# String Matching am Beispiel von DNA Sequenzvergleichen

Andreas Krug, Sarah Sallinger, Michael Ferdinand Moser

VP Wissenschaftliches Arbeiten und Präsentieren (WS 2014/2015)

23. Januar 2015

# Motivation

- String-Matching Algorithmen gehören zur Klasse der Zeichenkettenalgorithmen. Es geht dabei um das Finden von Textsegmenten in Strings anhand eines vorgegebenen Suchmusters.

# Motivation

- String-Matching Algorithmen gehören zur Klasse der Zeichenkettenalgorithmen. Es geht dabei um das Finden von Textsegmenten in Strings anhand eines vorgegebenen Suchmusters.
- Wir suchen nach:
  - Exakten Übereinstimungen (*matches*)
  - Ungefähren Übereinstimmungen:
    – Hier muss das Konzept der Ähnlichkeit von zwei Strings definiert werden.

# Anwendungsgebiete

- Textverarbeitungsprogramme / Editoren: – Bestimmte Muster in einem Text suchen.

# Anwendungsgebiete

- Textverarbeitungsprogramme / Editoren: – Bestimmte Muster in einem Text suchen.
- Internetsuchmaschinen: – Finden von Websites.

# Anwendungsgebiete

- Textverarbeitungsprogramme / Editoren: – Bestimmte Muster in einem Text suchen.
- Internetsuchmaschinen: – Finden von Websites.
- `strg-f`, `grep`, ...

# Anwendungsgebiete

- Textverarbeitungsprogramme / Editoren: – Bestimmte Muster in einem Text suchen.
- Internetsuchmaschinen: – Finden von Websites.
- `strg-f`, `grep`, ...
- DNA–Sequenzanalysen in der Bioinformatik.

# Exaktes String–Matching (I): Definition

## Voraussetzungen

- Ein Text in einem Feld $T[1...n]$ der Länge $n$.

# Exaktes String–Matching (I): Definition

## Voraussetzungen

- Ein Text in einem Feld $T[1...n]$ der Länge $n$.
- Ein Textmuster in einem Feld $P[1...m]$ der Länge $m \leq n$, welches in $T$ mit Shift $s \in \mathbb{N}_0$ enthalten ist:
  $T[s+1, ..., s+m] = P[1...m]$

# Exaktes String–Matching (I): Definition

## Voraussetzungen

- Ein Text in einem Feld $T[1...n]$ der Länge $n$.
- Ein Textmuster in einem Feld $P[1...m]$ der Länge $m \leq n$, welches in $T$ mit Shift $s \in \mathbb{N}_0$ enthalten ist:
  $T[s+1, ..., s+m] = P[1...m]$
- Wenn $P$ mit Verschiebung $s$ in $T$ vorkommt, gilt diese Verschiebung als gültig.

# Exaktes String–Matching (I): Definition

## Voraussetzungen

- Ein Text in einem Feld $T[1...n]$ der Länge $n$.
- Ein Textmuster in einem Feld $P[1...m]$ der Länge $m \leq n$, welches in $T$ mit Shift $s \in \mathbb{N}_0$ enthalten ist:
  $T[s+1, ..., s+m] = P[1...m]$
- Wenn $P$ mit Verschiebung $s$ in $T$ vorkommt, gilt diese Verschiebung als gültig.
- Ein endliches Alphabet $\Sigma$, das alle Zeichen von $T$ und $P$ enthält:
  $\forall i \in \{1, \ldots, m\} : P[i] \in \Sigma \quad \wedge \quad \forall j \in \{1, \ldots, n\} : T[j] \in \Sigma$

# Exaktes String–Matching (I): Definition

## Voraussetzungen

- Ein Text in einem Feld $T[1...n]$ der Länge $n$.
- Ein Textmuster in einem Feld $P[1...m]$ der Länge $m \leq n$, welches in $T$ mit Shift $s \in \mathbb{N}_0$ enthalten ist:
  $T[s+1, ..., s+m] = P[1...m]$
- Wenn $P$ mit Verschiebung $s$ in $T$ vorkommt, gilt diese Verschiebung als gültig.
- Ein endliches Alphabet $\Sigma$, das alle Zeichen von $T$ und $P$ enthält:
  $\forall i \in \{1, \ldots, m\} : P[i] \in \Sigma \quad \wedge \quad \forall j \in \{1, \ldots, n\} : T[j] \in \Sigma$

z.B. $\Sigma := \{0, 1\}$ oder $\Sigma := \{a, b, c, ..., z\}$.

# Exaktes String–Matching (I): Definition

## Voraussetzungen

- Ein Text in einem Feld $T[1...n]$ der Länge $n$.
- Ein Textmuster in einem Feld $P[1...m]$ der Länge $m \leq n$, welches in $T$ mit Shift $s \in \mathbb{N}_0$ enthalten ist:
  $T[s+1, ..., s+m] = P[1...m]$
- Wenn $P$ mit Verschiebung $s$ in $T$ vorkommt, gilt diese Verschiebung als gültig.
- Ein endliches Alphabet $\Sigma$, das alle Zeichen von $T$ und $P$ enthält:
  $\forall i \in \{1, \ldots, m\} : P[i] \in \Sigma \quad \wedge \quad \forall j \in \{1, \ldots, n\} : T[j] \in \Sigma$

z.B. $\Sigma := \{0, 1\}$ oder $\Sigma := \{a, b, c, ..., z\}$.

## Problem

Finde alle gültige Verschiebungen $s$, mit denen ein gegebenes Muster $P$ in einem Text $T$ auftritt.

# Exact–String–Matching (II): Naïver Algorithmus

**Naïver Algorithmus:**
Verschiebe $P$ entlang von $T$ und vergleiche an jeder Position die Symbole von links nach rechts.

Sei $P =$ „ab“, $T =$ „aaaaaab“:

**a a a a a a b**

# Exact–String–Matching (II): Naïver Algorithmus

**Naïver Algorithmus:**
Verschiebe $P$ entlang von $T$ und vergleiche an jeder Position die Symbole von links nach rechts.

Sei $P =$ „ab“, $T =$ „aaaaaab“:

```
a  a  a  a  a  a  b
a  b                 no match!
```

# Exact–String–Matching (II): Naïver Algorithmus

**Naïver Algorithmus:**
Verschiebe $P$ entlang von $T$ und vergleiche an jeder Position die Symbole von links nach rechts.

Sei $P =$ „ab“, $T =$ „aaaaaab“:

```
a a a a a a b
a b                 no match!
  a b               no match!
```

# Exact–String–Matching (II): Naïver Algorithmus

**Naïver Algorithmus:**
Verschiebe $P$ entlang von $T$ und vergleiche an jeder Position die Symbole von links nach rechts.

Sei $P =$ „ab“, $T =$ „aaaaaab“:

```
a a a a a a b
a b                 no match!
  a b               no match!
    a b             no match!
```

# Exact–String–Matching (II): Naïver Algorithmus

**Naïver Algorithmus:**
Verschiebe $P$ entlang von $T$ und vergleiche an jeder Position die Symbole von links nach rechts.

Sei $P =$ „ab“, $T =$ „aaaaaab“:

```
a a a a a a b
a b                 no match!
  a b               no match!
    a b             no match!
      a b           no match!
```

# Exact–String–Matching (II): Naïver Algorithmus

**Naïver Algorithmus:**
Verschiebe $P$ entlang von $T$ und vergleiche an jeder Position die Symbole von links nach rechts.

Sei $P =$ „ab“, $T =$ „aaaaaab“:

```
a a a a a a b
a b                 no match!
  a b               no match!
    a b             no match!
      a b           no match!
        a b         no match!
```

# Exact–String–Matching (II): Naïver Algorithmus

**Naïver Algorithmus:**
Verschiebe $P$ entlang von $T$ und vergleiche an jeder Position die Symbole von links nach rechts.

Sei $P =$ „ab“, $T =$ „aaaaaab“:

```
a a a a a a b
a b                 no match!
  a b               no match!
    a b             no match!
      a b           no match!
        a b         no match!
          a b       match!
```

# Exact–String–Matching (III): Naïver Algorithmus

- Naïver Algorithmus hat eine Laufzeitkomplexität von $\Theta(m \cdot (n-m+1))$.

# Exact–String–Matching (III): Naïver Algorithmus

- Naïver Algorithmus hat eine Laufzeitkomplexität von $\Theta(m \cdot (n-m+1))$.
- In der Praxis oft trotzdem sehr schnell, da in natürlichen Texten Fehler nach ein bis zwei Zeichen auftauchen.

# Exact–String–Matching (III): Naïver Algorithmus

- Naïver Algorithmus hat eine Laufzeitkomplexität von $\Theta(m \cdot (n-m+1))$.
- In der Praxis oft trotzdem sehr schnell, da in natürlichen Texten Fehler nach ein bis zwei Zeichen auftauchen.
- Zu effizienteren Algorithmen zählen u.A. der *Rabin–Karp*–, *Knuth–Morris*– und der *Boyer–Moore*–Algorithmus.

# Nicht–Exaktes String Matching: Edit Distance

## Edit Distance

Die *Edit Distance* $E(S, T)$ von zwei Strings $S[1...m]$ und $T[1...n]$ mit $|S| =: m$ und $|T| =: n$ sei definiert als die minimale Anzahl von Änderungsoperationen (Einfügen, Löschen, Ändern), die notwendig sind, um $S$ in $T$ zu verwandeln.

# Nicht–Exaktes String Matching: Edit Distance

## Edit Distance

Die *Edit Distance* $E(S, T)$ von zwei Strings $S[1...m]$ und $T[1...n]$ mit $|S| =: m$ und $|T| =: n$ sei definiert als die minimale Anzahl von Änderungsoperationen (Einfügen, Löschen, Ändern), die notwendig sind, um $S$ in $T$ zu verwandeln.

z.B.: $E(\text{„interestingly“}, \text{„bioinformatics“}) = 11$:

| | |
|---|---|
| S := interestingly | `_i__nterestingly` |
| T := bioinformatics | `bioinformatics__` |
| | `10110110110011111` |

# DNA–Aufbau

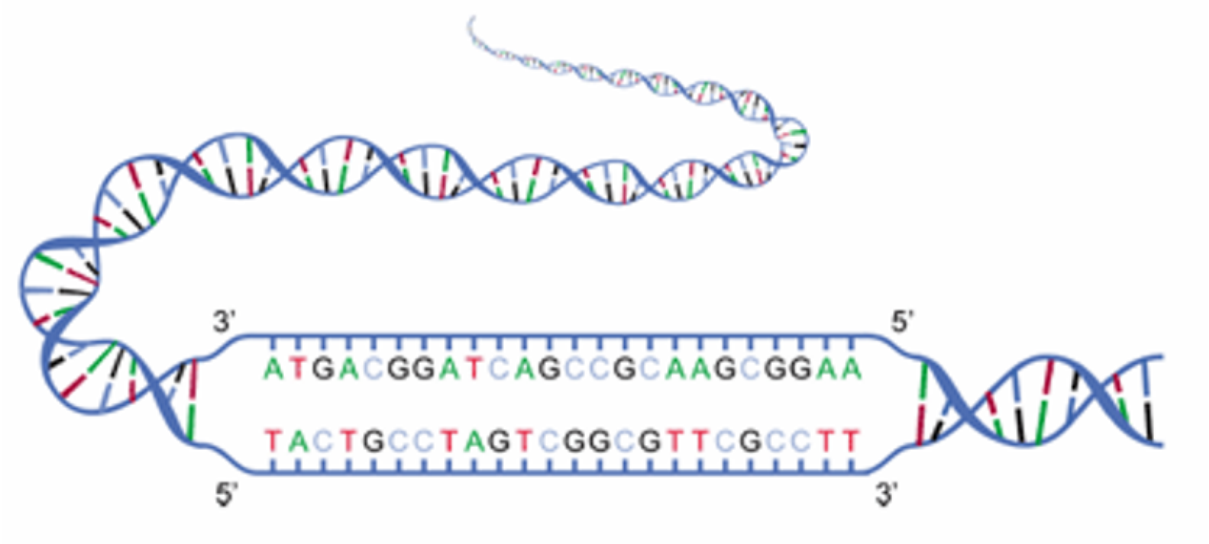



Abbildung: `www.biologie-schule.de%2Fdesoxyribonukleinsaeure-dna.php&h=-AQEwRF2I`

- Träger des Erbguts.

# DNA–Aufbau

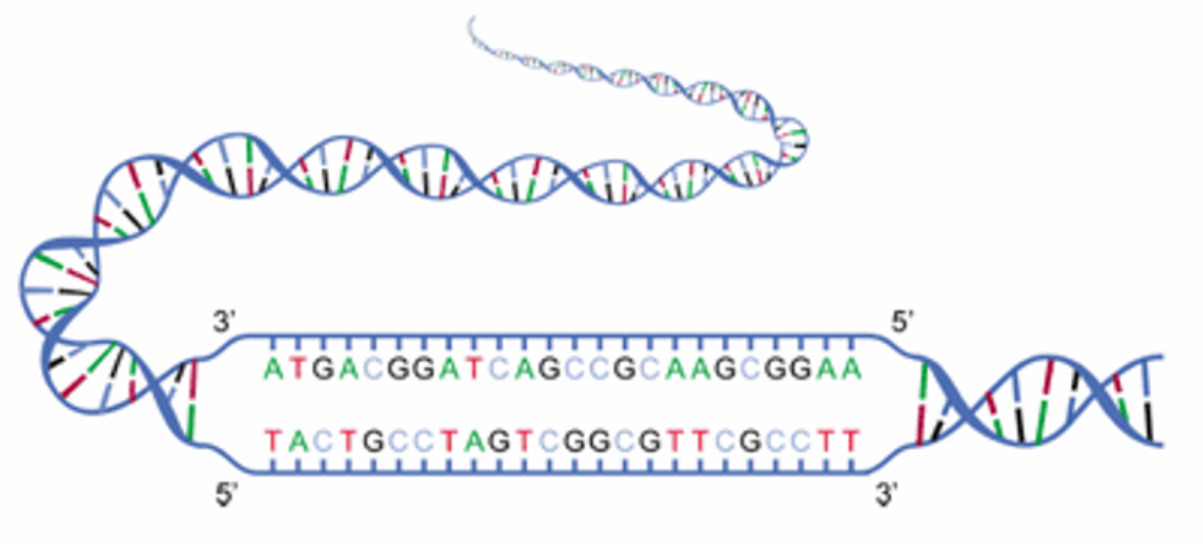



Abbildung: `www.biologie-schule.de%2Fdesoxyribonukleinsaeure-dna.php&h=-AQEwRF2I`

- Träger des Erbguts.
- Doppelhelix aus zwei komplementären Einzelsträngen.

# DNA–Aufbau


Abbildung: `www.biologie-schule.de%2Fdesoxyribonukleinsaeure-dna.php&h=-AQEwRF2I`

- Träger des Erbguts.
- Doppelhelix aus zwei komplementären Einzelsträngen.
- Anordnung der vier Basen (Adenin und Thymin, Guanin und Cytosin) ergibt Gencode.

# Genetischer Code

- Ein Gen codiert ein Protein ← Phänotyp.

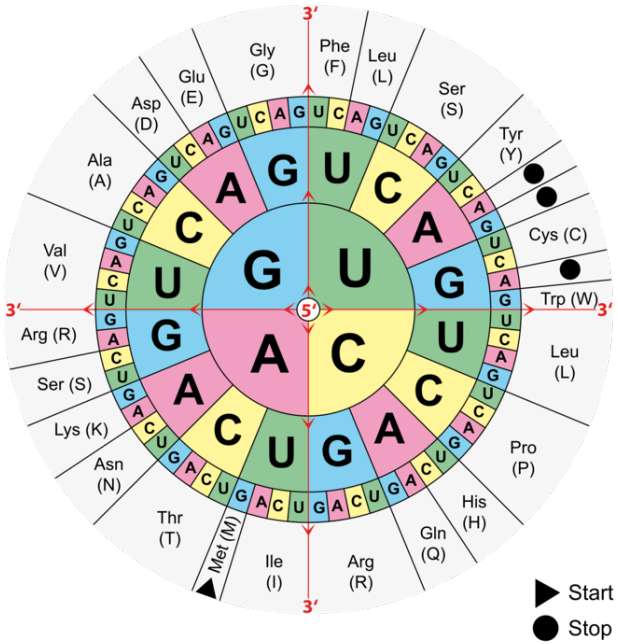



Abbildung: `http://de.wikipedia.org/wiki/Genetischer_Code`

# Genetischer Code

- Ein Gen codiert ein Protein ← Phänotyp.
- Je 3 Basen verschlüsseln eine Animosäure.

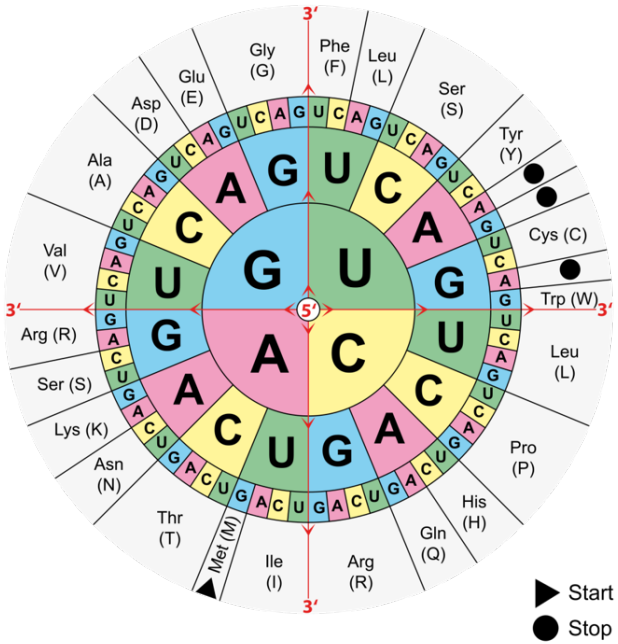



Abbildung: `http://de.wikipedia.org/wiki/Genetischer_Code`

# Genetischer Code

- Ein Gen codiert ein Protein ← Phänotyp.
- Je 3 Basen verschlüsseln eine Animosäure.
- Proteine sind aus 20 verschiedenen Aminosäuren aufgebaut.

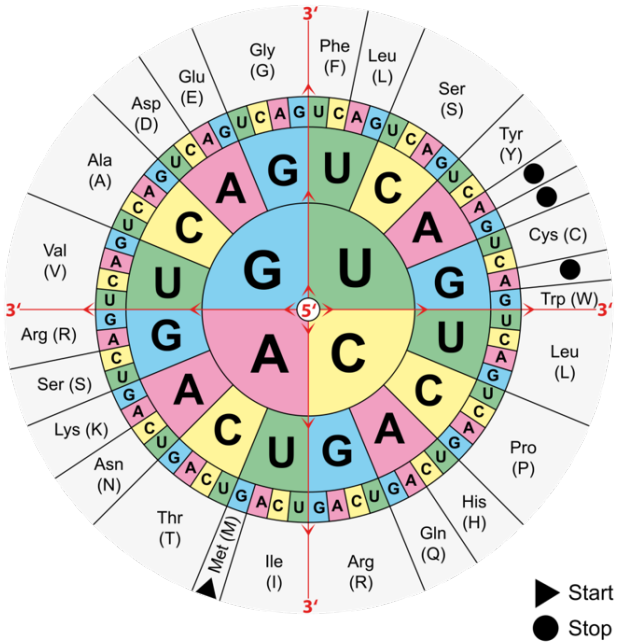



Abbildung: `http://de.wikipedia.org/wiki/Genetischer_Code`

# Genetischer Code

- Ein Gen codiert ein Protein ← Phänotyp.
- Je 3 Basen verschlüsseln eine Animosäure.
- Proteine sind aus 20 verschiedenen Aminosäuren aufgebaut.
- Zentraler Vorgang der Genexpression: Proteinbiosynthese.

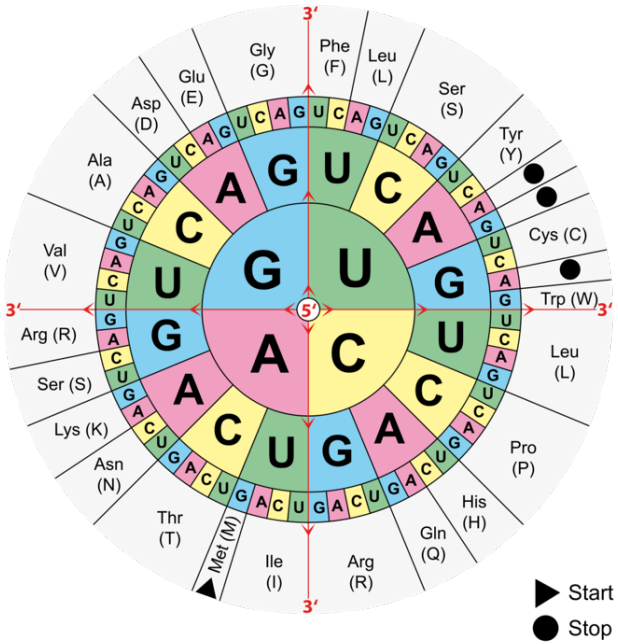



Abbildung: `http://de.wikipedia.org/wiki/Genetischer_Code`

# Mutationen

- Genommutation

# Mutationen

- Genommutation
- Chromosomenmutation

# Mutationen

- Genommutation
- Chromosomenmutation
- Genmutation: `AATGACG`
  - Substitution `ATTGACG`
  - Deletion `A_TGACG`
  - Insertion `AAATGACG`
  - Triplettrepeatexpansion `AATGACGAATGACG`

1. String–Matching
   - Motivation
   - Anwendungsgebiete
   - Exaktes String–Matching
   - Nicht exaktes String–Matching
2. Exkurs: DNA
   - DNA–Aufbau
   - Genetischer Code
   - Mutationen
3. String–Matching–Algorithmen in der Bioinformatik
   - Warum String–Matching in der DNA–Analyse?
   - Exaktes String–Matching in der Bioinformatik
   - Nicht–Exaktes String Matching in der Bioinformatik
   - Globales Alignment: Needleman–Wunsch

# Warum String–Matching in der DNA–Analyse?

- Hohe Sequenzähnlichkeit heißt sehr oft ähnliche Struktur und Funktion im Phänotyp!

# Warum String–Matching in der DNA–Analyse?

- Hohe Sequenzähnlichkeit heißt sehr oft ähnliche Struktur und Funktion im Phänotyp!
- Vergleichen von DNA Sequenzen.

# Warum String–Matching in der DNA–Analyse?

- Hohe Sequenzähnlichkeit heißt sehr oft ähnliche Struktur und Funktion im Phänotyp!
- Vergleichen von DNA Sequenzen.
- Durchsuchen von Datenbanken nach ähnlichen Sequenzen.

# Warum String–Matching in der DNA–Analyse?

- Hohe Sequenzähnlichkeit heißt sehr oft ähnliche Struktur und Funktion im Phänotyp!
- Vergleichen von DNA Sequenzen.
- Durchsuchen von Datenbanken nach ähnlichen Sequenzen.
- Muster suchen, die häufig in DNA vorkommen.

# Warum String–Matching in der DNA–Analyse?

- Hohe Sequenzähnlichkeit heißt sehr oft ähnliche Struktur und Funktion im Phänotyp!
- Vergleichen von DNA Sequenzen.
- Durchsuchen von Datenbanken nach ähnlichen Sequenzen.
- Muster suchen, die häufig in DNA vorkommen.
- Rekonstruieren von DNA Strängen aus sich überlappenden Teilen.

# Exaktes String–Matching in der Bioinformatik

- Für die Kartierung unbekannter DNA-Sequenzen.

# Exaktes String–Matching in der Bioinformatik

- Für die Kartierung unbekannter DNA-Sequenzen.
- Als Teilproblem bei Datenbanksuche.

# Exaktes String–Matching in der Bioinformatik

- Für die Kartierung unbekannter DNA-Sequenzen.
- Als Teilproblem bei Datenbanksuche.
- Als Teilproblem von multiplen Sequenzalignments.

# Nicht–Exaktes String–Matching in der Bioinformatik

Wichtig aufgrund von Mutationen und möglichen Fehlern beim Sequenzieren.

# Nicht–Exaktes String–Matching in der Bioinformatik

Wichtig aufgrund von Mutationen und möglichen Fehlern beim Sequenzieren.

Statt zu betrachten, wie verschieden zwei Strings $S$ und $T$ sind, wie bei der Edit Distance, analysiert man in der Bioinformatik oft eher deren Ähnlichkeiten.

# Nicht–Exaktes String–Matching in der Bioinformatik

Wichtig aufgrund von Mutationen und möglichen Fehlern beim Sequenzieren.

Statt zu betrachten, wie verschieden zwei Strings $S$ und $T$ sind, wie bei der Edit Distance, analysiert man in der Bioinformatik oft eher deren Ähnlichkeiten.

## Definition: Alignment

Ein (globales) *Alignment* von zwei Strings $S_1$ und $S_2$ erhält man, indem man zuerst Platzhalter entweder in die Mitte oder am Ende von $S_1$ und $S_2$ einfügt und dann die zwei resultierenden Strings übereinander legt, sodass jedes (Leer)Zeichen in einem der beiden Strings gegenüber einem eindeutigen (Leer)Zeichen des anderen Strings liegt.

# Nicht–Exaktes String–Matching in der Bioinformatik

- Dabei wird eine Ähnlichkeitsfunktion $\delta : (\Sigma \cup \{\_\}) \times (\Sigma \cup \{\_\}) \rightarrow \mathbb{Z}$ verwendet.

# Nicht–Exaktes String–Matching in der Bioinformatik

- Dabei wird eine Ähnlichkeitsfunktion $\delta : (\Sigma \cup \{\_\}) \times (\Sigma \cup \{\_\}) \to \mathbb{Z}$ verwendet.
- Hierbei sei $\Sigma$ die Menge der möglichen in $S$ und $T$ enthaltenen Zeichen und _ ein Null–Symbol. $\delta(x, y)$ bezeichnet dann die Ähnlichkeit von zwei Zeichen $x, y \in \Sigma \cup \{\_\}$.

# Nicht–Exaktes String–Matching in der Bioinformatik

- Dabei wird eine Ähnlichkeitsfunktion $\delta : (\Sigma \cup \{\_\}) \times (\Sigma \cup \{\_\}) \to \mathbb{Z}$ verwendet.
- Hierbei sei $\Sigma$ die Menge der möglichen in $S$ und $T$ enthaltenen Zeichen und _ ein Null–Symbol. $\delta(x, y)$ bezeichnet dann die Ähnlichkeit von zwei Zeichen $x, y \in \Sigma \cup \{\_\}$.

**Problem**

Finde zu zwei gegebenen Strings $S[1...m]$ und $T[1...n]$ ein Alignment $A$, das $\sum_{(x,y)\in A} \delta(x, y)$ maximiert.

# Needleman–Wunsch Algorithmus

z.B.: Finde das optimale globale Alignment $A$ für $S$ := „ACAATCC“ und $T$ := „AGCATGC“.

| | |
|---|---|
| S := „ACAATCC“ | `A_CAATCC` |
| T := „AGCATGC“ | `AGCA_TGC` |

wobei $\Sigma := \{\_, \text{A}, \text{G}, \text{C}, \text{T}\}$, $\delta(x, y) := \begin{cases} 2\,, & \text{falls } x = y \\ -1\,, & \text{sonst} \end{cases}$.

# Needleman–Wunsch Algorithmus

z.B.: Finde das optimale globale Alignment $A$ für $S$ := „ACAATCC“ und $T$ := „AGCATGC“.

| | |
|---|---|
| S := „ACAATCC“ | `A_CAATCC` |
| T := „AGCATGC“ | `AGCA_TGC` |

wobei $\Sigma := \{\_, \text{A}, \text{G}, \text{C}, \text{T}\}$, $\delta(x, y) := \begin{cases} 2\,, & \text{falls } x = y \\ -1\,, & \text{sonst} \end{cases}$.

- Dann liegt der Ähnlichkeitsscore von $S$ und $T$ bei 7.

# Needleman–Wunsch Algorithmus

z.B.: Finde das optimale globale Alignment $A$ für $S$ := „ACAATCC“ und $T$ := „AGCATGC“.

| | |
|---|---|
| S := „ACAATCC“ | `A_CAATCC` |
| T := „AGCATGC“ | `AGCA_TGC` |

wobei $\Sigma := \{\_, \text{A}, \text{G}, \text{C}, \text{T}\}$, $\delta(x, y) := \begin{cases} 2\,, & \text{falls } x = y \\ -1\,, & \text{sonst} \end{cases}$.

- Dann liegt der Ähnlichkeitsscore von $S$ und $T$ bei 7.
- Um dieses Alignment zu finden, wird im Needleman–Wunsch Algorithmus dynamische Programmierung eingesetzt.

# Needleman–Wunsch: Rekurrenzgleichung

Gegeben: Zwei Strings $S$ und $T$ mit $|S| = m$ und $|T| = n$. Definiere $V(i, j)$ als den Score des optimalen Alignments von den Substrings $S[1...i]$, $1 \leq i \leq m$ von $S$ und $T[1...j]$, $1 \leq j \leq n$ von $T$. Um $V(i, j)$ zu finden, gilt:

# Needleman–Wunsch: Rekurrenzgleichung

Gegeben: Zwei Strings $S$ und $T$ mit $|S| = m$ und $|T| = n$. Definiere $V(i, j)$ als den Score des optimalen Alignments von den Substrings $S[1...i]$, $1 \leq i \leq m$ von $S$ und $T[1...j]$, $1 \leq j \leq n$ von $T$. Um $V(i, j)$ zu finden, gilt:

$$
\begin{aligned}
V(0,0) &= 0 \\
V(i,0) &= V(i-1,0) + \delta(S[i], \_) \\
V(0,j) &= V(0,j-1) + \delta(\_, T[j]) \\
V(i,j) &= \texttt{max} \begin{cases} V(i-1,j-1) + \delta(S[i], T[j]) \\ V(i-1,j) + \delta(S[i], \_) \\ V(i,j-1) + \delta(\_, T[j]) \end{cases}
\end{aligned}
$$

# Beispiel: Needleman–Wunsch Algorithmus

Finde das optimale Alignment für $S :=$ „ACAATCC“, $T :=$ „AGCATGC“.

| | _ | A | G | C | A | T | G | C |
|---|---|---|---|---|---|---|---|---|
| _ | | | | | | | | |
| A | | | | | | | | |
| C | | | | | | | | |
| A | | | | | | | | |
| A | | | | | | | | |
| T | | | | | | | | |
| C | | | | | | | | |
| C | | | | | | | | |

# Beispiel: Needleman–Wunsch Algorithmus

Finde das optimale Alignment für $S :=$ „ACAATCC“, $T :=$ „AGCATGC“.

| | _ | A | G | C | A | T | G | C |
|---|---|---|---|---|---|---|---|---|
| _ | $0 \leftarrow$ | $-1 \leftarrow$ | $-2 \leftarrow$ | $-3 \leftarrow$ | $-4 \leftarrow$ | $-5 \leftarrow$ | $-6 \leftarrow$ | $-7$ |
| A | $\uparrow -1$ | | | | | | | |
| C | $\uparrow -2$ | | | | | | | |
| A | $\uparrow -3$ | | | | | | | |
| A | $\uparrow -4$ | | | | | | | |
| T | $\uparrow -5$ | | | | | | | |
| C | $\uparrow -6$ | | | | | | | |
| C | $\uparrow -7$ | | | | | | | |

$$V(i,j) = \texttt{max} \begin{cases} V(i-1,j-1) + \delta(S[i], T[j]) \\ V(i-1,j) + \delta(S[i], \_) \\ V(i,j-1) + \delta(\_, T[j]) \end{cases}$$

$$\delta(x,y) = \begin{cases} 2, & \text{falls } x = y \\ -1, & \text{sonst} \end{cases}$$

# Beispiel: Needleman–Wunsch Algorithmus

Finde das optimale Alignment für $S :=$ „ACAATCC“, $T :=$ „AGCATGC“.

| | _ | A | G | C | A | T | G | C |
|---|---|---|---|---|---|---|---|---|
| _ | $0$ | $\leftarrow -1$ | $\leftarrow -2$ | $\leftarrow -3$ | $\leftarrow -4$ | $\leftarrow -5$ | $\leftarrow -6$ | $\leftarrow -7$ |
| A | $\uparrow -1$ | $2 \leftarrow$ | $1 \leftarrow$ | $0$ | | | | |
| C | $\uparrow -2$ | | | | | | | |
| A | $\uparrow -3$ | | | | | | | |
| A | $\uparrow -4$ | | | | | | | |
| T | $\uparrow -5$ | | | | | | | |
| C | $\uparrow -6$ | | | | | | | |
| C | $\uparrow -7$ | | | | | | | |

$$V(i,j) = \texttt{max} \begin{cases} V(i-1,j-1) + \delta(S[i], T[j]) \\ V(i-1,j) + \delta(S[i], \_) \\ V(i,j-1) + \delta(\_, T[j]) \end{cases}$$

$$\delta(x,y) = \begin{cases} 2, & \text{falls } x = y \\ -1, & \text{sonst} \end{cases}$$

# Beispiel: Needleman–Wunsch Algorithmus

Finde das optimale Alignment für $S :=$ „ACAATCC“, $T :=$ „AGCATGC“.

| | _ | A | G | C | A | T | G | C |
|---|---|---|---|---|---|---|---|---|
| _ | 0 | ← −1 | ← −2 | ← −3 | ← −4 | ← −5 | ← −6 | ← −7 |
| A | ↑ −1 | ↖ 2 | ← 1 | ← 0 | ← −1 | ← −2 | ← −3 | ← −4 |
| C | ↑ −2 | ↑ 1 | ↖ 1 | ↖ 3 | | | | |
| A | ↑ −3 | | | | | | | |
| A | ↑ −4 | | | | | | | |
| T | ↑ −5 | | | | | | | |
| C | ↑ −6 | | | | | | | |
| C | ↑ −7 | | | | | | | |

# Beispiel: Needleman–Wunsch Algorithmus

Finde das optimale Alignment für $S :=$ „ACAATCC“, $T :=$ „AGCATGC“.

| | _ | A | G | C | A | T | G | C |
|---|---|---|---|---|---|---|---|---|
| _ | 0 | −1 | −2 | −3 | −4 | −5 | −6 | −7 |
| A | −1 | 2 | 1 | 0 | −1 | −2 | −3 | −4 |
| C | −2 | 1 | 1 | 3 | 2 | 1 | 0 | −1 |
| A | −3 | 0 | 0 | 2 | 5 | 4 | 3 | 2 |
| A | −4 | −1 | −1 | 1 | 4 | 4 | 3 | 2 |
| T | −5 | −2 | −2 | 0 | 3 | 6 | 5 | 4 |
| C | −6 | −3 | −3 | 0 | 2 | 5 | 5 | 7 |
| C | −7 | −4 | −4 | −1 | 1 | 4 | 4 | 7 |

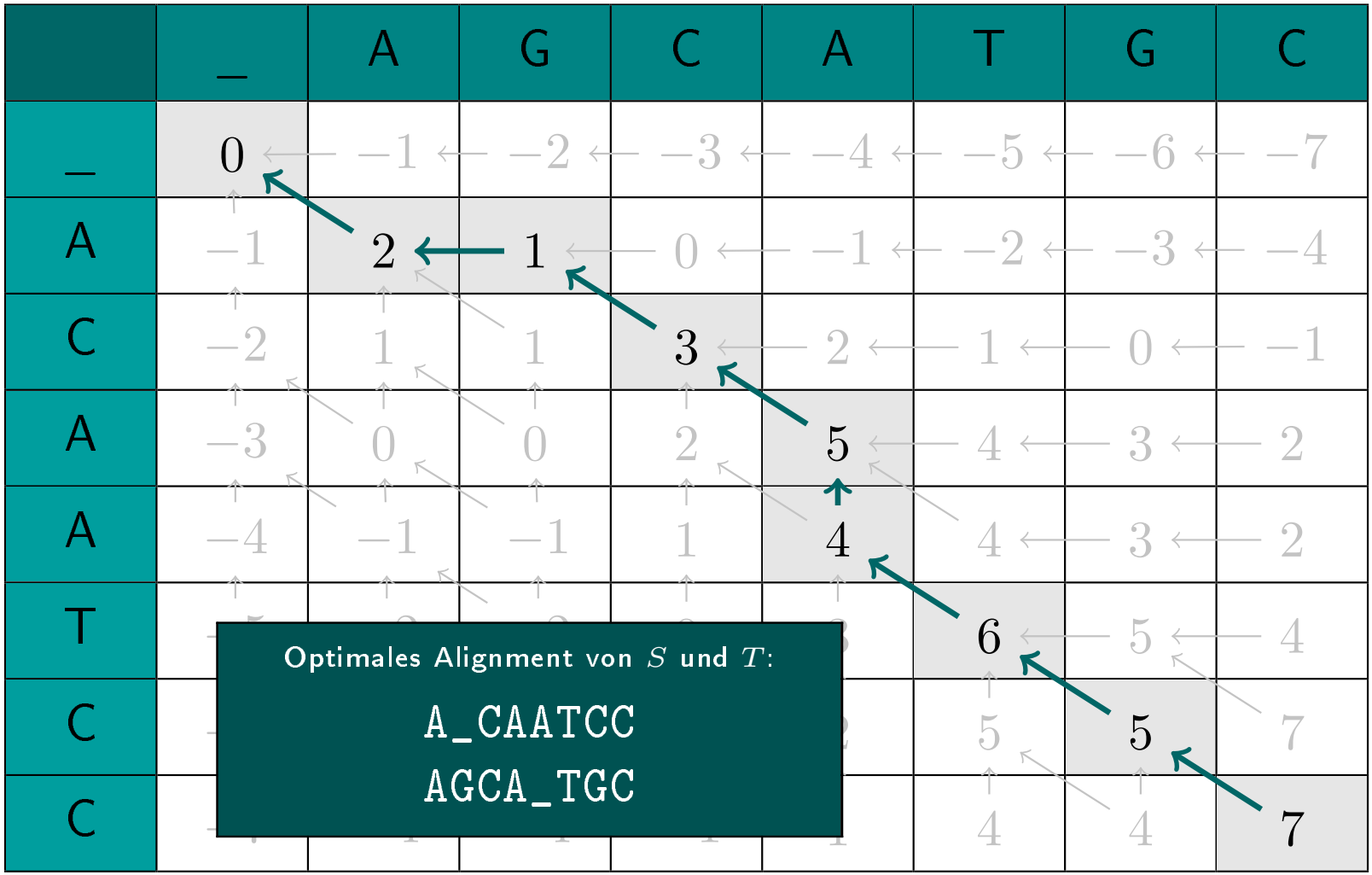
# Beispiel: Needleman–Wunsch Algorithmus

Finde das optimale Alignment für $S :=$ „ACAATCC“, $T :=$ „AGCATGC“.

| | _ | A | G | C | A | T | G | C |
|---|---|---|---|---|---|---|---|---|
| _ | 0 | −1 | −2 | −3 | −4 | −5 | −6 | −7 |
| A | −1 | 2 | 1 | 0 | −1 | −2 | −3 | −4 |
| C | −2 | 1 | 1 | 3 | 2 | 1 | 0 | −1 |
| A | −3 | 0 | 0 | 2 | 5 | 4 | 3 | 2 |
| A | −4 | −1 | −1 | 1 | 4 | 4 | 3 | 2 |
| T | −5 | −2 | −2 | 0 | 3 | 6 | 5 | 4 |
| C | −6 | −3 | −3 | 0 | 2 | 5 | 5 | 7 |
| C | −7 | −4 | −4 | −1 | 1 | 4 | 4 | 7 |

# Beispiel: Needleman–Wunsch Algorithmus

Finde das optimale Alignment für $S :=$ „ACAATCC“, $T :=$ „AGCATGC“.



# Needleman–Wunsch: Zeit– / Speicherkomplexitätsanalyse

- Man muss alle Einträge in einer $(m \times n)$ – Matrix auffüllen.

# Needleman–Wunsch: Zeit– / Speicherkomplexitätsanalyse

- Man muss alle Einträge in einer $(m \times n)$ – Matrix auffüllen.
- Das Auffüllen einer einzelnen Zelle in der Matrix kostet $O(1)$ Zeit, demnach beträgt die Gesamtlaufzeit von Needleman–Wunsch $O(mn)$, wobei $m = |S|$ sowie $n = |T|$.

# Needleman–Wunsch: Zeit– / Speicherkomplexitätsanalyse

- Man muss alle Einträge in einer $(m \times n)$ – Matrix auffüllen.
- Das Auffüllen einer einzelnen Zelle in der Matrix kostet $O(1)$ Zeit, demnach beträgt die Gesamtlaufzeit von Needleman–Wunsch $O(mn)$, wobei $m = |S|$ sowie $n = |T|$.
- Die Laufzeitkomplexität kann auf $O((2d + 1) \cdot (n + m))$ vermindert werden, wobei $0 < d \leq n + m$ eine vorbestimmte obere Schranke für die Anzahl der erlaubten Einfüge– und Löschoperationen ist. Man füllt dabei nur den entsprechenden Teil entlang der Diagonale in der Matrix aus.

# Needleman–Wunsch: Zeit– / Speicherkomplexitätsanalyse

- Man muss alle Einträge in einer $(m \times n)$ – Matrix auffüllen.
- Das Auffüllen einer einzelnen Zelle in der Matrix kostet $O(1)$ Zeit, demnach beträgt die Gesamtlaufzeit von Needleman–Wunsch $O(mn)$, wobei $m = |S|$ sowie $n = |T|$.
- Die Laufzeitkomplexität kann auf $O((2d + 1) \cdot (n + m))$ vermindert werden, wobei $0 < d \leq n + m$ eine vorbestimmte obere Schranke für die Anzahl der erlaubten Einfüge– und Löschoperationen ist. Man füllt dabei nur den entsprechenden Teil entlang der Diagonale in der Matrix aus.
- Mit einem Divide & Conquer–Ansatz von Needleman–Wunsch lässt sich die Speicherkomplexität von $O(mn)$ auf $O(m + n)$ vermindern.

# Danke für die Aufmerksamkeit!

# Quellen

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C., et al. (2001). *Introduction to algorithms*, volume 2. MIT press Cambridge.

Fernandez, E., Najjar, W., Harris, E., & Lonardi, S. (2010). Exploration of short reads genome mapping in hardware. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on* (pp. 360–363).: IEEE.

Gusfield, D. (1997). *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press.

Sung, W.-K. (2011). *Algorithms in bioinformatics: A practical introduction*. CRC Press.

`http://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_string_search_algorithm`