

# Algorithmen und Datenstrukturen

## Kapitel 9: Dynamische Programmierung

**Prof. Dr. Wolfgang Mühlbauer**

Fakultät für Informatik

`wolfgang.muehlbauer@th-rosenheim.de`

**Wintersemester 2019/2020**

# Memory



Quelle: [3]

- ❑ Dynamic Programming
  - "Man merkt sich auf "Vorrat" Teilergebnisse!"

# Überblick

---

- ❑ **Einführung: Fibonacci-Zahlen**
- ❑ Rod-Cutting
- ❑ Längste gemeinsame Teilfolge
- ❑ Levenshtein-Editierdistanz

# Dynamische Programmierung

- ❑ Kein Algorithmus, sondern ***algorithmisches Prinzip***
  
- ❑ Häufig verwendet für ***Optimierungsprobleme***
  - Es existieren *mehrere* Lösungen für ein Problem.
  - Finde davon die *beste* Lösung, d.h. Lösungen werden *bewertet*.
  
- ❑ ***Ähnlichkeit zu Divide-and-Conquer***
  - *Divide-and-Conquer*: Zerlege Problem in ***unabhängige*** Teilprobleme.
  - *Dynamische Programmierung*: Teilprobleme ***überlappen***. Jedes Teilproblem wird dennoch nur einmal gelöst.
  
- ❑ ***Ansatz: Problemlösen „auf Vorrat“***
  - Löse nur Teilprobleme, die auch wirklich für Gesamtproblem benötigt werden.
  - Löse jedes Teilproblem nur ***einmal***.

# Fibonacci: Rekursion / Top-Down

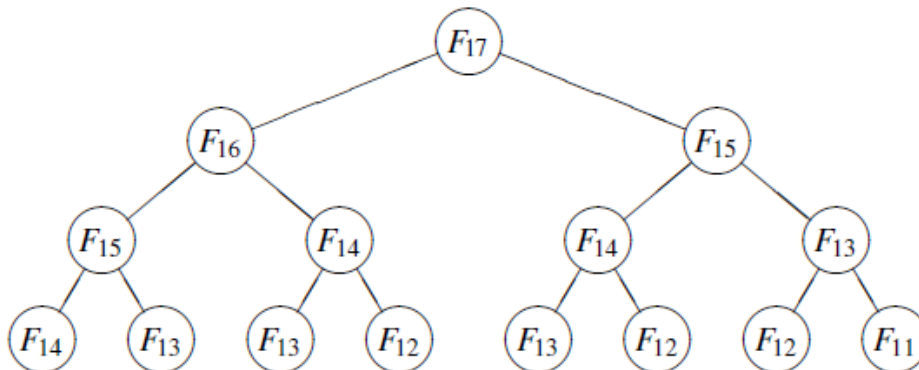
## □ **Rekursive Definition** der Fibonaccizahlen $F_n$

- $F_0 = 0$
- $F_1 = 1$
- $F_i = F_{i-1} + F_{i-2}$

```
Quellcode: Fibonacci.java  
Methode: fibTopDown
```

## □ **Laufzeit** der rekursiven Implementierung

- Exponentiell mit der Basis  $\frac{1+\sqrt{5}}{2}$ , d.h.  $O(2^n)$
- Geht es schneller?



Beobachtung: Im Rekursionsbaum gibt es doppelte Berechnungen!

# Fibonacci: Rekursion / Memoisation

❑ Behalte Rekursion bei!

❑ **Zusätzlich: Memoisation**

- Speichern von bereits berechneten Ergebnissen.
- Z.B. in Array oder in `HashMap`

## FIB-TOP-DOWN-MEMO(*n*)

```
1  let m[0..n] be a new array ("memory")
2  m[0] = 0 // fib(0)
3  m[1] = 1 // fib(1)
4  for i = 2 to n
5      m[i] = -∞
6  return FIB-TOP-DOWN-AUX(n, m)
```

Berechne Fibonacci-zahl  $F_n$ ,  
Wrapper-Funktion

Speicherplatz für Memoisation

## FIB-TOP-DOWN-MEMO-AUX(*n*, *m*)

```
7  if m[n] ≥ 0 // problem already solved?
8      return m[n]
9  else
10     result = FIB-TOP-DOWN-MEMO-AUX(n-1, m) +
11              FIB-TOP-DOWN-MEMO-AUX(n-2, m)
12     m[n] = result
11     return result
```

Rekursion

Quellcode: `Fibonacci.java`  
Methode: `fibTopDownMemoization`

# Fibonacci: Iterativ / Bottom Up

## ❑ Idee

- $F_0, F_1$  sind bekannt.
- Berechne der Reihe nach  $F_2, F_3, F_4, \dots$ 
  - Man merkt sich jeweils *Vorgänger* und *Vorvorgänger*

## ❑ Übung: Java-Programm schreiben

Quellcode: Fibonacci.java  
Methode: fibBottomUp

- Siehe Quellcode

## ❑ Vergleich der Laufzeiten:

Variante	Laufzeit gemessen für $n = 50$	Laufzeit in O-Notation
Top-Down	86 s	$O(2^n)$
Top-Down mit Memoisation	0 s	$O(n)$
Bottom-Up	0 s	$O(n)$

# Erstes Fazit

## ❑ **Animation**

- <https://www.cs.usfca.edu/~galles/visualization/DPFib.html>

## ❑ **Lehre:** Nicht ohne Nachdenken Rekursion verwenden!

- Wiederholen sich Aufrufe im Rekursionsbaum?
- Falls ja, führt das evtl. zu ineffizienter Laufzeit.

## ❑ **Memoisation** senkt die Laufzeit dramatisch

- Auf Kosten des Speichers!

## ❑ **Dynamische Programmierung**

- Iterativer / **Bottom-Up** Ansatz.
- Rekursives Problemlösen wird ersetzt durch Iteration und Abspeichern der bereits berechneten Teilergebnisse.
- Dynamische Programmierung **löst jedes Teilproblem** einer Rekursion **nur einmal**. Speichern des Ergebnisses ähnlich wie bei Memoisation in einer Tabelle.



# Überblick

---

- ❑ Einführung: Fibonacci-Zahlen
- ❑ **Rod-Cutting**
- ❑ Längste gemeinsame Teilfolge
- ❑ Levenshtein-Editierdistanz

# Das "Rod-Cutting Problem"

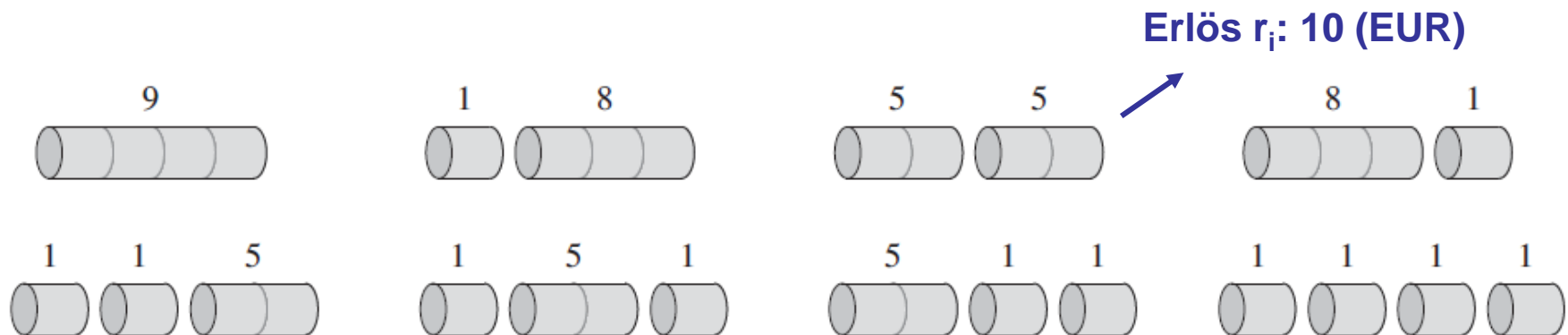
- ❑ Deutsch: (1-dimensionales) **Zuschnittproblem**
- ❑ Wie unterteilt man einen langen Stahlstab in kleine Stücke und maximiert gleichzeitig den erzielten Erlös?
- ❑ **Annahmen**
  - Jeder Schnitt ist kostenlos.
  - Die Länge des Ausgangsstabes **und** der Zuschnitte sind ganzzahlige Zentimeter-Werte.
  - Die kleinste Länge eines Zuschnittes ist 1 cm.
- ❑ **Eingabe**
  - Länge  $n$  (in Zentimeter) des originalen Stahlstabes
  - Tabelle mit Preisen  $p_i$  für ein Stahlstab der Länge  $i$  Zentimeter
- ❑ **Ausgabe**
  - Maximal erzielbarer Erlös (engl. "revenue")  $r_i$
  - Zuschnitt gemäß obiger Annahmen!
  - Erlös entspricht Summe der Preise für die Zuschnitte.



# Rod-Cutting: Beispiel

Länge $i$	1 (cm)	2	3	4	5	6	7	8
Preis $p_i$	1	5	8	9	10	17	17	20

- Anzahl Möglichkeiten, um Stab der **Länge  $n=4$**  zuzuschneiden?
  - $2^{n-1} = 8 \rightarrow$  exponentiell viele Möglichkeiten!
  - Man kann nach jedem Zentimeter teilen oder nicht teilen!
  - Bei einem sehr hohen Preis für  $p_8$  müsste man unter Umständen gar nicht teilen.
- Was ist der maximale Erlös bei einem Stab der **Länge  $n=4$** ?



# Rod-Cutting: Beispiel

Länge $i$	1 (cm)	2	3	4	5	6	7	8
Preis $p_i$	1	5	8	9	10	17	17	20

## □ **Definition $r_i$ :**

- Maximaler Erlös (engl.: "revenue") für Stab der Länge  $i$

## □ Bestimme die Werte von $r_i$ ("Augenmaß"):

$i$	$r_i$	optimale Lösung
1	1	1 (kein Zuschnitt)
2	5	2 (kein Zuschnitt)
3	8	3 (kein Zuschnitt)
4	10	2 + 2 (siehe vorherige Folie)
5	13	2 + 3
6	17	6 (kein Zuschnitt)
7	18	???
8	22	???

Lösung (=Zuschnitt)  
kann durch Summe  
der einzelnen  
Stablängen  
beschrieben  
werden.

**Beispiel:** 2+3 ergibt  
maximalen Erlös  
 $r_5$  für Stab der  
Länge 5

# Finde optimale Substruktur (1)

- **Beobachtung:** Optimale Lösung ist aufgebaut aus optimalen Lösungen von Teilproblemen.
  - Nach einem Schnitt hat man 2 kleinere Teilprobleme.
  - Für **beide** muss man die optimale Lösung berechnen.
  
- **Optimaler Erlös  $r_n$**  ist das **Maximum** aus
  - $p_n$ : Erlös, falls man gar nicht unterteilt.
  - $r_1 + r_{n-1}$ : Maximaler Erlös aus Stab der Länge 1 und der Länge  $n-1$
  - $r_2 + r_{n-2}$ : Maximaler Erlös aus Stab der Länge 2 und der Länge  $n-2$
  - ...
  - Kurz:  $r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1)$
  
- **Beispiel:  $n=7$** 
  - Mögliche optimale Lösung unterteilt Stab in die Längen 3 und 4.
  - Die optimale Lösung für das Problem der Länge  $n=4$  (siehe Vorvorgängerfolie) wird deshalb in der optimalen Lösung für das Gesamtproblem  $n=7$  wiederverwendet.

# Finde optimale Substruktur (2)

- ❑ **Vereinfachung:** Es kommt nicht darauf an, in welcher Reihenfolge man zuschneidet, z.B. egal ob
  - $r_7 = r_4 + r_3 = r_2 + r_2 + r_3$  ("schneide zuerst rechts") oder
  - $r_7 = r_2 + r_5 = r_2 + r_2 + r_3$  ("schneide zuerst links")
- ❑ Idee: Teile immer Stab (gedanklich) in
  - **Erstes Stück** der Länge  $i$ , das **links** abgeschnitten wird und später **nie** weiter unterteilt wird.
  - **Reststück** der Länge  $n-i$ , das **rechts** übrig bleibt und ggfs. weiter unterteilt wird.
  - Rekursion also nur für Reststück!
  - Sonderfall, dass **überhaupt kein** Zuschnitt nötig ist:
    - Erstes Stück hat Länge  $n$  mit Erlös  $r_n$
    - Reststück hat Länge 0 mit Erlös  $r_0 = 0$
- ❑  $r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$

# Rekursive Definition der Lösung

Berechne maximalen Erlös  $r_n$  für Stab der Länge  $n$ .  
 $p$  sei Array, das Preise speichert.

```
CUT-ROD-REC( $p, n$ )
1  if  $n == 0$ 
2    return 0
3   $q = -\infty$       // store maximum revenue seen so far
4  for  $i = 1$  to  $n$ 
5     $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$ 
6  return  $q$       // returns optimal revenue  $r_n$ 
```

Quellcode: RodCutting.java  
Methode: cutRodRec

- ❑ Direkte, rekursive Implementierung der identifizierten, optimalen Substruktur
  - Algorithmus gibt maximalen Erlös  $r_n$  aus, aber nicht wie man zuschneiden muss.
- ❑ Asymptotische Laufzeit katastrophal: **Exponentiell!**
  - Grund: Rekursion löst Teilprobleme erneut, obwohl diese vorab bereits gelöst wurden.
  - Übung: Zeichne Rekursionsbaum für das Beispiel von Folie 10 und den Aufruf von  $n=4$

# Berechnung der optimalen Lösung

## □ **Ziel**

- Jede Teillösung soll nur einmal berechnet werden.

## □ **Idee**

- Speichere Ergebnisse der Teillösungen in einer **Tabelle**.
- Schlage das bereits berechnete Ergebnis nach, und zwar jedes Mal wenn es erneut benötigt wird.

## □ **2 Ansätze**

- **Top-down** mit **Memoisation** (engl. "Memoization")
- **Bottom-up** / **Dynamische Programmierung**



# Top-Down mit Memoisation

## □ Rekursiver Ansatz

## □ Memoisation:

- Erinnern, was man bereits berechnet hat.
- Bereits berechnete Ergebnisse werden hier im Array  $r$  gespeichert.

## □ Lösen eines Teilproblems

- Schau in Tabelle nach ob Lösung bereits existiert.
- Falls nein, berechne Lösung und speichere Lösung in Tabelle.

## □ Laufzeit: $\Theta(n^2)$

- Ohne Beweis

Berechne maximalen Erlös  $r_n$  für Stab der Länge  $n$ .  
 $p$  sei Array, das Preise speichert.

### MEMOIZED-CUT-ROD( $p, n$ )

```
1  let  $m[0..n]$  be a new array ("memory")
2  for  $i = 0$  to  $n$ 
3       $m[i] = -\infty$  // memo: initialize
4  return MEMOIZED-CUT-ROD-AUX( $p, n, m$ )
```

### MEMOIZED-CUT-ROD-AUX( $p, n, m$ )

```
5  if  $m[n] \geq 0$  // memo: already solved?
6      return  $m[n]$ 
7  if  $n == 0$ 
8       $q = 0$ 
9  else
10      $q = -\infty$ 
11     for  $i = 1$  to  $n$ 
12          $q = \max(q, p[i] +$ 
13              $\text{MEMOIZED-CUT-ROD-AUX}(p, n - i, m))$ 
13      $m[n] = q$  // memo: save solution
14     return  $q$ 
```

# Bottom-Up, Dynamische Programmierung

## Iterativer Ansatz

- Berechne von "unten nach oben", d.h. erst maximaler Erlös für Stab der Länge 1, dann Länge 2, usw.
- Löst man ein Teilproblem, kann man sicher sein, dass man schon alle kleineren Teilprobleme gelöst hat.

## Erklärung

- Zeile 3: for-Schleife berechnet Lösung für Teilproblem der Größe  $j$
- Zeile 5: Teste alle möglichen Zerlegungen (an  $i$ . ter Position).

Berechne maximalen Erlös  $r_n$  für Stab der Länge  $n$ .  
 $p$  sei Array, das Preise speichert.

### BOTTOM-UP-CUT-ROD( $p, n$ )

```
1  let  $r[0..n]$  be a new array
2   $r[0] = 0$ 
3  for  $j = 1$  to  $n$ 
4       $q = -\infty$ 
5      for  $i = 1$  to  $j$ 
6          if  $q < p[i] + r[j - i]$ 
7               $q = p[i] + r[j - i]$ 
8       $r[j] = q$ 
9  return  $r[n]$ 
```

Quellcode: RodCutting.java  
Methode: cutRodBottomUp

Länge $i$	1 (cm)	2	3	4
Preis $p_i$	1	5	8	9

## Laufzeit: $\Theta(n^2)$

- "2 verschachtelte Schleifen"

Index	0	1	2	3	4
$r[i]$	0	1	5	8	10

# Publikums-Joker: Rod Cutting

Wie hoch (*O-Notation*) ist der Speicherbedarf des "Bottom-Up" Algorithmus beim Rod Cutting Problem, falls der Stab die Länge  $n$  hat?

- A.  $O(1)$
- B.  $O(\log n)$
- C.  $O(n)$
- D.  $O(n^2)$



# Rekonstruktion der Lösung

- Bislang wurde nur der optimale Erlös  $r_n$  berechnet.
- Woher weiß man aber nun, wie man einen Stahlstab zuschneiden muss (= an welchen Stellen) um den optimalen Erlös zu erhalten?
- **Idee:** Speichere die Schnittpositionen im *Bottom-Up* Ansatz mit.
  - Dazu: Gesondertes **Array  $s$**
  - Speichere die Größe  **$i$**  des ("ersten") linken Stückes (siehe Folie 14), falls gerade Teilproblem der Größe  **$j$**  gelöst wird (Zeile 8)

## EXTENDED-BOTTOM-UP-CUT-ROD( $p, n$ )

```
1  let  $r[0..n]$  and  $s[0..n]$  be new arrays
2   $r[0] = 0$ 
3  for  $j = 1$  to  $n$ 
4       $q = -\infty$ 
5      for  $i = 1$  to  $j$ 
6          if  $q < p[i] + r[j - i]$ 
7               $q = p[i] + r[j - i]$ 
8               $s[j] = i$ 
9       $r[j] = q$ 
10 return  $r$  and  $s$ 
```

*Berechne maximalen Erlös  $r_n$  sowie Position der Schnitte  $s$  für Stab der Länge  $n$ .*

# Rekonstruktion der Lösung (2)

## EXTENDED-BOTTOM-UP-CUT-ROD( $p, n$ )

```

1  let  $r[0..n]$  and  $s[0..n]$  be new arrays
2   $r[0] = 0$ 
3  for  $j = 1$  to  $n$ 
4       $q = -\infty$ 
5      for  $i = 1$  to  $j$ 
6          if  $q < p[i] + r[j - i]$ 
7               $q = p[i] + r[j - i]$ 
8               $s[j] = i$ 
9       $r[j] = q$ 
10 return  $r$  and  $s$ 
    
```

*Im Array  $r$  stehen die maximalen Erlöse, im Array  $s$  die dazugehörigen Zuschnitt-positionen.*

- Welche Werte werden für das Array  $r$  und  $s$  beim Aufruf EXTENDED-BOTTOM-UP-CUT-ROD für  $n = 8$  berechnet?

Länge $i$	1 (cm)	2	3	4	5	6	7	8
Preis $p_i$	1	5	8	9	10	17	17	20

Index $i$	0	1	2	3	4	5	6	7	8
$r[i]$	0	1	5	8					
$s[i]$									

# Rekonstruktion der Lösung (3)

## EXTENDED-BOTTOM-UP-CUT-ROD( $p, n$ )

```

1  let  $r[0..n]$  and  $s[0..n]$  be new arrays
2   $r[0] = 0$ 
3  for  $j = 1$  to  $n$ 
4       $q = -\infty$ 
5      for  $i = 1$  to  $j$ 
6          if  $q < p[i] + r[j - i]$ 
7               $q = p[i] + r[j - i]$ 
8               $s[j] = i$ 
9       $r[j] = q$ 
10 return  $r$  and  $s$ 
    
```

Gibt aus, in welcher Reihenfolge man  
in welcher Größe Stücke abschneiden muss,  
um den maximalen Erlös  $r[n]$  zu erhalten.

## PRINT-CUT-ROD-SOLUTION( $p, n$ )

```

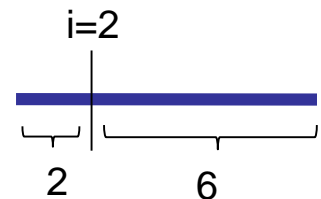
1  ( $r, s$ ) = EXTENDED-BOTTOM-UP-
    CUT-ROD( $p, n$ )
2  while  $n > 0$ 
3      print  $s[n]$ 
5       $n = n - s[n]$ 
    
```

## Ergebnis:

Index $i$	0	1	2	3	4	5	6	7	8
$r[i]$	0	1	5	8	10	13	17	18	22
$s[i]$	0	1	2	3	2	2	6	1	2

### □ Ausgabe von PRINT-CUT-ROD-SOLUTION im konkreten Fall $n=8$ ?

- Nachschlagen bei  $i=8$ , Ausgabe von 2
  - $n$  um 2 reduzieren.
- Nachschlagen bei  $i=6$  (kein Schnitt!), Ausgabe 6 ausgegeben,
  - $n$  um 6 reduzieren.
- Dann Abbruch, da  $n = 0$ .



# Dynamische Programmierung: Allgemein

- ❑ Finde **optimale Substruktur** der Lösung
  - Optimale (Gesamt)Lösung muss sich aus optimalen Teillösungen kleinerer Probleme herleiten lassen.
  - $r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$
- ❑ **Rekursive Definition** der Lösung
  - Ergibt sich meist unmittelbar aus optimalen Substruktur.
  - Die direkte Variante führt aber meist zu "katastrophalen" Laufzeiten.
- ❑ Berechne optimale Lösung
  - Jede Teillösung soll nur einmal berechnet werden → Tabelle.
  - 2 Möglichkeiten: **Top-down mit Memoisation** oder **Bottom-Up**.
- ❑ Rekonstruktion der Lösung.
  - Zurückhangeln in Tabelle.

# Überblick

---

- ❑ Einführung: Fibonacci-Zahlen
- ❑ Rod-Cutting
- ❑ **Längste gemeinsame Teilfolge**
- ❑ Levenshtein-Editierdistanz



# Längste gemeinsame Teilfolge (LGT)

□ Englisch: Longest Common Subsequence (LCS)

□ **Eingabe:** Gegeben seien 2 Textsequenzen

- $X = \langle x_1, \dots, x_m \rangle$
- $Y = \langle y_1, \dots, y_n \rangle$

□ **Ausgabe:**

- Längste gemeinsame Teilfolge
- **Ungleich:** Längster gemeinsamer Substring!
  - Eine Teilfolge muss **nicht** aus aufeinanderfolgenden Zeichen bestehen, aber die Zeichen müssen in der korrekten Reihenfolge kommen.


□ **Beispiel**

- $X = \text{s p r i n g t i m e}$
- $Y = \text{p i o n e e r}$

Die längste gemeinsame Teilfolge ist: "pine"

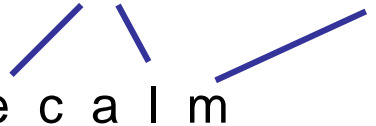
# Weitere Beispiele

## □ Beispiel 1

- $X = \text{h o r s e b a c k}$
  - $Y = \text{s n o w f l a k e}$
- 

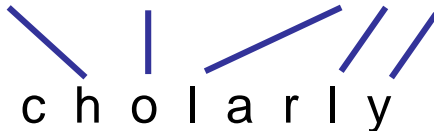
Die längste gemeinsame Teilfolge ist: "oak"

## □ Beispiel 2

- $X = \text{m a e l s t r o m}$
  - $Y = \text{b e c a l m}$
- 

Die längste gemeinsame Teilfolge ist z.B.: "elm"

## □ Beispiel 3

- $X = \text{h e r o i c a l l y}$
  - $Y = \text{s c h o l a r l y}$
- 

Die längste gemeinsame Teilfolge ist: "holly"

# Anwendungen in der Praxis

## □ Sequenzierung von DNA und Proteinen

- DNA-Sequenz = String über Alphabet {A, C, G, T}.
- Wie ähnlich sind zwei DNAs?
  - S1= ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
  - S2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA
  - LGT = GTCGTCGGAAGCCGGCCGAA
- Längste gemeinsame Teilfolge kann Maß für Ähnlichkeit sein.

## □ Grundlage des Tools `diff`

- [https://en.wikipedia.org/wiki/Diff\\_utility#Algorithm](https://en.wikipedia.org/wiki/Diff_utility#Algorithm)

## □ Wird verwendet in Versionsverwaltungssystemem wie `git`.

- [https://en.wikipedia.org/wiki/Longest\\_common\\_subsequence\\_problem](https://en.wikipedia.org/wiki/Longest_common_subsequence_problem)

# "Brute-Force" Ansatz

## □ Eingabe

- $X = \langle x_1, \dots, x_m \rangle$
- $Y = \langle y_1, \dots, y_n \rangle$

## □ Prüfe für jede (!) Teilfolge von $X$ ob sie Teilfolge von $Y$ ist.

## □ Wie viele mögliche Teilfolgen von $X$ gibt es?

- $2^m$  (ohne Beweis)

## □ Man müsste also $2^m$ Teilfolgen prüfen, jede Prüfung benötigt $\Theta(n)$

## □ Laufzeit insgesamt: $\Theta(n2^m)$

# Finde optimale Substruktur

## □ Notation

- $X_i$  = Präfix  $\langle x_1, \dots, x_i \rangle$ , d.h. alles bis zur  $i$ -ten Position
- $Y_i$  = Präfix  $\langle y_1, \dots, y_i \rangle$ , d.h. alles bis zur  $i$ -ten Position
- Beispiel:  $X = \text{maelstrom}$ , dann ist  $X_3 = \text{mae}$

## □ Aussage (ohne Beweis): Es sei $Z = \langle z_1, \dots, z_k \rangle$ eine mögliche LGT von $X$ und $Y$ .

### 1. Letzter Buchstabe ist gleich und gehört damit zwingend zur LGT!

- $x_m = y_n \rightarrow z_k := x_m = y_n$  und  $Z_{k-1}$  ist eine LGT von  $X_{m-1}$  und  $Y_{n-1}$ .
- Beispiel:  $X = \text{maelstrom}$  und  $Y = \text{becalm}$

### 2. Letzter Buchstabe verschieden, LGT enthält nicht letzten Buchstaben von $X$ .

- $x_m \neq y_n$  und  $z_k \neq x_m \rightarrow Z$  ist auch LGT von  $X_{m-1}$  und  $Y$ .

### 3. Letzter Buchstabe verschieden, LGT enthält nicht letzten Buchstaben von $Y$

- $x_m \neq y_n$  und  $z_k \neq y_n \rightarrow Z$  ist eine LGT von  $X$  und  $Y_{n-1}$
- Beispiel:  $X = \text{springtime}$  und  $Y = \text{pioneer}$

- Man muss alle 3 Fälle untersuchen.
- Mindestens 1 Sequenz wird verkleinert.

# Rekursive Definition der Lösung

## □ Erinnerung: Notation – Zusammenfassung

- $X = \langle x_1, \dots, x_m \rangle$ , Folge mit  $m$  Zeichen
- $Y = \langle y_1, \dots, y_n \rangle$ , Folge mit  $n$  Zeichen
- $X_i$  = prefix  $\langle x_1, \dots, x_i \rangle$ , d.h. alles bis zur  $i$ -ten Position

## □ Definition: $c[i, j]$ = Länge der LGT von $X_i$ und $Y_j$ .

- **Länge** der LGT falls nur Präfixe der Länge  $i$  bzw. der Länge  $j$  von  $X$  und  $Y$  betrachtet werden.
- Gesamtproblem also:  $c[m, n]$

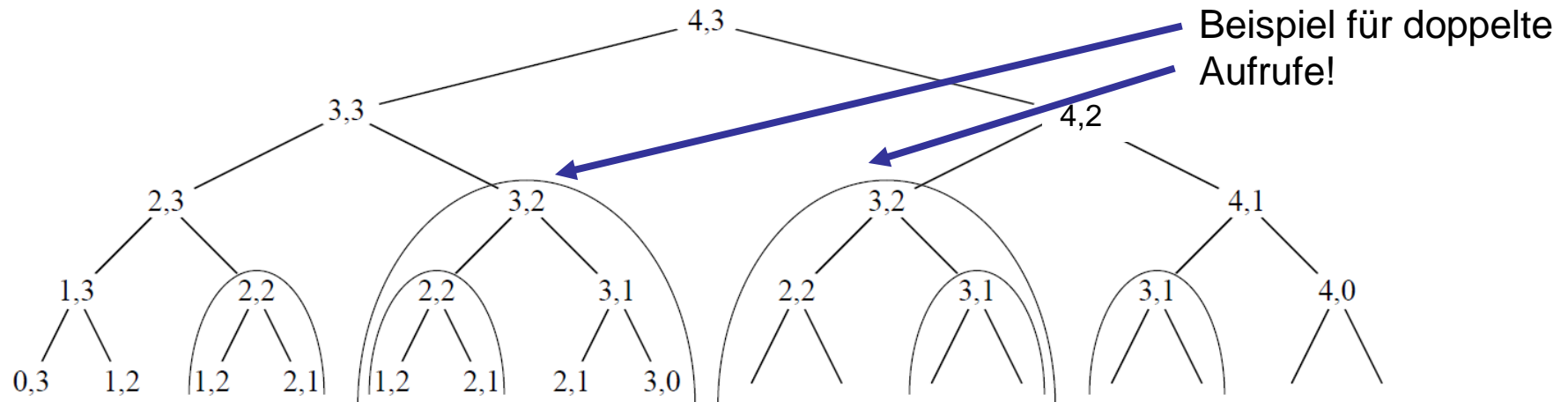
*Falls 2/3 der letzten Folie*

*Fall 1 der letzten Folie*

□ **Rekursion:** 
$$c[i, j] = \begin{cases} 0 & \text{falls } i = 0 \text{ oder } j = 0 \\ c[i - 1, j - 1] + 1 & \text{falls } i, j > 0 \text{ und } x_i = y_j \\ \max(c[i - 1, j], c[i, j - 1]) & \text{falls } i, j > 0 \text{ und } x_i \neq y_j \end{cases}$$

# Rekursive Definition der Lösung

- Mit der vorherigen Rekursionsformel ließe sich bereits ein Programm schreiben.
- **Problem:** Sehr ineffizient → exponentielle Laufzeit!
  - Viele Teilprobleme werden mehrfach gelöst, siehe Rekursionsbaum!
- **Abhilfe:** Mitspeichern von Ergebnissen in einer Tabelle.
  - Beobachtung: Beim Berechnen von  $c[m, n]$  gibt es nur  $m \cdot n$  verschiedene Teilprobleme.
  - Dynamische Programmierung verspricht also eine effiziente Lösung!



# LGT mit dynamischer Programmierung

Berechne LGT für die Zeichenketten  $X$  bzw.  $Y$  mit den Längen  $m$  bzw.  $n$ .

## LCS-LENGTH( $X, Y, m, n$ )

```
1  let  $b[1..m, 1..n]$  and  $c[0..m, 0..n]$  be new tables
2  for  $i = 1$  to  $m$ 
3       $c[i, 0] = 0$ 
4  for  $j = 0$  to  $n$ 
5       $c[0, j] = 0$ 
6  for  $i = 1$  to  $m$ 
7      for  $j = 1$  to  $n$ 
8          if  $x_i == y_j$ 
7           $c[i, j] = c[i - 1, j - 1] + 1$ 
8           $b[i, j] = "\nwarrow"$ 
9          else
10             if  $c[i - 1, j] \geq c[i, j - 1]$ 
11                  $c[i, j] = c[i - 1, j]$ 
12                  $b[i, j] = "\uparrow"$ 
13             else
14                  $c[i, j] = c[i, j - 1]$ 
15                  $b[i, j] = "\leftarrow"$ 
16  return  $c$  and  $b$ 
```

Initialisierung für Fall, dass eine der beiden Teilfolgen leer ist.

Berechne zeilenweise

Fall 1: Letzter Buchstabe gleich

Fall 2/3: Letzter Buchstabe verschieden.

## □ Eingabe:

- Sequenzen  $X$  und  $Y$

## □ $c[m, n]$ : Länge der LGTs

- 2-dimensionales Array.
- $c[i, j]$  speichert in der  $i$ . Zeile und  $j$ . Spalte die Länge der LGT falls man  $i$  Zeichen von  $X$  und  $j$  Zeichen von  $Y$  betrachtet.
- Die Tabelle wird zeilenweise berechnet (1. Zeile, dann 2. Zeile, usw.)

## □ $b[1..m, 1..n]$ erlaubt Rekonstruktion der Lösung ("Was ist die LGT")

- $b[i, j]$  zeigt auf Tabelleneintrag, dessen Teillösung verwendet wurde um  $c[i, j]$  zu berechnen.
- Siehe nächste Folie



# Beispiel

□ Berechne LGT von  $X = \langle \text{ABCBDAB} \rangle$  und  $Y = \langle \text{BDCABA} \rangle$ :

- Die ersten beiden Zeilen sind bereits vorgegeben.
- Berechne die nächste 3. Zeile!

**LCS-LENGTH( $X, Y, m, n$ )**

```
1  let  $b[1..m, 1..n]$  and  $c[0..m, 0..n]$  be tables
2  for  $i = 1$  to  $m$ 
3     $c[i, 0] = 0$ 
4  for  $j = 0$  to  $n$ 
5     $c[0, j] = 0$ 
6  for  $i = 1$  to  $m$ 
7    for  $j = 1$  to  $n$ 
8      if  $x_i == y_j$ 
7       $c[i, j] = c[i - 1, j - 1] + 1$ 
8       $b[i, j] = "\nwarrow"$ 
9      else
10       if  $c[i - 1, j] \geq c[i, j - 1]$ 
11          $c[i, j] = c[i - 1, j]$ 
12          $b[i, j] = "\uparrow"$ 
13       else
14          $c[i, j] = c[i, j - 1]$ 
15          $b[i, j] = "\leftarrow"$ 
16  return  $c$  and  $b$ 
```

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

# Rekonstruktion: Wie sieht die LGT aus?

## □ $b[i,j]$

- Pfeile
- Zeigt auf Teilproblem, das verwendet wurde um die LGT für  $X_i$  und  $Y_j$  zu bestimmen.

## □ Starte bei $b[m,n]$

## □ Laufe durch die Tabelle

## □ Falls " $\nwarrow$ "

- Element gehört zur LGT
- Fall 1 der optimalen Substruktur (Folie 27)

Ergebnistabelle [1]

- Zahlenwerte:  $c[i,j]$ , d.h. Länge der LGT
- Pfeile:  $b[i,j]$ , benötigt zur Rekonstruktion der LGT

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

**BCBA ist LGT**

# Rekonstruktion der LGT

## PRINT-LCS(b,X,i,j)

```

1  if i == 0 or j == 0
2    return
3  if b[i,j] == "↖"
4    PRINT-LCS(b,X,i-1,j-1)
5    print xi
6  elseif b[i,j] == "↑"
7    PRINT-LCS(b,X,i-1,j)
8  else
9    PRINT-LCS(b,X,i,j-1)
    
```

- ❑ Ergebnis
  - LGT=B C B A
- ❑ Man hangelt sich an den Pfeilen ausgehend von b[7,6] zurück.

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

# Publikums-Joker: LGT

Gegeben seien zwei Strings:

1/ "PQRSTPQRS" und

2/ "PRATPBRQRPS"

Welche Länge hat die längste gemeinsame Teilfolge?

A. 9

B. 8

C. 7

D. 6



- ❑ Animation
  - <https://www.cs.usfca.edu/~galles/visualization/DPLCS.html>
- ❑ Dynamische Programmierung erlaubt hier eine effiziente Lösung des Problems.
- ❑ Brute Force:  $\Theta(n2^m)$
- ❑ Dynamische Programmierung:  $\Theta(mn)$

# Überblick

---

- ❑ Einführung: Fibonacci-Zahlen
- ❑ Rod-Cutting
- ❑ Längste gemeinsame Teilfolge
- ❑ **Levenshtein-Editierdistanz**

# Levenshtein – Editierdistanz (LSD)

## □ **Anwendung:** Unterschied bzw. Ähnlichkeiten von Zeichenketten

- Bioinformatik, DNA Vergleich
- Plagiaterkennung
- Fuzzy-Suche in Suchmaschinen und Datenbanken
- Spamfilter

## □ **Definition:** Editierdistanz

- *Minimale* Anzahl an Editieroperationen, um Zeichenkette  $X$  in Zeichenkette  $Y$  zu überführen.
- $X = \langle x_1, \dots, x_m \rangle$  und  $Y = \langle y_1, \dots, y_n \rangle$  Zeichenketten der Länge  $m$  bzw.  $n$
- Editieroperationen: Einfügen, Löschen, Substitution
- Eng verwandt zur "Längsten Gemeinsamen Teilfolge"

## □ **Beispiel:**

○ hello  $\xRightarrow{\text{Substitution}}$  wello  $\xRightarrow{\text{Substitution}}$  welto  $\xRightarrow{\text{Löschen}}$  welt

# Finde optimale Substruktur

- ❑ Betrachte letzten Buchstaben beider Zeichenketten
  - Stimmt überein → keine Aktion nötig!
  - Stimmt nicht überein: Der letzte Buchstabe der 1. Zeichenkette wird entweder *eingefügt*, *gelöscht* oder *ersetzt*, um zur 2. Zeichenkette zu kommen.
- ❑ Betrachtet man Zeichenketten ohne letzten Buchstaben, gelangt man zu kleinerem Teilproblem.
- ❑ **Notation**
  - Eingabe: Zeichenketten  $X = \langle x_1, \dots, x_m \rangle$  und  $Y = \langle y_1, \dots, y_n \rangle$  der Länge  $m$  bzw.  $n$
  - Präfix einer Zeichenkette (hier von  $X$ ):  $X_i = \langle x_1, \dots, x_i \rangle$  mit  $i \leq m$
  - Beispiel:  $X = \text{hallowelt}$ , dann ist  $X_3 = \text{hal}$
- ❑ **Definition:  $D[i, j]$  = Editierdistanz**
  - Kosten falls man nur Präfix der Länge  $i$  ( $=X_i$ ) bzw. Präfix der Länge  $j$  ( $=Y_j$ ) betrachtet.
  - Kosten für Gesamtproblem:  $D[m, n]$



# Publikums-Joker: Levenshtein

Wie hoch sind die Editierkosten für  $D[0, j]$ ?

- A. Kann man aufgrund der Angabe nicht sagen, zwischen 0 und  $j$ .
- B. 0
- C.  $j/2$
- D.  $j$



# Rekursive Definition der Lösung

## □ Erinnerung: Notation – Zusammenfassung

- $X = \langle x_1, \dots, x_m \rangle$ , Folge mit  $m$  Zeichen
- $Y = \langle y_1, \dots, y_n \rangle$ , Folge mit  $n$  Zeichen
- $X_i$  = prefix  $\langle x_1, \dots, x_i \rangle$ , d.h. alles bis zur  $i$ -ten Position

## □ Definition: $D[i, j]$ = Levenshtein-Editierdistanz von $X_i$ und $Y_j$ .

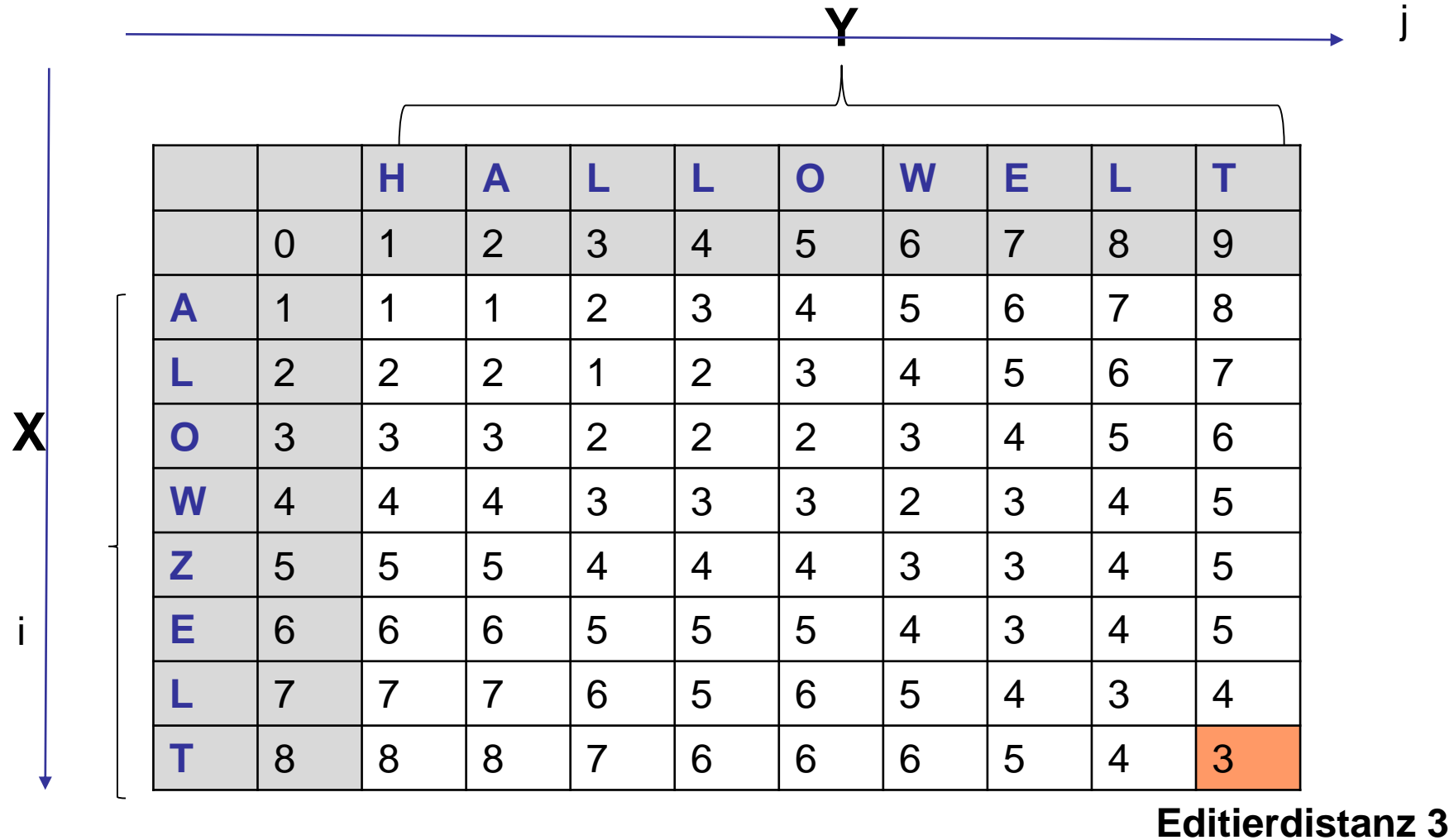
- Editierdistanz falls nur Präfixe der Länge  $i$  bzw. der Länge  $j$  von  $X$  und  $Y$  betrachtet werden.

□ Rekursion:  $D[i, j] = \left\{ \begin{array}{ll} D[i-1, j-1] + 0 & \text{Match} \\ D[i-1, j-1] + 1 & \text{Substitution} \\ D[i, j-1] + 1 & \text{Einfügen} \\ D[i-1, j] + 1 & \text{Löschen} \end{array} \right\}$  Fallunterscheidung bzgl. des letzten Buchstabens

## □ Terminierung

- $D[0, 0] = 0$
- $D[0, j] = j$
- $D[i, 0] = i$

# Beispiel: Levenshtein-Editierdistanz



# Beispiel: Rekonstruktion der optimalen Lösung

**Y**

**X**

		H	A	L	L	O	W	E	L	T
	0	1	2	3	4	5	6	7	8	9
A	1	1	1	2	3	4	5	6	7	8
L	2	2	2	1	2	3	4	5	6	7
O	3	3	3	2	2	2	3	4	5	6
W	4	4	4	3	3	3	2	3	4	5
Z	5	5	5	4	4	4	3	3	4	5
E	6	6	6	5	5	5	4	3	4	5
L	7	7	7	6	5	6	5	4	3	4
T	8	8	8	7	6	6	6	5	4	3

**Editierdistanz 3**

Hinweis: Es kann mehrere Lösungen geben.

# Weitere Beispiele für dynamische Programmierung

- ❑ Rod-Cutting-Problem
- ❑ Längste gemeinsame Teilfolge
- ❑ Rucksackproblem
- ❑ Floyd-Warshall Algorithmus zur Berechnung aller kürzesten Pfade (ASAP)
- ❑ Kettenmultiplikation von Matrizen
- ❑ Zahlreiche String-Algorithmen
- ❑ Optimale binäre Suchbäume, falls bekannt ist wie häufig welche Schlüssel gesucht werden.
- ❑ ...

# Zusammenfassung

- ❑ Rekursives Problemlösen wird ersetzt durch Iteration und Abspeichern der bereits berechneten Teilergebnisse.
- ❑ "Verbesserung" von Divide-and-Conquer
  - Teilprobleme überlappen. Jedes Teilproblem wird dennoch nur einmal gelöst
  - Viele exponentielle Probleme lassen sich damit in polynomieller Zeit lösen!
- ❑ Wird häufig verwendet für Optimierungsprobleme
- ❑ Beispiele
  - Fibonacci
  - Rod Cutting
  - Längste gemeinsame Teilfolge
  - Rucksackproblem
  - ....

# Quellenverzeichnis

---

- [1] Cormen, Leiserson, Rivest and Stein. *Introduction to Algorithms*, Third Edition, The MIT Press, 2009.
- [2] Ottmann, Widmayer. *Algorithmen und Datenstrukturen*, 5. Auflage, Spektrum Akademischer Verlag, 2012.
- [3] <https://www.luettundfien.de/shop/out/pictures/master/product/3/omm-memory-3.jpg>