

Bioinformatik

Zeichenketten und Stringalgorithmen

Ulf Leser

Wissensmanagement in der
Bioinformatik

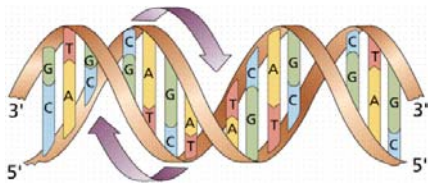


Inhalt dieser Vorlesung

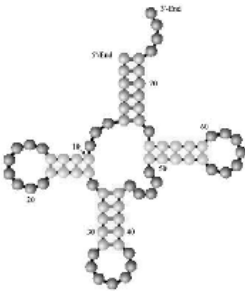
- Motivation
- Strings und Matching
- Naiver Algorithmus
- Z-Box Algorithmus

Biomoleküle

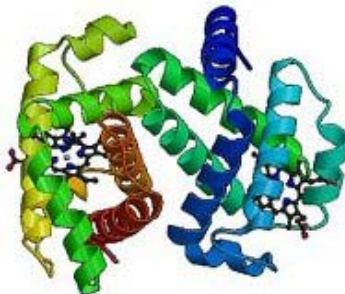
- DNA, RNA und Proteine lassen sich als Zeichenkette über festem Alphabet darstellen



DNA
A C G T



RNA
A C G U



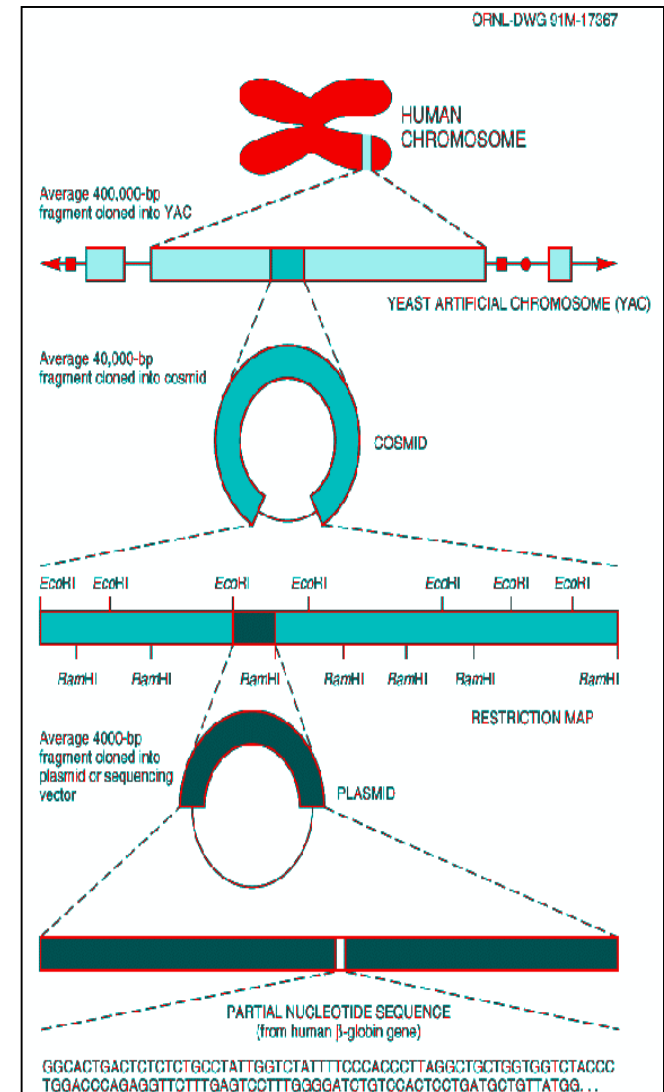
Protein
A C D E F G H I K L M N P Q R S T V W Y

Teil 1: Drei Anwendungen von Stringalgorithmen in der Bioinformatik

- Sequenzierung
 - Shotgun und Whole Genome Shotgun
 - Assembly von Teilssequenzen
- cDNA Clustering
 - All-against-all Vergleiche
 - Hohe Fehlerraten
- Funktionale Annotation
 - Finden homologer Sequenzen
 - Schnelle Suche in Sequenzdatenbanken

1.1 Stringvergleiche beim Sequenzieren

- Sequenzierungsverfahren bekannt?
- **Shotgun Sequenzierung**
 - Gegeben Clone der Länge X
 - Zerbrechen des Clones in Teilstücke von 500 – 1.500 Bp
 - Sequenzieren jedes Teilstückes (Read)
 - **Assembly**: Berechnung der Clonesequenz aus den Reads
 - **Redundanz**: Cosmid (30.000 Bp) mit 600 Reads a 500 Bp = 10-fache Überdeckung



Sequenzierung

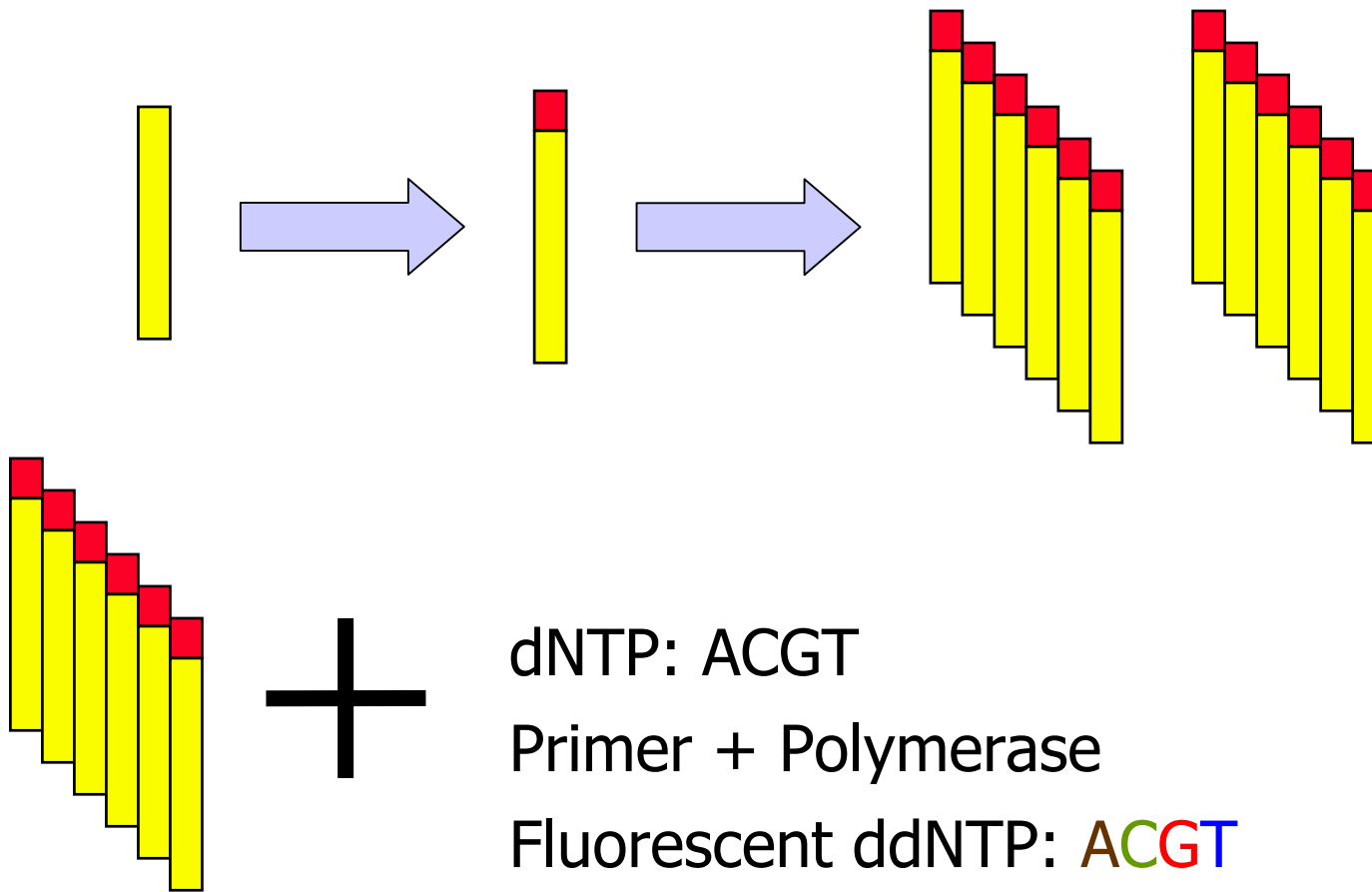
- Gegeben: Clone unbekannter Sequenz
- Gesucht: Sequenz

- Unmöglich: Ansehen, Messen, Mikroskop, etc.
- Verfahren von Sanger, 1972: „Radioactive Dideoxy Sequencing“

Sequenzierung nach Sanger

- Zwei Voraussetzungen
- Polymerase
 - Enzym
 - Bindet an spezifischen Primer
 - Verlängert einsträngige DNA entlang Template
- Deoxy versus Dideoxy Nucleotide
 - DNA besteht aus Deoxy Nucleotiden (dNTP)
 - Einbau von Dideoxy Nucleotiden (ddNTP) möglich
 - ddNTP stoppt Polymerase

Schritt 1 und 2



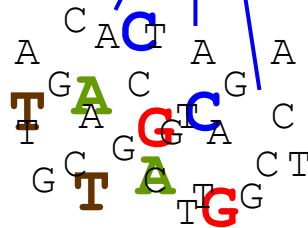
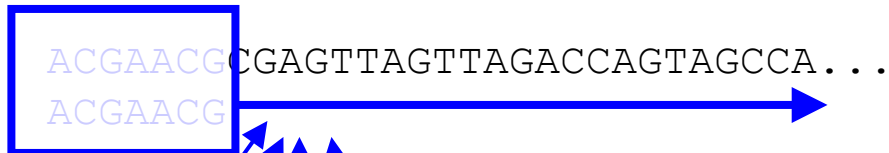
Schritt 3

Primer

Template

ACGAACGCGAGTTAGTTAGACCAGTAGCCA...

Polymerase



ACGAACGCGAGTTAGTTAGACCAGTAGCCA...

ACGAACGCGAGTT**A**

ACGAACGCGAGTTAGTTAGACCAGTAGCCA...

ACGAACGCGA**G**

ACGAACGCGAGTTAGTTAGACCAGTAGCCA...

ACGAACGCGAGTTAGT**T**

ACGAACGCGAGTTAGTTAGACCAGTAGCCA...

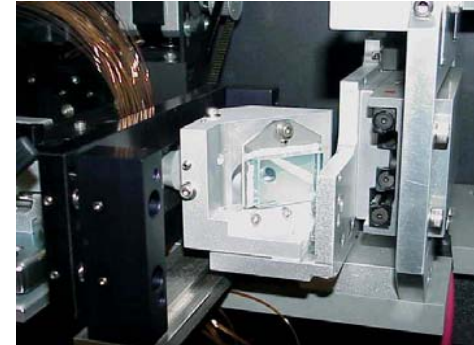
ACGAACGCGAGTTAGTTAG**T**

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**



Heute

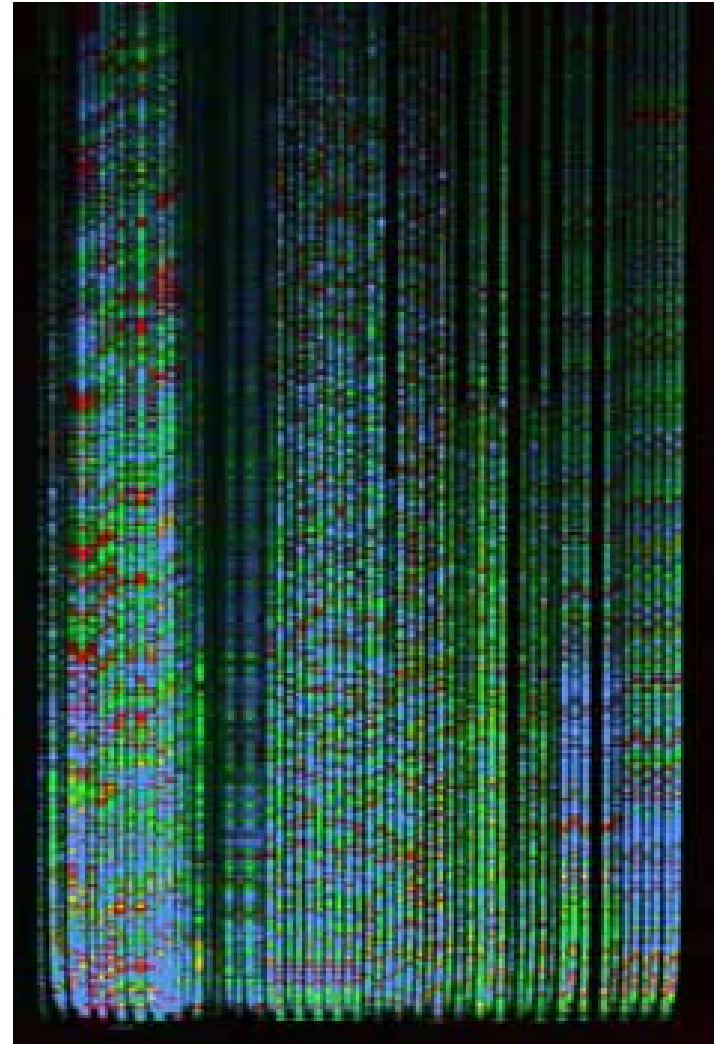
- Fluoreszente Markierung
- Hochdurchsatz
- Billig



Quelle: <http://www.geneticsplace.com>

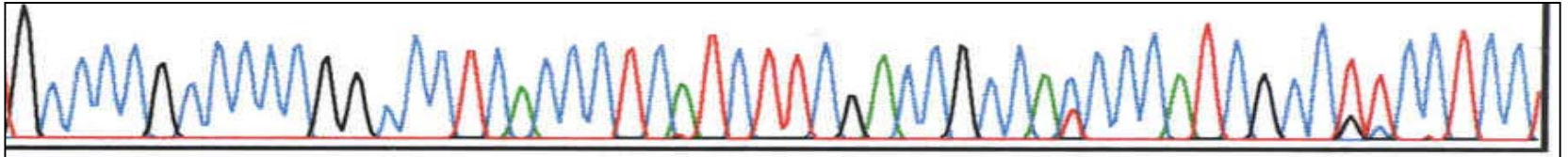
Ergebnis (roh)

- Heutige Geräte
 - > 36 Lanes parallel
 - Kapillarelektrophorese statt Gel
 - Direktes Laden von 96 Well Plates
- Sanger
 - Radioaktive Markierung
 - 4 Mischungen (A,G,T,P)
 - 4 Gel - Lanes



Ergebnis (Zwischenprodukt)

- Signalverarbeitung (Rauschen, ...)



- Übersetzung in Traces
 - 4 Arrays, jedes für eine Farbe
 - Intensitätswerte in regelmäßigen Zeitabschnitten
- Theoretisch
 - Peaks entdecken
 - Immer nur eine Farbe
 - Sequenz zuordnen

Vom Tracefile zur Sequenz

- Tracefiles sind Rohdaten der Sequenzierung
- Verschiedene Verfahren / Tools, um aus Tracefiles Sequenzen zu berechnen
- Komplexe Probleme
 - Base Calling
 - Assembly
 - Finishing

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

Greedy ?

```
accgtaaagcaaagatta
  aagattattgaaccgtt
    aaagcaaagattattg
      attattgccagta

accgtaaagcaaagatta
aagattattgaaccgtt
  aaagcaaagattattg
    attattgccagta
```

Fehler ?

```
tggacaagcaaagattga
  acattggtgaac
    gcaaagattgctg

tggacaagcaaagattga
acattggtgaac
  gcaaagattgctg
```

Assembly – 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)
- NP-vollständig

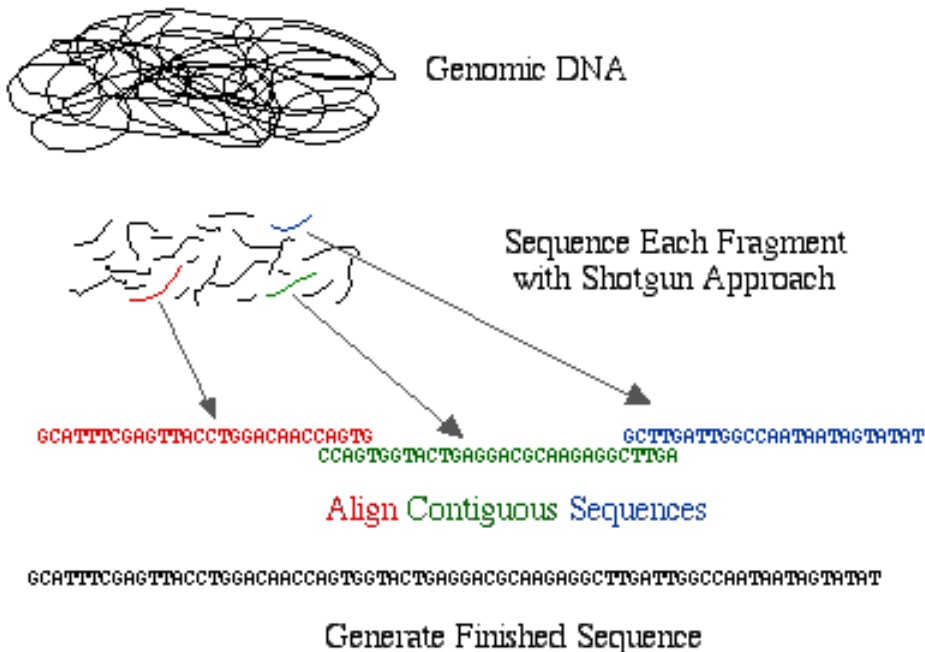
- **Assembly: Verschärfungen von SUPERSTRING**

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

- **Heuristische Verfahren**

Problemdimension: Whole Genome Shotgun

Whole Genome Shotgun Sequencing Method



- Zerbrechen von **kompletten Genomen** in Stücke 1KB-100KB
- Alle Stücke (an-)sequenzieren
- Celera:
 - Homo sap.: Genom: 3 GB, 28.000.000 Reads
 - Drosophila: Genom: 120 MB, 3.200.000 Reads
- Schnelle Algorithmen notwendig!

1.3 Stringvergleiche zur funktionalen Annotation

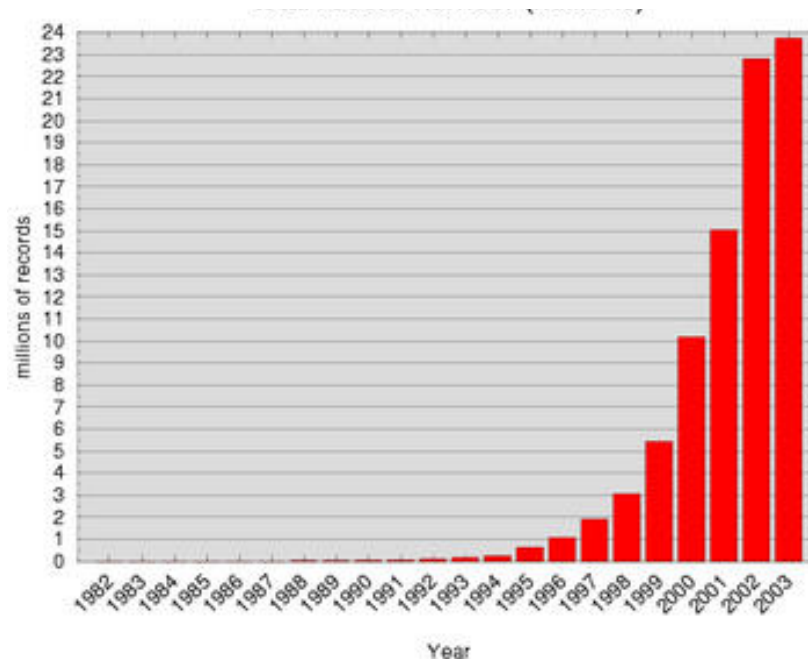
- Sequenzen bestimmen Funktionen
 - Gleiche Gensequenzen \sim gleiche Proteinsequenzen (Dif. Spl.!)
 - Gleiche Proteinsequenzen \sim gleich Eigenschaften (Struktur, Form, Ladung, ...) (Modifikationen!)
- Grundannahme der Bioinformatik
 - Gleiche Sequenzen – gleiche Funktion
 - Sehr ähnliche Sequenzen – sehr ähnliche/gleiche Funktion
 - Etwas ähnliche Sequenzen – verwandte Funktion?
- Stimmt oft ... aber nicht immer
 - Viele Gegenbeispiele bekannt
 - Naive Übertragung führt zu vielen Fehlern (siehe EMBL)
 - Besonders schwierig (und lohnend!) zwischen verschiedenen Spezies

Standardvorgehen

- Gegeben: Eine neue Sequenz (Genom)
- **Annotationspipeline basierend auf bereits annotiertem Genom**
 - Suche nach ähnlichen Gensequenzen
 - Suche nach ähnlichen Promotersequenzen
 - Suche nach ähnlichen Proteinen (Übersetzung- Rückübersetzung)
 - Suche nach neuen 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** (Knock-Out, Yeast2Hybrid)

Problemdimension

- BLAST gegen EMBL



Quelle: EMBL, Genome Monitoring Tables, Stand 12.2.2003

- Schnelle Stringvergleichsalgorithmen notwendig

Fazit

- Stringalgorithmen sind an sehr vielen Stellen der Bioinformatik essentiell
- Sowohl exakte als auch approximative Suche notwendig
- Wegen Größe der Datenmengen ist hohe Performance sehr wichtig

Exaktes Matching

- Gegeben: P (Pattern) und T (Text), $|P| \leq |T|$
- Gesucht: Sämtliche Vorkommen von P in T

Beispiel

Auffinden der Erkennungssequenzen von Restriktionsenzymen

Eco RV - GATATC

```
tcagcttactaattaaaaattctttctagtaagtgctaagatcaagaaaaataaattaaaaataatggaacatggcacatcttccctaaactcttcacagattgctaatga
ttattaattaaagaataaatggttataatctttttatggtaacggaatttcctaaaatattaattcaagcaccatggaatgcaaataagaaggactctgttaattgggtact
atccaactcaatgcaagtggaaactaagttgggtattaatactctttttacatatatatgtagttatcttttaggaagcgaaggacaatttcacatctgctaataaagggtact
atatttattttgtgaatataaaaaatagaaaagtatggtatcagattaaactcttttgagaaaaggtaagtagaagtaagctgtatactccagcaataagttcaaataaggc
gaaaaactcttttaataacaaaagttaaataatcattttgggaattgaaatgtcaaagataattacttcacgataagtagttgaagatagtttaaattttctctttgtatt
acttcaatgaaggtaacgcaacaagattagagtatatatggccaataaggtttgctgtaggaaaattattctaaggagatcgcgagagggcttctcaaatttattcaga
gatggatggttttagatgggtgggttaagaaaagcagattaaatccagcaaaactagaccttaggttttattaaagcagggcaataagtttaattggaattgtaaaaagatat
ctaaattctcttcatttgggtggaggaaaactagtttaacttcttaccocatgcagggccataggggtcgaatacagatctgtcactaagcaaaaggaaaatgtgagtgtagact
ttaaaccatttttattaatgacttttagagaatcatgcatttgatgttactttcttaacaatgtgaacatatttatgcgattaagatgagttatgaaaaaggcgaatata
tattcagttacatagagattatagctgggtctattcttagttataggacttttgacaagatagcttagaaaaaagattatagagcttaataaaagagaacttcttggaa
tagctgcctttgggtgcagctgtaattggctattgggtatggctccagcttactgggttaggttttaatagaaaaattccccatgattgctaattataatctatcctattgagaa
caacgtgcgaagatgagtggtgcaaatgggtcattattaactgctgggtgctatagtagttatccttagaaagatatataaatctgataaagcaaaatcctggggaaaatat
tgctaactgggtgctggtaggggtttggggattggattatctctacaagaaatttgggtggttactgatatccttataaataatagagaaaaaattaataaagatgat
```

Teil 3. Zeichenketten

- Definition

*Ein **String** S ist eine von links nach rechts angeordnete Liste von Zeichen eines Alphabets Σ*

- *$|S|$ 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*

Notation

- Wir suchen im Folgenden immer P in T
- $|T| = m \neq 0$
- $|P| = n \neq 0$
- $m \gg n$
- Alphabet Σ endlich
- P, T sind Strings über Σ
- Kosten für Vergleich zweier Zeichen aus Σ : 1

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 auf P, gehe zu 2
 - P vollständig durchlaufen \Rightarrow Merke Vorkommen von P in T
3. Verschiebe P um ein Zeichen nach rechts
4. Wenn P noch nicht über $|T|-|P|$ hinaus, gehe zu 2

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

Naiver Ansatz (cont.)

```
for i = 1 to |T| - |P| + 1
  match := true;
  j := 1;
  while ((match) and (j <= |P|))
    if (T(i + j - 1) <> P(j)) then
      match := false;
    else
      j := j + 1;
  end while;
  if (match) then
    -> OUTPUT
end for;
```

Worst-case

| | |
|----------|---------------|
| T | aaaaaaaaaaaaa |
| P | aaaaat |
| | aaaaat |
| | aaaaat |
| | aaaaat |
| | ... |

Vergleiche : $(n-1) * (m-n+1) \Rightarrow O(m*n)$

Optimierungsideen

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

1. Beobachtung: Zeichen

T xabxyabxyabxz
P abxyabxz
 abxyabxz
 abxyabxz

Nächstes a in T
an Position 6

- Substring in T muss mit a beginnen; nächstes a kommt erst an Position 6 – springe 4 Positionen
- Vorkommen von Buchstaben in T kann während Vergleich von Position 2 „gelernt“ werden

Optimierungsideen 2

2. Beobachtung: Substrings

T xabxyabxyabxz
P abxyabxz
 abxyabxz
 abxyabxz

- Nächstes abx in T an Position 6
- abx doppelt in P

- Algorithmus kann sich interne Struktur von P merken
 - $P[1..3] = P[6..8]$
 - $P[1..3]$ kommt nicht vor Position 6 wieder in P vor
- Vergleich findet: $P[1..8] = T[2..9]$
- Daher muss $P[1..3] = T[7..9]$, und zwischen 2 und 7 kann in T kein Treffer für P liegen

Teil 4. Z-Algorithmus

- Zerlegung des Matching Problem in **zwei Phasen**
 - **Preprocessing**: Lerne möglichst viel über die Struktur von T und / oder P
 - **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
- **Achtung**: Auch das Preprocessing muss schnell sein
 - Forderung kann aufgehoben werden, wenn nur T (P) voranalysiert werden muss und danach viele P (T) zum Matching kommen
 - Beispiel: Sequenzsuchserver von Genbank:
 - „Statische“ Datenbank (T), viele Anfragen (P)
 - Verwendung von Suffixbäumen

Z-Algorithmus: Preprocessing

- Definition

- Sei $i > 1$. Dann ist $Z_i(S)$ die Länge des *größten Substrings* x von S mit

- $x = S[i..i+|x|]$ (x startet an Position i in S)

- $S[i..i+|x|] = S[1..|x|]$ (x ist auch Präfix von S)

- x heißt *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

5

4

3

2

1

S = baaaaa

0

0

0

0

0

Beispiel 2

| | | | | | | | | | | | | | |
|-----------------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|---|
| | A | C | A | T | A | C | A | C | A | T | A | G | |
| Z ₂ | C | A | T | A | C | A | C | A | T | A | G | | 0 |
| Z ₃ | | A | T | A | C | A | C | A | T | A | G | | 1 |
| Z ₄ | | | T | A | C | A | C | A | T | A | G | | 0 |
| Z ₅ | | | | A | C | A | C | A | T | A | G | | 3 |
| Z ₆ | | | | | C | A | C | A | T | A | G | | 0 |
| Z ₇ | | | | | | A | C | A | T | A | G | | 5 |
| Z ₈ | | | | | | | C | A | T | A | G | | 0 |
| Z ₉ | | | | | | | | A | T | A | G | | 1 |
| Z ₁₀ | | | | | | | | | T | A | G | | 0 |
| Z ₁₁ | | | | | | | | | | A | G | | 1 |
| Z ₁₂ | | | | | | | | | | | G | | 0 |

Linearer Stringmatching Algorithmus

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

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

- Komplexität
 - Schleife wird m-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
 - $O((m+n)^2) + O(m) = O(m^2)$
- Aber es geht schneller ...

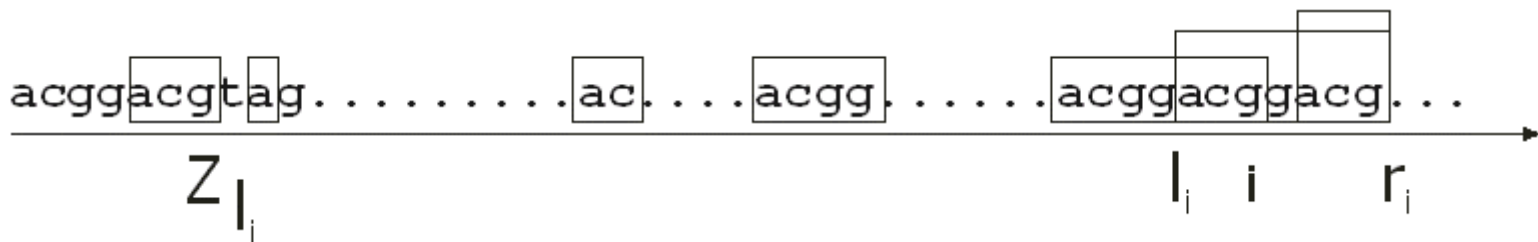
Vorarbeiten

- 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 Z-Box, die bei r_i endet

- Sprich: $S[l_i..r_i]$ ist die Z-Box, die Position i von S enthält und am weitesten nach rechts reicht



Lineare Berechnung der Z_i Werte

- Trick

- Verwenden von bereits berechneten Z_i zur Berechnung von Z_k ($k > i$)
- Lineares Durchlaufen des Strings
- Kontinuierliches Vorhalten der aktuellen Werte l und r
- Größe der Z-Box an Position i ergibt sich mit konstantem Aufwand

- Induktive Erklärung

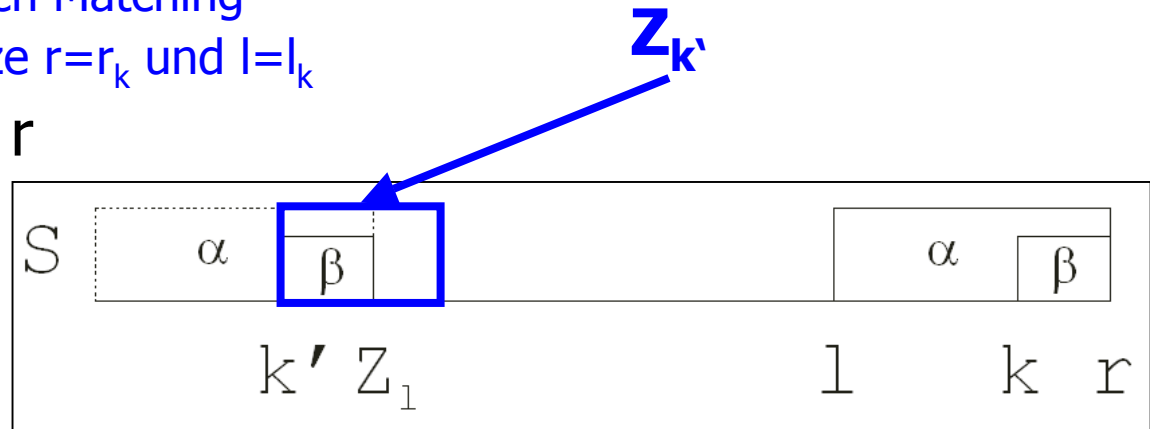
- Beginne mit $i=2$. Berechne Z_2 . Wenn $Z_2 > 0$, setze $r=r_2$ und $l=l_2$, sonst $r=l=0$
- Nun stehen wir an Position $k > 2$ und wissen r , l und alle Z_j , $j < k$

Z-Algorithmus 1

- Annahme 1: $k > r$
 - D.h., dass es keine Z-Box gibt, die k enthält
 - Dann tappen wir weiter im Dunkeln
 - Berechne Z_k durch Matching
 - Wenn $Z_k > 0$, setze $r=r_k$ und $l=l_k$

- Annahme 2: $k \leq r$

- Die Situation:



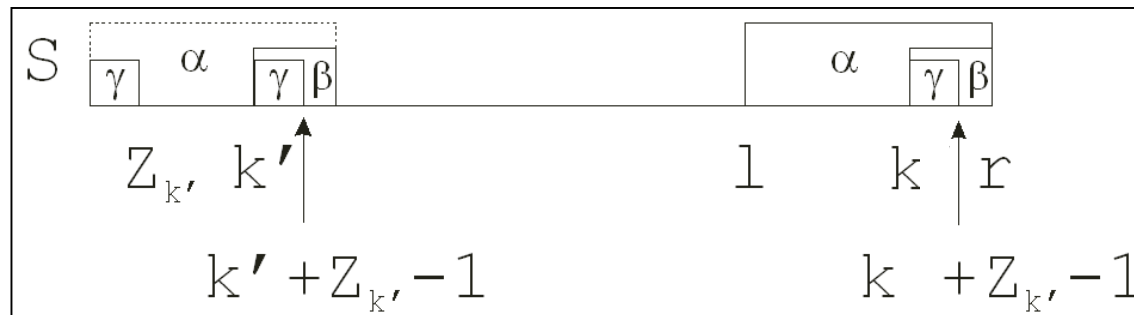
- Also

- Z-Box Z_1 ist Präfix von S
- Substring $\beta=S[k..r]$ kommt auch an Pos $k'=k-l+1$ von S vor
- Was wissen wir über diesen Substring? Natürlich: $Z_{k'}$
- D.h., dass $S[k'..]$ ist Präfix von S mit mindestens Länge $\min(Z_{k'}, |\beta|)$

Z-Algorithmus 2

- Fallunterscheidung

- $Z_{k'} < |\beta|$: 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



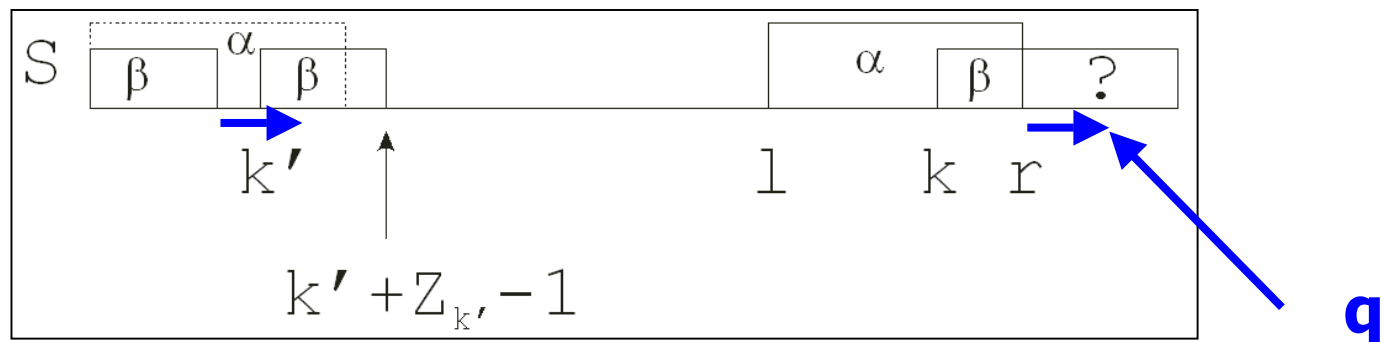
Z-Algorithmus 3

- $Z_{k'} \geq \beta$: Dann ist β ein Präfix von S
 - ... dass sich vielleicht sogar verlängern lässt
 - ... denn i.A. ist $S(k'+Z_{k'}+1) \neq S(r+1) = S(k+|\beta|+1)$
 - ... und $S(|\beta|+1)$ ist noch nicht mit $S(r+1)$ gematched worden

Matche Substring $S[r+1.. ?]$ mit $S[|\beta|+1.. ?]$

Sei der erste Mismatch an Position q

Dann: $Z_k = q - k$; $l = k$; $r = q - 1$



Wrap Up

```
compute Z2, l, r;
for k = 3 to |S|
  if k > r then
    compute 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; l := k; r := q-1;
    end if;
  end if;
end for;
```

- „Compute“: Explizites Matchen

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 $O(|S|)$

Fazit

- Z-Algorithmus
 - Findet alle Vorkommen von P in T
 - Preprocessing & Search: Berechnung Z_i Werte für P\$T
 - Führt einen Schritt für jedes der ersten $(m-n)$ Zeichen von T aus
 - Anzahl Vergleiche pro Schritt geringer als in naivem Alg. durch Ausnutzen der Struktur von P und bereits Gesehenem von T
 - Komplexität $O(m+n)$ ($=O(|S|)$)
- Ist bereits nahezu optimal
- Aber: In der Praxis sind andere Alg. schneller, speicherplatzeffizienter oder leichter erweiterbar
 - Boyer-Moore: Average Case sublinear
 - Knuth-Morris-Pratt: Erweiterung zu vielen P

123456789012345678901
 abxyabxz\$xabxyabxyabxz

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

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

| 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 | $9 > 0$; Neues Matching, 1 Mismatch | 0 | 5 | 7 |
| 10 | $10 > 0$; Neues Matching, 1 Mismatch | 0 | 5 | 7 |
| 11 | $11 > 0$; Neues Matching, 7 Matches, 1 Mismatch | 7 | 11 | 17 |
| 12 | $13 \leq 17$; $k'=2$; $b=6$; $Z_2=0$; $Z_k < b$, 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 | | | |

1234567890123456
 aaaat\$aaaaaaaaaaa

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

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

| k | Bemerkung | Z_k | l | r |
|----|--|-------|-----|-----|
| 2 | Induktionsanfang | 3 | 2 | 4 |
| 3 | $k < r$; $k'=2$; $b=2$; $Z_2=3$; $Z_k \geq b$; matche S[5..] mit S[3..]; 1 Mismatch; $q=5$ | 2 | 3 | 4 |
| 4 | $k \leq r$; $k'=3$; $b=1$; $Z_3=2$; $Z_k \geq b$; matche S[5..] mit S[3..]; 1 Mismatch; $q=5$ | 1 | 4 | 4 |
| 5 | $k > r$; Neues Matching, 1 Mismatch | 0 | 4 | 4 |
| 6 | $k > r$; Neues Matching, 1 Mismatch | 0 | 4 | 4 |
| 7 | $K > r$; Neues Matching, 4 Matches, 1 Mismatch | 4 | 7 | 10 |
| 8 | $8 \leq 10$; $k'=2$; $b=3$; $Z_2=3$; $Z_k \geq b$; matche S[11..] mit S[4..]; 1 / 1; $q=12$ | 4 | 8 | 11 |
| 9 | $9 \leq 11$; $k'=2$; $b=3$; $Z_2=3$; $Z_k \geq b$; matche S[12..] mit S[4..]; 0 / 1; $q=12$ | 4 | 8 | 11 |
| 10 | $10 > 0$; Neues Matching, 1 Mismatch | 0 | 8 | 11 |
| .. | ... | ... | ... | ... |