



# Übungslektion 10 – Dynamic Programming II

Informatik II

29. / 30. April 2025

# Heutiges Programm

---

- Wiederholung: Rod Cutting
- Längste gemeinsame Teilsequenz
- DNA-Sequenzvergleich
- Wrap-Up

# 1. Wiederholung: Rod Cutting

---

# Wiederholung: Rod Cutting

- Um eine optimale Lösung zu finden, müssen möglicherweise viele Optionen evaluiert werden.

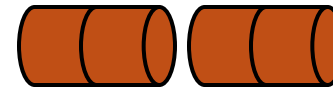
(a)



(b)



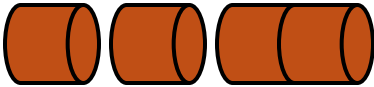
(c)



(d)



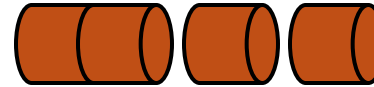
(e)



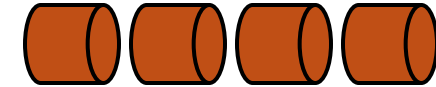
(f)



(g)



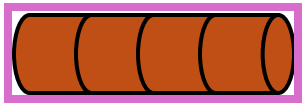
(h)



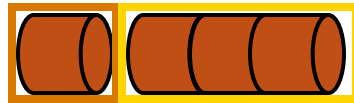
# Wiederholung: Rod Cutting

- Wir können diese Teilprobleme mit Hilfe von Rekursion berechnen.

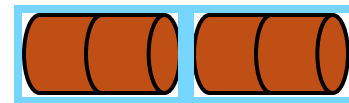
(a)



(b)



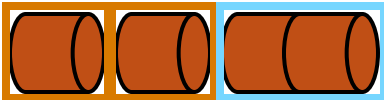
(c)



(d)



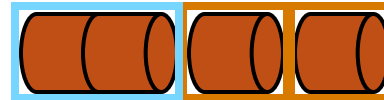
(e)



(f)



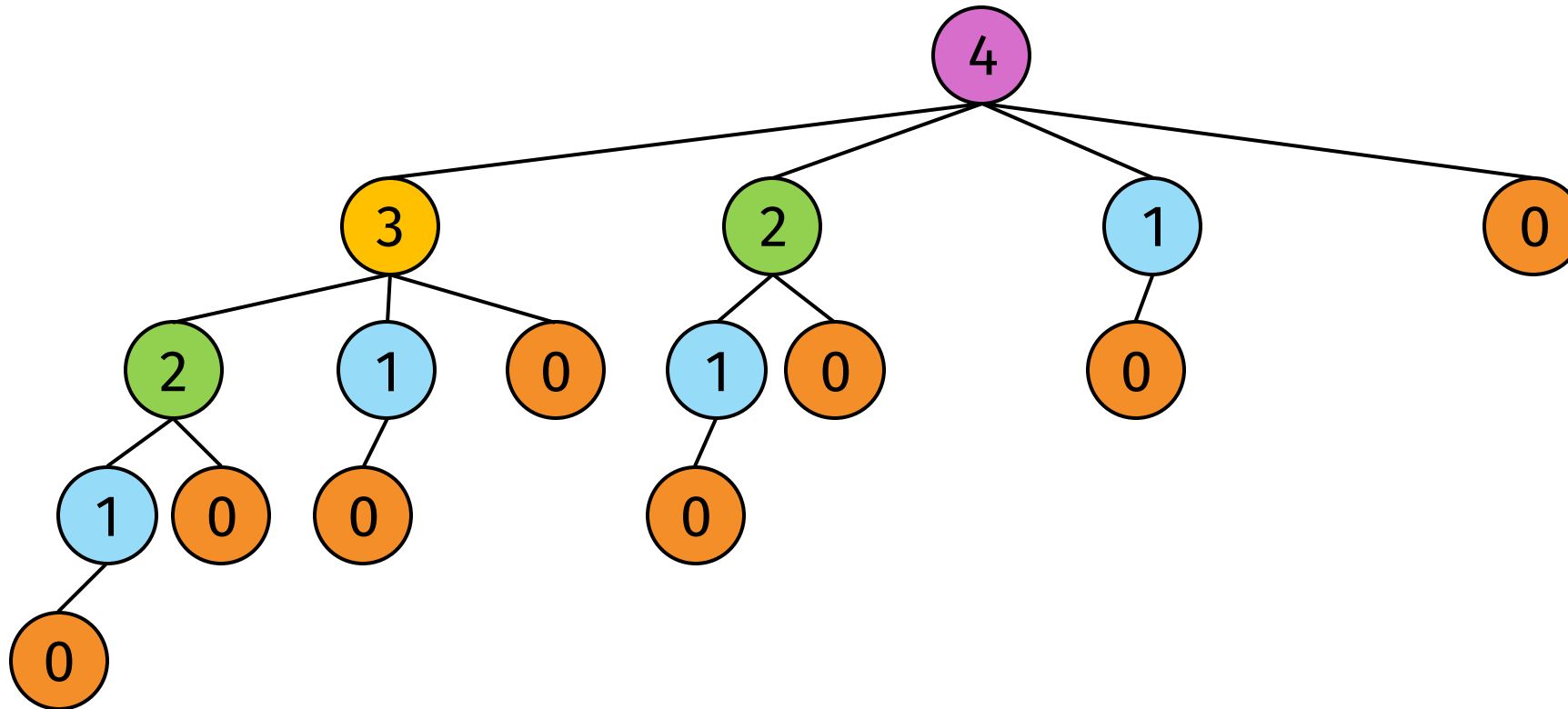
(g)



(h)



# Wiederholung: Rod Cutting

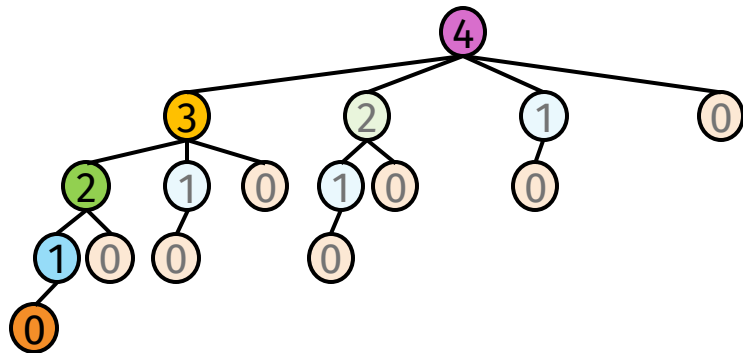


- Viele Teilprobleme überlappen jedoch – das bedeutet, dass wir dieselben Berechnungen mehrfach durchführen.
- Dies verhindern wir durch den Einsatz von Memoisierung oder Dynamic Programming.

# Wiederholung: Rod Cutting

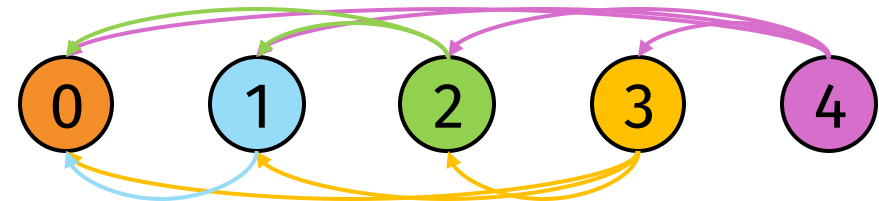
## Memoisierung

- Die Rekursion bleibt erhalten.
- Bereits berechnete Lösungen von Teilproblemen werden in einer Tabelle gespeichert (memoisiert), um doppelte Berechnungen zu vermeiden.



## Dynamic Programming

- Das Problem wird von unten nach oben gelöst – mithilfe einer geeigneten Datenstruktur.
- Voraussetzung dafür ist das Erkennen der Abhängigkeiten zwischen den Teilproblemen.



## 2. Längste gemeinsame Teilsequenz



# Was ist eine Teilsequenz?

Eine Teilsequenz  $Z$  von einer Sequenz  $X$  wird gebildet, indem wir einige (oder keine) der Elemente aus der originalen Sequenz entfernen, ohne die Reihenfolge zu verändern.

Beispiel:  $Z_i$  sind mögliche Teilsequenzen von  $X = [A, B, C, B, D, A, B]$

- $Z_1 = [A, B, C]$
- $Z_2 = [B, C, B, A]$
- $Z_3 = [B, C, D, B]$
- Keine gültige Teilsequenz:  $F = [B, A, C]$

# Was ist eine gemeinsame Teilsequenz?

Eine gemeinsame Teilsequenz erscheint in weiteren zwei oder mehr Sequenzen in der gleichen Reihenfolge, aber nicht zwingend aufeinanderfolgend.

Beispiel: Gegeben seien die Sequenzen  $X$  und  $Y$ ; die  $Z_i$  stellen mögliche gemeinsame Teilsequenzen dar.

$X = [A, \textcolor{green}{B}, \textcolor{red}{C}, B, D, \textcolor{blue}{A}, B]$

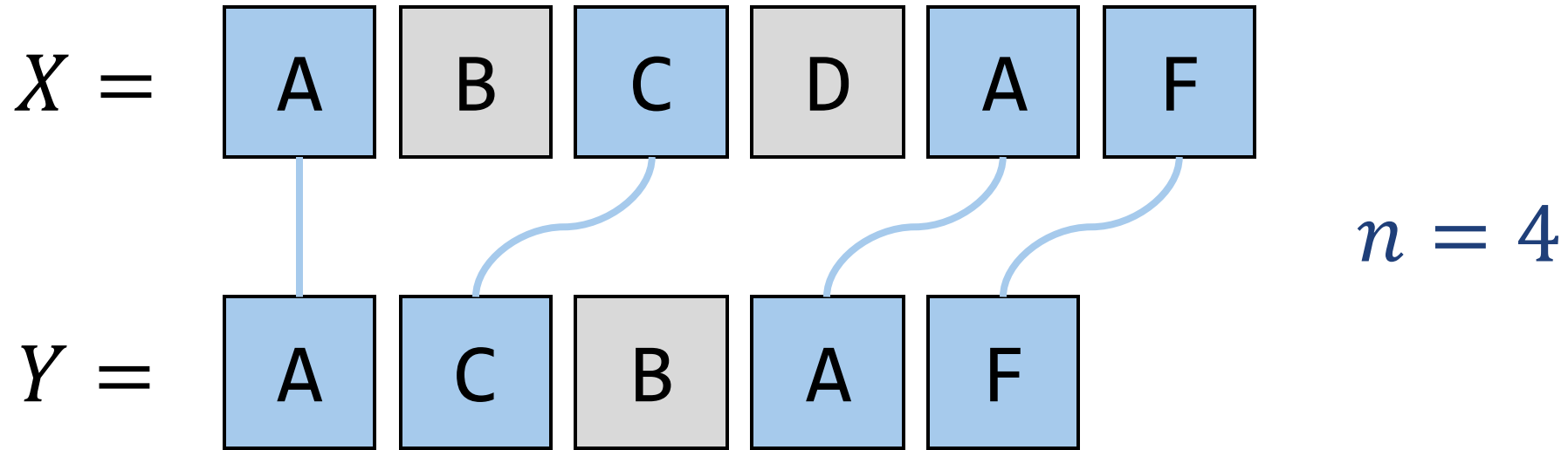
$Y = [\textcolor{green}{B}, D, \textcolor{red}{C}, \textcolor{blue}{A}, B, A]$

$Z_1 = [\textcolor{green}{B}, \textcolor{red}{C}, \textcolor{blue}{A}]$

$Z_2 = [B, C, B, A]$

$Z_3 = [B, C, D, B]$

# Problemstellung: Längste gemeinsame Teilsequenz



- **Input:** Zwei Sequenzen,  $X = [x_1, x_2, \dots, x_m]$  und  $Y = [y_1, y_2, \dots, y_n]$ , mit Längen  $m$  bzw.  $n$ . Die Längen der Sequenzen müssen dabei nicht übereinstimmen.
- **Output:** Die **Länge  $n$**  der längsten gemeinsamen Teilsequenz (LGT) von beiden Sequenzen  $X$  und  $Y$ .

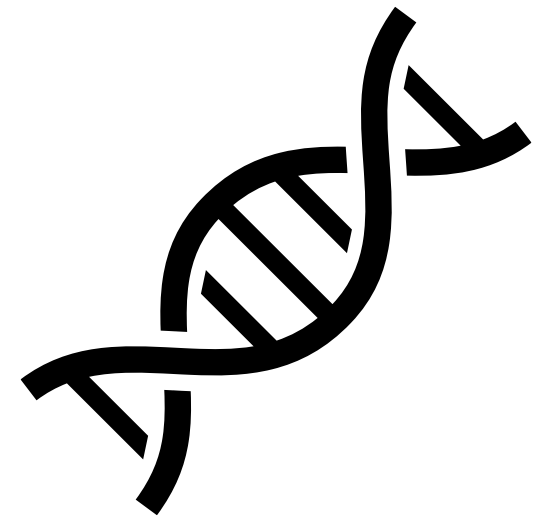
# Mögliche Echtwelt Anwendung

- **DNA-Sequenzvergleich:** Gegeben sind zwei DNA-Sequenzen  $S_1$  und  $S_2$ , das Ziel ist die längste gemeinsame Teilsequenz (LGT)  $Z$  zu finden.

$S_1 = ACCGTCTGAGTGCTCGGAAGG$

$S_2 = GTCGTTCGGAATGCC$

$\rightarrow Z = GTCGTTCGGAAG$



# Herausforderungen

Warum ist es so schwer, die LGT zu finden?

- Zu viele Möglichkeiten: Eine Sequenz mit Länge  $m$  hat  $2^m$  Teilsequenzen.

Wenn wir den Brute-Force-Ansatz verwenden:


1. Erzeuge alle Teilsequenzen von  $X$ .
2. Überprüfe, jede dieser Teilsequenzen auch eine Teilsequenz von  $Y$  ist.
3. Verfolge und speichere die längste gemeinsame Teilsequenz.

Die Zeitkomplexität ist  $\mathcal{O}(2^m \cdot 2^n)$ , d.h. exponentielles Wachstum

→ Ineffizient für lange Sequenzen.

# Vier Schritte für Dynamic Programming

Unterteile das Problem in kleinere Teilprobleme und speichere die Ergebnisse zur Wiederverwendung, um die Effizienz zu steigern.

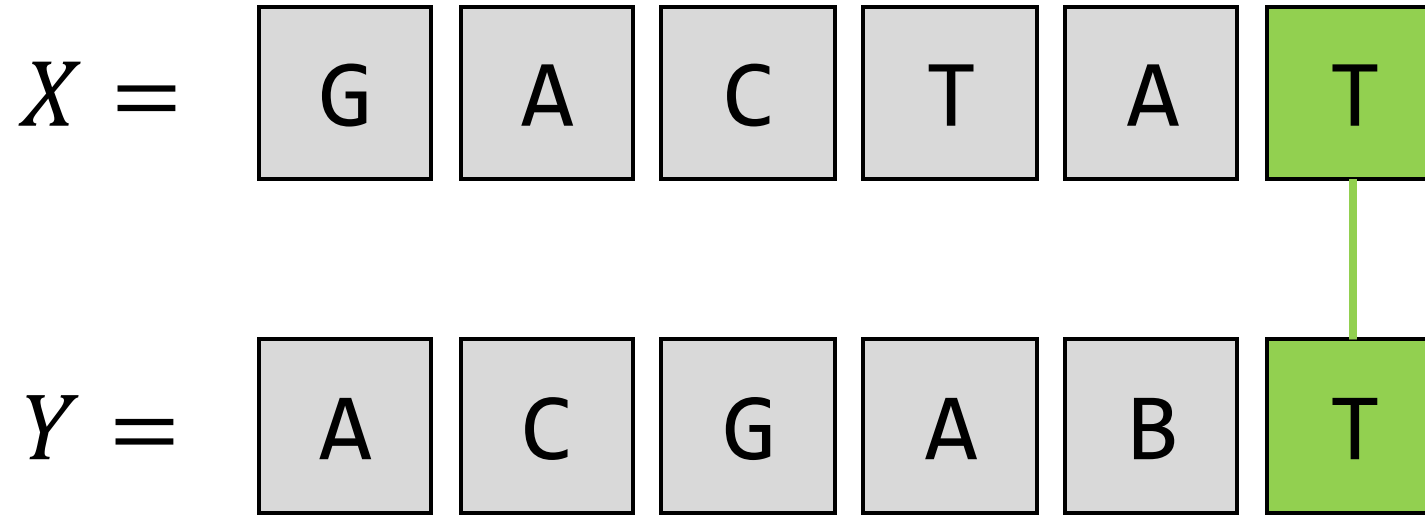
1. Charakterisiere die optimale Substruktur für eine längste gemeinsame Teilsequenz 
2. Finde eine rekursive Lösung
3. Berechne die Länge der LGT
4. Konstruiere ein LGT

# Optimale Substruktur einer LGT

Vergleiche die letzten Elemente von zwei Sequenzen:

- Wenn  $x_m = y_n$ :
  - Der letzte Buchstabe **muss** Teil der LGT sein.
  - Wir können das Problem zur LGT von  $X[:m - 1]$  und  $Y[:n - 1]$  reduzieren.
  - $LGS(X, Y) = LGS(X[:m - 1], Y[:n - 1]) + 1$
- Wenn  $x_m \neq y_n$ :
  - Der letzte Buchstabe von  $X$  oder  $Y$  gehört nicht zur LGT. Es entstehen verschiedene Teilprobleme, und wir müssen die maximale Lösung wählen.
    - Ignoriere das letzte Element von  $X$ :  $LGS(X, Y) = LGS(X[:m - 1], Y)$
    - Ignoriere das letzte Element von  $Y$ :  $LGS(X, Y) = LGS(X, Y[:n - 1])$

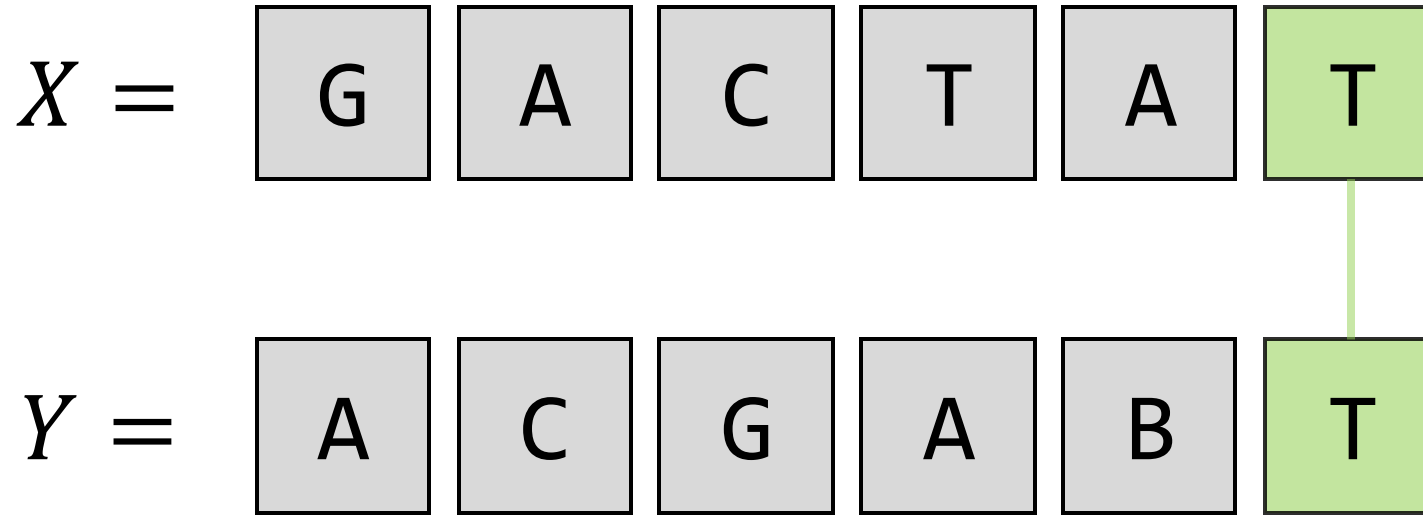
# Beispiel: Vergleiche das letzte Element



- Vergleiche die letzten beiden Elemente → sie sind gleich und gehören somit zur LGT.

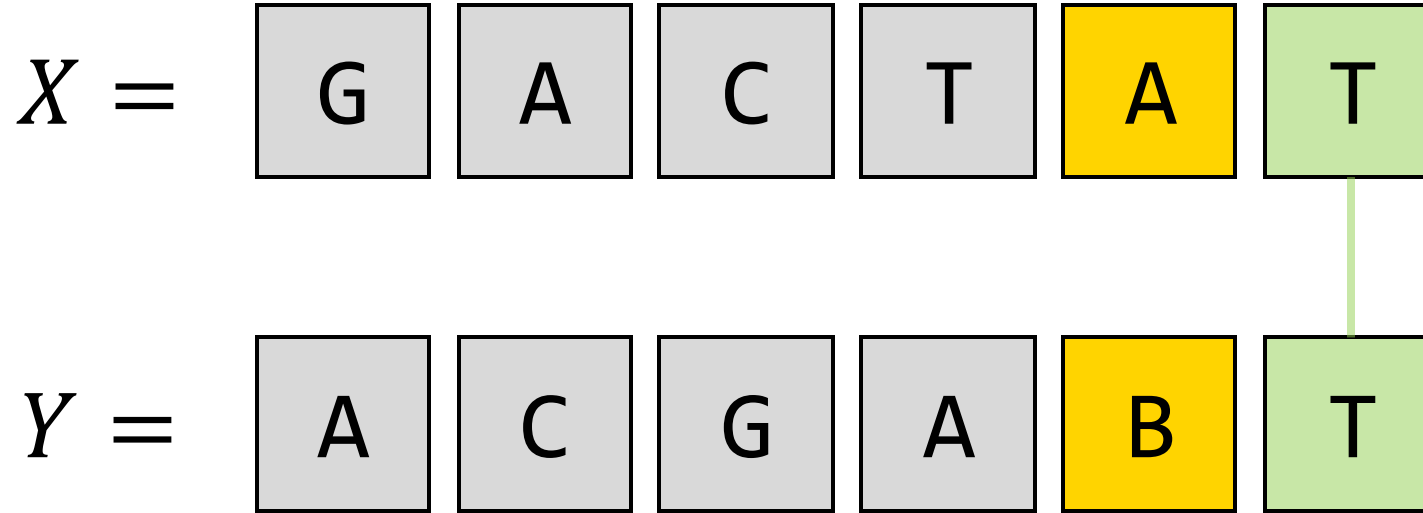


# Beispiel: Vergleiche das letzte Element



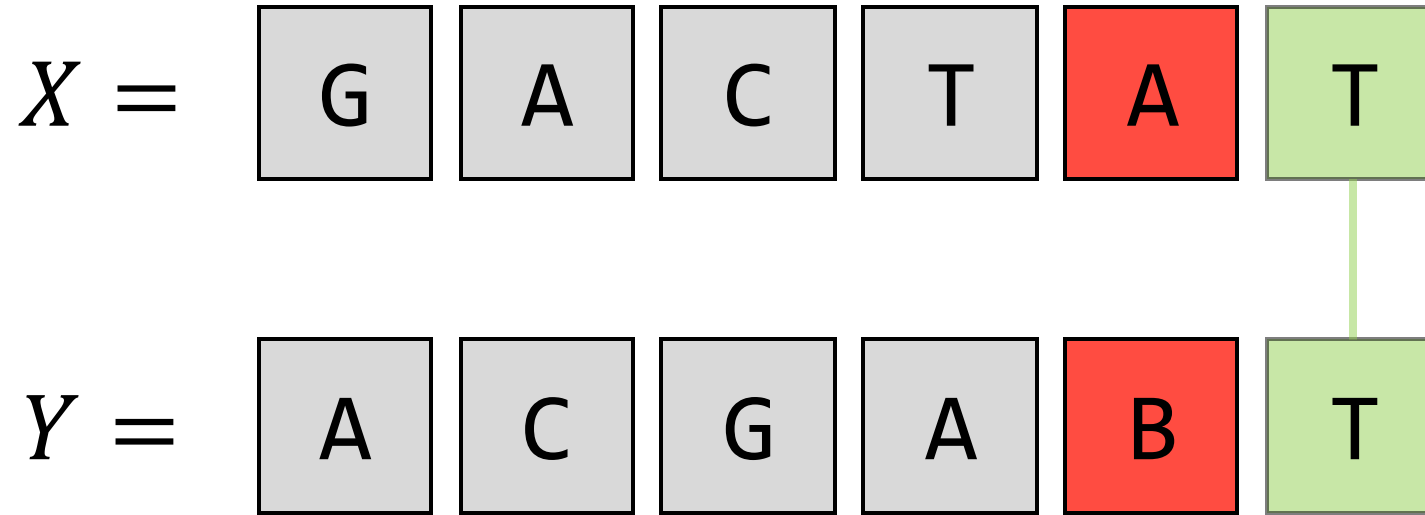
- Vergleiche die letzten beiden Elemente → sie sind gleich und gehören somit zur LGT.
- Entferne das letzte Element aus beiden Sequenzen und fahre mit der Suche fort.

# Beispiel: Vergleiche das letzte Element



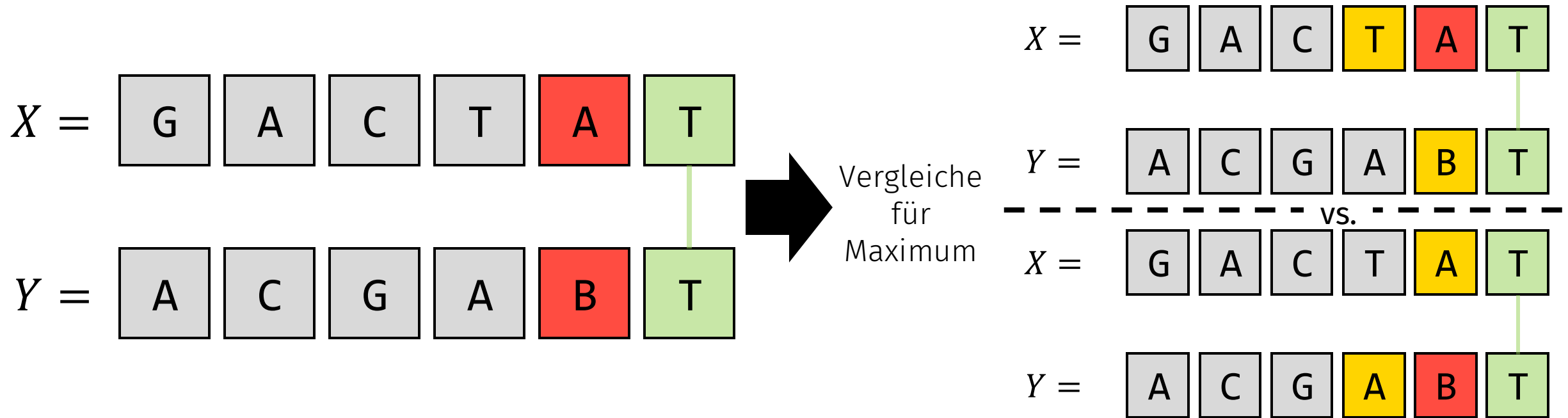
- Vergleiche die letzten beiden Elemente.
- Entferne das letzte Element aus beiden Sequenzen.
- Vergleiche die neuen letzten beiden Elemente

# Beispiel: Vergleiche das letzte Element



- Vergleiche die letzten beiden Elemente
- Entferne das letzte Element aus beiden Sequenzen
- Vergleiche die neuen letzten beiden Elemente → Sie sind ungleich und wir suchen nun die LGT für zwei Fälle, jeweils ohne  $x_m$  oder  $y_n$ .

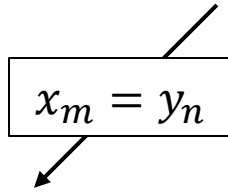
# Beispiel: Vergleiche das letzte Element



- Vergleiche die letzten beiden Elemente.
- Entferne das letzte Element aus beiden Sequenzen.
- Vergleiche die neuen letzten beiden Elemente → Sie sind ungleich und wir suchen nun die LGT für zwei Fälle, jeweils ohne  $x_m$  oder  $y_n$ .
- Nun vergleichen wir beide Fälle – In welchem ist  $n$  maximal?

# Struktur des Teilproblems

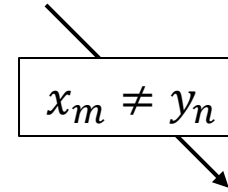
$LGS(X, Y)$



$LGS(X[:m-1], Y[:n-1])$



$$LGS(X, Y) = LGS(X[:m-1], Y[:n-1]) + 1$$



$\max(LGS(X[:m-1], Y), LGS(X, Y[:n-1]))$



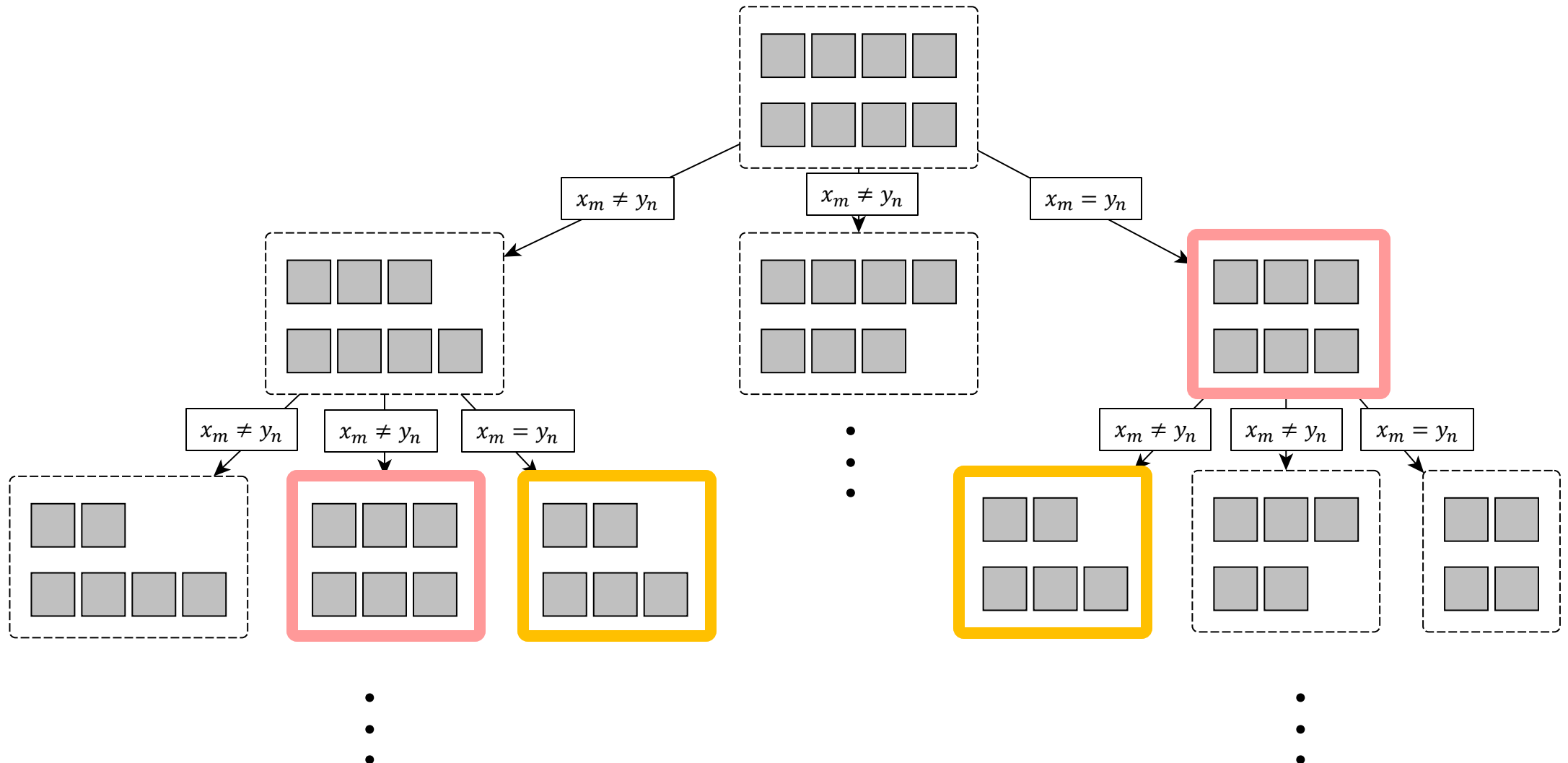
$$LGT(X, Y) = \max(LGS(X[:m-1], Y), LGS(X, Y[:n-1]))$$

# Vier Schritte für Dynamische Programmierung

Unterteile das Problem in kleinere Teilprobleme und speichere die Ergebnisse zur Wiederverwendung, um die Effizienz zu steigern.

1. Charakterisiere die optimale Substruktur für eine längste gemeinsame Teilsequenz ✓
2. Finde eine rekursive Lösung ←
3. Berechne die Länge der LGT
4. Konstruiere ein LGT

# Rekursiver Aufrufs-Baum



# Die rekursive Lösung

Wir bezeichnen mit  $Z[i, j]$  die Länge des LGT für die ersten  $i$  Elemente von  $X$  und die ersten  $j$  Elemente von  $Y$

$$Z[i, j] = \begin{cases} 0 & \text{wenn } i = 0 \text{ oder } j = 0 \\ Z[i - 1, j - 1] + 1 & \text{wenn } x_i = y_j \\ \max(Z[i, j - 1], Z[i - 1, j]) & \text{wenn } x_i \neq y_j \end{cases}$$

- Der Basisfall tritt ein, wenn eine der beiden Sequenzen leer ist ( $i = 0$  **oder**  $j = 0$ )  $\rightarrow$  da eine leere Sequenz keine Teilsequenz enthalten kann.



# Vier Schritte für Dynamische Programmierung

Unterteile das Problem in kleinere Teilprobleme und speichere die Ergebnisse zur Wiederverwendung, um die Effizienz zu steigern.

1. Charakterisiere die optimale Substruktur für eine längste gemeinsame Teilsequenz ✓
2. Finde eine rekursive Lösung ✓
3. Berechne die Länge der LGT ←
4. Konstruiere ein LGT

# Zeitkomplexität

- Ohne speichern der Lösungen für die Teilprobleme ist die Zeitkomplexität der rekursiven Lösung  $\mathcal{O}(2^{\min(n,m)})$ .
- Wir können mit DP redundante Rechnungen vermeiden, um die Zeitkomplexität auf  $\mathcal{O}(n \cdot m)$  zu reduzieren!

$$Z[i, j] = \begin{cases} 0 & \text{wenn } i = 0 \text{ oder } j = 0 \\ Z[i - 1, j - 1] + 1 & \text{wenn } x_i = y_j \\ \max(Z[i, j - 1], Z[i - 1, j]) & \text{wenn } x_i \neq y_j \end{cases}$$

- $Z[i, j]$  hängt von drei vorher berechneten Lösungen ab  $Z[i - 1, j - 1]$ ,  $Z[i - 1, j]$ ,  $Z[i, j - 1]$ .

# DP-Tabelle

- Zur Lösung verwenden wir eine DP-Tabelle der Grösse  $m \times n$ , um die Ergebnisse der Teilprobleme zu speichern.
  - Position  $i$  in Sequenz  $X$  sind die Spalten.
  - Position  $j$  in Sequenz  $Y$  sind die Zeilen.
- Wir müssen die Tabelle in einer sinnvollen Reihenfolge durchlaufen, sodass zu jedem Zeitpunkt alle Teilprobleme, von denen das aktuelle abhängt, bereits gelöst sind.

# DP-Tabelle

	$j = 0$ $Y[: 0]$	$j = 1$ $Y[: 1]$	$j = 2$ $Y[: 2]$	$j = 3$ $Y[: 3]$	$j = 4$ $Y[: 4]$
$i = 0$ $X[: 0]$					
$i = 1$ $X[: 1]$					
$i = 2$ $X[: 2]$					
$i = 3$ $X[: 3]$					
$i = 4$ $X[: 4]$					

- Die endgültige Lösung ist in  $Z[m, n]$  gespeichert und befindet sich in der unteren rechten Ecke der DP-Tabelle.

# DP-Tabelle

	$j = 0$ $Y[: 0]$	$j = 1$ $Y[: 1]$	$j = 2$ $Y[: 2]$	$j = 3$ $Y[: 3]$	$j = 4$ $Y[: 4]$
$i = 0$ $X[: 0]$	0	0	0	0	0
$i = 1$ $X[: 1]$	0				
$i = 2$ $X[: 2]$	0				
$i = 3$ $X[: 3]$	0				
$i = 4$ $X[: 4]$	0				


- Die Basisfälle treten ein, wenn  $i = 0$  oder  $j = 0$  gilt  $\rightarrow$  in diesen Fällen ist  $Z[i, j] = 0$ .

# DP-Tabelle

	$j = 0$ $Y[: 0]$	$j = 1$ $Y[: 1]$	$j = 2$ $Y[: 2]$	$j = 3$ $Y[: 3]$	$j = 4$ $Y[: 4]$
$i = 0$ $X[: 0]$	0	0	0	0	0
$i = 1$ $X[: 1]$	0	...			
$i = 2$ $X[: 2]$	0				
$i = 3$ $X[: 3]$	0				
$i = 4$ $X[: 4]$	0				

- Wir gehen zu  $Z[1,1]$  und benutzen die rekursive Formel von vorhin.

# DP-Tabelle

	$j = 0$ $Y[: 0]$	$j = 1$ $Y[: 1]$	$j = 2$ $Y[: 2]$	$j = 3$ $Y[: 3]$	$j = 4$ $Y[: 4]$
$i = 0$ $X[: 0]$	0	0	0	0	0
$i = 1$ $X[: 1]$	0				
$i = 2$ $X[: 2]$	0				
$i = 3$ $X[: 3]$	0				
$i = 4$ $X[: 4]$	0				

- Wir iterieren durch die Zeile, indem wir  $j$  erhöhen und  $i$  erhalten.



# DP-Tabelle

	$j = 0$ $Y[: 0]$	$j = 1$ $Y[: 1]$	$j = 2$ $Y[: 2]$	$j = 3$ $Y[: 3]$	$j = 4$ $Y[: 4]$
$i = 0$ $X[: 0]$	0	0	0	0	0
$i = 1$ $X[: 1]$	0	...	...	...	...
$i = 2$ $X[: 2]$	0	...	...	...	...
$i = 3$ $X[: 3]$	0	...			
$i = 4$ $X[: 4]$	0				

- Nach der Zeile erhöhen wir  $i$  um 1 und wiederholen das gleiche.

# Beispiel: DP-Tabelle füllen

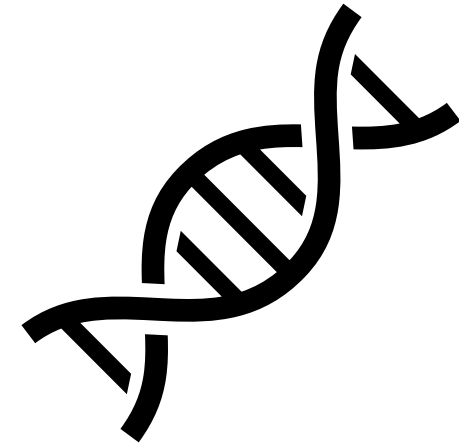
	$j = 0$ $Y[: 0]$	$j = 1$ $Y[: 1]$	$j = 2$ $Y[: 2]$	$j = 3$ $Y[: 3]$	$j = 4$ $Y[: 4]$
$i = 0$ $X[: 0]$	0	0	0	0	0
$i = 1$ $X[: 1]$	0	...	...	...	...
$i = 2$ $X[: 2]$	0	...	...	...	...
$i = 3$ $X[: 3]$	0	...			
$i = 4$ $X[: 4]$	0				

```
for i in range(1, m+1):
    for j in range(1, n+1):
        if X[i-1] == Y[j-1]:
            L = Z[i-1][j-1]
            Z[i][j] = L + 1
        else:
            L1 = Z[i][j-1]
            L2 = Z[i-1][j]
            Z[i][j] = max(L1, L2)
```

# Übung: Fülle die Tabelle für DNA-Sequenzen

	$j = 0$ $Y[: 0]$	$j = 1$ $Y[: 1]$	$j = 2$ $Y[: 2]$	$j = 3$ $Y[: 3]$	$j = 4$ $Y[: 4]$
$i = 0$ $X[: 0]$					
$i = 1$ $X[: 1]$					
$i = 2$ $X[: 2]$					
$i = 3$ $X[: 3]$					
$i = 4$ $X[: 4]$					
$i = 5$ $X[: 5]$					

$X = [T, A, G, C, A]$  und  $Y = [A, G, C, G]$



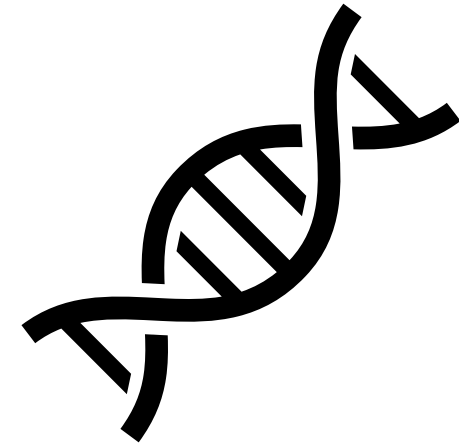
Wie lang ist die LGT?

$n = \underline{\quad}$

# Übung: Fülle die Tabelle für DNA-Sequenzen

	$j = 0$ $Y[: 0]$	$j = 1$ $Y[: 1]$	$j = 2$ $Y[: 2]$	$j = 3$ $Y[: 3]$	$j = 4$ $Y[: 4]$
$i = 0$ $X[: 0]$	0	0	0	0	0
$i = 1$ $X[: 1]$	0	0	0	0	0
$i = 2$ $X[: 2]$	0	1	1	1	1
$i = 3$ $X[: 3]$	0	1	2	2	2
$i = 4$ $X[: 4]$	0	1	2	3	3
$i = 5$ $X[: 5]$	0	1	2	3	3

$X = [T, A, G, C, A]$  und  $Y = [A, G, C, G]$



Wie lang ist die LGT?

$n = 3$

# Python Implementation, um LGT mit DP zu lösen

```
def lcs_len(X, Y):
    m, n = len(X), len(Y)
    Z = [[0]*(n+1) for _ in range(m+1)] #Erstelle DP-Tabelle + Basisfälle

    for i in range(1, m+1): #Iteriere durch alle Zeilen
        for j in range(1, n+1): #Iteriere durch alle Zellen der Zeile
            if X[i-1] == Y[j-1]: #Berechne Ergebnis mit unserer Formel
                L = Z[i-1][j-1]
                Z[i][j] = L + 1
            else:
                L1 = Z[i][j-1]
                L2 = Z[i-1][j]
                Z[i][j] = max(L1, L2)

    return Z[m][n] #Resultat für komplette Sequenzen
```

# Vier Schritte für Dynamische Programmierung

Unterteile das Problem in kleinere Teilprobleme und speichere die Ergebnisse zur Wiederverwendung, um die Effizienz zu steigern.

1. Charakterisiere die optimale Substruktur für eine längste gemeinsame Teilsequenz ✓
2. Finde eine rekursive Lösung ✓
3. Berechne die Länge der LGT ✓
4. Konstruiere ein LGT ←

# Problembeschreibung

	$j = 0$ $Y[:0]$	$j = 1$ $Y[:1]$	$j = 2$ $Y[:2]$	$j = 3$ $Y[:3]$	$j = 4$ $Y[:4]$
$i = 0$ $X[:0]$	0	0	0	0	0
$i = 1$ $X[:1]$	0	0	0	0	0
$i = 2$ $X[:2]$	0	1	1	1	1
$i = 3$ $X[:3]$	0	1	2	2	2
$i = 4$ $X[:4]$	0	1	2	3	3
$i = 5$ $X[:5]$	0	1	2	3	3

- **Input:** Eine Tabelle  $Z$  mit den optimalen Lösungen für jede Kombination von Teilsequenzen von  $X$  und  $Y$ .
- **Output:** Das LGT von  $X$  und  $Y$ .

# Wann sollte ein Element Teil der LGT sein?

- Wenn  $x_m = y_n$  gilt, ist das letzte Element Teil unserer Teilsequenz.
- Wenn  $x_m \neq y_n$  gilt, ist keine Aussage möglich und es müssen zwei neue Fälle betrachtet werden:
  - $x_m$  bleibt und bei  $Y$  wird das letzte Element entfernt.
  - $y_n$  bleibt und bei  $X$  wird das letzte Element entfernt.



# Wann sollte ein Element Teil der LGT sein?

- Mit unserer Tabelle können wir die Teilsequenz rekonstruieren.
- Wir beginnen bei  $Z[m, n]$ 
  - Wenn  $x_m = y_n$  bewegen wir uns diagonal
  - Wenn  $x_m \neq y_n$  bewegen wir uns entweder hoch oder nach links zum Maximum von  $Z[i, j - 1]$  oder  $Z[i - 1, j]$ .

# LGT-Konstruktion

		<b>B</b>	<i>D</i>	<b>C</b>	<i>A</i>
	0	0	0	0	0
<i>A</i>	0	0	0	0	1
<b>B</b>	0	1	1	1	1
<b>C</b>	0	1	1	2	2
<i>B</i>	0	1	1	2	2
<i>D</i>	0	1	2	2	2

■ Wir beginnen bei  $Z[m, n]$

- Wenn  $x_m = y_n$  bewegen wir uns diagonal.
- Wenn  $x_m \neq y_n$  bewegen wir uns hoch oder nach links links zum Maximum von  $Z[i, j - 1]$  oder  $Z[i - 1, j]$ .

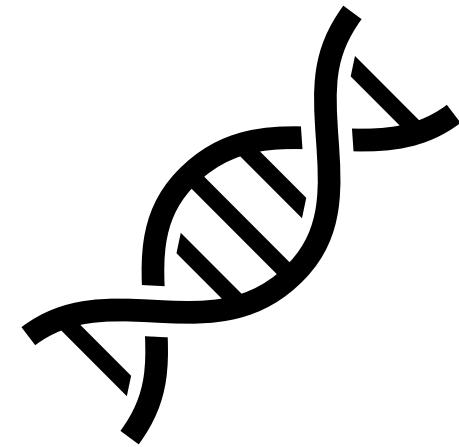
# Python-Code, um LGT zu rekonstruieren

```
def reconstruct_LGT(X, Y, Z):  
    m, n = len(X), len(Y)  
    i, j = m, n  
    lgt = []  
  
    while i > 0 and j > 0: #Iteriere durch die Tabelle bis zum Rand  
        if X[i-1] == Y[j-1]: #Gehe diagonal und merke dir das Element  
            lgt.append(X[i-1])  
            i -= 1  
            j -= 1  
        elif Z[i-1][j] >= Z[i][j-1]: #Gehe vertikal  
            i -= 1  
        else: #Gehe horizontal  
            j -= 1  
    return lgt[::-1] #Kehre das LGT um für die richtige Reihenfolge
```

# Übung: Mit unserer Tabelle, finde die LGT.

		<i>A</i>	<i>G</i>	<i>C</i>	<i>G</i>
	0	0	0	0	0
<i>T</i>	0	0	0	0	0
<i>A</i>	0	1	1	1	1
<i>G</i>	0	1	2	2	2
<i>C</i>	0	1	2	3	3
<i>A</i>	0	1	2	3	3

$X = [T, A, G, C, A]$  und  $Y = [A, G, C, G]$



Wie lang ist die LGT?

$n = 3$

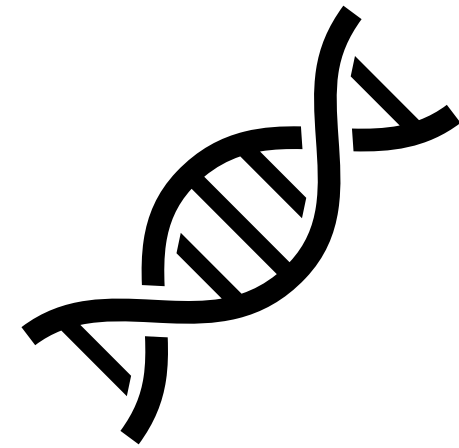
Wie lautet eine LGT?

LGT = \_\_\_\_\_

# Übung: Mit unserer Tabelle, finde die LGT.

		A	G	C	G
	0	0	0	0	0
T	0	0	0	0	0
A	0	1	1	1	1
G	0	1	2	2	2
C	0	1	2	3	3
A	0	1	2	3	3

$X = [T, A, G, C, A]$  und  $Y = [A, G, C, G]$



Wie lang ist die LGT?

$n = 3$

Wie lautet eine LGT?

$LGT = [A, G, C]$

### 3. DNA-Sequenzvergleich

---

# Problemstellung: DNA-Sequenzvergleich

$Q =$ 

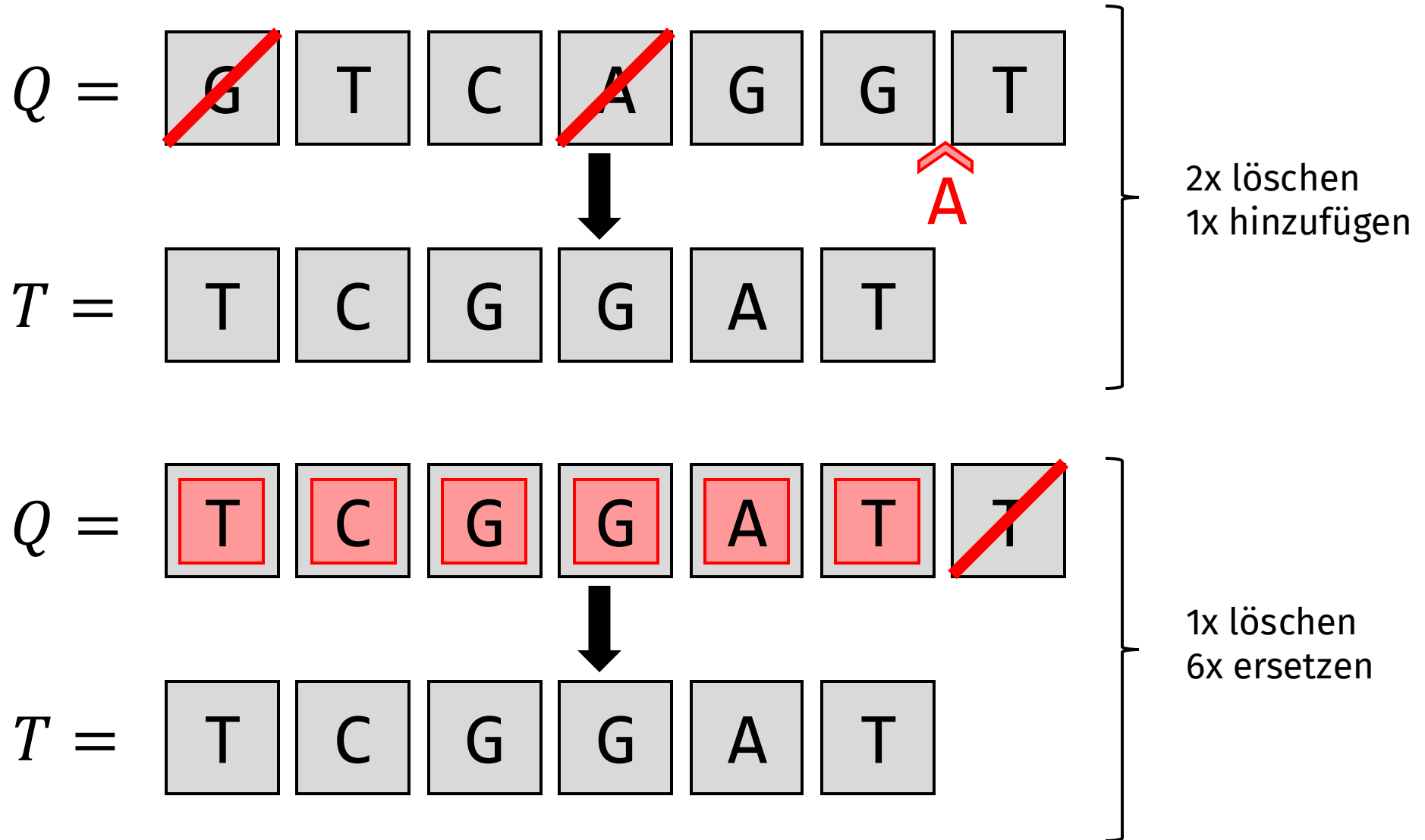
G	T	C	A	G	G	T
---	---	---	---	---	---	---

$T =$ 

T	C	G	G	A	T
---	---	---	---	---	---

- **Input:** Zwei Sequenzen Query  $Q$  und Target  $T$ .
- **Aufgabe:** Verändere  $Q$  mit so wenigen verfügbaren Operationen wie möglich, um gleich wie  $T$  zu sein.
- **Output:** Minimale Anzahl nötiger Operationen.

# Beispiel






# Problembeschreibung: DNA-Sequenzvergleich

- Zur Angleichung von  $Q$  an  $T$  stehen drei Operationen zur Verfügung:
  - Löschen
  - Einfügen
  - Ersetzen
- Problem: Wie gezeigt, existieren dabei viele mögliche Lösungswege, um  $Q$  an  $T$  anzugleichen.

# Vier Schritte für Dynamische Programmierung

Unterteile das Problem in kleinere Teilprobleme und speichere die Ergebnisse zur Wiederverwendung, um die Effizienz zu steigern.

1. Charakterisiere die Struktur einer optimalen Lösung 
2. Finde eine rekursive Lösung
3. Berechne den Wert der optimalen Lösung
4. Konstruiere die optimale Lösung – hier out of scope

# Optimale Substruktur

$Q =$ 

A	G	T	T	C	A	G	T
---	---	---	---	---	---	---	---

$T =$ 

A	T	T	G	C	A	A
---	---	---	---	---	---	---

 Teilproblem  
(5, 3)

- Wir erkennen die Ähnlichkeit zum LGT-Problem. Wie erwartet eine optimale Substruktur, und die Teilprobleme einer optimalen Lösung müssen wiederum optimal gelöst werden.
- Wie beim LGT-Problem beschreiben wir mit  $(i, j)$  das Teilproblem, bei dem die ersten  $i$  Buchstaben von  $Q$  und die ersten  $j$  Buchstaben von  $T$  abgeglichen werden.

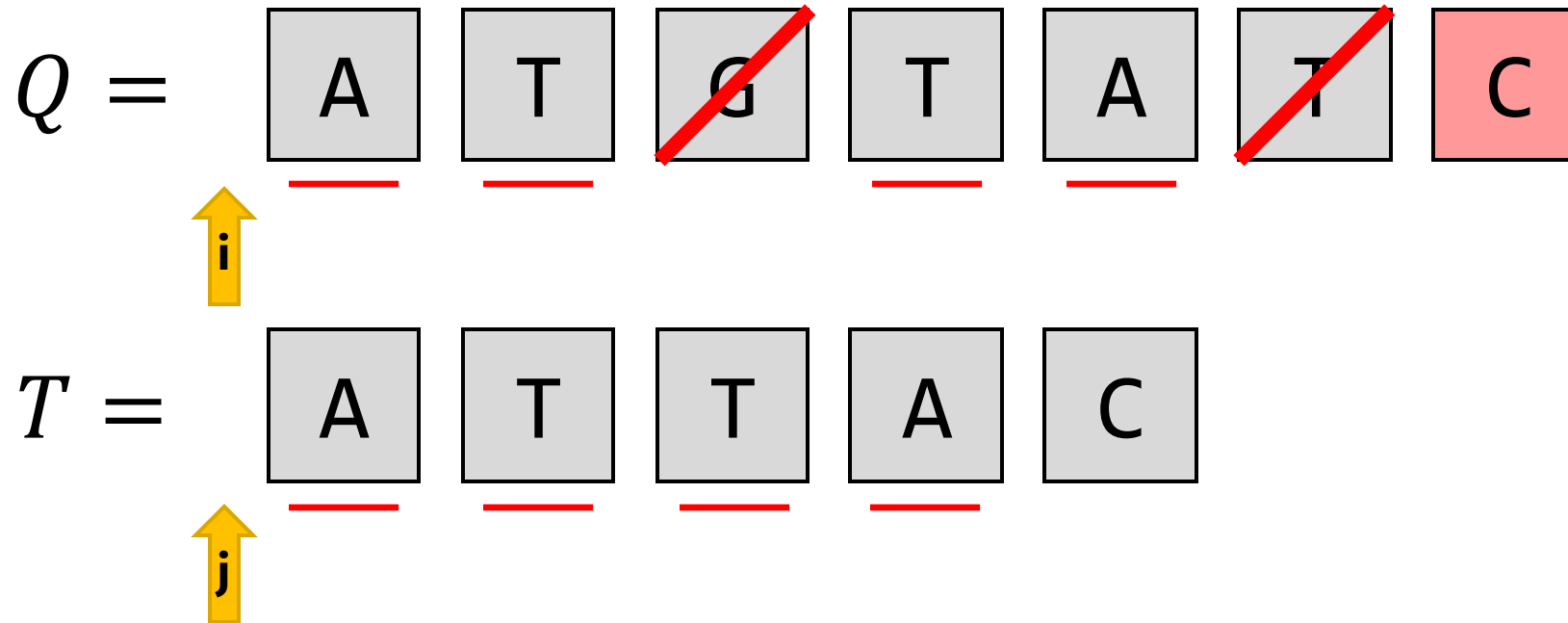
# Vier Schritte für Dynamische Programmierung

Unterteile das Problem in kleinere Teilprobleme und speichere die Ergebnisse zur Wiederverwendung, um die Effizienz zu steigern.

1. Charakterisiere die Struktur einer optimalen Lösung ✓
2. Finde eine rekursive Lösung ←
3. Berechne den Wert der optimalen Lösung
4. Konstruiere die optimale Lösung – hier out of scope

# Rekursive Struktur

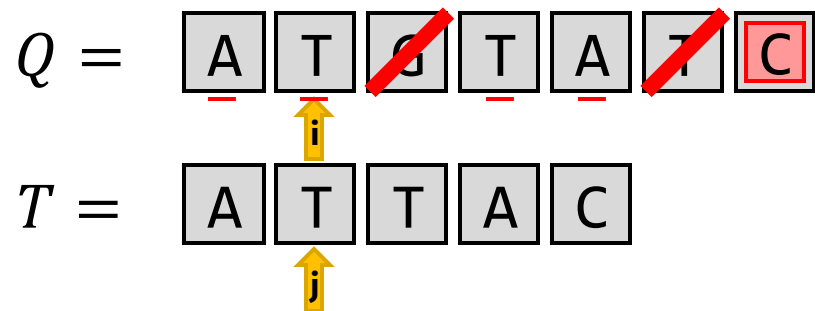
Schauen wir eine optimale Lösung an.



Output: [replace, delete, -, -, delete, -, -]  $\rightarrow$  3 Operationen

# Rekursive Lösung

- Indem wir unsere Operationen betrachten, können wir die Rekursion identifizieren:
  1. Wenn zwei Buchstaben an der Position  $(i, j)$  übereinstimmen, behalten wir sie und bewegen uns zu  $(i - 1, j - 1)$ . Die Anzahl der Operationen steigt nicht.



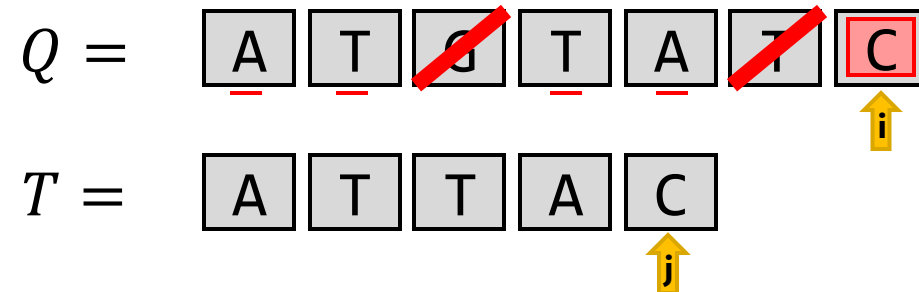
# Rekursive Lösung

- Mit  $c[i, j]$  (c für cost) bezeichnen wir die Anzahl Operationen, die nötig sind, um den Substring  $Q[:i + 1]$  an  $T[:j + 1]$  anzugleichen.

$$c[i, j] = \begin{cases} c[i - 1, j - 1] & \text{wenn } q_i = t_j \\ \end{cases}$$

# Rekursive Lösung

- Indem wir unsere Operationen betrachten, können wir die Rekursion identifizieren:
  1. Wenn zwei Buchstaben an der Position  $(i, j)$  übereinstimmen, behalten wir sie und bewegen uns zu  $(i - 1, j - 1)$ . Die Anzahl der Operationen steigt nicht.
  2. Wenn zwei Buchstaben nicht übereinstimmen, können wir sie ersetzen und uns zu  $(i - 1, j - 1)$  bewegen.





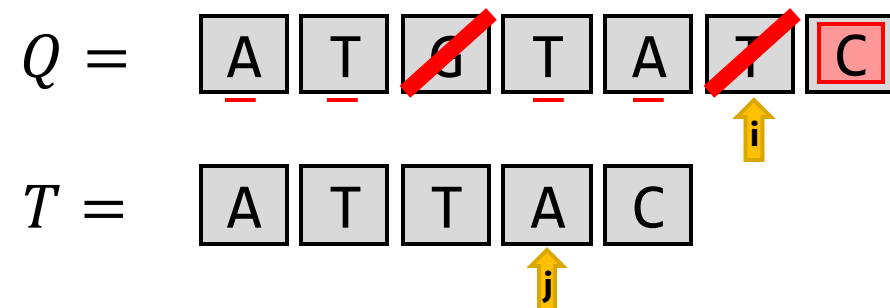
# Rekursive Lösung

- Mit  $c[i, j]$  (c für cost) bezeichnen wir die Anzahl Operationen, die nötig sind, um den Substring  $Q[:i + 1]$  an  $T[:j + 1]$  anzugleichen.

$$c[i, j] = \begin{cases} c[i - 1, j - 1] & \text{wenn } q_i = t_j \\ c[i - 1, j - 1] + 1 & \text{wenn } q_i \neq t_j \end{cases} \rightarrow \text{ersetzen}$$

# Rekursive Lösung

- Indem wir unsere Operationen betrachten, können wir die Rekursion identifizieren:
  1. Wenn zwei Buchstaben an der Position  $(i, j)$  übereinstimmen, behalten wir sie und bewegen uns zu  $(i - 1, j - 1)$ . Die Anzahl der Operationen steigt nicht.
  2. Wenn zwei Buchstaben nicht übereinstimmen, können wir sie ersetzen und uns zu  $(i - 1, j - 1)$  bewegen.
  3. Wir können auch löschen und uns zu  $(i - 1, j)$  bewegen.



# Rekursive Lösung

- Mit  $c[i, j]$  (c für cost) bezeichnen wir die Anzahl Operationen, die nötig sind, um den Substring  $Q[:i + 1]$  an  $T[:j + 1]$  anzugleichen.

$$c[i, j] = \begin{cases} c[i - 1, j - 1] & \text{wenn } q_i = t_j \\ c[i - 1, j - 1] + 1 & \text{wenn } q_i \neq t_j \quad \rightarrow \text{ersetzen} \\ c[i - 1, j] + 1 & \quad \rightarrow \text{löschen} \end{cases}$$

# Rekursive Lösung

- Indem wir unsere Operationen betrachten, können wir die Rekursion identifizieren:
  1. Wenn zwei Buchstaben an der Position  $(i, j)$  übereinstimmen, behalten wir sie und bewegen uns zu  $(i - 1, j - 1)$ . Die Anzahl der Operationen steigt nicht.
  2. Wenn zwei Buchstaben nicht übereinstimmen, können wir sie ersetzen und uns zu  $(i - 1, j - 1)$  bewegen.
  3. Wir können auch löschen und uns zu  $(i - 1, j)$  bewegen.
  4. Einfügen ist das Gegenteil von Löschen: Wir bewegen uns zu  $(i, j - 1)$ , wobei das Einfügen hinter unserem aktuellen Index stattfindet.

# Rekursive Lösung

- Mit  $c[i, j]$  (c für cost) bezeichnen wir die Anzahl Operationen, die nötig sind, um den Substring  $Q[:i + 1]$  an  $T[:j + 1]$  anzugleichen.

$$c[i, j] = \begin{cases} c[i - 1, j - 1] & \text{wenn } q_i = t_j \\ c[i - 1, j - 1] + 1 & \text{wenn } q_i \neq t_j \quad \rightarrow \text{ersetzen} \\ c[i - 1, j] + 1 & \rightarrow \text{löschen} \\ c[i, j - 1] + 1 & \rightarrow \text{einfügen} \end{cases}$$

# Rekursive Lösung

- Uns ist egal, ob wir ersetzen, löschen, oder hinzufügen. Uns interessiert nur die minimale Anzahl an Operationen.

$$c[i, j] = \begin{cases} c[i - 1, j - 1] & \text{wenn } q_i = t_j \\ c[i - 1, j - 1] + 1 & \text{wenn } q_i \neq t_j \\ c[i - 1, j] + 1 \\ c[i, j - 1] + 1 \end{cases}$$



$$c[i, j] = \begin{cases} c[i - 1, j - 1] & \text{wenn } q_i = t_j \\ \textcolor{red}{min}(c[i - 1, j - 1], c[i - 1, j], c[i, j - 1]) + 1 & \text{wenn } q_i \neq t_j \end{cases}$$

# Vier Schritte für Dynamische Programmierung

Unterteile das Problem in kleinere Teilprobleme und speichere die Ergebnisse zur Wiederverwendung, um die Effizienz zu steigern.

1. Charakterisiere die Struktur einer optimalen Lösung ✓
2. Finde eine rekursive Lösung ✓
3. Berechne den Wert der optimalen Lösung ←
4. Konstruiere die optimale Lösung – hier out of scope

# Datenstruktur

- Die Problemstruktur ist gleich wie bei LGT, also können wir auch die gleiche DP-Struktur benutzen.

	$j = 0$	$j = 1$ $A$	$j = 2$ $T$	$j = 3$ $G$	$j = 4$ $T$	$j = 5$ $A$
$i = 0$						
$i = 1$ $A$						
$i = 2$ $T$						
$i = 3$ $T$						
$i = 4$ $A$						



# Datenstruktur

- Eine leere Sequenz ( $i = 0$ ) in eine Sequenz der Länge  $j$  zu verwandeln, braucht  $j$  Einfüge-Operationen.

	$j = 0$	$j = 1$ $A$	$j = 2$ $T$	$j = 3$ $G$	$j = 4$ $T$	$j = 5$ $A$
$i = 0$	0	1	2	3	4	5
$i = 1$ $A$						
$i = 2$ $T$						
$i = 3$ $T$						
$i = 4$ $A$						

# Datenstruktur

- Eine Sequenz der Länge  $i$  in eine leere Sequenz ( $j = 0$ ) zu verwandeln, braucht  $i$  Löschoperationen.

	$j = 0$	$j = 1$ $A$	$j = 2$ $T$	$j = 3$ $G$	$j = 4$ $T$	$j = 5$ $A$
$i = 0$	0	1	2	3	4	5
$i = 1$ $A$	1					
$i = 2$ $T$	2					
$i = 3$ $T$	3					
$i = 4$ $A$	4					

# Rekursive Lösung

- Damit sind die Basisfälle abgedeckt.

- Ab hier können wir die gefundene Formel nutzen:

$$c[i, j] = \begin{cases} c[i - 1, j - 1] & \text{wenn } q_i = t_j \\ \min(c[i - 1, j - 1], c[i - 1, j], c[i, j - 1]) + 1 & \text{wenn } q_i \neq t_j \end{cases}$$

# Datenstruktur

- Wenn die Buchstaben übereinstimmen, gilt:

$$c[i, j] = c[i - 1, j - 1]$$

	$j = 0$	$j = 1$ <i>A</i>	$j = 2$ <i>T</i>	$j = 3$ <i>G</i>	$j = 4$ <i>T</i>	$j = 5$ <i>A</i>
$i = 0$	0	1	2	3	4	5
$i = 1$ <i>A</i>	1	0				
$i = 2$ <i>T</i>	2					
$i = 3$ <i>T</i>	3					
$i = 4$ <i>A</i>	4					

# Datenstruktur

- Wenn die Buchstaben nicht übereinstimmen, gilt:

$$c[i, j] = \min \begin{pmatrix} c[i-1, j-1], \\ c[i-1, j], \\ c[i, j-1] \end{pmatrix} + 1$$

	$j = 0$	$j = 1$ <i>A</i>	$j = 2$ <i>T</i>	$j = 3$ <i>G</i>	$j = 4$ <i>T</i>	$j = 5$ <i>A</i>
$i = 0$	0	1	2	3	4	5
$i = 1$ <i>A</i>	1	0	1			
$i = 2$ <i>T</i>	2					
$i = 3$ <i>T</i>	3					
$i = 4$ <i>A</i>	4					

# Datenstruktur

- Die DP-Tabelle wird analog zum LGT-Problem traversiert, jedoch mit der anderen, neuen rekursiven Formel.

	$j = 0$	$j = 1$ $A$	$j = 2$ $T$	$j = 3$ $G$	$j = 4$ $T$	$j = 5$ $A$
$i = 0$	0	1	2	3	4	5
$i = 1$ $A$	1	0	1	...		
$i = 2$ $T$	2					
$i = 3$ $T$	3					
$i = 4$ $A$	4					

# Datenstruktur

- Die Lösung finden wir, wie gewohnt, in  $c[m, n]$ .

	$j = 0$	$j = 1$ $A$	$j = 2$ $T$	$j = 3$ $G$	$j = 4$ $T$	$j = 5$ $A$
$i = 0$	0	1	2	3	4	5
$i = 1$ $A$	1	0	1	2	3	4
$i = 2$ $T$	2	1	0	1	2	3
$i = 3$ $T$	3	2	1	1	1	2
$i = 4$ $A$	4	3	2	2	2	1

# Implementation in Python

```
import numpy as np

def match(query, target):
    #Erstelle DP-Tabelle
    m, n = len(target), len(query)
    table = np.zeros((m+1,n+1))
    #Basisfälle der Berechnung
    table[0,:] = np.arange(n+1)
    table[:,0] = np.arange(m+1)
    #Fülle Tabelle, starte bei (i, j) = (1, 1)
    for i in range(1, m+1):
        for j in range(1, n+1):
            #Formel anwenden
            if query[j-1] == target[i-1]:
                table[i,j] = table[i-1,j-1]
            else:
                table[i,j] = max(table[i-1,j-1], table[i-1,j], table[i,j-1]) + 1
    #Ergebnis zurückgeben
    return table[m, n]
```



## 4. Wrap-Up

---

# Wrap-Up: LGT

1. Datenstruktur initialisieren:

- 2D-Matrix

```
Z = [[0]*(n+1) for _ in range(m+1)]
```

2. Rekursive Lösung finden

$$Z[i,j] = \begin{cases} 0 & \text{wenn } i = 0 \text{ oder } j = 0 \\ Z[i-1,j-1] + 1 & \text{wenn } x_i = y_j \\ \max(Z[i,j-1], Z[i-1,j]) & \text{wenn } x_i \neq y_j \end{cases}$$

3. Datenstruktur ausfüllen:

- Iteriere durch Zeilen

```
for i in range(1, m+1):  
    for j in range(1, n+1):  
        ...
```

4. Resultat zurückgeben

```
return Z[m][n]
```

## 5. Hausaufgaben

---

# Übung 9: Dynamic Programming II

Auf <https://expert.ethz.ch/enrolled/SS25/mavt2/exercises>

Übung 9: DP II

- Mission Mars mit Lava
- Binomialkoeffizienten
- Dreieck
- Längste gemeinsame Teilsequenz

Abgabedatum: Montag 05.05.2025, 20:00 MEZ

**KEINE HARDCODIERUNG**