

# Bioinformatik

Z-Box Algorithmus – Lineares Stringmatching

Ulf Leser

Wissensmanagement in der  
Bioinformatik

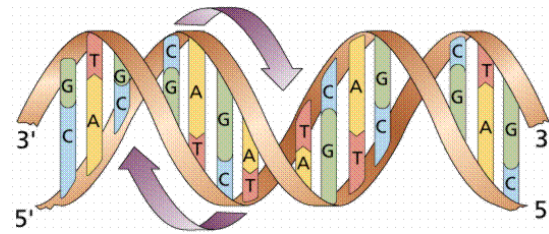


# Sequenz - Funktion

---

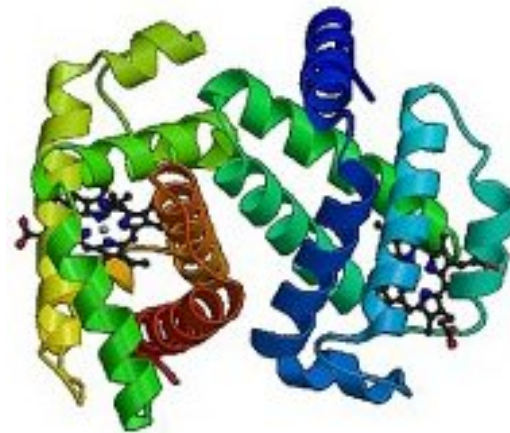
- DNA

- Vererbung
- Genotyp
- Produktion von Proteinen



- Proteine

- Phänotyp
- Struktur
- Bindungsverhalten



# Drei Anwendungen

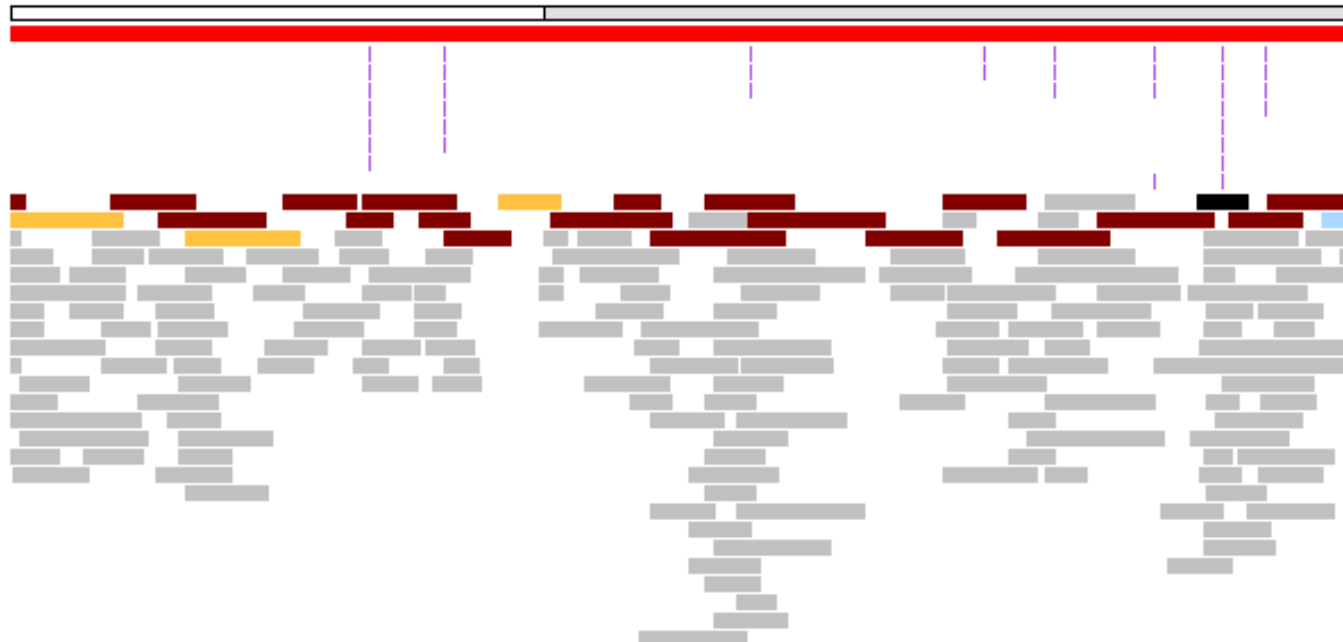
---

- Sequenzierung
  - Assembly von Teilsequenzen
- Funktionale Annotation
  - Schnelle Suche in Sequenzdatenbanken

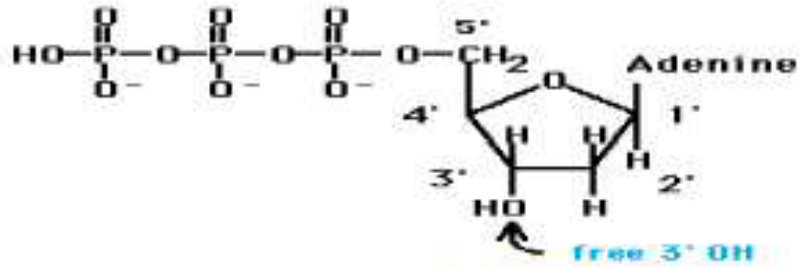
# Mapping und Sequenzierung

---

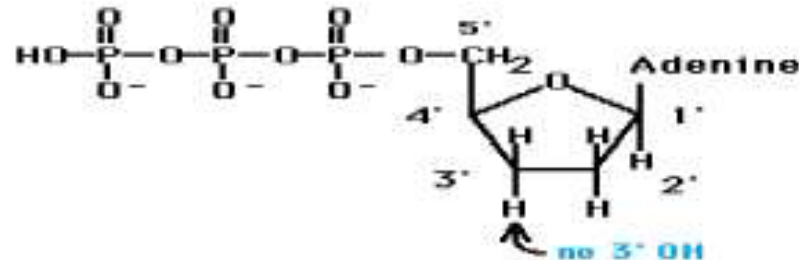
- Zerlegung in Bruchstücke (Clonierung)
- Berechnung aller Überlappungen
- Bestimmung der wahrscheinlichsten Gesamtsequenz
- Variante: Bestimmung des **Minimum Tiling Paths**



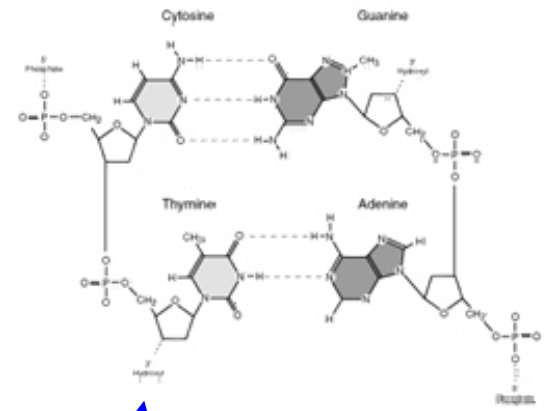
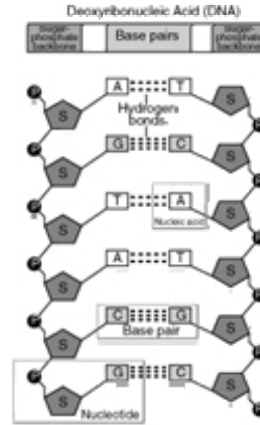
# dNTP versus ddNTP



dNTP



ddNTP



- Dideoxy-Base: keine freie OH Gruppe
- Dideoxy-Base wird eingebaut
- Danach können keine weiteren Basen mehr angehängt werden
- Polymerase fällt ab



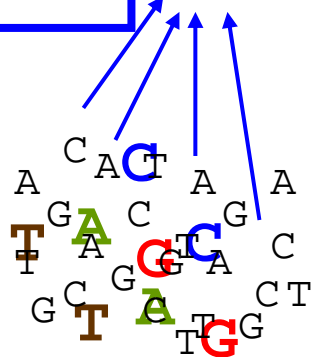
# Schritt 3

Primer

Template

ACGAACGCGAGTTAGTTAGACCAGTAGCCA...

Polymerase



ACGAACGCGAGTTAGTTAGACCAGTAGCCA...

ACGAACGCGAGTT**A**

ACGAACGCGAGTTAGTTAGACCAGTAGCCA...

ACGAACGCGA**G**

ACGAACGCGAGTTAGTTAGACCAGTAGCCA...

ACGAACGCGAGTTAGT**T**

ACGAACGCGAGTTAGTTAGACCAGTAGCCA...

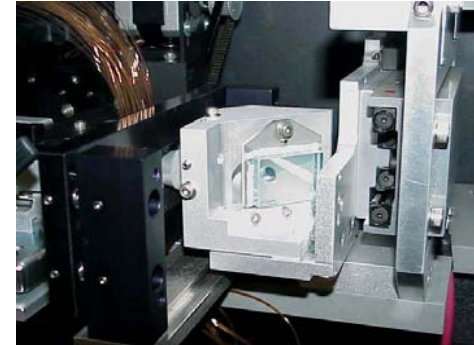
ACGAACGCGAGTTAGTTAG**T**

ACGAACGCGAGTTAGTTAGACCAGTAGCCA...

ACGAACGCG**A**

# Schritt 4

Laser &  
Detektoren



ACGAACGCGAGTT**A**

ACGAACGCGA**G**

ACGAACGCGAGTTAGT**T**

ACGAACGCGAGTTAGTTAG**T**

ACGAACGCG**A**

Gel / Kapillar  
Elektrophorese

ACGAACG**C**

ACGAACG**C**G

ACGAACGCG**A**

ACGAACGCGA**G**

ACGAACGCGAG**T**

ACGAACGCGAGT**T**

ACGAACGCGAGTT**A**

ACGAACGCGAGGTTA**G**



# Assembly

---

- Finden von Überlappungen von Sequenzen
- Redundanz ist
  - Notwendig: Verbindung von Teilstücken nur durch Überlappungen
  - Konflikträchtig: Widersprüche, Mehrdeutigkeit
- Beachtung von Sequenzierfehlern
- Auswahl der plausibelsten Anordnungen

## Fehler ?

```
tggacaagcaaagattga
      acatttttgaac
gcaaagattggtg

tggacaagcaaagattga
acatttttgaac
      gcaaagattggtg
```

# Abstrakte Formulierung

---

- **SUPERSTRING**

- Geg.: Menge  $S$  von Strings
- Ges.: String  $T$  so, dass
  - (a)  $\forall s \in S: s \in T$  (s Substring von  $T$ )
  - (b)  $\forall T'$ , für die (a) gilt, gilt:  $|T| \leq |T'|$  ( $T$  ist minimal)
- Sehr schwieriges Problem
  - Exponentielle Komplexität
  - „NP-vollständig“

- **Assembly: Verschärfungen von SUPERSTRING**

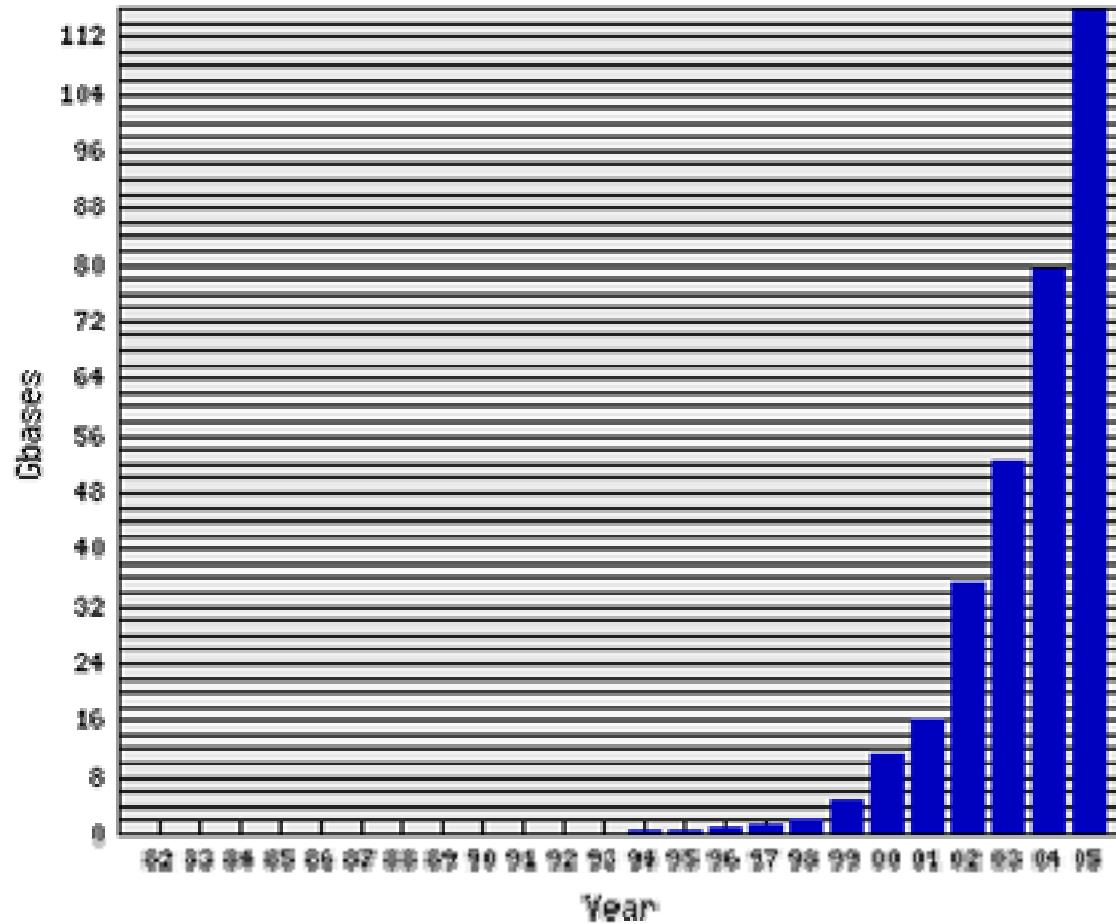
- Fehler in Sequenzen (s „ungefähr Substring“ von  $T$ )
- Zwei Orientierungen von  $s$  möglich

# Standardvorgehen

---

- Gegeben: Eine frisch sequenzierte DNA Sequenz
- **Annotationspipeline**
  - Suche nach ähnlichen Gensequenzen
  - Suche nach ähnlichen Promotersequenzen
  - Suche nach ähnlichen Proteinen (Übersetzung- Rückübersetzung)
  - Vorhersage neuer Genen durch Programme (trainiert auf bekannten Gensequenzen)
  - Suche nach ähnlichen Proteindomänen durch Programme (trainiert auf bekannten Proteindomänen)
  - ...
- **Alternative: Experimentelle Überprüfung**
  - Teuer, auch nicht fehlerfrei
  - Ethische / technische Machbarkeit

# Problemdimension



Quelle: EMBL, Genome Monitoring Tables, Stand 9.11.2005

- 
- Strings und Matching - Problemklassen

# Problemklassen

---

- **Exaktes Matching**
  - Gegeben: Strings  $P, T$
  - Gesucht: Alle Auftreten von  $P$  in  $T$
  - Variante: Gegeben  $P_1, \dots, P_n, T$ : Vorkommen aller  $P_i$  in  $T$ ?
  - Anwendungen: Schnittstellen von Restriktionsenzymen, manche Bindungsstellen, Vektorsequenzen, ...
- **Approximatives Matchen**
  - Gegeben: Strings  $S, T$
  - Gesucht: Wie ähnlich sind sich  $S$  und  $T$ ?
  - Variante: Ist  $P$  in  $T$  mit höchstens  $k$  Fehlern enthalten?
  - Anwendungen: Finden konservierter Bereiche, Proteindomänen, Übertragung funktionaler Information
- **Suche in Datenbanken**
  - Gegeben: Datenbank  $D$  von Sequenzen, String  $P$
  - Gesucht: Die Top- $k$  zu  $P$  ähnlichsten Sequenzen in  $D$
  - Anwendungen: Approximatives Matching gegen 1000de Sequenzen

# Exaktes Matching

- Gegeben: P (Pattern) und T (Text)
  - Trivialerweise verlangen wir  $|P| \leq |T|$
  - In der Regel nehmen wir an  $|P| \ll |T|$
- Gesucht: Sämtliche Vorkommen von P in T
- Beispiel
  - Auffinden der Erkennungssequenzen von **Restriktionsenzymen**

## Eco RV - GATATC

```
tcagcttactaattaaaaattctttctagtaagtgctaagatcaagaaaaataaattaaaaataatggaacatggcacattttcctaaactcttcacagattgctaataga
ttattaattaaagaataaatggttataatttttatggtaacggaatttcctaaaatattaattcaagccatggaatgcaataagaaggactctgtaattgggtact
attcaactcaatgcaagtggaaactaagttgggtattaatactctttttacatatatatgtagttattttaggaagcgaaggacaatttcactctgctaataaagggattac
atatttatttttgtgaatataaaaaatagaaagtatggtatcagattaaacttttgagaaaggtaagatgaagtaaagctgtatactccagcaataagttcaaataggc
gaaaaactttttaataacaaaagttaataatcattttgggaattgaaatgtcaaagataattacttcacgataagtagttgaagatagtttaaattttctttttgtatt
acttcaatgaaggtaacgcaacaagattagagtatatatggccaataagggttgctgtaggaaaattattctaaggagatacgcgagaggggcttctcaaatttattcaga
gatggatggttttagatgggtgggttaagaaaaagcagatttaaatccagcaaaactagaccttaggtttattaagcgaggccaataagttaattggaattgtaaaagatat
cctaattcttcttcatttgggtggaggaaaactagtttaacttcttaccocatgcagggccataggggtcgaatacgcctgtcactaagcaaaaggaaaaatgtgagtgtagact
ttaaaccatttttattaatgactttagagaatcatgcatttgatgttactttcttaacaatgtgaacatatttatgcgattaagatgagttatgaaaaaggcgaatataat
tattcagttacatagagattatagctgggtctattcttagttataggacttttgacaagatagcttagaaaataagattatagagcttaataaaagagaacttcttggaaat
tagctgcctttgggtgcagctgtaattggctattgggtatggctccagcttactgggttaggttttaatagaaaaattccccatgattgctaattatatctatcctattgagaa
caacgtgcgaagatgagtggaatgggttctatttaactgctgggtgctatagtagttatccttagaaagatatataaatctgataaagcaaaatcctggggaaaatat
tgctaactgggtgctggtaggggtttggggatgggattatctcctctacaagaaatttgggtggttacttgatatcctataaataatagagaaaaaatttaataaagatgat
```

# Zeichenketten

---

- Definition

*Ein **String**  $S$  ist eine von links nach rechts angeordnete Liste von Zeichen eines Alphabets  $\Sigma$*

- *$|S|$  ist die Länge des Strings*
- *Positionen in  $S$  sind  $1, \dots, |S|$* 
  - *Wir zählen ab 1*
- *$S(i)$  beschreibt das Zeichen an der Position  $i$  im String  $S$*
- *$S[i..j]$  ist der Substring, welcher an Position  $i$  beginnt und an Position  $j$  endet*
- *$S[i..j]$  ist ein leerer String, falls  $i > j$*
- *$S[1..i]$  heißt **Präfix** von  $S$  bis zur Position  $i$*
- *$S[i..]$  ist das **Suffix** von  $S$ , welches an Position  $i$  beginnt*
- ***Echte Präfixe und echte Suffixe** umfassen nicht den gesamten String  $S$  und sind nicht leer*

# Naiver Ansatz

---

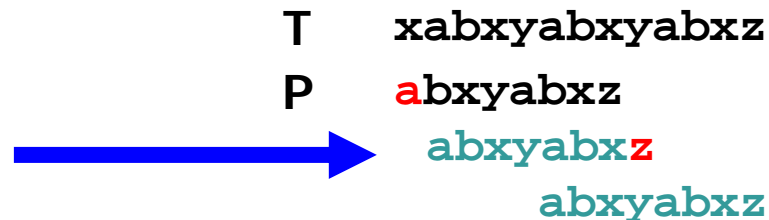
1. P und T an Position 1 ausrichten
2. Vergleiche P mit T von links nach rechts
  - Zwei ungleiche Zeichen  $\Rightarrow$  Gehe zu 3
  - Zwei gleiche Zeichen
    - P noch nicht durchlaufen  $\Rightarrow$  Verschiebe Pointer nach rechts, gehe zu 2
    - P vollständig durchlaufen  $\Rightarrow$  Merke Vorkommen von P in T
3. Verschiebe P um ein Zeichen nach rechts
4. Solange Startposition  $\leq |T| - |P|$ , gehe zu 2

```
T   ctgagatcgcgta
P   gagatc
    gagatc
     gagatc
      gagatc
       gagatc
        gatatc
         gatatc
          gatatc
```

# Optimierungsidee

---

- Anzahl der Vergleiche reduzieren
  - P um mehr als ein Zeichen verschieben
  - Aber nie soweit, dass ein Vorkommen von P in T nicht erkannt wird
- Beobachtung: Zeichen



- Substring in T muss mit a beginnen
- Nächstes a in T erst an Position 6 – springe 4 Positionen
- Vorkommen von Buchstaben in T kann während des Vergleichs ab Position 2 gelernt werden

# Z-Algorithmus

---

- Zerlegung des Problems in **zwei Phasen**
  - **Preprocessing**: Lerne möglichst viel über die Struktur der beiden Strings
  - **Search**: Nutze das Gelernte, um
    - P weiter nach rechts verschieben zu können
    - Bei einem Vergleich von P mit Substring von T nicht an Position 1 von P starten zu müssen
- Auch das Preprocessing muss schnell sein
  - Forderung kann aufgehoben werden, wenn bei vielen Suchen P oder T gleich ist
    - Beispiel: Suche Promotersequenz in vielen Gensequenzen

# Z-Algorithmus: Preprocessing

---

- Im Folgenden:  $S$ 
  - (wird gleich aus  $P$  und  $T$  zusammengesetzt)
- Definition
  - Sei  $i > 1$ . Dann ist  $Z_i(S)$  die Länge des *längsten Substrings*  $x$  von  $S$  mit
    - $x = S[i..i+|x|-1]$  ( $x$  startet an Position  $i$  in  $S$ )
    - $S[i..i+|x|-1] = S[1..|x|-1]$  ( $x$  ist auch Präfix von  $S$ )
  - $x$  ist die *Z-Box* von  $S$  an Position  $i$  mit Länge  $Z_i(S)$



# Beispiele

---

S = aabcaabxaaz

1(a)

0

0

3(aab)

1(a)

0

0

2(aa)

1(a)

0

S = aaaaaa

S = baaaaa

# Beispiel 2

	<b>A</b>	<b>C</b>	<b>A</b>	<b>T</b>	<b>A</b>	<b>C</b>	<b>A</b>	<b>C</b>	<b>A</b>	<b>T</b>	<b>A</b>	<b>G</b>	
Z <sub>2</sub>		C	A	T	A	C	A	C	A	T	A	G	0
Z <sub>3</sub>			<b>A</b>	T	A	C	A	C	A	T	A	G	1
Z <sub>4</sub>				T	A	C	A	C	A	T	A	G	0
Z <sub>5</sub>					<b>A</b>	<b>C</b>	<b>A</b>	C	A	T	A	G	3
Z <sub>6</sub>						C	A	C	A	T	A	G	0
Z <sub>7</sub>							<b>A</b>	<b>C</b>	<b>A</b>	<b>T</b>	<b>A</b>	G	5
Z <sub>8</sub>								C	A	T	A	G	0
Z <sub>9</sub>									<b>A</b>	T	A	G	1
Z <sub>10</sub>										T	A	G	0
Z <sub>11</sub>											<b>A</b>	G	1
Z <sub>12</sub>												G	0

# Linearer Stringmatching Algorithmus

---

- Annahme: Z-Boxen lassen sich in  $O(|S|)$  berechnen
  - Wie zeigen wir später
- Verwendung der Z-Boxen für String Matching

```
S := P||`$`||T; // ($ ∉ Σ)
compute Z-Boxes for S;
for i = |P|+2 to |S|
    if (Zi(S)=|P|) then
        print i-|P|-1; // P in T at position i
    end if;
end if;
```

- Komplexität
  - Schleife wird  $|S|$ -mal durchlaufen =>  $O(m)$

# Berechnung der Z-Boxen

---

- Naiver Algorithmus braucht  $O(|S|^2)$

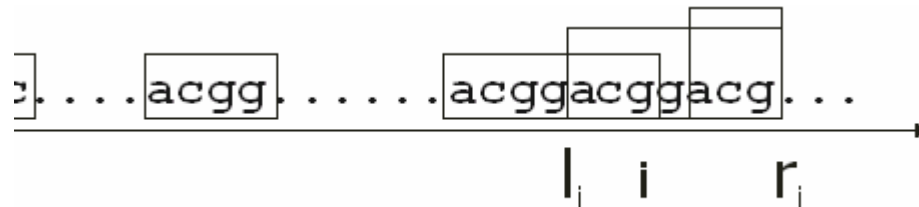
```
for i = 2 to |S|
  Zi := 0;
  j := 1;
  while ((S[j] = S[i + j - 1]) and (j <= |S|))
    Zi := Zi + 1;
    j := j + 1;
  end while;
end for;
```

- Damit wäre nichts gewonnen
  - $O((m+n)^2) + O(m) = O(m^2)$

# Vorarbeiten

---

- Definition
  - Sei  $i > 1$ . Dann ist
    - $r_i$  der *maximale Endpunkt* aller Z-Boxen, die bei oder vor  $i$  beginnen
    - $l_i$  ist die *Startposition* der längsten Z-Box, die bei  $r_i$  endet
- $l_i$  ist eindeutig, da an jeder Position nur eine Z-Box beginnt
- $S[l_i..r_i]$  ist die Z-Box, die die Position  $i$  von  $S$  enthält, am weitesten nach rechts reicht und am längsten ist



# Lineare Berechnung der $Z_i$ Werte

---

- Trick
  - Verwenden von bereits bekannten  $Z_i$  zur Berechnung von  $Z_k$  ( $k > i$ )
- Grundaufbau
  - Lineares Durchlaufen des Strings (Laufvariable  $k$ )
  - Kontinuierliches Vorhalten der aktuellen Werte  $l=l_k$  und  $r=r_k$
  - Größe der Z-Box an Position  $k$  ergibt sich mit konstantem Aufwand
- Induktive Erklärung
  - Induktionsanfang: Position  $k=2$ 
    - Berechne  $Z_2$ .
    - Wenn  $Z_2 > 0$ , setze  $r=r_2$  ( $=2+Z_2-1$ ) und  $l=l_2$  ( $=2$ ), sonst  $r=l=0$
  - Induktionsschritt: Position  $k > 2$ 
    - Bekannt sind  $r, l$  und  $\forall j < k: Z_j$

# Z-Algorithmus, Fall 1

- Möglichkeit 1:  $k > r$ 
  - D.h., dass es keine Z-Box gibt, die  $k$  enthält
  - Wir wissen nichts über den Bereich in  $S$  ab  $k$
  - Dann gehen wir primitiv vor
    - Berechne  $Z_k$  durch Zeichen-für-Zeichen Matching
    - Wenn  $Z_k > 0$ , setze  $r = r_k$  und  $l = l_k$

Beispiel

$k$   
CTCGAGTTGCAG  
0  
1  
0  
?

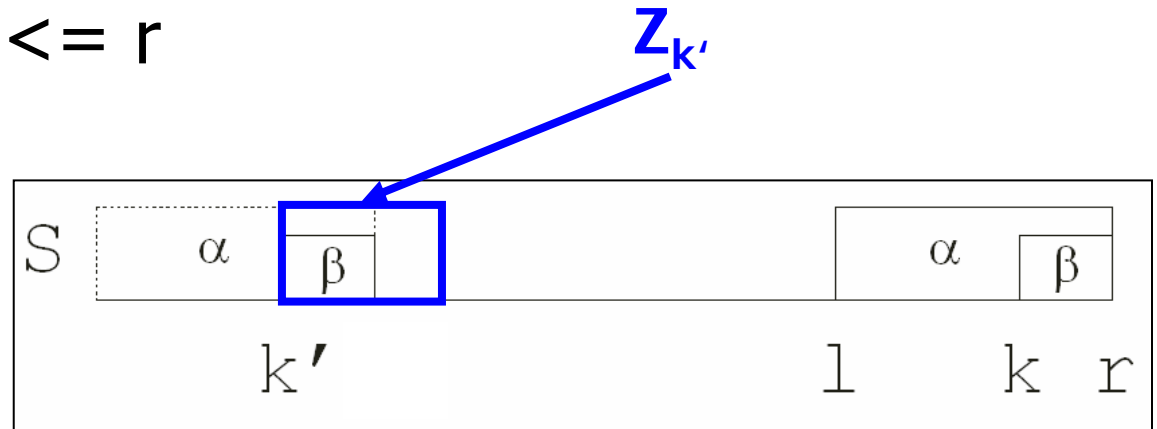
Gegenbeispiel

$lk$   $r$   
CTACTACTTTGCAG  
0  
0  
5  
?

# Z-Algorithmus, Fall 2

- Möglichkeit 2:  $k \leq r$

- Die Situation:



- Also

- Z-Box  $Z_l$  ist Präfix von  $S$
- Substring  $\beta = S[k..r]$  kommt auch an Position  $k' = k - l + 1$  von  $S$  vor
- Was wissen wir über diesen Substring? Natürlich:  $Z_{k'}$
- $Z_{k'}$  und  $Z_k$  können aber länger oder kürzer als  $|\beta| = r - k + 1$  sein
- $S[r+1..]$  kennen wir noch nicht;  $S[k'+1..]$  schon

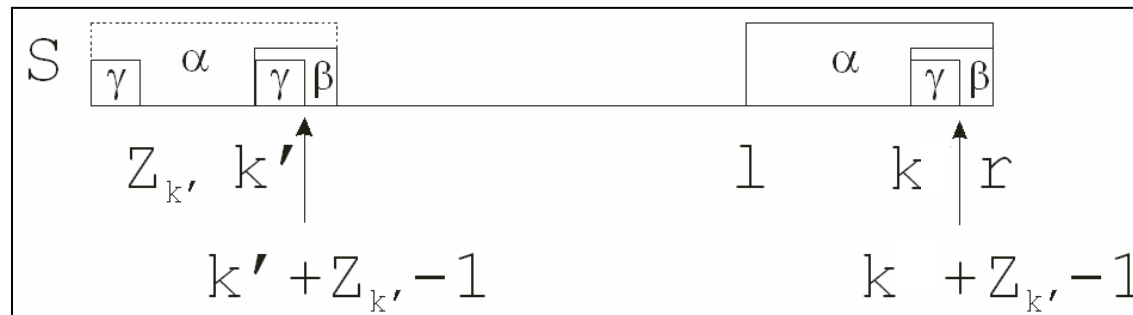
# Z-Algorithmus, Fall 2.1

- Fallunterscheidung

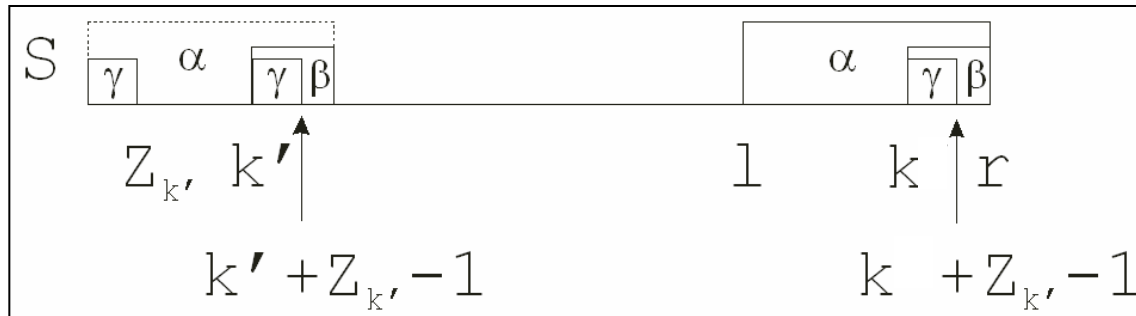
- $Z_{k'} < |\beta| = r - k + 1$

Dann ist das Zeichen an  $k' + Z_{k'}$  ein Mismatch bei der Präfixverlängerung. Dann ist das Zeichen  $S(k + Z_k)$  der gleiche Mismatch. Also:

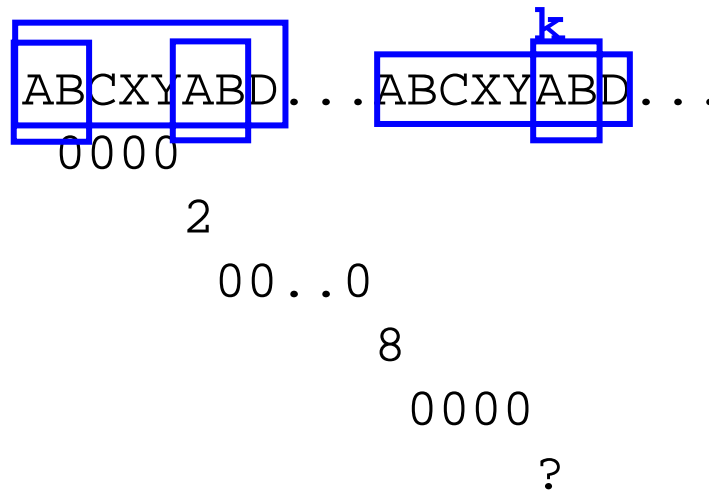
$Z_k = Z_{k'}$ ;  $r$  und  $l$  unverändert



# Beispiel



## Beispiel



$$\beta = |ABD|; k' = 6; Z_6 = 2 < |\beta|$$

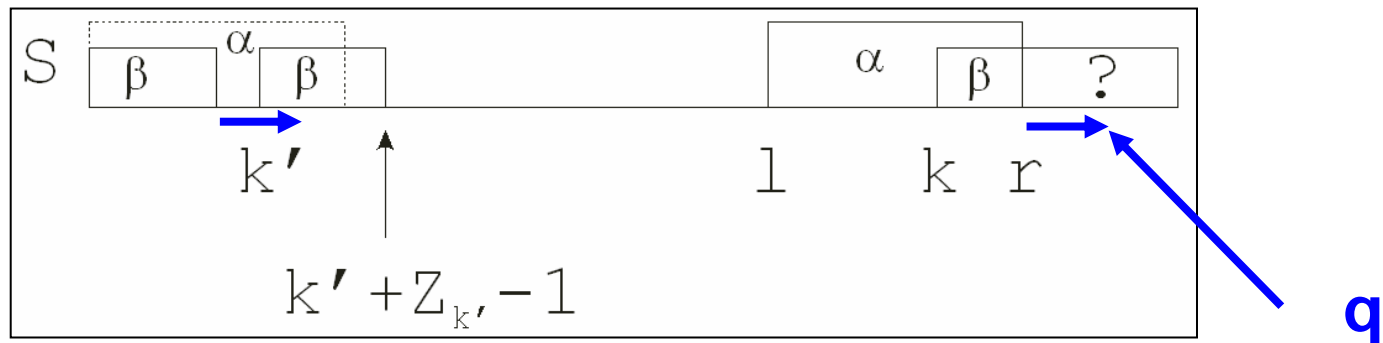
# Z-Algorithmus, Fall 2.2

- $Z_{k'} \geq |\beta|$ : Dann ist  $\beta$  ein Präfix von  $S$ 
  - ... dass sich vielleicht noch verlängern lässt
  - Wir wissen, dass  $S(|\beta|+1)=S(k'+|\beta|)$
  - Wir wissen aber nichts über  $S(r+1)$  – dieses Zeichen wurde noch nie betrachtet

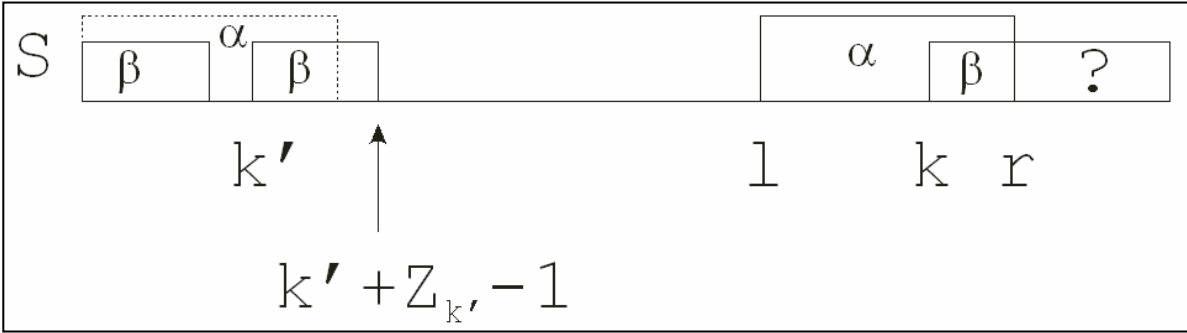
Matche Zeichen-für-Zeichen  $S[r+1..]$  mit  $S[|\beta|+1..]$

Sei der erste Mismatch an Position  $q$

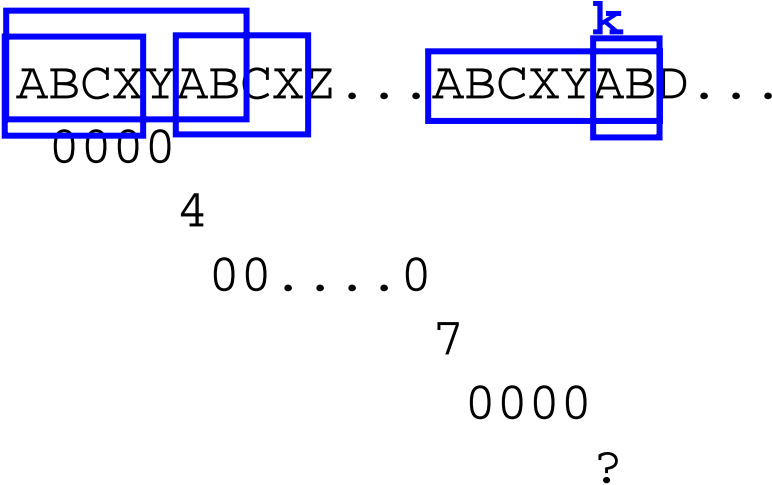
Dann:  $Z_k = q - k$ ;  $r = q - 1$ ; wenn  $q \neq r + 1$ :  $l = k$



# Beispiel



## Beispiel



$$\beta = |AB|; k' = 6; Z_6 = 4 > |\beta|$$

# Algorithmus

---

```
match Z2;
set l,r;
for k = 3 to |S|
    if k>r then
        match Zk;
        set r,l;
    else
        k' := k-1+1;
        b := r-k+1; // This is |β|
        if Zk' < b then
            Zk := Zk';
        else
            match S[r+1.. ] with S[b+1.. ] until q;
            Zk := q-k; r := q-1;
            if q!=r+1 then l := k;
        end if;
    end if;
end for;
```

# Komplexität

---

- Der Algorithmus berechnet alle  $Z_i$ -Werte in  $O(|S|)$
- Beweis
  - Idee: Abschätzung der maximalen Anzahl von Matches und Mismatches
  - Beides ist kleiner-gleich  $|S|$
  - Damit ist das Preprocessing  $O(|S|)$
  - Und damit ist der gesamte Algorithmus  $O(|S|)$

123456789012345678901  
 abxyabxz\$xabxyabxyabxz

$$k' := k-1+1; \quad b := r-k+1;$$

$$Z_k := q-k; \quad l := k; \quad r := q-1;$$

k	Bemerkung	$Z_k$	l	r
2	Induktionsanfang	0	0	0
3	$k > r$ ; Neues Matching, 1 Mismatch	0	0	0
4	$k > r$ ; Neues Matching, 1 Mismatch	0	0	0
5	$k > r$ ; Neues Matching, 3 Matches, 1 Mismatch	3	5	7
6	$6 \leq 7$ ; $k'=2$ ; $b=2$ ; $Z_2=0$ ; Also $Z_k < b$ , damit $Z_k = Z_{k'}$	0	5	7
7	$7 \leq 7$ ; $k'=3$ ; $b=1$ ; $Z_3=0$ ; Also $Z_k < b$ , damit $Z_k = Z_{k'}$	0	5	7
8	$8 > 7$ ; Neues Matching, 1 Mismatch	0	5	7
9	$9 > 7$ ; Neues Matching, 1 Mismatch	0	5	7
10	$10 > 7$ ; Neues Matching, 1 Mismatch	0	5	7
11	$11 > 7$ ; Neues Matching, 7 Matches, 1 Mismatch	7	11	17
12	$12 \leq 17$ ; $k'=2$ ; $b=6$ ; $Z_2=0$ ; $Z_k < b$ , damit $Z_k = Z_{k'}$	0	11	17
13	$13 \leq 17$ ; $k'=3$ ; $b=5$ ; $Z_3=0$ ; $Z_k < b$ , damit $Z_k = Z_{k'}$	0	11	17
14	$14 \leq 17$ ; $k'=4$ ; $b=4$ ; $Z_4=0$ ; $Z_k < b$ , damit $Z_k = Z_{k'}$	0	11	17
15	$15 \leq 17$ ; $k'=5$ ; $b=3$ ; $Z_5=3$ ; Also $Z_k \geq b$ ; matche $S[18..]$ mit $S[4..]$ ; 5 Matches und Erfolg			

# Fazit

---

- Z-Box Algorithmus
  - Findet alle Vorkommen von P in T
  - Berechnung Z Werte für P\$T in linearer Zeit
  - Danach alle Vorkommen in linearer Zeit
  - Komplexität  $O(m+n)$
- Als Worst-Case ist das bereits optimal
- Aber
  - Boyer-Moore: Average Case sublinear
  - Knuth-Morris-Pratt: Elegant erweiterbar zu vielen P