei Zeichenreihen über  $\Sigma$  sind, möchte man herausfinden, wie **ähnlich** (oder unähnlich) sie sind.

„Freizeit“ und „Arbeit“ sind nicht identisch, aber intuitiv einander ähnlicher als „Informatik“ und „Schwimmen“.

Wie messen?

Wir definieren Elementarschritte (Editier-Operationen), die einen String verändern:

- 
- Lösche einen Buchstaben aus einem String:  
Aus  $uav$  wird  $uv$  ( $u, v \in \Sigma^*$ ,  $a \in \Sigma$ )

- 
- Lösche einen Buchstaben aus einem String:  
Aus  $uav$  wird  $uv$  ( $u, v \in \Sigma^*$ ,  $a \in \Sigma$ )
  - Füge einen Buchstaben in einen String ein:  
Aus  $uv$  wird  $uav$  ( $u, v \in \Sigma^*$ ,  $a \in \Sigma$ )

- 
- Lösche einen Buchstaben aus einem String:  
Aus  $uav$  wird  $uv$  ( $u, v \in \Sigma^*$ ,  $a \in \Sigma$ )
  - Füge einen Buchstaben in einen String ein:  
Aus  $uv$  wird  $uav$  ( $u, v \in \Sigma^*$ ,  $a \in \Sigma$ )
  - Ersetze einen Buchstaben:  
Aus  $uav$  wird  $ubv$  ( $u, v \in \Sigma^*$ ,  $a, b \in \Sigma$ )

---

Der „Abstand“, oder technisch die **Editierdistanz** von  $x$  und  $y$ , in Zeichen  $d(x, y)$ , ist die minimale Anzahl von Editieroperationen, die benötigt werden, um  $x$  in  $y$  zu transformieren.

---

Der „Abstand“, oder technisch die **Editierdistanz** von  $x$  und  $y$ , in Zeichen  $d(x, y)$ , ist die minimale Anzahl von Editieroperationen, die benötigt werden, um  $x$  in  $y$  zu transformieren.

**Überlege:** Äquivalenz zu Zuordnung:

F	r	e	i	z	e	i	t
A	r	b	$\varepsilon$	$\varepsilon$	e	i	t

---

Der „Abstand“, oder technisch die **Editierdistanz** von  $x$  und  $y$ , in Zeichen  $d(x, y)$ , ist die minimale Anzahl von Editieroperationen, die benötigt werden, um  $x$  in  $y$  zu transformieren.

**Überlege:** Äquivalenz zu Zuordnung:

F	r	e	i	z	e	i	t
A	r	b	$\varepsilon$	$\varepsilon$	e	i	t

---

Der „Abstand“, oder technisch die **Editierdistanz** von  $x$  und  $y$ , in Zeichen  $d(x, y)$ , ist die minimale Anzahl von Editieroperationen, die benötigt werden, um  $x$  in  $y$  zu transformieren.

**Überlege:** Äquivalenz zu Zuordnung:

F	r	e	i	z	e	i	t
A	r	b	$\varepsilon$	$\varepsilon$	e	i	t

Man schreibt Strings aus Buchstaben und dem Sonderzeichen  $\varepsilon$  untereinander, wobei verlangt wird, dass die Konkatenation der beiden Zeilen jeweils das Wort  $x$  bzw.  $y$  ergeben, wenn man die  $\varepsilon$ 's ignoriert.

---

Der „Abstand“, oder technisch die **Editierdistanz** von  $x$  und  $y$ , in Zeichen  $d(x, y)$ , ist die minimale Anzahl von Editieroperationen, die benötigt werden, um  $x$  in  $y$  zu transformieren.

**Überlege:** Äquivalenz zu Zuordnung:

F	r	e	i	z	e	i	t
A	r	b	$\varepsilon$	$\varepsilon$	e	i	t

Man schreibt Strings aus Buchstaben und dem Sonderzeichen  $\varepsilon$  untereinander, wobei verlangt wird, dass die Konkatenation der beiden Zeilen jeweils das Wort  $x$  bzw.  $y$  ergeben, wenn man die  $\varepsilon$ 's ignoriert.

Die **Kosten** einer solchen Anordnung: Die Anzahl der Positionen, an denen die Einträge nicht übereinstimmen.

---

Die minimalen Kosten, die eine solche Anordnung erzeugt, sind gleich der Editierdistanz. (Im Beispiel: 4.)

---

Die minimalen Kosten, die eine solche Anordnung erzeugt, sind gleich der Editierdistanz. (Im Beispiel: 4.)

**Input:**  $x = a_1 \dots a_m, y = b_1 \dots b_n.$

---

Die minimalen Kosten, die eine solche Anordnung erzeugt, sind gleich der Editierdistanz. (Im Beispiel: 4.)

**Input:**  $x = a_1 \dots a_m, y = b_1 \dots b_n$ .

**Aufgabe:** Berechne  $d(x, y)$  und eine Editierfolge, die  $x$  in  $y$  transformiert.

---

Die minimalen Kosten, die eine solche Anordnung erzeugt, sind gleich der Editierdistanz. (Im Beispiel: 4.)

**Input:**  $x = a_1 \dots a_m, y = b_1 \dots b_n$ .

**Aufgabe:** Berechne  $d(x, y)$  und eine Editierfolge, die  $x$  in  $y$  transformiert.

**Ansatz:** **Dynamische Programmierung**

---

Die minimalen Kosten, die eine solche Anordnung erzeugt, sind gleich der Editierdistanz. (Im Beispiel: 4.)

**Input:**  $x = a_1 \dots a_m, y = b_1 \dots b_n$ .

**Aufgabe:** Berechne  $d(x, y)$  und eine Editierfolge, die  $x$  in  $y$  transformiert.

**Ansatz:** **Dynamische Programmierung**

Unser Beispiel:  $x = \text{Exponentiell}$  und  $y = \text{Polynomiell}$ .

---

Die minimalen Kosten, die eine solche Anordnung erzeugt, sind gleich der Editierdistanz. (Im Beispiel: 4.)

**Input:**  $x = a_1 \dots a_m$ ,  $y = b_1 \dots b_n$ .

**Aufgabe:** Berechne  $d(x, y)$  und eine Editierfolge, die  $x$  in  $y$  transformiert.

**Ansatz:** **Dynamische Programmierung**

Unser Beispiel:  $x = \text{Exponentiell}$  und  $y = \text{Polynomiell}$ .

Relevante **Teilprobleme** identifizieren!

---

Die minimalen Kosten, die eine solche Anordnung erzeugt, sind gleich der Editierdistanz. (Im Beispiel: 4.)

**Input:**  $x = a_1 \dots a_m$ ,  $y = b_1 \dots b_n$ .

**Aufgabe:** Berechne  $d(x, y)$  und eine Editierfolge, die  $x$  in  $y$  transformiert.

**Ansatz:** **Dynamische Programmierung**

Unser Beispiel:  $x = \text{Exponentiell}$  und  $y = \text{Polynomiell}$ .

Relevante **Teilprobleme** identifizieren!

Betrachte Präfixe  $x_i = a_1 \dots a_i$  und  $y_j = b_1 \dots b_j$ .

---

Die minimalen Kosten, die eine solche Anordnung erzeugt, sind gleich der Editierdistanz. (Im Beispiel: 4.)

**Input:**  $x = a_1 \dots a_m$ ,  $y = b_1 \dots b_n$ .

**Aufgabe:** Berechne  $d(x, y)$  und eine Editierfolge, die  $x$  in  $y$  transformiert.

**Ansatz:** **Dynamische Programmierung**

Unser Beispiel:  $x = \text{Exponentiell}$  und  $y = \text{Polynomiell}$ .

Relevante **Teilprobleme** identifizieren!

Betrachte Präfixe  $x_i = a_1 \dots a_i$  und  $y_j = b_1 \dots b_j$ .

**$E(i, j)$**  :=  $d(x_i, y_j)$ , für  $1 \leq i \leq m$ ,  $1 \leq j \leq n$ .

---

Die minimalen Kosten, die eine solche Anordnung erzeugt, sind gleich der Editierdistanz. (Im Beispiel: 4.)

**Input:**  $x = a_1 \dots a_m$ ,  $y = b_1 \dots b_n$ .

**Aufgabe:** Berechne  $d(x, y)$  und eine Editierfolge, die  $x$  in  $y$  transformiert.

**Ansatz:** **Dynamische Programmierung**

Unser Beispiel:  $x = \text{Exponentiell}$  und  $y = \text{Polynomiell}$ .

Relevante **Teilprobleme** identifizieren!

Betrachte Präfixe  $x_i = a_1 \dots a_i$  und  $y_j = b_1 \dots b_j$ .

**$E(i, j)$**  :=  $d(x_i, y_j)$ , für  $1 \leq i \leq m$ ,  $1 \leq j \leq n$ .

*Beispiel:*  **$E(7, 6)$** : Berechne  $d(\text{Exponen}, \text{Polyno})$ .

---

Um von  $x_i = a_1 \dots a_i$  gegenüber  $y_j = b_1 \dots b_j$  anzuordnen, gibt es drei Möglichkeiten:

---

Um von  $x_i = a_1 \dots a_i$  gegenüber  $y_j = b_1 \dots b_j$  anzuordnen, gibt es drei Möglichkeiten:

1. Fall		2. Fall		3. Fall	
$x_{i-1}$	$a_i$	$x_{i-1}$	$a_i$	$x_i$	$\varepsilon$
$y_{i-1}$	$b_j$	$y_j$	$\varepsilon$	$y_{j-1}$	$b_j$

---

Um von  $x_i = a_1 \dots a_i$  gegenüber  $y_j = b_1 \dots b_j$  anzuordnen, gibt es drei Möglichkeiten:

1. Fall	2. Fall	3. Fall
$x_{i-1} \quad a_i$	$x_{i-1} \quad a_i$	$x_i \quad \varepsilon$
$y_{i-1} \quad b_j$	$y_j \quad \varepsilon$	$y_{j-1} \quad b_j$

Im 1. Fall kostet die letzte Stelle  $\text{diff}(a_i, b_j)$ , das ist 1 falls  $a_i \neq b_j$ , und 0 falls  $a_i = b_j$ ,

Um von  $x_i = a_1 \dots a_i$  gegenüber  $y_j = b_1 \dots b_j$  anzuordnen, gibt es drei Möglichkeiten:

1. Fall	2. Fall	3. Fall
$x_{i-1} \quad a_i$	$x_{i-1} \quad a_i$	$x_i \quad \varepsilon$
$y_{i-1} \quad b_j$	$y_j \quad \varepsilon$	$y_{j-1} \quad b_j$

Im 1. Fall kostet die letzte Stelle  $\text{diff}(a_i, b_j)$ , das ist 1 falls  $a_i \neq b_j$ , und 0 falls  $a_i = b_j$ ,  $x_{i-1}$  und  $y_{j-1}$  sind einander optimal zuzuordnen:

Um von  $x_i = a_1 \dots a_i$  gegenüber  $y_j = b_1 \dots b_j$  anzuordnen, gibt es drei Möglichkeiten:

1. Fall	2. Fall	3. Fall
$x_{i-1} \quad a_i$	$x_{i-1} \quad a_i$	$x_i \quad \varepsilon$
$y_{i-1} \quad b_j$	$y_j \quad \varepsilon$	$y_{j-1} \quad b_j$

Im 1. Fall kostet die letzte Stelle  $\text{diff}(a_i, b_j)$ , das ist 1 falls  $a_i \neq b_j$ , und 0 falls  $a_i = b_j$ ,  $x_{i-1}$  und  $y_{j-1}$  sind einander optimal zuzuordnen:

$$d(x_i, y_j) = d(x_{i-1}, y_{j-1}) + \text{diff}(a_i, b_j).$$

Um von  $x_i = a_1 \dots a_i$  gegenüber  $y_j = b_1 \dots b_j$  anzuordnen, gibt es drei Möglichkeiten:

1. Fall	2. Fall	3. Fall
$x_{i-1} \quad a_i$	$x_{i-1} \quad a_i$	$x_i \quad \varepsilon$
$y_{i-1} \quad b_j$	$y_j \quad \varepsilon$	$y_{j-1} \quad b_j$

Im 1. Fall kostet die letzte Stelle  $\text{diff}(a_i, b_j)$ , das ist 1 falls  $a_i \neq b_j$ , und 0 falls  $a_i = b_j$ ,  $x_{i-1}$  und  $y_{j-1}$  sind einander optimal zuzuordnen:

$$d(x_i, y_j) = d(x_{i-1}, y_{j-1}) + \text{diff}(a_i, b_j).$$

Im 2. Fall kostet die eine unpassende Stelle 1, und es ist  $x_{i-1}$  und  $y_j$  optimal zuzuordnen:

Um von  $x_i = a_1 \dots a_i$  gegenüber  $y_j = b_1 \dots b_j$  anzuordnen, gibt es drei Möglichkeiten:

1. Fall		2. Fall		3. Fall	
$x_{i-1}$	$a_i$	$x_{i-1}$	$a_i$	$x_i$	$\varepsilon$
$y_{i-1}$	$b_j$	$y_j$	$\varepsilon$	$y_{j-1}$	$b_j$

Im 1. Fall kostet die letzte Stelle  $\text{diff}(a_i, b_j)$ , das ist 1 falls  $a_i \neq b_j$ , und 0 falls  $a_i = b_j$ ,  $x_{i-1}$  und  $y_{j-1}$  sind einander optimal zuzuordnen:

$$d(x_i, y_j) = d(x_{i-1}, y_{j-1}) + \text{diff}(a_i, b_j).$$

Im 2. Fall kostet die eine unpassende Stelle 1, und es ist  $x_{i-1}$  und  $y_j$  optimal zuzuordnen:

$$d(x_i, y_j) = 1 + d(x_{i-1}, y_j).$$

Um von  $x_i = a_1 \dots a_i$  gegenüber  $y_j = b_1 \dots b_j$  anzuordnen, gibt es drei Möglichkeiten:

1. Fall		2. Fall		3. Fall	
$x_{i-1}$	$a_i$	$x_{i-1}$	$a_i$	$x_i$	$\varepsilon$
$y_{i-1}$	$b_j$	$y_j$	$\varepsilon$	$y_{j-1}$	$b_j$

Im 1. Fall kostet die letzte Stelle  $\text{diff}(a_i, b_j)$ , das ist 1 falls  $a_i \neq b_j$ , und 0 falls  $a_i = b_j$ ,  $x_{i-1}$  und  $y_{j-1}$  sind einander optimal zuzuordnen:

$$d(x_i, y_j) = d(x_{i-1}, y_{j-1}) + \text{diff}(a_i, b_j).$$

Im 2. Fall kostet die eine unpassende Stelle 1, und es ist  $x_{i-1}$  und  $y_j$  optimal zuzuordnen:

$$d(x_i, y_j) = 1 + d(x_{i-1}, y_j).$$

Im 3. Fall kostet die eine unpassende Stelle 1, und es ist  $x_i$  und  $y_{j-1}$  optimal zuzuordnen:

Um von  $x_i = a_1 \dots a_i$  gegenüber  $y_j = b_1 \dots b_j$  anzuordnen, gibt es drei Möglichkeiten:

1. Fall		2. Fall		3. Fall	
$x_{i-1}$	$a_i$	$x_{i-1}$	$a_i$	$x_i$	$\varepsilon$
$y_{i-1}$	$b_j$	$y_j$	$\varepsilon$	$y_{j-1}$	$b_j$

Im 1. Fall kostet die letzte Stelle  $\text{diff}(a_i, b_j)$ , das ist 1 falls  $a_i \neq b_j$ , und 0 falls  $a_i = b_j$ ,  $x_{i-1}$  und  $y_{j-1}$  sind einander optimal zuzuordnen:

$$d(x_i, y_j) = d(x_{i-1}, y_{j-1}) + \text{diff}(a_i, b_j).$$

Im 2. Fall kostet die eine unpassende Stelle 1, und es ist  $x_{i-1}$  und  $y_j$  optimal zuzuordnen:

$$d(x_i, y_j) = 1 + d(x_{i-1}, y_j).$$

Im 3. Fall kostet die eine unpassende Stelle 1, und es ist  $x_i$  und  $y_{j-1}$  optimal zuzuordnen:  $d(x_i, y_j) = 1 + d(x_i, y_{j-1})$ .

---

**Zusammengefasst:** Bellman'sche Optimalitätsgleichungen für Editierdistanz:

---

**Zusammengefasst:** Bellman'sche Optimalitätsgleichungen für Editierdistanz:

$$E(i, j) = \min\{E(i-1, j-1) + \text{diff}(a_i, b_j), 1 + E(i-1, j), 1 + E(i, j-1)\}.$$

Der Induktionsanfang wird von den leeren Präfixen  $x_0$  und  $y_0$  definiert.

---

**Zusammengefasst:** Bellman'sche Optimalitätsgleichungen für Editierdistanz:

$$E(i, j) = \min\{E(i-1, j-1) + \text{diff}(a_i, b_j), 1 + E(i-1, j), 1 + E(i, j-1)\}.$$

Der Induktionsanfang wird von den leeren Präfixen  $x_0$  und  $y_0$  definiert.

Es ist klar, dass  $d(\varepsilon, y_j) = j$  und  $d(x_i, \varepsilon) = i$  ist (man muss jeweils alle vorhandenen Buchstaben streichen).

---

Die Zahlen  $E(i, j)$  tragen wir in eine Matrix  $E[0..m, 0..n]$  ein:

---

Die Zahlen  $E(i, j)$  tragen wir in eine Matrix  $E[0..m, 0..n]$  ein:

**Initialisierung:**

---

Die Zahlen  $E(i, j)$  tragen wir in eine Matrix  $E[0..m, 0..n]$  ein:

**Initialisierung:**

$$E[0, j] \leftarrow j, \text{ für } j = 0, \dots, n.$$

---

Die Zahlen  $E(i, j)$  tragen wir in eine Matrix  $E[0..m, 0..n]$  ein:

**Initialisierung:**

$$E[0, j] \leftarrow j, \text{ für } j = 0, \dots, n.$$

$$E[i, 0] \leftarrow i, \text{ für } i = 0, \dots, m.$$

---

Die Zahlen  $E(i, j)$  tragen wir in eine Matrix  $E[0..m, 0..n]$  ein:

**Initialisierung:**

$$E[0, j] \leftarrow j, \text{ für } j = 0, \dots, n.$$

$$E[i, 0] \leftarrow i, \text{ für } i = 0, \dots, m.$$

Dann füllen wir die Matrix (z. B.) zeilenweise aus, genau nach den Bellman'schen Optimalitätsgleichungen:

---

Die Zahlen  $E(i, j)$  tragen wir in eine Matrix  $E[0..m, 0..n]$  ein:

**Initialisierung:**

$E[0, j] \leftarrow j$ , für  $j = 0, \dots, n$ .

$E[i, 0] \leftarrow i$ , für  $i = 0, \dots, m$ .

Dann füllen wir die Matrix (z. B.) zeilenweise aus, genau nach den Bellman'schen Optimalitätsgleichungen:

**for**  $i$  **from** 1 **to**  $m$  **do**  
    **for**  $j$  **from** 1 **to**  $n$  **do**

---

Die Zahlen  $E(i, j)$  tragen wir in eine Matrix  $E[0..m, 0..n]$  ein:

**Initialisierung:**

$E[0, j] \leftarrow j$ , für  $j = 0, \dots, n$ .

$E[i, 0] \leftarrow i$ , für  $i = 0, \dots, m$ .

Dann füllen wir die Matrix (z. B.) zeilenweise aus, genau nach den Bellman'schen Optimalitätsgleichungen:

```
for  $i$  from 1 to  $m$  do  
    for  $j$  from 1 to  $n$  do  
         $E[i, j] \leftarrow$ 
```

---

Die Zahlen  $E(i, j)$  tragen wir in eine Matrix  $E[0..m, 0..n]$  ein:

### **Initialisierung:**

$E[0, j] \leftarrow j$ , für  $j = 0, \dots, n$ .

$E[i, 0] \leftarrow i$ , für  $i = 0, \dots, m$ .

Dann füllen wir die Matrix (z. B.) zeilenweise aus, genau nach den Bellman'schen Optimalitätsgleichungen:

**for**  $i$  **from** 1 **to**  $m$  **do**

**for**  $j$  **from** 1 **to**  $n$  **do**

$E[i, j] \leftarrow$

$\min\{E[i-1, j-1] + \text{diff}(a_i, b_j),$   
                 $1 + E[i-1, j], 1 + E[i, j-1]\};$

---

Die Zahlen  $E(i, j)$  tragen wir in eine Matrix  $E[0..m, 0..n]$  ein:

### **Initialisierung:**

$E[0, j] \leftarrow j$ , für  $j = 0, \dots, n$ .

$E[i, 0] \leftarrow i$ , für  $i = 0, \dots, m$ .

Dann füllen wir die Matrix (z. B.) zeilenweise aus, genau nach den Bellman'schen Optimalitätsgleichungen:

**for**  $i$  **from** 1 **to**  $m$  **do**

**for**  $j$  **from** 1 **to**  $n$  **do**

$E[i, j] \leftarrow$

$\min\{E[i-1, j-1] + \text{diff}(a_i, b_j),$   
                 $1 + E[i-1, j], 1 + E[i, j-1]\};$

**return**  $E[m, n]$ .

<i>E</i>		P	o	l	y	n	o	m	i	e	l	l
	0	1	2	3	4	5	6	7	8	9	10	11
E	1	1	2	3	4	5	6	7	8	9	10	11
x	2	2	2	3	4	5	6	7	8	9	10	11
p	3	3	3	3	4	5	6	7	8	9	10	11
o	4	4	<u>3</u>	4	4	5	<u>5</u>	6	7	8	9	10
n	5	5	4	4	5	<u>4</u>	5	6	7	8	9	10
e	6	6	5	5	5	5	5	6	7	<u>7</u>	8	9
n	7	7	6	6	6	<u>5</u>	6	6	7	8	8	9
t	8	8	7	7	7	6	6	7	7	8	9	9
i	9	9	8	8	8	7	7	7	<u>7</u>	8	9	10
e	10	10	9	9	9	8	8	8	8	<u>7</u>	8	9
l	11	11	10	<u>9</u>	10	9	9	9	9	8	<u>7</u>	<u>8</u>
l	12	12	11	<u>10</u>	10	10	10	10	10	9	<u>8</u>	<u>7</u>

---

Rot, unterstrichen: die Positionen  $(i, j)$  mit  $a_i = b_j$ .

---

Für die Arbeit von Hand ist es bequem, in die Matrix zuerst die Werte  $\text{diff}(a_i, b_j)$  einzutragen!

---

Für die Arbeit von Hand ist es bequem, in die Matrix zuerst die Werte  $\text{diff}(a_i, b_j)$  einzutragen!

## **Ermittlung der Editier-Operationen**

Gegeben  $x$  und  $y$ , möchte man nicht nur die Editierdistanz  $d(x, y)$ , sondern auch die Schritte einer möglichst kurzen Transformation ermitteln, die  $x$  in  $y$  überführt.

---

Für die Arbeit von Hand ist es bequem, in die Matrix zuerst die Werte  $\text{diff}(a_i, b_j)$  einzutragen!

## **Ermittlung der Editier-Operationen**

Gegeben  $x$  und  $y$ , möchte man nicht nur die Editierdistanz  $d(x, y)$ , sondern auch die Schritte einer möglichst kurzen Transformation ermitteln, die  $x$  in  $y$  überführt.

Wenn die Matrix  $E[0..m, 0..n]$  vorliegt, die alle Werte  $E(i, j)$  enthält, ist das leicht:

---

Wir wandern einen Weg von  $(m, n)$  nach  $(0, 0)$ .

---

Wir wandern einen Weg von  $(m, n)$  nach  $(0, 0)$ . Dabei gehen wir von  $(i, j)$  nach

---

Wir wandern einen Weg von  $(m, n)$  nach  $(0, 0)$ . Dabei gehen wir von  $(i, j)$  nach

$$(1) (i - 1, j - 1), \quad \text{falls } E(i, j) = E(i - 1, j - 1) + \text{diff}(a_i, b_j),$$

---

Wir wandern einen Weg von  $(m, n)$  nach  $(0, 0)$ . Dabei gehen wir von  $(i, j)$  nach

- (1)  $(i - 1, j - 1)$ , falls  $E(i, j) = E(i - 1, j - 1) + \text{diff}(a_i, b_j)$ ,
- (2)  $(i - 1, j)$ , falls  $E(i, j) = 1 + E(i - 1, j)$ ,

---

Wir wandern einen Weg von  $(m, n)$  nach  $(0, 0)$ . Dabei gehen wir von  $(i, j)$  nach

- (1)  $(i - 1, j - 1)$ , falls  $E(i, j) = E(i - 1, j - 1) + \text{diff}(a_i, b_j)$ ,
- (2)  $(i - 1, j)$ , falls  $E(i, j) = 1 + E(i - 1, j)$ ,
- (3)  $(i, j - 1)$ , falls  $E(i, j) = 1 + E(i, j - 1)$ .

---

Wir wandern einen Weg von  $(m, n)$  nach  $(0, 0)$ . Dabei gehen wir von  $(i, j)$  nach

- (1)  $(i - 1, j - 1)$ , falls  $E(i, j) = E(i - 1, j - 1) + \text{diff}(a_i, b_j)$ ,
- (2)  $(i - 1, j)$ , falls  $E(i, j) = 1 + E(i - 1, j)$ ,
- (3)  $(i, j - 1)$ , falls  $E(i, j) = 1 + E(i, j - 1)$ .

Wenn mehr als eine Gleichung zutrifft, können wir einen der möglichen Schritte beliebig wählen.

---

Wir wandern einen Weg von  $(m, n)$  nach  $(0, 0)$ . Dabei gehen wir von  $(i, j)$  nach

- (1)  $(i - 1, j - 1)$ , falls  $E(i, j) = E(i - 1, j - 1) + \text{diff}(a_i, b_j)$ ,
- (2)  $(i - 1, j)$ , falls  $E(i, j) = 1 + E(i - 1, j)$ ,
- (3)  $(i, j - 1)$ , falls  $E(i, j) = 1 + E(i, j - 1)$ .

Wenn mehr als eine Gleichung zutrifft, können wir einen der möglichen Schritte beliebig wählen.

Der so gefundene Weg endet in  $(0, 0)$ .

---

Den so ermittelten Weg lesen wir dann (weil wir von links nach rechts lesen) rückwärts, also von  $(0, 0)$  nach  $(m, n)$  laufend.

Er repräsentiert in offensichtlicher Weise einer Transformation von  $x$  in  $y$ , wobei man Buchstaben gleich lässt, ersetzt, einfügt oder löscht.

---

Den so ermittelten Weg lesen wir dann (weil wir von links nach rechts lesen) rückwärts, also von  $(0, 0)$  nach  $(m, n)$  laufend.

Er repräsentiert in offensichtlicher Weise einer Transformation von  $x$  in  $y$ , wobei man Buchstaben gleich lässt, ersetzt, einfügt oder löscht.

(1)  $(i - 1, j - 1) \rightarrow (i, j)$  entspricht einem identischen Buchstaben oder einer Ersetzung an Stelle  $i$ ;

---

Den so ermittelten Weg lesen wir dann (weil wir von links nach rechts lesen) rückwärts, also von  $(0, 0)$  nach  $(m, n)$  laufend.

Er repräsentiert in offensichtlicher Weise einer Transformation von  $x$  in  $y$ , wobei man Buchstaben gleich lässt, ersetzt, einfügt oder löscht.

(1)  $(i - 1, j - 1) \rightarrow (i, j)$  entspricht einem identischen Buchstaben oder einer Ersetzung an Stelle  $i$ ;

(2)  $(i - 1, j) \rightarrow (i, j)$  entspricht dem Streichen von  $a_i$ ;

---

Den so ermittelten Weg lesen wir dann (weil wir von links nach rechts lesen) rückwärts, also von  $(0, 0)$  nach  $(m, n)$  laufend.

Er repräsentiert in offensichtlicher Weise einer Transformation von  $x$  in  $y$ , wobei man Buchstaben gleich lässt, ersetzt, einfügt oder löscht.

(1)  $(i - 1, j - 1) \rightarrow (i, j)$  entspricht einem identischen Buchstaben oder einer Ersetzung an Stelle  $i$ ;

(2)  $(i - 1, j) \rightarrow (i, j)$  entspricht dem Streichen von  $a_i$ ;

(3)  $(i, j - 1) \rightarrow (i, j)$  entspricht dem Einfügen von  $b_j$ .

---

Im Beispiel:  $(0, 0) \rightarrow (1, 1) \rightarrow (2, 2) \rightarrow (3, 3) \rightarrow (4, 4) \rightarrow (5, 5) \rightarrow (6, 6) \rightarrow (7, 6) \rightarrow (8, 7) \rightarrow (9, 8) \rightarrow (10, 9) \rightarrow (11, 10) \rightarrow (12, 11)$ .

Dies führt zu der folgenden Zuordnung, wobei die sieben „kostenpflichtigen“ Positionen markiert sind:

E	x	p	o	n	e	n	t	i	e	l	l
<hr/>											
P	o	l	y	n	o	-	m	i	e	l	l
<hr/>											
!	!	!	!	!	!	!	!	!	!	!	!

---

Weiteres Beispiel, mit Streichungen und Einfügungen:

$x = \text{speziell}$  und  $y = \text{beliebig}$ .

Ausfüllen der Matrix liefert:

$E$		b	e	l	i	e	b	i	g
	0	1	2	3	4	5	6	7	8
s	1	1	2	3	4	5	6	7	8
p	2	2	2	3	4	5	6	7	8
e	3	3	<u>2</u>	3	4	<u>4</u>	5	6	7
z	4	4	3	3	4	5	5	6	7
i	5	5	4	4	<u>3</u>	4	5	<u>5</u>	6
e	6	6	<u>5</u>	5	4	<u>3</u>	4	5	6
l	7	7	6	<u>5</u>	5	4	4	5	6
l	8	8	7	<u>6</u>	6	5	5	5	6

---

Weiteres Beispiel, mit Streichungen und Einfügungen:

$x = \text{speziell}$  und  $y = \text{beliebig}$ .

Ausfüllen der Matrix liefert:

$E$		b	e	l	i	e	b	i	g
	0	1	2	3	4	5	6	7	8
s	1	1	2	3	4	5	6	7	8
p	2	2	2	3	4	5	6	7	8
e	3	3	<u>2</u>	3	4	<u>4</u>	5	6	7
z	4	4	3	3	4	5	5	6	7
i	5	5	4	4	<u>3</u>	4	5	<u>5</u>	6
e	6	6	<u>5</u>	5	4	<u>3</u>	4	5	6
l	7	7	6	<u>5</u>	5	4	4	5	6
l	8	8	7	<u>6</u>	6	5	5	5	6

Rot, unterstrichen: die Positionen  $(i, j)$  mit  $a_i = b_j$ .

---

Zurückverfolgen liefert nach Umdrehen zum Beispiel die folgende Transformationsfolge:

$(0, 0) \rightarrow (1, 1) \rightarrow (2, 1) \rightarrow (3, 2) \rightarrow (4, 3) \rightarrow (5, 4) \rightarrow (6, 5) \rightarrow (6, 6) \rightarrow (7, 7) \rightarrow (8, 8)$ .

---

Zurückverfolgen liefert nach Umdrehen zum Beispiel die folgende Transformationsfolge:

$(0, 0) \rightarrow (1, 1) \rightarrow (2, 1) \rightarrow (3, 2) \rightarrow (4, 3) \rightarrow (5, 4) \rightarrow (6, 5) \rightarrow (6, 6) \rightarrow (7, 7) \rightarrow (8, 8)$ .

Dies führt zu der folgenden Zuordnung, wobei die sechs „kostenpflichtigen“ Positionen markiert sind:

s	p	e	z	i	e	-	l	l
b	-	e	l	i	e	b	i	g
!	!	!	!	!	!	!	!	!

Es gibt aber auch andere legale Transformationsfolgen.