

# Bioinformatik

Approximative Stringvergleiche

Ulf Leser

Wissensmanagement in der  
Bioinformatik



# Problemstellung

---

- Bisherige Algorithmen
  - Gegeben ein Template  $T$  ( $m$ ) und ein Pattern  $P$  ( $n$ )
  - Suche alle Vorkommen von  $P$  in  $T$
  - Dazu: Preprocessing von  $P$  in  $O(n)$ , dann Suche in  $O(m)$
- Jetzt betrachtetes Szenario
  - Gegeben sei eine Datenbank von Sequenzen ( $T$ )
  - Benutzer weltweit schicken kontinuierlich sich ändernde Sequenzstücke ( $P$ )
- Also:  $T$  (und nicht  $P$ ) vorverarbeiten
- Für die Suche wird  $O(n)$  angestrebt
- Lösung: **Suffixbäume**

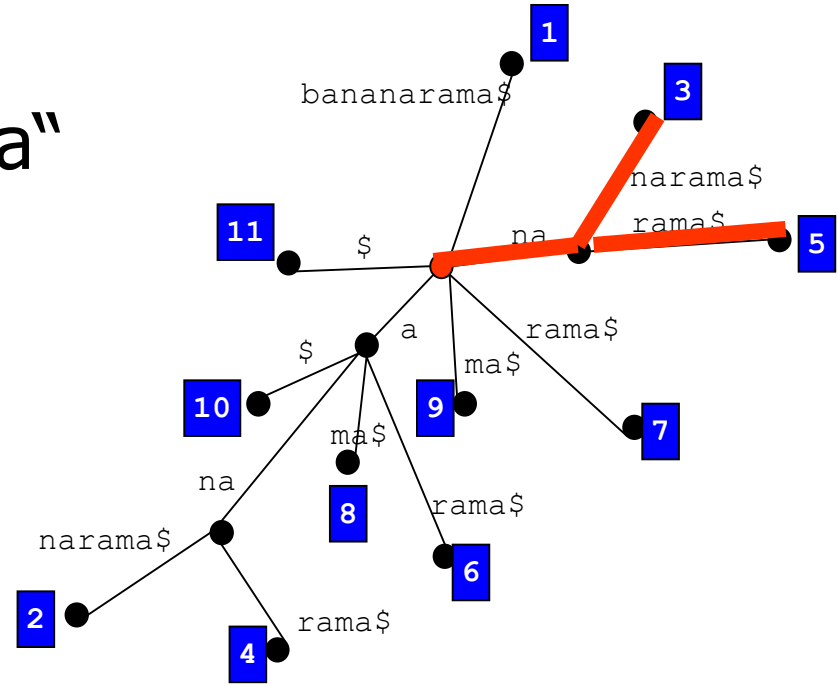
# Suffixbäume

---

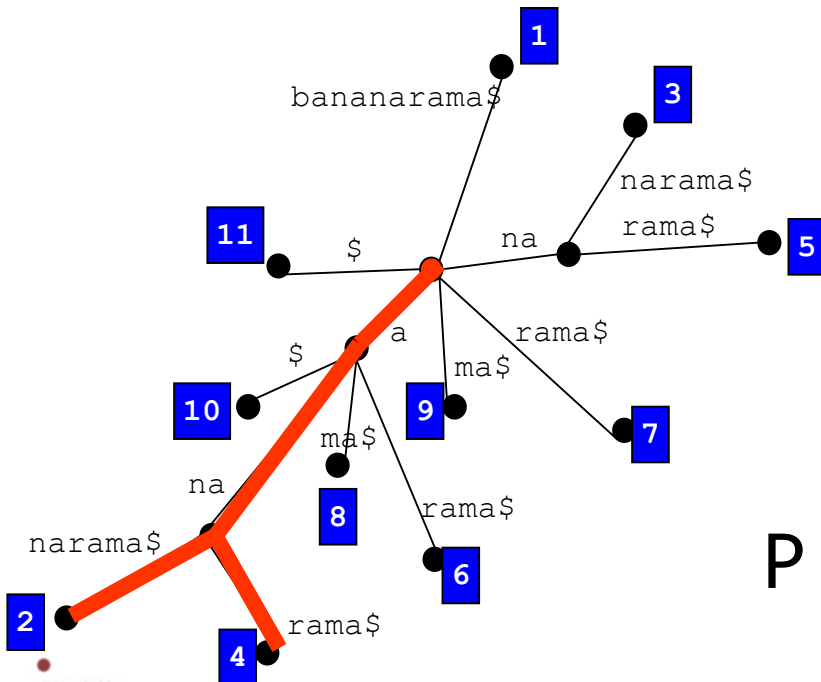
- Intuition: Kompakte Repräsentation **aller Suffixe** von  $S$  in einem Baum
- Definition: *Suffixbaum  $T$  für  $S$  ist ein Baum:*
  - *$m$  Blättern beschriftet mit  $1..,m$*
  - *Jede Kante  $E$  ist mit einem Substring  $label(E) \neq \emptyset$  von  $S$  beschriftet*
  - *Jeder **interne Knoten**  $K$  hat  $\geq 2$  Kinder. Die Label aller Kanten aus  $K$  beginnen mit unterschiedlichen Zeichen*
  - *Sei  $\{K_1, K_2, \dots, K_n\}$  ein Pfad von der Wurzel zu einem Blatt mit Beschriftung  $i$ . Dann ist die **Konkatenation der Label des Pfades gleich  $S[i..m]$*** 
    - Also das Suffix von  $S$ , das an Position  $i$  startet

# Beispiel: bananarama\$

P = „na“



P = „ana“



# Naive Konstruktion von Suffixbäumen

---

- Gegeben: String  $S$ . Gesucht: Suffixbaum  $T$  für  $S$
- Bilde Baum  $T_0$  mit Wurzelknoten und einer Kante mit Label „ $S$ “ zu einem Blatt mit Label 1
- **Induktion: konstruiere  $T_{i+1}$  aus  $T_i$  wie folgt**
  - Betrachte das Suffix  $S_{i+1} = S[i+1..]$
  - Matche  $S_{i+1}$  in  $T_i$  so weit wie möglich
  - Wenn  $S_{i+1}$  auf einer Kante  $P$  mit Label  $L$  an Position  $j$  aufgebraucht ist, füge in  $P$  an Position  $j$  einen Knoten ein mit neuem Kind „ $\$$ “
  - Wenn  $S_{i+1}$  auf einer Kante  $P$  mit Label  $L$  an Position  $j$  nicht mehr matched (der Mismatch in  $S_{i+1}$  sei  $j'$ ), füge in  $P$  an Position  $j$  einen Knoten ein mit neuem Kind „ $S[j'..]\$$ “

# Längster gemeinsamer Substring

---

- Gegeben zwei Strings  $S_1$  und  $S_2$
- Gesucht: Längster gemeinsamer Substring  $s$
- Vorschläge ?
- Lösung
  - Konstruiere Suffixbaum  $T$  für  $S_1\$S_2\%$
  - Durchlaufe  $T$  depth-first und markiere alle internen Knoten mit 1, wenn im Baum darunter ein Blatt aus  $S_1$  kommt; markiere Knoten mit 2 für  $S_2$
  - Suche breadth-first den tiefsten Knoten mit Beschriftung 1 und 2
- Komplexität:  $O(|S_1| + |S_2|)$

# Suffixarrays

- Definition

- Ein *Suffixarray*  $a$  für String  $S$  ist ein Integerarray der Länge  $|S|$ , in dem  $a[i]$  den Index des  $i$ -ten Suffix von  $S$ , sortiert nach lexikographischer Ordnung, enthält

- Beispiel

12345678901
mississippi

mississippi	i	$a[1]=11$
ississippi	ippi	$a[2]=8$
ssissippi	issippi	$a[3]=5$
sissippi	issippi	$a[4]=2$
issippi	mississippi	$a[5]=1$
ssippi	pi	$a[6]=10$
sippi	ppi	$a[7]=9$
ippi	sippi	$a[8]=7$
ppi	sissippi	$a[9]=4$
pi	ssippi	$a[10]=6$
i	ssissippi	$a[11]=3$

# Inhalt dieser Vorlesung

---

- Motivation: Approximative Stringvergleiche
- Dotplots
- Alignment
- Dynamische Programmierung

# Beispiel

---

- „AGGTAG“ in „AGTAGGTAGGATAGCTCAGA“
  - 1: AGTAGGTAGGATAGCTCAGA
  - 2: AGTAGGTAGGATAGCTCAGA
  - 3: AGTAGGTAGGATAGCTCAGA
  - 4: AGTAGGTAGGATAGTTCAGA
- Welche Matches sind besser?
  - 1: „G“ fehlt
  - 2: Perfekt
  - 3: „A“ zuviel
  - 4: „T“ durch „G“ ersetzen oder „T“ löschen und „G“ einfügen, zweites „C“ zuviel

# Voraussetzungen

---

- Relevant ist die biologische Funktion, nicht die Sequenz
- Sequenz und Funktion hängen eng zusammen, aber nicht direkt ableitbar (genetischer Code, Sequenz-Struktur)
- Bestimmung von Funktion ist extrem aufwändig (wenn überhaupt möglich), Bestimmung von Sequenzen dagegen sehr billig
  
- Also: Annäherung der Funktion über Sequenzähnlichkeiten
  - Grundparadigma moderner Biotechnologie
  - Basiert auf approximativem Stringmatching

# Genetischer Code

## Wildtyp

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

DNA

Leu Ser Asp Tyr Gly Lys

Protein

## Fatale Mutationen

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

DNA

Leu Ser Asp **Stop-Codon**

Protein

## Leseraster Mutationen

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

DNA

Leu Ser His Asp Leu Thr

Protein

## neutrale Mutationen

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

DNA

Leu Ser Asp Tyr Gly Lys

Protein

## Funktionale Mutationen

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

DNA

Leu Ser Glu Tyr Gly Lys

Protein

# Approximatives Matchen außerhalb der Bioinformatik

- Anwendungen außerhalb der Bioinformatik
  - Unschärfe Volltextsuche
    - Suche mit „Xylofon“ und finde auch „Xhylophon“
    - Fehler – oder deutsche Rechtschreibreform?
  - Personenabgleich
    - Ist „Herr Müller, 27, Stargarder Str 54“ identisch zu „Hr. Mueller, 27, Stagarder Str. 54“ ?
  - Phonetische Suche
    - Finde alle Meyer, Meier, Maier, Mair, ...
  - Vorschlagen ähnlicher Suchbegriffe



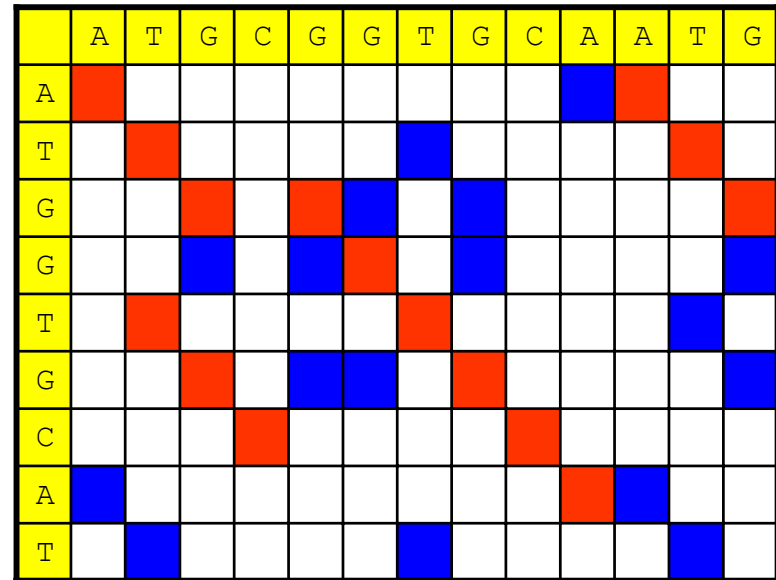
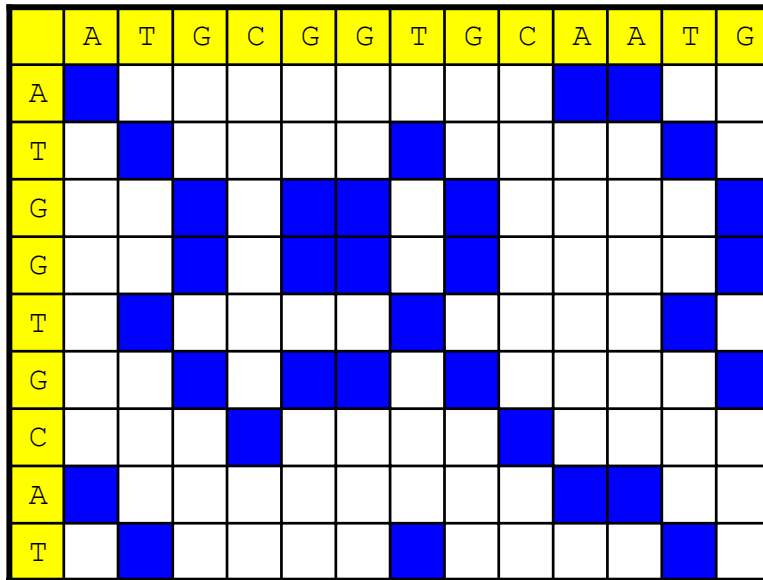
# Dotplot

- Definition:  
*Ein **Dotplot** zweier Strings  $A$ ,  $B$  ist eine Matrix  $M$ :*
  - Die Spalten entsprechen den Zeichen von  $A$
  - Die Zeilen entsprechen den Zeichen von  $B$
  - $M[a,b]=1$  gdw.  $A[a] = B[b]$
- Beispiel

	A	T	G	C	G	G	T	G	C	A	A	T	G
A	1									1	1		
T		1					1					1	
G			1		1	1		1					1
G			1		1	1		1					1
T		1					1					1	
G			1		1	1		1					1
C				1					1				
A	1									1	1		
T		1					1					1	

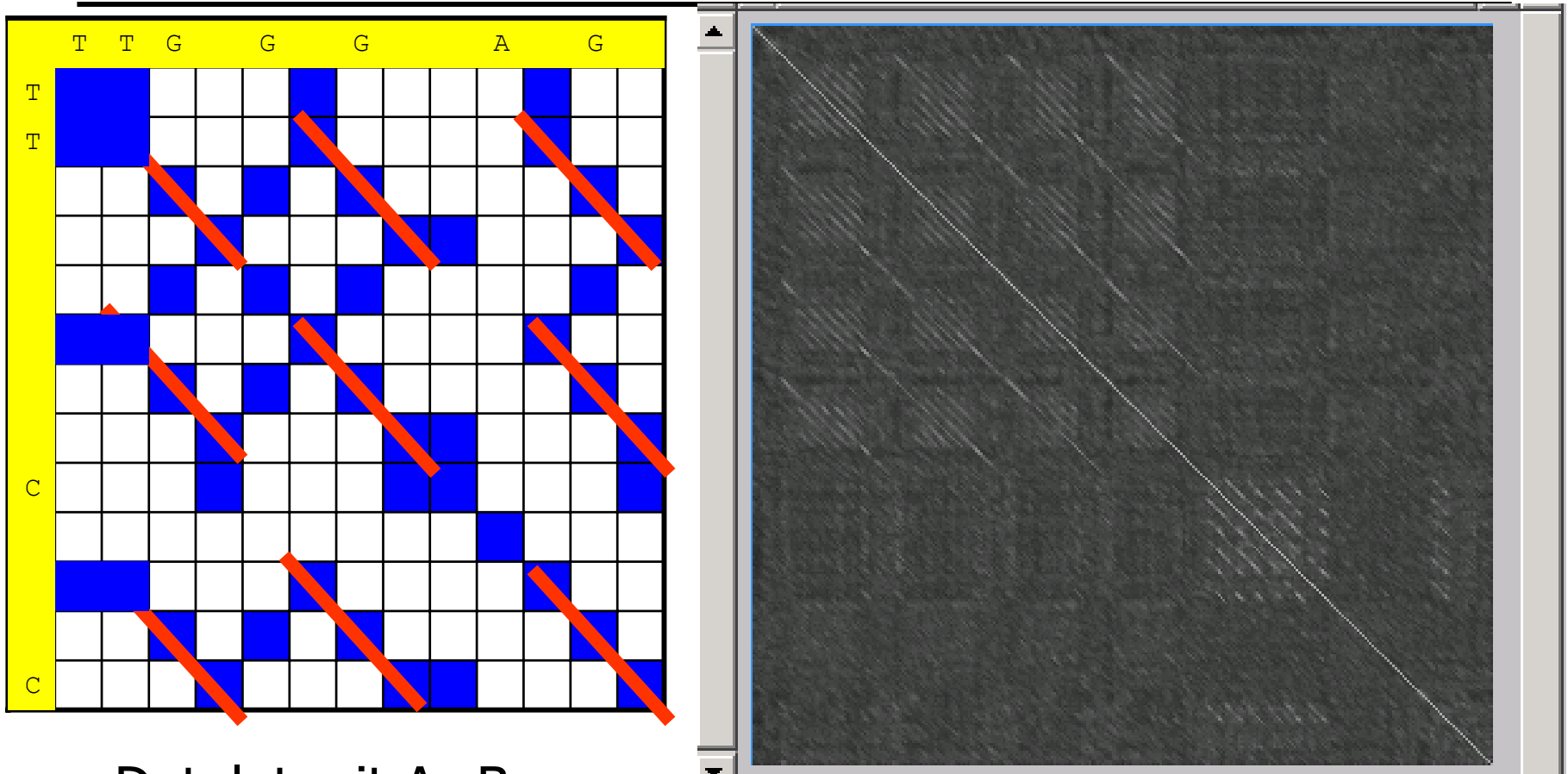
# Dotplot und gleiche Teilstrings

- Wie erkennt man gleiche Teilstrings im Dotplot?



- Diagonalen von Links-oben Rechts-Unten
  - Größter gemeinsamer Teilstring – längste Diagonale
  - Visuell bei kurzen Strings möglich

# Repetitive Sequenzen



- Dotplot mit  $A=B$

- Zitat (Genbank, P24014):

- [SIMILARITY] CONTAINS 7 EGF-LIKE DOMAINS.

- [SIMILARITY] Contains 24 leucine-rich (LRR) repeats.

# Abstandsmaße

---

- Approximatives Stringmatching sucht Ähnlichkeiten
  - Welcher Substring von  $T$  ist am ähnlichsten zu  $P$  ?
  - Welcher String  $T_1, \dots, T_n$  ist am ähnlichsten zu  $T$  ?
- Voraussetzung dafür
  - Was heißt ähnlich?
  - Was heißt „am ähnlichsten“?
- Quantifizierung des Abstandes zweier Strings
  - In der Regel eine sehr schwierige Aufgabe
  - Ähnlichkeit ist abhängig vom Gegenstand und Aufgabe
    - Wann sind sich Gesichter ähnlich - Haarfarbe zählt weniger als Augenfarbe ?
    - Wann sind sich Texte ähnlich – gleiche Wörter oder gleicher Inhalt?
  - Wir tun im folgenden so, als ob es einfach wäre
    - Und kommen auf das „Schwere“ später zurück

# Mögliche Maße

---

- Hammingabstand
  - Voraussetzung:  $|A|=|B|$
  - Vergleiche A und B Zeichen für Zeichen
  - Hammingabstand = Anzahl der Mismatches
  - Beispiel:  $ha(\text{CGTGCTCGC}, \text{ACGTGCTCGC})= 9$
  - Das kann nicht in unserem Sinne sein ...
- Biologischen Hintergrund nicht vergessen
  - Situation: Wir haben humane Gensequenz A und suchen ähnliche Sequenzen (B) in anderen Organismen
  - Annahme also: A und B haben gemeinsamen Vorfahren und sind durch **evolutionäre Prozesse** entstanden
  - Einfaches Modell: **Basenaustausch, Baseneinfügung, Basenlöschen**

# Alignment

---

- Definition

- Ein (*globales*) *Alignment* zweier Strings  $A, B$  ist eine Untereinanderanordnung von  $A$  und  $B$ , jeweils mit beliebigen zusätzlichen Spaces ( $\_$ )
  - Achtung: Zeichen dürfen beliebig matchen oder nicht
- Der *Alignmentabstand* zweier Strings  $A$  und  $B$  ist das Minimum der Anzahl von Spaces plus Anzahl von Mismatches in allen Alignments von  $A$  und  $B$

- Beispiel:  $A = \text{„ATGTA“}$ ,  $B = \text{„AGTGTC“}$

```
A_TGT_A
AGTGTC_
```

```
_AGAGAG
GAGAGA_
```

```
A__TGT_A
_AGTGTC_
```

```
AGAGAG_
_GAGAGA_
```



# Alignments und Dotplots 2

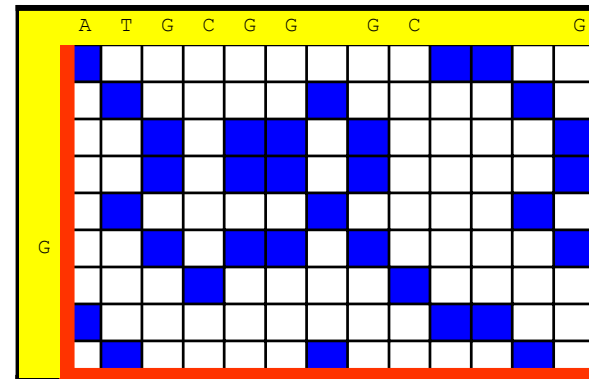
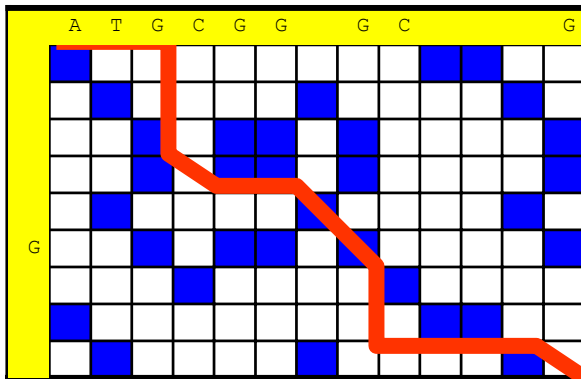
- Übersetzung von Pfaden im Dotplot in Alignments
  - Dotplot: Sei A horizontal und B vertikal aufgetragen
  - Alignment: Sei A über B angeordnet
  - Schritt nach rechts: Einfügen von \_ in B
  - Schritt nach unten: Einfügen von \_ in A
  - Schritt nach rechts-unten: Zeichen von A und B antragen

```

ATG__CGGTG__CAATG
__ATGG__TGCA__T
    
```

```

____ATGCGGTGCAATG
ATGGTGCCAT_____
    
```



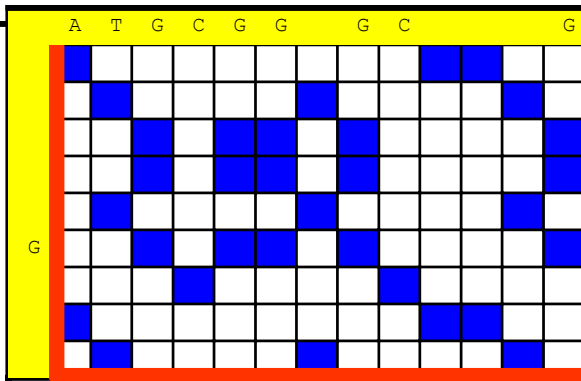
# Pfadgüte

---

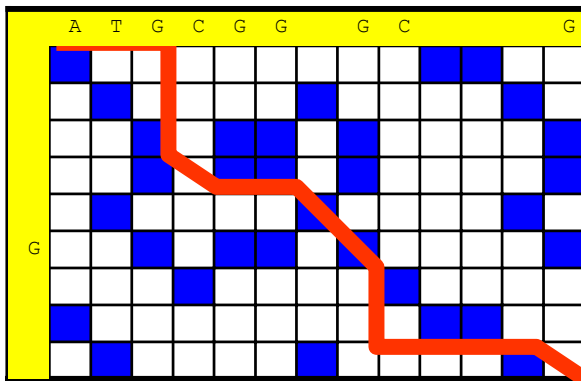
- „Gute“ Alignments haben viele Matches
  - Alignmentabstand: Anzahl von \_ plus Anzahl von Mismatches
  - Matches im Dotplot sind die 1`er Felder
- Definition

*Die **Güte eines Pfades**  $P$  durch einen Dotplot  $M$  für Strings  $A, B$  ist die Anzahl an durchquerten 1`er Feldern*
- Bemerkung
  - Schritte nach rechts oder unten zählen nicht
  - Der beste Pfad kann also höchstens Güte  $\max(m,n)$  haben

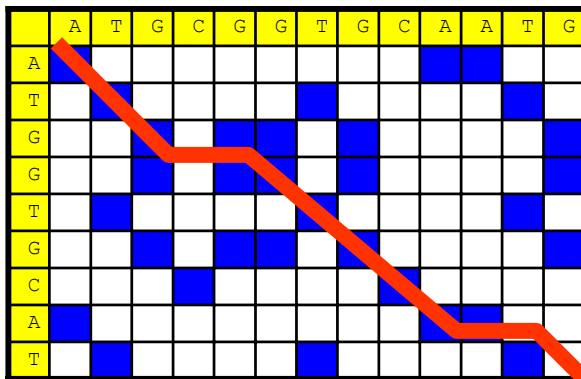
# Beispiele



Pfadgüte: 0



Pfadgüte: 2

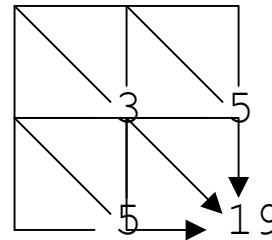
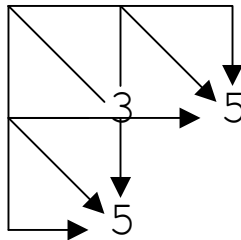
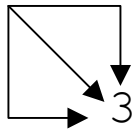


Pfadgüte: 8

# Algorithmus

---

- Naives Verfahren um den besten Pfad zu finden
  - Alle Pfade aufzählen
  - Das sind **exponentiell viele**



- Nur Pfade „um“ die Hauptdiagonale:  $3^{\min(m,n)}$
- **Inakzeptable Laufzeit**
- Tatsächliche Komplexität des Problems:  $O(m^2)$
- Trick: **Dynamische Programmierung**

# Inhalt dieser Vorlesung

---

- Editabstand durch dynamische Programmierung
- Varianten
  - Editgraphen
  - Needleman-Wunsch Algorithmus
  - Gewichtete Editabstände
  - Ähnlichkeit

# Editskripte

---

- Definition

*Ein **Editskript**  $e$  für zwei Strings  $A, B$  aus  $\Sigma^*$  ist*

*– Eine Sequenz von Editieroperationen*

- $I$  (*Einfügen* eines Zeichen  $c \in \Sigma$  in  $A$ )
- $D$  (*Löschen* eines Zeichen  $c$  in  $A$ )
  - Wird dargestellt als Einfügung in  $B$
- $R$  (*Ersetzen* eines  $c$  in  $A$  mit  $c' \in \Sigma$ )
- $M$  (*Match*, d.h.,  $A[i]=B[i]$  an dieser Stelle)

*– so, dass  $e(A)=B$*

- Beispiel:  $A=$ „ATGTA“,  $B=$ „AGTGTC“

– MIMMMR  
A TGTA  
AGTGTC

IRMMMDI  
ATGTA  
AGTGT C



# Editabstände

---

- Definition. Gegeben zwei Strings  $A$ ,  $B$  mit  $|A|=n$ ,  $|B|=m$ 
  - Die Funktion  $\text{dist}(A,B)$  berechnet den *Editabstand* von  $A$  und  $B$
  - Die Funktion  $d(i,j)$ ,  $0 \leq i \leq n$  und  $0 \leq j \leq m$ , berechnet den Editabstand zwischen  $A[1..i]$  und  $B[1..j]$
- Bemerkungen
  - Editskript besteht aus Match (M – zählt 0), Einfügungen (I), Löschungen (D) und Ersetzungen (R)
    - Achtung: Jedes R kann durch Kombination  $\{I,D\}$  ersetzt werden; im Augenblick werden also R bevorzugt
  - Offensichtlich gilt:  $d(n, m) = \text{dist}(A,B)$
  - Definition von  $D(i,j)$  dient zur rekursiven Berechnung von  $\text{dist}(A,B)$

# Rekursive Berechnung 1

---

- Wir betrachten die Berechnung von  $d(i,j)$  für  $A,B$
- **Fallunterscheidung** über das letzte Symbol im Editskript
  - **I. Entspricht einer Einfügung in A**
    - Situation: 
$$\begin{array}{c} \text{XXX} \\ \text{XXX}\underline{\quad} \\ \text{XXX}\text{T} \end{array}$$
    - Also benutzen wir ein Zeichen mehr von B
    - $d(i,j-1)$  ist der Editabstand von  $A[1..i]$  zu  $B[1..j-1]$ 
      - Symbolisiert durch die XXX
    - Damit:  $d(i,j) = d(i, j-1) + 1$

# Rekursive Berechnung 2

---

- Wir betrachten die Berechnung von  $d(i,j)$  für  $A,B$
- Fallunterscheidung über das letzte Symbol im Editskript
  - **D. Entspricht einer Einfügung in B**
    - Situation:  $\begin{array}{l} \text{XXXT} \\ \text{XXX\_} \end{array}$
    - Umgekehrte Situation
    - Wir benutzen ein Zeichen mehr von A
    - $d(i-1,j)$  ist der Editabstand von  $A[1..i-1]$  zu  $B[1..j]$
    - Damit:  $d(i,j) = d(i-1, j) + 1$

# Rekursive Berechnung 3

---

- Wir betrachten die Berechnung von  $d(i,j)$  für  $A,B$
- Fallunterscheidung über das letzte Symbol im Editskript
  - M. Entspricht einem Match
    - Situation:        XXXT  
                      XXXT
    - Wir benutzen ein Zeichen mehr von A und eines mehr von B
    - Match kostet nichts
    - Damit:  $d(i,j) = d(i-1, j-1)$

# Rekursive Berechnung 4

---

- Wir betrachten die Berechnung von  $d(i,j)$  für A,B
- Fallunterscheidung über das letzte Symbol im Editskript
  - R. Entspricht einem Mismatch
    - Situation:        XXXXG  
                      XXXXC
    - Wir benutzen ein Zeichen mehr von A und eines mehr von B
    - Mismatch kostet
    - Damit:  $d(i,j) = d(i-1, j-1) + 1$

# Rekursionsgleichung

---

- Wir leiten das nächste Symbol im Editskript aus schon bekannten Editabständen ab
- Wir suchen das **kürzeste Skript**, also

$$d(i, j) = \min \left\{ \begin{array}{l} d(i, j-1) + 1 \\ d(i-1, j) + 1 \\ d(i-1, j-1) + 1 \\ d(i-1, j-1) \end{array} \right.$$

- Ist nicht Fall 4 immer besser als Fall 3 ?
  - Ob man Fall (3) oder (4) nehmen darf hängt von  $A[i]$  und  $A[j]$  ab

# Kleinigkeiten

---

- **Randbedingungen** nicht vergessen
  - $d(i,0) = i$ 
    - Um  $A[1..i]$  zu „“ zu transformieren braucht man  $i$  D's
  - $d(0,j) = j$ 
    - Um  $A[1..0]=$ „“ zu  $B[1..j]$  zu transformieren braucht man  $j$  I's
- **Definition**

*Die Funktion  $t(i,j)$  ist definiert als*

  - $t(i,j) = 0$  gdw  $A[i] = B[j]$
  - $t(i,j) = 1$  gdw  $A[i] \neq B[j]$

# Zusammen

---

- Theorem

- Der *Editabstand zweier Strings*  $A, B$  mit  $|A|=n$ ,  $|B|=m$  berechnet sich mit Startbedingung

$$d(i,0) = i \quad d(0,j) = j$$

als  $d(n,m)$  mit folgender *Rekursionsgleichung*

$$d(i,j) = \min \left\{ \begin{array}{l} d(i,j-1) + 1 \\ d(i-1,j) + 1 \\ d(i-1,j-1) + t(i,j) \end{array} \right\}$$

# Rekursiver Algorithmus

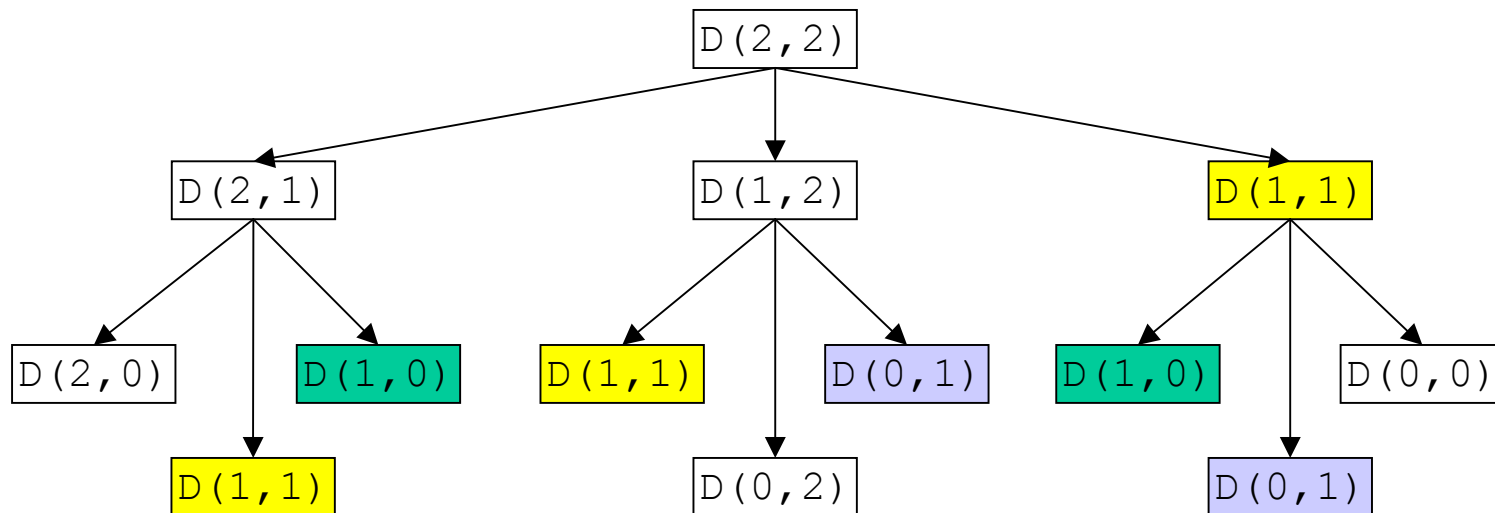
```
function d(A,B,i,j) {
    if (i = 0)           return i;
    else if (j = 0)      return j;
    else
        return min (    d(i-1,j) + 1,
                        d(i,j-1) + 1,
                        d(i-1,j-1) + t(A[i],B[j]));
}
function t(c1, c2) {
    if (c1 = c2)      return 0;
    else                 return 1;
}
```

- Komplexität?

- Für jedes  $n,m$  erfolgen 3 Aufrufe, die wiederum jeweils 3 Aufrufe auslösen, die ...
- Komplexität damit mindestens  $O(3^{\max(n,m)})$

# Sicher nicht optimal

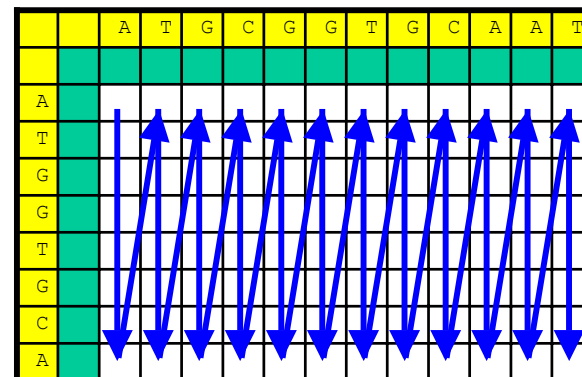
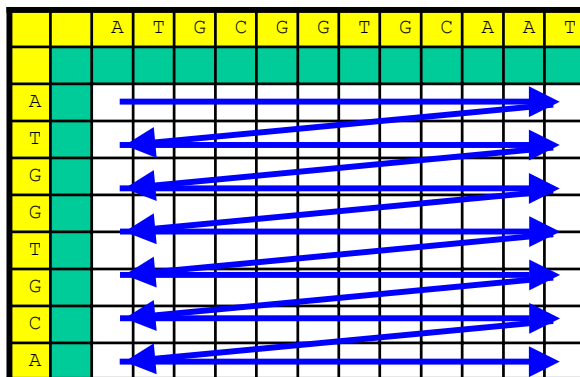
- Durch die Rekursionsgleichung werden viele Teillösungen mehrfach berechnet



- Es gibt nur  $(n+1) \cdot (m+1)$  verschiedene Aufrufe
- Wie kann man die redundanten Berechnungen sparen?

# Tabellarische Berechnung

- Grundidee
  - Speichern der Teillösungen in Tabelle
  - Bei Berechnung Wiederverwendung wo immer möglich
- Aufbau der Tabelle: Bottom-Up (statt rekursiv Top-Down)
  - **Initialisierung** mit festen Werten  $d(i,0)$  und  $d(0,j)$
  - **Sukzessive Berechnung** von  $d(i,j)$  mit steigendem  $i,j$
  - Für  $d(i,j)$  brauchen wir  $d(i,j-1)$ ,  $d(i-1,j)$  und  $d(i-1,j-1)$
  - Verschiedene Reihenfolgen möglich



# Beispiel

$$d(i, j) = \min \left\{ \begin{array}{l} d(i, j-1) + 1 \quad (I) \\ d(i-1, j) + 1 \quad (D) \\ d(i-1, j-1) + 1 \quad (R) \\ d(i-1, j-1) \quad (M) \end{array} \right.$$

		A	T	G	C	G	G	T
	0	1	2	3	4	5	6	7
A	1							
T	2							
G	3							
G	4							

		A	T	G	C	G	G	T
	0	1	2	3	4	5	6	7
A	1	0						
T	2							
G	3							
G	4							

		A	T	G	C	G	G	T
	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
T	2							
G	3							
G	4							

		A	T	G	C	G	G	T
	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
T	2	1	0	1	2	3	4	5
G	3							
G	4							

		A	T	G	C	G	G	T
	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
T	2	1	0	1	2	3	4	5
G	3	2	1	0	1	2	3	4
G	4							

		A	T	G	C	G	G	T
	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
T	2	1	0	1	2	3	4	5
G	3	2	1	0	1	2	3	4
G	4	3	2	1	2	1	2	3

# Was ist gewonnen?

		A	T	G	C	G	G	T
	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
T	2	1	0	1	2	3	4	5
G	3	2	1	0	1	2	3	4
G	4	3	2	1	2	1	2	3

- Editabstand von ATGG, ATGCGGT ist 3
- Wir suchen aber ein Alignment, nicht nur den Abstand
- Extraktion aus der Tabelle durch „Tracebacking“
  - Bei Berechnung von  $d(i,j)$  behalte Pointer auf minimale Vorgängercelle
  - Das muss nicht eindeutig sein

		A	T	G	C	G	G	T
	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
T	2	1	0	1	2	3	4	5
G	3	2	1	0	1	2	3	4
G	4	3	2	1	2	1	2	3

		A	T	G	C	G	G	T
	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
T	2	1	0	1	2	3	4	5
G	3	2	1	0	1	2	3	4
G	4	3	2	1	2	1	2	3

		A	T	G	C	G	G	T
	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
T	2	1	0	1	2	3	4	5
G	3	2	1	0	1	2	3	4
G	4	3	2	1	2	1	2	3

# Vom Pfad zum Alignment

	A	T	G	C	G	G	T	
	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
T	2	1	0	1	2	3	4	5
G	3	2	1	0	1	2	3	4
G	4	3	2	1	2	1	2	3

- Jeder Pfad von  $(n,m)$  nach  $(1,1)$  ist ein optimales Alignment
  - Starte von  $(1,1)$
  - Nach rechts: Space in A
  - Nach unten: Space in B
  - Diagonal: Match/Replace

	A	T	G	C	G	G	T	
	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
T	2	1	0	1	2	3	4	5
G	3	2	1	0	1	2	3	4
G	4	3	2	1	2	1	2	3

ATGCGGT  
ATG \_ G \_ \_

	A	T	G	C	G	G	T	
	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
T	2	1	0	1	2	3	4	5
G	3	2	1	0	1	2	3	4
G	4	3	2	1	2	1	2	3

ATGCGGT  
AT \_ \_ GG \_

# Beispiel 2

	0	1	2	3	4	5	6	7
		w	r	i	t	e	r	s
0	0	1	2	3	4	5	6	7
1	v	1	1	2	3	4	5	6
2	i	2	2	2	2	3	4	5
3	n	3	3	3	3	3	4	5
4	t	4	4	4	4	3	4	5
5	n	5	5	5	5	4	4	5
6	e	6	6	6	6	5	4	5
7	r	7	7	6	7	6	5	4

	0	1	2	3	4	5	6	7
		w	r	i	t	e	r	s
0	0	1	2	3	4	5	6	7
1	v	1	2	3	4	5	6	7
2	i	2	2	2	3	4	5	7
3	n	3	3	3	3	4	5	6
4	t	4	4	4	4	3	4	5
5	n	5	5	5	5	4	5	6
6	e	6	6	6	6	5	5	6
7	r	7	7	6	7	6	5	4

	0	1	2	3	4	5	6	7
		w	r	i	t	e	r	s
0	0	1	2	3	4	5	6	7
1	v	1	2	3	4	5	6	7
2	i	2	2	2	3	4	5	7
3	n	3	3	3	3	4	5	6
4	t	4	4	4	4	4	5	6
5	n	5	5	5	5	4	5	6
6	e	6	6	6	6	5	5	6
7	r	7	7	6	7	6	5	4

	0	1	2	3	4	5	6	7
		w	r	i	t	e	r	s
0	0	1	2	3	4	5	6	7
1	v	1	1	2	3	4	5	6
2	i	2	2	2	3	4	5	7
3	n	3	3	3	3	4	5	6
4	t	4	4	4	4	3	4	5
5	n	5	5	5	5	4	5	6
6	e	6	6	6	6	5	5	6
7	r	7	7	6	7	6	5	4

v i n t n e r -  
w r i t - e r s

v - i n t n e r -  
w r i - t - e r s

- v i n t n e r -  
w r i - t - e r s

# Komplexität

---

- Aufbau der Tabelle
  - Jede Zelle muss genau drei andere Zellen betrachten
  - Konstante Zeit pro Tabelle
  - Insgesamt:  $O(m*n)$
- Traceback
  - Man kann einen beliebigen Pfad wählen
  - Es muss einen Pfad von  $(n,m)$  nach  $(1,1)$  geben
    - Jede Zelle hat einen wegzeigenden Pointer
    - Keine Zelle zeigt aus der Tabelle hinaus
  - Worst-Case Pfadlänge ist  $O(m+n)$
- Zusammen
  - Quadratische Komplexität (für  $m*n > m+n$ )

# Prinzip des dynamischen Programmierens

---

- Ziel: Optimierung einer Zielfunktion
  - Hier: Editabstand
- Prinzip
  - Berechnung von Lösungen für „große“ Problemen aus Lösungen für „kleinere“ Probleme
  - Weiteres Beispiel: Kürzeste Wege in Graphen
- Drei Bestandteile
  - Berechnung aus **optimalen Teillösungen**
    - Hier: Rekursionsgleichung
  - Wenn möglich **Zwischenspeichern der Teillösungen**
    - Hier: Tabelle mit Werten  $d(i,j)$
  - **Rückverfolgung** der Lösung aus Tabelle
    - Hier: Traceback des Alignments

# Needleman-Wunsch Algorithmus

---

- Historisch der **erste Algorithmus** zum globalen Alignment zweier Sequenzen
  - S.B. Needleman, C.D. Wunsch, „A general method applicable to the search for similarities in the amino acid sequence of two proteins“, J. of Molecular Biology, 48, 1970
- Variante des Algorithmus, wie wir ihn kennen gelernt haben
- Komplexität ist  $O(n^3)$  (wenn  $m=n$ )
- Wird nicht mehr verwendet

# Ähnlichkeit

---

- Welche Frage will man eigentlich beantworten?
  - Wie weit entfernt sind diese beiden Sequenzen
  - Wie ähnlich sind sich zwei Sequenzen
- Ähnlichkeit ist oft einfacher
  - Intuitives Maß – je ähnlicher, desto höher ist die Wahrscheinlichkeit für ähnliche Funktion
  - Programme berechnen i.d.R. Ähnlichkeit
  - **Logisch ist Ähnlichkeit und Abstand äquivalent**
- Differenzierte Betrachtung
  - Ähnlichkeit einzelner Zeichen / Basen / Aminosäuren
  - Ähnliche Zeichen – hohe positive Werte
  - Unähnliche Zeichen – negative Werte

# Formal

---

- Definition

*Gegeben Alphabet  $\Sigma' = \Sigma \cup \_'$ , Strings  $A, B$  über  $\Sigma'$  mit  $|A| = |B| = n$*

- Eine *Scoringfunktion* ist eine Funktion  $s: \Sigma' \times \Sigma' \rightarrow \text{Integer}$ 
  - Substitutionsmatrix
- Die *Ähnlichkeit* von  $A, B$  bzgl. der Scoringfunktion  $s$  ist

$$\text{sim}(A, B) = \sum_{i=1}^n s(A[i], B[i])$$

- Bemerkung

- Wir betrachten i.d.R. Alignments, nicht beliebige  $A, B$
- Optimales Alignment  $\sim$  Alignment mit höchster Ähnlichkeit

# Beispiel

$$\Sigma' = \{A, C, G, T, \_ \}$$

	A	C	G	T	_
A	4	-2	-2	-2	0
C		4	-2	-2	-2
G			4	-1	0
T				4	-2
_					0

A	C	_	G	T	C
A	G	G	T	_	C

= **3**

A	C	G	T	C
A	G	G	T	C

= **18**

# Rekursionsgleichung

---

- Nur kleine Veränderung

$$d(i,0) = \sum_{i=1}^n s(A[i], \_) \quad d(0, j) = \sum_{i=1}^m s(\_, B[i])$$

$$d(i, j) = \max \left\{ \begin{array}{l} d(i, j-1) + s(\_, B[i]) \\ d(i-1, j) + s(A[i], \_) \\ d(i-1, j-1) + s(A[i], B[j]) \end{array} \right\}$$

# Zusammenfassung

---

- Editabstand und Alignierung ineinander überführbar
- Berechnung eines optimalen Alignments hat **quadratische Komplexität**
  - Mittels dynamischer Programmierung
  - Tabelle aufbauen, Pfad zurückverfolgen
- **Platzbedarf ist auch quadratisch**
  - Das ist kritisch
  - Es gibt Algorithmen mit linearem Platzbedarf
- Diverse Erweiterungen
  - Berücksichtigen die **Biologie der Aufgabe**: Mutationswahrscheinlichkeiten, Aminosäureähnlichkeiten, Evolutionsmodell, ...
  - Hohe praktische Relevanz: Erweiterungen machen approximatives Stringmatching erst anwendbar in der Bioinformatik