1 Grundfragen

1.1 Was ist ein Algorithmus?

- Lösung soll auf einem Rechner laufen
- Ergebnis soll **effizient** berechenbar sein.
- Bsp. Multiplication:
  - Eingabe: Zahlen a, b in Binärdarstellung
  - Ergebnis: $a \cdot b$
  - "Kochrezept":

$$\begin{array}{cccccccc} 1 & 0 & 1 & \cdot & 1 & 0 & 1 & 1 \\ \hline & & & & & 1 & 0 & 1 \\ & & & & 1 & 0 & 1 & \\ & & 1 & 0_1 & 1 & & & \\ \hline & & 1 & 1 & 0 & 1 & 1 & 1 \end{array}$$

- Algorithmus kann als Programm aufgeschrieben werden
- Fragen:
  - Ist das Verfahren korrekt
  - Wie schwer ist es, das Problem $a \cdot b$ zu berechnen? (Wieviele Schritte).

1.2 Warum betrachtet man Sequenzen, Strings?

- Dogma der Informationsverarbeitung in der Biologie
- 3 Bestandteile

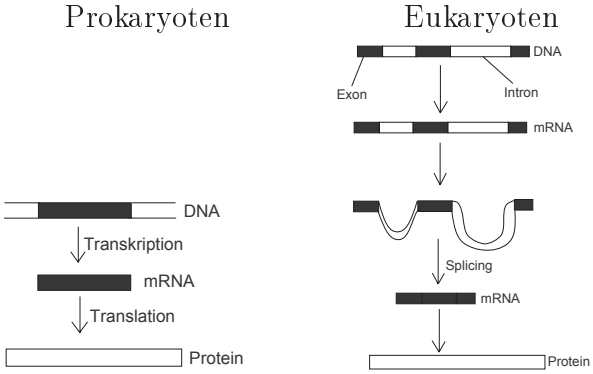
1. DNA:
   - aus Nukleotiden
   - Basen:
     - Adenin
     - Guanin
     - Cytosin
     - Thymin
     - Uracil
     - Verbindung durch Wasserstoffbrücken $\Rightarrow$ 2 elektronegative Atome und ein H-Atom $\Rightarrow$ wichtige Rolle für H_2O
     - hydrophil $\Rightarrow$ Ausbildung von Wasserstoffbrücken möglich

- nicht-kovalente Bindungen ⇒ notwendig für Informationsverarbeitung in der Biologie ⇐ passende Strukturen notwedig
- gerichtete Sequenzen: durch Aneinanderreihung von Basen (Phosphatgruppe am 3'-end)
- Doppelhelix
- Speicherung und Replikation genetischer Information

2. RNA:
   - durch Ribose verschiedene Strukturen ⇒ z.B. transfer-RNA
   - Anwendung
     * Riboenzyme
     * Übersetzung von DNA zu Protein

   ⇒ Dogma der Informationsverarbeitung
   - "The central dogma states that once 'information' has passed into a protein it cannot get out again. The transfer of information from nucleic acid, or from nucleic acid to protein, may be possible, but transfer from protein to protein, or from protein to nucleic acid, is impossible. Information here means the precise determination of sequence, either of bases in the nucleic acid or of amino acid residues in the protein." (Francis Crick 1958)

3. Proteine:
   - 20 (bzw. 21) Aminosäuren ⇒ Verbindung zu langen Ketten über Peptidbindungen
   - Übersetzung DNA ⇒ RNA ⇒ Protein



  - Codon:
    * Triplet von Nukleotiden
    * Definition einer Aminosäure, eines Start- oder Stopcodons

    ⇒ genetischer Code
  - Stopcodon = Code für 21. Aminosäure

- Welcher Bereich der DNA wird übersetzt?
- Erkennung von Introns/ Exons, bzw. "Mustern" und Teilsequenzen

⇒ Suche von Mustern in Strings (PATTERN MATCHING)

2 Einteilung der Bioinformatik

- Sequenzanalyse
  - Mustersuche, Alignmentverfahren
  - Identifizierung von Genen
  - Sequenzierung der DNA
- Strukturvorhersage
  - RNA-Sekundärstruktur
  - Proteinstruktur
- Funktionalanalyse
- Aufklärung von Stoffwechselwegen
  - Regulation
  - metabolische Pfade
  - Proteomics
- medizinische Diagnostik
  - Indentifizierung von Krankheitsgenen
  - Patientenspezifische Therapie

3 Mustersuche/Pattern Matching

Definition 3.1 (Alphabet) *Ein endliches Alphabet Σ ist eine endliche Menge von Zeichen. Mit Σ^i bezeichnet man alle Wörter der Länge i, die mit Hilfe von Zeichen aus Σ gebildet werden können. Mit Σ^* bezeichnet man die Menge der Wörter beliebiger Länge, d.h.*

$$\Sigma^* = \bigcup_{i \geq 0} \Sigma^i$$

Definition 3.2 (String) *Ein String S ist eine Folge von Zeichen $S = S_1...S_n$, wobei $n = |S|$ die Länge von S. S_i ist das Zeichen an der Position i in S. Der leere String wird mit ε bezeichnet*

Definition 3.3 (Prefix,Suffix,Substring) *Sei $S = S_1, S_2.....S_n$ ein String der Länge n. Ein String T heißt **Prefix** (bzw. **Suffix**) von S falls T=$S_1, S_2, \ldots, S_k$ (bzw. $T' = S_k, S_{k+1}, \ldots, S_n$) mit $0 \leq k \leq n+1$. Ein String T heißt **Substring** von S falls $T = S_k, \ldots, S_l$ für $1 \leq k, l \leq n$*

Bemerkung 3.4 *Der leere String ist Prefix, Suffix bzw. Teilstring eines jeden anderen Strings.*

Im folgenden werden wir folgendes Problem behandeln:

Gegeben:
- Langer Text T
- Muster (Pattern) S

Gesucht: alle Vorkommen von S in T

3.1 Naiver Algorithmus

Definition 3.5 (Match,Mismatch) *Sei P und T zwei Strings, $1 \leq i \leq |P|$ und $1 \leq j \leq |T|$ zwei Positionen in P bzw. T. Falls $P_i = T_j$, dann spricht man von einem **match** zwischen P und T an den Positionen i, j. Falls $P_i \neq T_j$, so heisst das **mismatch***

Grundidee Lege P unter den Anfang T, vergleiche bis zum 1. Mismatch, oder bis P zu Ende ist ($\Rightarrow$ dann gefunden).

Im nächsten Schritt wird P um eine Position unter T weitergeschoben, mit denselben Vergleichen ("Fenster"). Dies ist im Programm 3.1 aufgezeichnet

Beschreibung von Algorithmen: sogenannter Pseudocode, der beschreibt, wie die einzelnen Rechenschritte ablaufen sollen. Verwendete Konstrukte:

- Variablen: $I, J, S, T \ldots$
- mathematische Ausdrücke $I + 1$, T_I, ...
- Wertzuweisung: verschieden von Gleichungen

$$\underset{\text{Variable}}{I} = \underset{\text{mathematische Ausdruck}}{I + 1}$$

Zuerst wird der Ausdruck ausgewertet (mit aktuellen Wert von I), dann der Wert als neuer Wert von I gesetzt.

Programmkonstrukte Steuer den Ablauf der Berechnung

- Schleife:
```
FOR I = 1 TO N DO
     PRINT I
ENDFOR
```

- Abfrage:
```
IF  <Bedingung> THEN
     <falls Bedingung erfüllt>
ELSE
     <falls Bedingung NICHT erfüllt>
ENDIF
```

- goto-Anweisung: Springe an eine Stelle (gefährlich!)
```
schleife:
     PRINT ich drucke immer dasselbe
     GOTO schleife
```

Im folgenden Beispiel bedeutet:
$*$ Mismatch
$\hat{}$ Match
T Suchtext
P Pattern

Beispiel 3.6 Bespiellauf

Programm 3.1 Der naive Vergleichsalgorithmus

```
FOR I = 1 TO |T| - |P| + 1 DO
   // Vergleiche der Reihe nach
   // T_i,T_{I+1} ... T_{I+|P|-1} mit P_1, P_2 ... P_{|P|}
   //
   J = 1    // (Laufvariable für Vergleich)
   schleife:
      IF J > |P| THEN
         PRINT P an Position I gefunden
      ELSE
         IF (T_{I+J-1} = P_J) THEN
            J = J + 1
            GOTO schleife
         ENDIF
      ENDIF
ENDFOR
```

Positionen:	1	2	3	4	5	6	7	8	9	10	11	12	13
T:	x	a	b	x	y	a	b	x	y	a	b	x	z
P:	a	b	x	y	a	b	x	z					

Vergleich an folgenden Positionen

```
a b x y a b x z
*
  a b x y a b x z
  ^ ^ ^ ^ ^ ^ ^ *
    a b x y a b x z
    *
      a b ...
      *
        ⋮
```

◀

Problem Sehr viele überflüssige Vergleiche

Beispiel Aus Wissen über T nach Schritt 2 (nach 9 Vergleichen):

↪ Buchstaben 1-7 von P „matchen“ Buchstaben 2-8 von T

$\hookrightarrow$ 1.Buchstabe von P (nämlich a) kommt bis Position 5 nicht mehr in P vor.

$\Rightarrow$ Der erste Buchstabe von P(nämlich a) kommt in T bis Position 6 nicht mehr vor.

Verbesserung aufgrund des Wissens über P

Positionen:	1	2	3	4	5	6	7	8	9	10	11	12	13
T:	x	a	b	x	y	a	b	x	y	a	b	x	z
P:	a	b	x	y	a	b	x	z					

Vergleich an folgenden Positionen

	a	b	x	y	a	b	x	z					
	*												
		a	b	x	y	a	b	x	z				
		^	^	^	^	^	^	^	*				
						a	b	x	y	a	b	x	z
						^	^	^	^	^	^	^	^

weitere Verbesserung:

- $P_1P_2P_3$ ist gleich mit $P_5P_6P_7$
- daher: die Vergleiche, an denen $P_1P_2P_3$ unter $P_5P_6P_7$ liegt, können eingespart werden.

Positionen:	1	2	3	4	5	6	7	8	9	10	11	12	13
T:	x	a	b	x	y	a	b	x	y	a	b	x	z
P:	a	b	x	y	a	b	x	z					

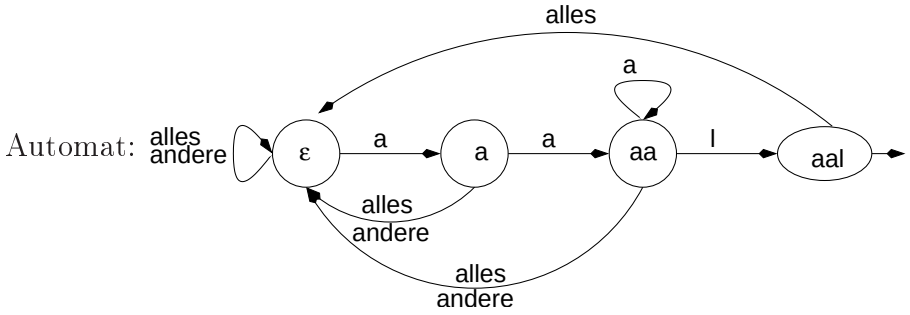
Vergleich an folgenden Positionen

	a	b	x	y	a	b	x	z					
	*												
		a	b	x	y	a	b	x	z				
		^	^	^	^	^	^	^	*				
						a	b	x	y	a	b	x	z
						□	□	□	^	^	^	^	^

Dafür: Preprocessing, das angibt, wo eine Prefix von P in P selbst wieder vorkommt $\Rightarrow$ Fundamentales Preprocessing bzw. Z-Algorithmus

Frage Wie kann man das in einen Algorithmus packen?

Beispiel Suchtext T =aaaaaalglatt, Suchword P =aal.



Zustände: Präfixe des Patterns P

Kanten: gelesene Buchstaben des Suchtextes

3.2 Fundamentales Preprocessing bzw. Z-Algorithmus

Definition 3.7 *Sei S ein String der Länge n, und $2 \leq i \leq n$ eine Position in S. $Z_i(s)$ ist die Länge des längsten Teilstrings von S, der an Position i beginnt, und ein Prefix von S ist.*

Beispiel 3.8

	1	2	3	4	5	6	7	8	9	10	11
S=	a	a	b	c	a	a	b	x	a	a	z

/	Z_2	Z_3	Z_4	Z_5	Z_6	Z_7	Z_8	Z_9	Z_{10}	Z_{11}
/	1	0	0	3	1	0	0	2	1	0

◀

- Frage: Was hilft es, Z_2(S)...Z_n(S) zu kennen?
- Behauptung: Falls man Z_2(S)...Z_n(S) effzient berechnen kann(für jedes S), dann kann man damit sehr einfach das Patternmatching Problem lösen.
- Idee:
  - S=P$T
  - ($ kommt in T,P nicht vor)
  - Berechne $Z_2(S), ...Z_n(S)$

– P in T gefunden falls $Z_i(S) = |P|$

⇒ Einfachster linearer Algorithmus (Programm 3.2), Gegeben: P(Pattern), T(Suchtext),

Programm 3.2 Einfachster lineare Algorithmus mit Z-Preprocessing

```
1 S = S = P$T
2 Berechne Z_i(S)
3 FOR I = 1 TO |T| − |P| + 1 DO
4       IF Z_{i+1+|P|}(S) = |P| THEN
5          PRINT S an Position I gefunden
6       ENDIF
7 ENDFOR
```

- Warum ist dieser Algorithmus korrekt?

i=	1	2	3	4	5	6	7	8	9	10	11				
T=	a	a	b	c	a	a	b	x	a	a	z				
P=	a	a	b												
S=	a	a	b	$	a	a	b	c	a	a	b	x	a	a	z
Z_i=	/	1	0	0	3	1	0	0	3	1	0	0	2	1	0

- Funktion des Algorithmus:
- FOR-Schleife:($|P| = 3$)
- i=1 ⇒ betrachte $Z_5(S)$ Da $Z_5(S) = 3 = |P|$ ⇒ OUTPUT P gefunden an 1
- i=2 ⇒ betrachte $Z_6(S)$ ⇒ nicht gefunden
- dasselbe, i=3, i=4, i=5 ⇒ $Z_9(S) = 3 = |P|$ ⇒ gefunden an i=5
- Begründung für Korrektheit des Algorithmus:

  1. Da $ weder in T noch in P vorkommt
     ⇒ $\forall_i : Z_i(P\$T) \leq |P|$
  2. Sei i mit $Z_{i+|P|+1}$(P$T)=$|P|$

$\Rightarrow$ P kommt in P\$T an Position $i + |P| + 1$ vor

$\Rightarrow$ P kommt in T an Position i vor

3. umgekehrt: Falls P an Position i in T vorkommt
   $\Rightarrow Z_{i+|P|+1}$(P\$T)$\geq |P|$ mit 1, $\Rightarrow Z_{i+|P|+1}$ (P\$T)=$|P|$

3.3 Was heißt eigentlich effizient?

Zur Abschätzung der Laufzeit (= Anzahl der Schritte, die benötigt wird, um ein Problem zu bearbeiten) wird die big-O Notation eingeführt.

Bemerkung 3.9 *linearer Algorithmus A: Die Zeit zum Lösen des Problems mit A wächst linear mit der Länge der Eingabe $|T| = n$.*

Definition 3.10 (Laufzeit) *Sei A ein Algorithmus. Mit $Schritte_A$:$\mathbb{N} \to \mathbb{N}$ bezeichnen wir die Funktion, die angibt, wieviele Schritte A benötigt um eine Eingabe der Länge zu bearbeiten.*

Beispiel 3.11 Wir möchten wissen, wieviele Schritte einfache Suchalgorithmus A (Programm 3.2) bei fester Patternlänge und variabler Suchtextlänge benötigt. Zum Beispiel bei einer Eingabe zweier Strings der Längen $|T| = n$ und $|P| = 5$ erhält man folgende Anzahl von Schritten:

- für die Berechnung der Z_i (Zeile 2): $(n + 5 + 1)$
- für die FOR-Schleife, um die Vorkommen zu finden (Zeile 3): n

Insgesamt erhält man $Schritte_A(n) = 2n + 7$. ◀

Definition 3.12 (Komplexität) *Algorithmus A hat Komplexität O(g) mit g:$\mathbb{N} \to \mathbb{N}$ falls $\exists n_o, c$ Konstanten mit $\forall n \geq n_0$: $Schritte_A(n) \leq cg(n)$*

- Beispiel für O(n):
  - $Schritte_A$(n) = 2n + 7
  - Ist die Komplexität von A O(n)?
  - $Schritte_A$(n) $\leq c \cdot$n

  Bsp.: c = 5; $n_0 > 2$
  $Schritte_A(4) = 2 \cdot 4 + 7 = 15 < 20$

  - im Prinzip linear
  - $\frac{Schritte_A(2n)}{Schritte_A(n)} = \frac{47}{27} = 1{,}9...$
  - $\frac{Schritte_A(3n)}{Schritte_A(n)} = \frac{67}{27} = 2{,}9...$

- Beispiele für O(n^2):
  - $Schritte_A(n) = 5n + 100000 \equiv 0n^2 + 5n + 100000$
  - $Schritte_A(n) = 7n^2 + 100000n + 10^{43}$

3.4 Analyse des trivialen Algorithmus

Der naive Suchalgorithmus zeigt, dass die Laufzeit nicht immer nur von der Längen der Eingabe abhängt, sondern auch von dem genauen Input. Bsp: trivialer Algorithmus:

Beispiel 1: 8×3 Schritte

T = aaaaa aaaaa a

P = aaa

Beispiel 1: 8×1 Schritte

T = abcde fghij k

P = xyz

Daher: Worst-Case-Analyse

- Idee des trivialen Algorithmus
  - T: $\sigma_1\sigma_2\sigma_3...\sigma_n$
  - S: $\tau_1\tau_2\tau_3...\tau_m$
  - bei Mismatch verschiebe um eine Stelle
- Worst-Case
  - in T ist S von $\tau_1...\tau_{n-1}$ enthalten
  - $|S|$ = m; $|T|$ = n

  $\hookrightarrow$ bei jeder Verschiebung m Vergleiche

  $\Rightarrow$ Schritte(A) = (n-m+1)m

  $\Rightarrow$ Schritte(A) = nm - m^2 + m $\leq$n·m (da n $\gg$ m)

  $\Rightarrow$ Komplexität in n = max(n,m) $\Rightarrow O(n^2)$
  - falls n = 2m $\rightarrow$ Schritte(A) = $2m^2$ - m^2 + m

3.5 besserer Suchalgorithmus

Wir betrachten im folgenden Z-Boxen. Die Z-Box ist das Intervall, das alle Positionen eines nicht-leeren Prefix von S enthält. r_i ist die rechte Grenze längsten (der am weitesten nach rechts gehenden) Z-Box, die vor i beginnt. **Achtung**: Sie muss nicht über i hinausgehen. Das ist immer der Fall, falls $Z_i(S) = 0$ ist.

Definition 3.13 (Z-Box) *$\forall i \leq |S|$ mit $Z_i(S) > 0$ bezeichne die Z-Box an i das Intervall von i bis i+Z_i(S)-1*

Definition 3.14 (rechte und linke Grenze) *$\forall i(1 < i \wedge i \leq |S|)$ bezeichne r_i den größten Wert von j+z_j-1 mit $1 < j \leq i$ und $Z_j(S) > 0$. Der Wert l_i bezeichnet das zu r_i gehörende linke Ende der Z-Box*

Beispiel 3.15

S=	a	a	b	a	a	b	c	a	x	a	a	b	c	y
Z_i=	$\emptyset$	1	0	3	1	0	0	1	0	3	1	0	0	0
l_i=	$\emptyset$	2	2	4	4	4	4	8	8	10	10	10	10	10
r_i=	$\emptyset$	2	2	6	6	6	6	8	8	12	12	12	12	12

◀

3.6 Z-Algorithmus

Aufgabe Bestimmung der Werte Z_k, r_k, l_k für alle Positionen $2 \leq k \leq |S$ für gegebenen String S in $O(n)$.

Initialisierung

- bestimme Z_2 durch Vergleich von $S = S_1 ... S_{|S|}$ mit $S_2 ... S_{|S|}$
- Worst-Case: $|S|$-Schritte

Schleife

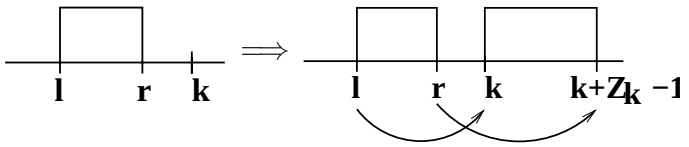
- Startet mit Position $k = 3$.
- Annahme: Werte von Z_i, r_i, l_i sind schon berechnet für alle i mit $2 \leq i \leq k-1$
- Laufvariablen und deren Bedeutung (zu Beginn der Schleife)

  k aktuelle Position im String
  r rechtes Ende der Z-Box für Position $k-1$
  l linkes Ende der Z-Box für Position $k-1$

- In der Schleife wird dann Z_k, r_k, l_k berechnet.

Für den Schleifendurchlauf betrachten wir folgende Fälle:

1.Fall: $k > r$ Das heisst, die aktuelle Position k ist nicht in vorhergehender Z-Box. Damit haben wir die folgende Situation:



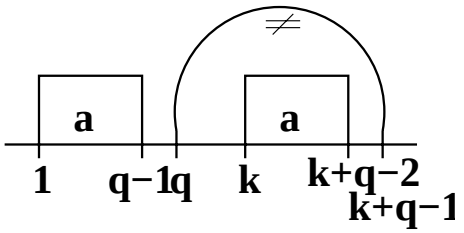
Damit muss die Z-Box an der Stelle k neu bestimmt werden:

- Bestimme Z_k durch Vergleich von $S_1 ... S_{|S|-k+1}$ mit $S_k ... S_{|S|}$
- IF ($Z_k > 0$) THEN r = k + Z_k - 1, l = k
- l_k = l, r_k = r
- Die Neuberechnung von Z_k,l_k,r_k geht also folgendermaßen von statten.

  Vergleiche $S_1 ... S_{|S|}$ mit $S_k ... S_{|S|}$ bis zum ersten Mismatch.

Ist der 1. Mismatch bei

$$\begin{array}{ccccc} S_1 & . & . & . & S_q \\ \| & \| & \| & \| & \nparallel \\ S_k & . & . & . & S_{q+k-1} \end{array}$$



- $\Rightarrow Z_k = q-1; r_k = k+q-2; l_k = k$
- Konkretes Beispiel:

  S=xabxacde

  k=1

  Vergleich:$S_1...S_{|S|}$=xabxacde

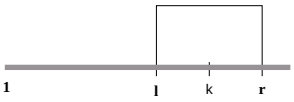
	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
	x	a	b	x	a	c	d	e
mit:$S_4...S_{|S|}$=xacde	$\|$	$\|$	$\nparallel$					
	x	a	c	d	e			
	S_4	S_5	S_6	S_7	S_8	q=3		

  Z_4= Länge von xa = q-1=2

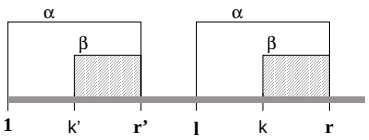
  r_4=4+3-2=5

  l_4=4

2.Fall: $k \leq r$ In diesem Fall kann man sehr viel über Z_k, l_k, r_k aus den Werten r, l vorberechnen und dadurch Rechenzeit einsparen. Wir sind in folgender Situation:



Da k innerhalb einer Z-box ist, gilt $S_1...S_{r-l+1} = S_l...S_r$. Den Teilstring $S_1...S_{r-l+1}$ nennen wir α. Da k innerhalb von α vorkommt α sowohl an Position 1 als auch an Position k vorkommt, wissen wir, dass auch der Teilstring $S_k \ldots S_r = \beta$ am Anfang vorkommen muss:



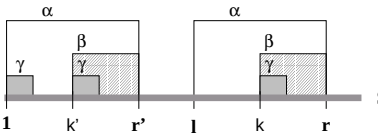
Die Werte k' und r' lassen sich einfach wie folgt berechnen:

- $k' = k - (l - 1)$
- $r' = r - l + 1$

Für die Berechnung von Z_k hängt es nun ab, ob bzw. wie lange β selbst ein Prefix von S ist. Diese Länge ist mit $Z_{k'}$ schon vorberechnet (da $k' < k$). Wir wissen auf alle Fälle schon dass $Z_k \geq Z_{k'}$ ist.

Wir unterscheiden folgende Unterfälle:

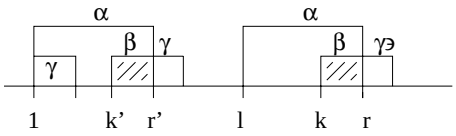
$Z'_k <| \beta | = r - k + 1 \Rightarrow Z_k \geq | \beta | = r - k + 1$. Wir habe folgende Situation:



Setze

- $Z_k = Z'_k$
- $r_k = r$
- $l_k = l$

$Z'_k \geq | \beta | = r - k + 1$. Folgende Situation:



$\Rightarrow$ Neue Z-Box muss berechnet werden, da im allgemeinen γ nicht gleich γ' sein wird.

- Berechne Z_k durch Vergleiche von $S_{r+1}....S_{|S|}$ mit $S_{r-k}...S_{|S|}$
- Sei q der erste Mismatch bei diesem Vergleich, und wir setzen

$$Z_k = r - k + q.$$

Zur Berechnng der neuen Z-Box müssen wir noch unterscheiden, ob $q = 1$ oder $q > 1$ ist. Falls $q = 1$, so ist die neu berechnete Z-Box völlig in der alten enthalten, und wir setzen

$$l_k = l$$
$$r_k = r$$

Ansonsten haben eine neue Z-Box, und wir setzten

$$r = r_k = k + Z_k - 1 l = l_k \qquad = k$$

Alternative Beschreibung:

$\rightarrow$ Vergleiche der Reihe nach

$$\begin{array}{cccccc} & S_{|\beta|+1} & S_{|\beta|+2} & \ldots & S_{|\beta|+q} \\ & \| & \| & \ldots & \nparallel \\ \text{mit} & S_{r+1} & S_{r+2} & \ldots & S_{r+q} \end{array}$$

1. Mismatch bei $S_{|\beta|+q}$ mit S_{r+q}

$Z_k =\mid \beta \mid +q-1 = r-(k-1)+q-1$

$\Rightarrow$ Wissen

$r_k = r + q - 1$

$r = r_k$

$l_k = k$

$l = l_k$

Frage: Warum ist Z-Algorithmus O($|S|$)?

Analyse: 1. Iteration. Welche Werte nimmt k an?

k=1 k=2 ... k=$|S|$

$\Rightarrow$ O($|S|$) viele Iterationen

Für jedes k bei weiteren Schritten folgen Matches oder Mismatches

Mismatches für ein einzelnes k

Fall 1 k>r

$\Rightarrow$ 1 Mismatch

Fall 2a: keine Vergleiche

$\Rightarrow$ 0 Mismatch

Fall 2b: 1 Mismatch

$\Rightarrow$ O($|S|$)

- Noch zu zählen wieviele Matches q_k beim k-ten Schritt.
- Es gilt: $r_k \geq r_{k-1}$

Fall 1 q_k-Matches. $k \geq r_{k-1}$, deshalb $r_k = k + q_k \geq r_{k-1} + q_k$

Fall 2 q_k-Matches. $\Rightarrow r_k \geq r_{k-1} + q_k$

$r_{|S|} \leq |S|$

Anzahl matches in der k-ten Iteration

$$r_k \geq r_2 + \sum_{3 \leq i \leq k} q_k$$

$\Rightarrow q_3 + ... + q_k + ...q_{|S|} \leq |S|$

$r_1 \quad r_2 \quad r_3 \quad r_{|S|} = |S|$

$\vee| \quad \vee| \quad \vee|$

$r_1 + q_1 + ...$

$\Rightarrow$ kein Match wird zweimal gemacht

$\Rightarrow$ viele Matches

$\Rightarrow$ Anzahl der Schritte $O(|S|) + O(|S|) + O(|S|) = O(|S|)$

- Beispiel für den Z-Algorithmus
- Situation:
  - Berechnung von z_k
  - $z_{k'}, r_{k'}, l_{k'}$ berechnet für k' < k
  - aktuelle z-Box enthält k $\rightarrow$ l < k < r
  - Fallunterscheidung nach $|\gamma| = Z_{k'}$

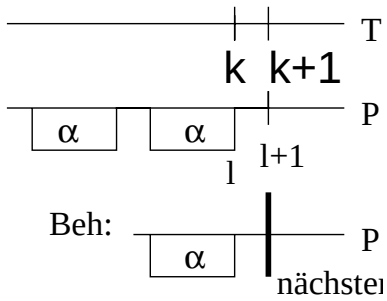
1. Fall: $Z_{k'} = |\gamma| < |\beta| \Rightarrow Z_k = Z_{k'}$
2. Fall: $Z_{k'} = |\gamma| \geq |\beta| \Rightarrow Z_k$ muß nach β neu berechnet werden
   $\rightarrow$ Neuberechnung von $Z_k, r_k, l_k (r_k, l_k \equiv$ Z-Box)

3.7 Knuth-Morris-Pratt-Algorithmus

Z-Algorithmus ist etwas kompliziert, aber häufig eine Unterprozedur einfacher Verfahren. Der naive Algorithmus wäre einfacher zu verstehen, ist aber komplizierter.

Ein Beispiel, wie man die Idee des naiven Algorithmus weiterentwickeln kann, ist der Knuth-Morris-Pratt Algorithmus. Die Grundidee des naiven Algorithmus ist ja, immer das Suchpattern P um eine Position weiterzubewegen ("shiften"). Der Knuth-Morris-Pratt (KMP) Algorithmus macht dieses "shiften" intelligenter und wird dadurch linear. Er wurde 1977 eingeführt und ist der bekanntester linearer Pattern-Matching-Algorithmus ($O(n)$). Er vewendet den Z-Algorithmus als Preprocessing. Es ist die Erweiterung auf multiple Pattern möglich. Ausserdem erlaubt er, den Suchtext aktuell (real-time) Buchstabe für Buchstabe der Reihe nach einzugeben. Dies ist mit dem Z-Algorithmus nicht möglich.

Die Grundidee des Algorithmus ist die Modifikation der Shiftregel bei Missmatch:



α der Teil des Strings, der am Ende der Übereinstimmung liegt und selbst wieder ein Prefix ist. Damit kann ich das Suchpattern P soweit verschieben, bis α als Suffix des gemachten Teils mit dem α als Prefix übereinstimmt. Wir benötigen zunächst eine Definition.

Definition 3.16 (proper suffix) *Für jede Position im Muster P sei $sp_i(P)$ (eng: proper suffix) die Länge des längsten echten Suffixes von $P_1...P_i$, der mit einem Präfix von P übereinstimmt.*

$sp_i(P)$ beschreibt die Länge von α, wie es bei der Shiftregel verwendet wird. Falls P aus dem Kontext bekannt ist, schreiben wir einfach sp_i Nun können wir die Aussagen der Shift-regel präzisieren.

Theorem 3.17 *Sei T ein Suchtext, und P ein Pattern das in T gesucht wird. Sei nun $P_1 \dots P_i = T_{k-i+1} \dots T_k$ Dann gilt*

$$P_1 \dots P_{sp_i} = T_{k-sp_i+1} \dots T_k$$

Im folgenden Beweis werden wir immer die Suffixe einer bestimmten Länge eines Strings benötigen. Sei T ein String, k eine Position in dem String. Dann ist der Suffix an der Position k der Länge l gleich

$$T_{k-l+1} \dots T_k$$

Beweis. Aufgrund der Definition von sp_i gilt

$$P_1 \dots P_{sp_i} = P_{i-sp_i+1} \dots P_i. \tag{1}$$

Ebenso gilt aufgrund der Vorraussetzung daß $P_1 \dots P_i = T_{k-i+1} \dots T_k$. Nachdem $sp_i \leq i$, gilt damit auch

$$P_{i-sp_i+1} \dots P_i = T_{k-sp_i+1} \dots T_k \tag{2}$$

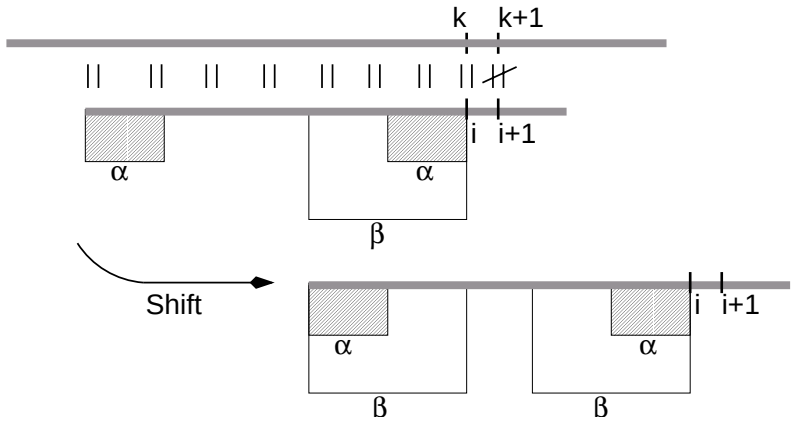
Die Gleichungen (1) und (2) zusammen ergeben das gewünschte Ergebnis. □

Beispiel 3.18 Sei T und P wie unten geben:

T:	x	y	a	b	c	x	a	b	c	x	a	d	c	c	d	g	f	e	g
										∦									
P:			a	b	c	x	a	b	c	d	e								
	↪ shift						a	b	c	x	a	b	c	d	e				
							$\underbrace{\quad\quad}_{\alpha}$					$\underbrace{\quad\quad}_{\alpha}$							

↪ α braucht nicht mehr verglichen. ◀

Es stellt sich die Frage, warum keine Position davor matchen kann (die Positionen danach werden ja bei späteren Shifts betrachtet). Die Grundidee ist, dass ein früherer Match einen neuen Suffix β von $P_1 \dots P_i$ erfordern würde, der gleichzeitig ein Prefix von $P_1 \dots P_i$ ist. Dieses β wäre aber länger als das α (dessen Länge sp_i ist), was im Widerspruch zur Definition von α als das längste solche Suffix steht. Dies wird am folgenden Bild verdeutlicht:



Theorem 3.19 *Sei T ein Suchtext, und P ein Pattern das in T gesucht wird. Sei nun $P_1 \dots P_i = T_{k-i+1} \dots T_k$. Dann gilt für alle Positionen $sp_i < j < i$ gilt*

$$P_1 \dots P_j \neq T_{k-j+1} \dots T_k$$

Beweis. Beweis durch Widerspruch. Wir nehmen an daß für ein $sp_i < j$ gilt

$$P_1 \dots P_j = T_{k-j+1} \dots T_k.$$

Aufgrund der Vorraussetzung daß $P_1 \dots P_i = T_{k-i+1} \dots T_k$. Nachdem $sp_i \leq i$, gilt damit auch

$$P_{i-j+1} \dots P_i = T_{k-j+1} \dots T_k.$$

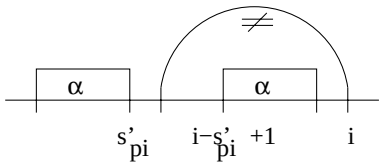
Dann folgt daraus

$$P_1 \dots P_j = P_{i-j+1} \dots P_i.$$

Somit ist $P_{i-j+1} \dots P_i$ ein Suffix, das auch ein Prefix von $P_1 \dots P_i$ ist. Das steht im Widerspruch zur Annahme, daß sp_i die Länge des längsten Strings mit dieser Eigenschaft ist. □

Dies impliziert, dass man im Fall von einem $sp_i = 0$ keinen Match haben kann, der im Bereich Position k in T beginnt. Dann kann man gleich mit dem Vergleichen von P_1 mit T_{k+1} beginnen.

Bevor wir die Shiftregel in einen Algorithmus fassen, wollen wir die Shiftregel noch verbessern. Sie wird nur angewendet, wenn T_{k+1} verschieden von P_{i+1} ist (Mismatch). Ist nun P_{sp_i+1} gleich P_{i+1}, dann kann wiederum kein Match erfolgen, und man muss die ganze Sequenz einen Match bilden. Wir haben folgendes Bild:



Es gilt immer $sp'_i \leq sp_i$.

Definition 3.20 *Für jede Position i in P sei $sp'_i(P)$ die Länge des längsten Suffixes von $P_1...P_i$, das ein Präfix von P ist, mit der zusätzlichen Bedingung: $P'_{sp_{i+1}} \neq P_{i+1}$*

Beispiel 3.21 Hier kann man den Zusammenhang zwischen sp_i, sp'_i und Z_i sehen:

	a	b	c	a	c	a	b	c	a	b	d	
i =	1	2	3	4	5	6	7	8	9	10	11	
sp_i =	0	0	0	1	0	1	2	3	4	2	0	
sp'_i =	0	0	0	1	0	0	0	0	4	2	0	rechte Grenze der Z-Box
Z_i =	0	0	0	1	0	4	0	0	2	0	0	

◀

Beispiel 3.22 Beispiel für Unterschied

T:	a	b	c	a	e	a	b	d	x	x	x	x	x	x	
									∦						
P:	a	b	c	a	e	a	b	c	a	b	d				
						a	b	c	a	e...	alte Shiftregel				
		↪ kein Match													
								a	b	c...	neue Shiftregel				

◀

Bevor wir aufzeigen, wie sp'_i überhaupt berechnet wird, wollen wir zunächsten den Algorithmus für KMP vorstellen. Hier wird der Shift nicht tatsächlich ausgeführt, sondern nur durch das Umsetzen der Laufvariablen erreicht. Es gibt hier zwei Laufvariablen, nämlich k für den (laufende) Zeiger auf die Position in T, und i für den (laufende) Zeiger auf die Position P. Dieser wird dann im Fall eines Mismatches um den Wert von sp_i weiterversetzt. Dafür verwenden wir die Definition der Fehlerfunktion.

Definition 3.23 *Wir definieren die* Fehlerfunktion $F'(i)$ *durch* $F'(i) = sp'_{i-1} + 1$, *wobei* sp_0 *auf* 0 *gesetzt wird.*

$F'(i)$ ist die Schrittweite, um den der Zeiger i von Position von P weitergesetzt wird in dem Fall, daß es einen Mismatch an Position i gibt. Damit haben den folgende Algorithmus:

```
BEGIN
  Berechne F'(i)=sp'_{i-1}+1 in einen Vorverarbeitungsschritt
  k = 1
  i = 1
  WHILE (k + (|P|-i)) < |T|)
      WHILE (T[k] = P[i]) und i ≤ n
         k = k + 1
         i = i + 1
      ENDWHILE
      IF (i = |P| + 1) THEN
               report match at k − |P|
      ENDIF
      IF (i = 1) THEN
         ## Mismatch an erster Position, daher
         ## muss das nächste Zeichen in T betrachtet werden
         k = k + 1
      ENDIF
      i = F'(i)
  ENDWHILE
END
```

Jetzt muss nur noch sp'_i berechnet werden. Dazu wollenb wir den Z-Algorithmus verwenden. Der Zusammenhang wurde ja schon im Beispiel 3.21 gezeigt. Wir benötigen noch eine Definition.

Definition 3.24 *Position* $j > 1$ *zeigt auf die Position i, falls:* $i = j + z_j(P) - 1$

Bemerkung: linkes Ende der Z-Box zeigt auf rechtes Ende

Theorem 3.25 *Für jedes* $i > 1$ *gilt* $sp'_i = Z_j$ *(i-j+1), wobei* $j > 1$ *die kleinste Position ist, die auf i zeigt. Falls es keine Position gibt, dann wird* $sp'_i = 0$ *gesetzt.*

Daraus kann man den folgenden Algorithmus zur Berechnung von sp'_i ableiten:

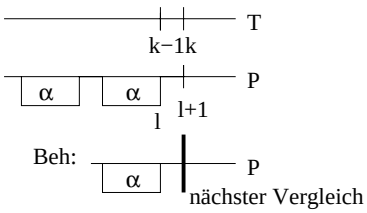
Algorithmus (für sp'_i) : FOR i=1...$|P|$
$sp'_i := 0$
END FOR
FOR j=$|P|$...2
$i := j + Z_j(P) - 1$
$sp_i := Z_j$
END FOR

3.8 Knuth-Morris-Pratt-Algorithmus

- 1977
- bekanntester linearer Pattern-Matching-Algorithmus ($O(n)$)
- Preprocessing:
  - Z-Algorithmus

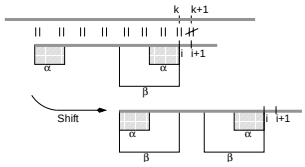
  ↪ Erweiterung auf multiple Pattern möglich

  ↪ Internet-Version (T fest und P variabel), Suchstring wird erst der Reihe nach eingegeben
- Grundidee: Shiftregel bei Missmatch



da α übereinstimmt → neue Position mögliche Anfangsposition

- Frage: Warum können Positionen davor keine Startpositionen sein?



$\Rightarrow$ β ein längerer Prefix $\rightarrow$ Widerspruch zu oben

- Beispiel:

T:	x	y	a	b	c	x	a	b	c	x	a	d	c	c	d	g	f	e	g
										∦									
P:			a	b	c	x	a	b	c	d	e								
	↪ shift						a	b	c	x	a	b	c	d	e				
							α				α								

$\hookrightarrow$ α wird nicht mehr verglichen (bei nur einem α in P $\rightarrow$ verschieben um 1

- Vorteil: Knuth-Morris-Pratt nur für P und nicht für T notwendig

> Für jede Position im Muster P sei sp_i(P) (eng: proper suffix) die Länge des längsten echten Suffixes von $P_1...P_i$, der mit einem Präfix von P übereinstimmt.

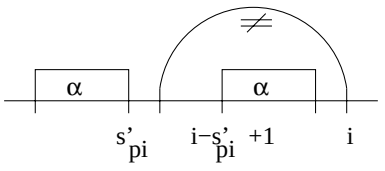
- Bemerkung: $P_1...P_{sp_i} = P_{i-sp_i+1}...P_i$

- Beispiel:

	a	b	c	a	c	a	b	c	a	b	d	
i =	1	2	3	4	5	6	7	8	9	10	11	
sp_i =	0	0	0	1	0	1	2	3	4	2	0	
sp'_i =	0	0	0	1	0	0	0	0	4	2	0	rechte Grenze der Z-Box
Z_i =	0	0	0	1	0	4	0	0	2	0	0	

- Es reicht folgendes zu betrachten, wegen Missmatch

> Für jede Position i in P sei sp'_i(P) die Länge des längsten Suffixes von $P_1...P_i$, das ein Präfix von P ist, mit der zusätzlichen Bedingung: $P'_{sp_{i+1}} \neq P_{i+1}$



$sp'_i \leq sp_i$

Beispiel siehe oben

- neue Shiftregel: man betrachtet nur die Suffixe/Präfixe, die durch sp_i definiert werden.
- Beispiel für Unterschied

T:	a	b	c	a	e	a	b	d	x	x	x	x	x	x
									∦					
P:	a	b	c	a	e	a	b	c	a	b	d			
						a	b	c	a	e...	alte Shiftregel			
			↪ kein Match											
								a	b	c...	neue Shiftregel			

- Wie berechne ich sp'_i ?

↪ Z-Algorithmus ⇒ Wie hängen Z-Boxen mit sp'_i zusammen?

Beispiel siehe oben

- Zusammenhang von Z_i und sp_i:

Position $j > 1$ zeigt auf die Position i, falls: $i = j + z_j(P) - 1$

- Bemerkung: linkes Ende der Z-Box zeigt auf rechtes Ende

SATZ: Für jedes $i > 1$ gilt $sp'_i = Z_j$ (i-j+1), wobei $j > 1$ die kleinste Position ist, die auf i zeigt. Falls es keine Position gibt, dann wird $sp'_i = 0$ gesetzt.

Algorithmus (für sp'_i) : FOR i=1...$|P|$
$sp'_i := 0$
END FOR
FOR j=$|P|$...2
$i := j + Z_j(P) - 1$
$sp_i := Z_j$
END FOR

```
k=1
WHILE (K < |T|)
   i=1
   WHILE (T[k] != '0' AND P[i+1] != '0' AND T[k]==P[i+1])
      k=k+1
```

```
      i=i+1
   ENDWHILE
   IF (P[i+1] = '0') THEN  MATCH FOUND
   ELSE
      i=max(1,sp'(i))
   ENDIF
ENDWHILE
```

4 Suffixbäume

Im folgenden woll wir das Problem behandeln, dass T nur einmal eingegeben wird, jedoch verschiedene S (Beispiel: T...Genom und S...bestimmte Gensequenz). Hierzu wollen wir die **Datenstruktur** der Suffixbäume einführen, die in der Bioinformatik eine wichtige Rolle spielen.

Was ist eine Datenstruktur? Eine Datenstruktur ist (vereinfacht) eine Mengen von (strukturierten) Objekten mit den dazugehörigen Operationen. Beispiele von Datenstrukturen sind Zahlen mit den arithmetischen Operation, oder Zeichenketten oder Arrays von Zahlen mit den Operationen "Zusammenfügen", "Auslesen eines Wertes" an Position i (wie z.B. in `if` S_i `=` T_i), "Setzen eines Wertes" (wie z.B. in $Z_i = 5$). Die Komplexität der einzelnen Operation ist natürlich wichtig, z.B. ist die Komplexität des Auslesen

- Problem
  - Zeichenkette S und T
  - array von ganzen Zahlen (z.B. Z-Boxen)
  - Operationen:

    Zusammenfügen (nur bei Z-Algorithmus)

    Auslesen eines Wertes an Position i

    ↪ Bsp.: if $(S_i = T_j)$

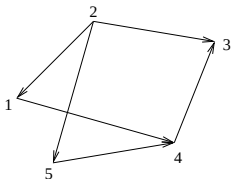
    Setzen eines Wertes

    ↪ Bsp.: $Z_i = 5$

    Komplexität von Auslesen (Setzen) = O(1)

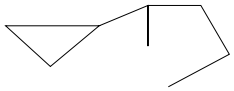
- Suffixbäume bekannteste Datenstruktur in der Bioinformatik
- Graphen: (siehe dazu Vorlesung graph-baume-vorl.pdf)
  - Punkte (Knoten)
  - Kanten (gerichtet oder ungerichtet)
  - Graph G = (V,E) V...vertex (Knoten), E...edges (Kanten)
  - Beispiel:

    $V = \{1, 2, 3, 4, 5\}$
    $E = \{(2, 1), (2, 5), (2, 3), (1, 4), (5, 4), (4, 3)\}$

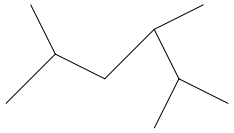
    

  - Darstellung am PC als Matrizen
  - Zyklus

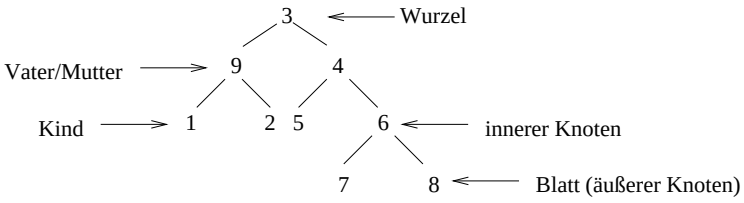
    Folge von Kanten
    Anfangsknoten = Endknoten

    
  - Baum

    zyklusfrei
    vollständig verbunden

    Spezialfall: gewurzelter Baum (mit ausgezeichnetem Knoten ≡ Wurzel)

Ein Suffixbaum τ für einen String S ist ein Baum mit Blättern 1...$|S|$(m) mit folgenden Eigenschaften:

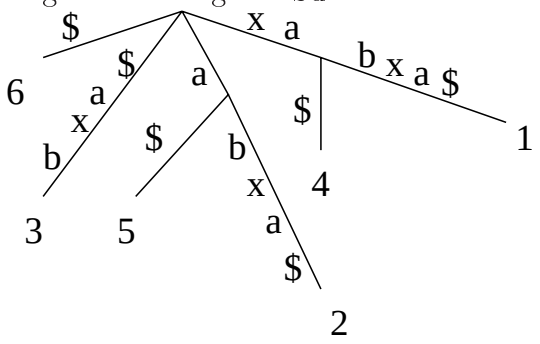
1. Jeder innerer Knoten hat mindestens 2 Kanten
2. Jede Kante ist mit einem nichtleeren Teilstring von S beschriftet.
3. Alle von einem inneren Knoten ausgehenden Kanten beginnen mit verschiedenen Buchstaben
4. Die Aneinanderhängung der Kantenbeschriftung von der Wurzel zum Blatt i ist das Teilwort $S_i...s_m$ (Suffix ab Position i)

- Beispiel:

S = xabxa$

Suffix:	$	a$	xa$	bxa$	abxa$	xabxa$
Pos.:	6	5	4	3	2	1

Beginn mit längsten Suffix



- Problem: Wenn ein Suffix Präfix eines anderen Suffixes ist → *kein* Suffixbaum

4.1 Anwendung von Suffix-Bäumen

1. Gegeben: Text T und dazugehöriger Suffixbaum, Suchstring S

   Finde alle exakten Vorkommen von S "Exact pattern matching"

   Aufgabe: Suffixbaum soll nicht neu aufgebaut werden.

Suffix der mit S beginnt

T: A C G A ... A T A C ...

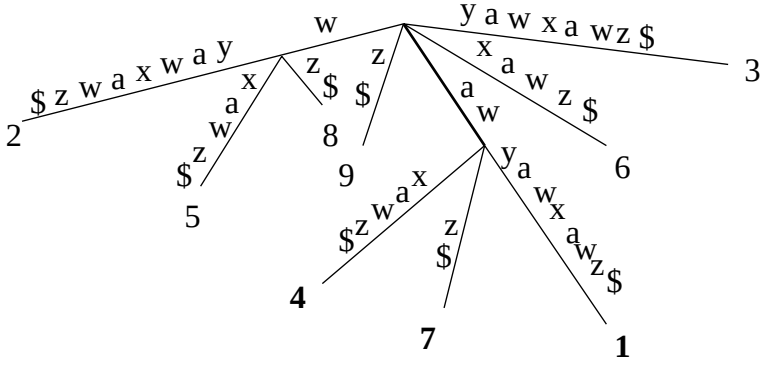
Suffix

- Beispiel:

	1	2	3	4	5	6	7	8	9
T=	a	w	y	a	w	x	a	w	z

Alle Suffixe:

a	w	y	a	w	x	a	w	z	$
	w	y	a	w	x	a	w	z	$
		y	a	w	x	a	w	z	$
			a	w	x	a	w	z	$
				w	x	a	w	z	$
					x	a	w	z	$
						a	w	z	$
							w	z	$
								z	$
									$



- Beispiel:
  - (a) Suche: S=aw⇒1,4,7
    Erklärung: Man suche die Folge aw im Baum
    ⇒ alle Blätter darunter sind Positionen von S
  - (b) S=a⇒ Folge der Kante, die mit a beginnt
    ⇒ darunter die Blätter sind die Positionen 1,4,7
  - (c) S=awy⇒1
  - (d) S=yawz⇒ wird im Baum nicht gefunden.⇒ kein Vorkommen in T
- Suche allgemein:

Gegeben: S, Suffixbaum τ

Gehe von Wurzel den Pfad S, bis entweder S aufgebraucht, oder τ.Es gibt 2 Fälle

(a) τ ist aufgebraucht⇒ S kommt nicht vor

(b) S ist aufgebraucht ⇒ alle Blätter unterhalb Treffer

2. DNA-Kontamination:

⇒ Dieses Problem tritt u.a. bei der Extraktion und Vermehrung von DNA durch die Polymerase Chain Reaction(PCR) auf.

- PCR: siehe Figure 1

  Problem:

  – Kontamination(falsche DNA) durch Hautschuppen
  – oder durch anderes DNA-Material(Laborprozeß)

- Führte im Experiment zu ähnlicher DNA zwischen Dinosaurier und Wirbeltier
- Algorithmisches Problem(DNA-Kontaminierung)

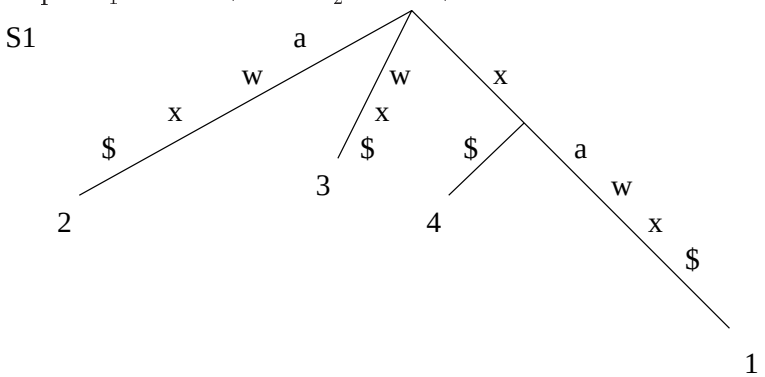
Gegeben: S_1, S_2(S_1 neue DNA-Sequenz(DINO))

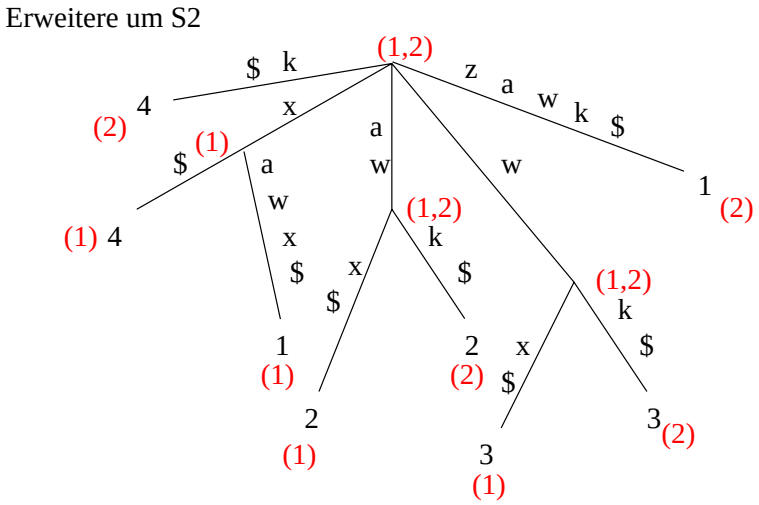
S_2 ist bekannte Sequenz(Quelle der Kontamination)

⇒ Aufgabe: Finde alle gemeinsamen Teilstrings von S_1, S_2 die eine Mindestlänge L haben.

Lösung: Baue Suffixbaum für S_1 und S_2

Bsp.: S_1= xawx\$ S_2=zawk\$





– Suche längsten Teilstring, der in S_1 und S_2 vorkommt
  * Wurzel ist mit (1,2) markiert
  * Steige von Wurzel so lange ab, solange Markierung 1,2 ist.

3. Repeats in der Biologie

- Bedeutung: Familien von wiederholten Sequenzen machen ca. ein Drittel des menschlichen Genoms aus.

Beispiel: C.Elegans:Bei 3.6 Mio Basenpaaren besitzt es 7000 Familien repetitiver Sequenzen.

1.Frage: Wie findet man repetitive Sequenzen?

2.Frage: Was sind repetitive Sequenzen?

- Versuch einer Definition:

Repeats sind Teilstrings, die mehrfach vorkommen.

Beispiel:

S= a a a a a a a
S= x a a x a a z

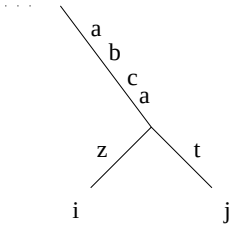
- Definition:

Maximaler Repeat: Wiederholte Teilsequenz eingerahmt von verschiedenen Buchstaben

Beispiel: ...xabcaz...sabcat

Maximaler Repeat

$\Rightarrow$ führt immer von der Wurzel zu einer Verzweigung.



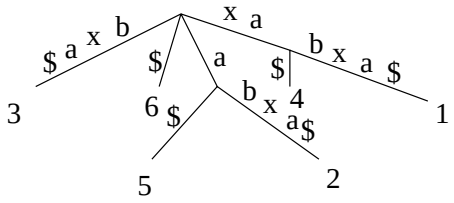
- Noch zu Vergleichen Position i-1 verschieden von j-1?

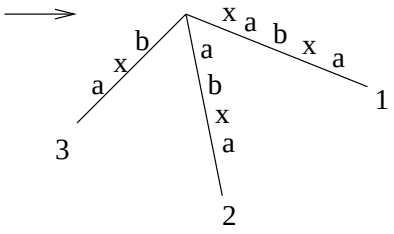
- Der "naive" Algorithmus O(n^3) bildet die Grundlage für O(n)

Das Label eines Pfades von der Wurzel zu einem inneren Knoten ist die Konkatenierung der Teilstrings der Beschriftungen des Pfades.

- Aus Konkatenierung zweier Strings S_1, S_2 $\Rightarrow$String $S_1 S_2$

Ein impliziter Baum für S entsteht aus einem Suffixbaum für S\$, indem alle auftreten von \$ ge"loscht werden, und alle leeren Kanten entfernt werden.





4.2 Naiver Algorithmus

– Baue impliziten Baum τ für $S_1 ... S_i$

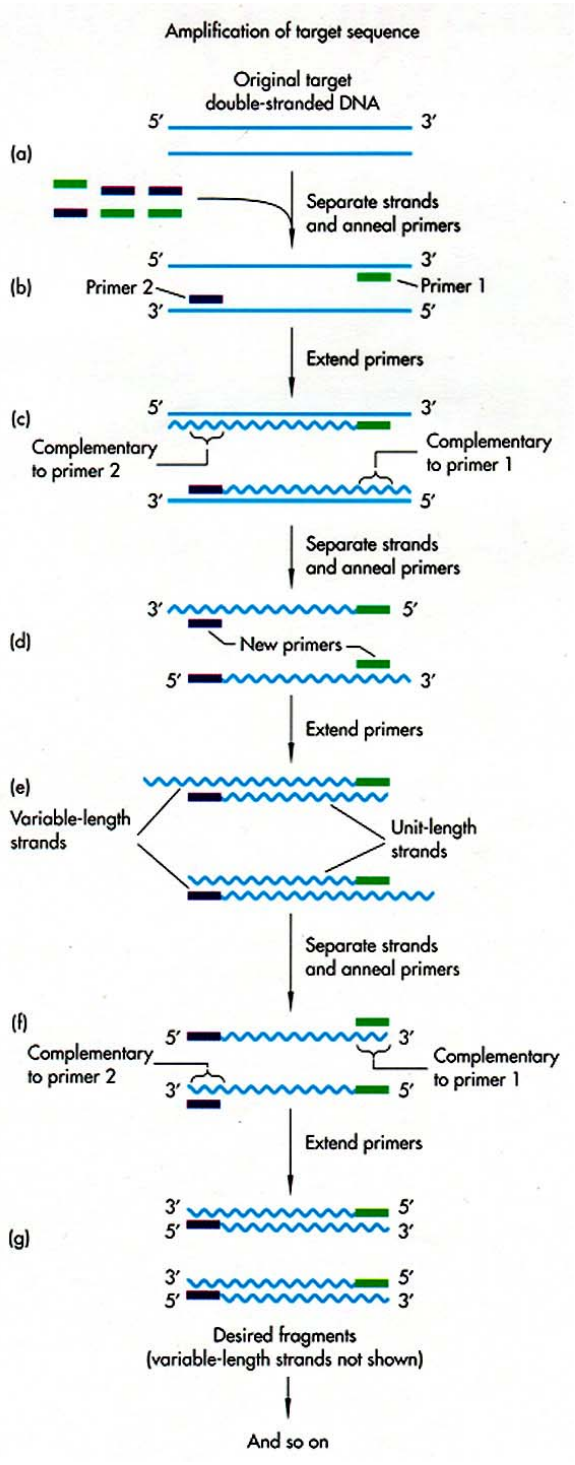


Figure 1: Polymerase Chain Reaction

- Dabei: Verwende τ_i für den Baum von τ_{i+1}
- Baum $\tau_i \rightarrow \tau_{i+1}$ füge alle Suffixe $S_j...S_{i+1}(1 \leq j \leq i+1)$hinzu.

4.3 Pseudocode:

Konstruiere τ_1
Phase i
  FOR i=1...|S|-1
  Erweiterungsschritte für i+1
    FOR j=1...i+1
    Erweiterung für $S_j...S_{i+1}$
    Finde das Ende des Pfades mit Label$S_j...S_i$
    Falls nötig erweitere Pfad um S_{i+1}
  END FOR
END FOR

Algorithmus stellt sicher, daß in der Phase i+1 alle Suffixe von $S_1...S_{i+1}$ in τ_{i+1}

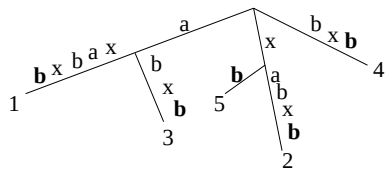
Sei β=$S_j...S_i$, d.h. βS_{i+1} in $\tau i+1$ hinzugefügt, nach folgenden Regeln:

1. β endet in einem Blatt$\Rightarrow$ S_{i+1} an Label angehängt.
2. Es gibt keinen Pfad am Ende von β, der mit S_{i+1} weitergeht $\Rightarrow$ neue Kante
3. Es gibt eine Verlängerung von β mit $S_{i+1} \Rightarrow$ gar nichts zu tun

Beispiel:

S=axabx|b

τ_5=



Hinzufügen von S_6 in Konstruktion von τ_6

j=	1	a	x	a	b	x	b	⇒	Regel 1
j=	2		x	a	b	x	b	⇒	Regel 1
j=	3			a	b	x	b	⇒	Regel 1
j=	4				b	x	b	⇒	Regel 1
j=	5					x	b	⇒	Regel 2
j=	6						b	⇒	Regel 3

Als Abschluß Erweiterung um \$ ⇒ Baum fertig,

5 Sequenzalignment

bisher: Suche nach exaktem" Mustern

jetzt: Suche nach ähnlichen Mustern

Motivation: Ähnlichkeit/Distanz von Wörtern

zum Beispiel: schimmlig/grimmig und Haus/Kaffee

Ähnlichkeit: schimmlig/grimmig

1. Mehrere gleiche Buchstaben
2. In der gleichen Reihenfolge

Levenstein Distanz für $S_1 S_2$ Minimale Anzahl von Einfügungen/Löschungen, die nötig sind um S_1 in S_2überzuführen.

schimmlig $\overset{4Lschungen}{\Rightarrow}$ immig $\overset{2Einfgungen}{\Rightarrow}$ grimmig ⇒ Distanz 6

Haus $\overset{3Lschungen}{\Rightarrow}$ a $\overset{5Einfgungen}{\Rightarrow}$ Kaffee ⇒ Distanz 8

Motivation 2: Homologe Proteinsequenzen

Zwei Proteine P,P' sind homolog, falls sie durch Evolution aus einem gemeinsamen Vorfahrprotein abstammen.

Bem.: Homologe Proteine haben meistens ähnliche Funktion und /oder Sequenzen

$\Rightarrow$ Modell der Sequenzevolution wird benötigt.

Evolution: Erlaubt Einfügen/Löschen und Ersetzen von Nukleotiden.

Anzahl der notwendingen evolutionären Operationen ist ein Maß für die Distanz.

Edit Operation:

Gegeben:AlphabetΣ(endliche Menge) mit -$\notin$ Σ(- gap-Symbol)

Eine edit-Operation ist ein Paar.

$(x, y) \in (\Sigma \cup \{-\}) \times (\Sigma \cup \{-\})$

(x,y) wird genannt:

* Substitution, falls x$\neq$- und y$\neq$-
* Einfügung, falls x= - und y$\neq$-
* Löschung, falls $x \neq$ - und y=-

Wir schreiben a$\rightarrow_{(x,y)}$ b falls a durch Ersetzen eines x durch y ((x,y) Substitution), bzw. eines Löschen von x ((x,y) Löschung), bzw. eines Einfügens eines y ((x,y)Einfügung) aus a generiert werden kann.

Löschen ATTACG$\rightarrow_{(T,-)}$ATACG

Substitution ACCA$\rightarrow_{(C,G)}$ACGA

Einfügen ATAT$\rightarrow_{(-,A)}$ATAAT

Sei $S = o_1 ... o_n$ eine Sequenz von Edit-Operationen
$a \Rightarrow_S b \leftrightarrow a = a^{(0)} \rightarrow_{o_1} a^{(1)} \rightarrow_{o_2} ... \rightarrow_{o_n} a^{(n)} = b$

- Needleman-Wunsch-Kostenfunktion

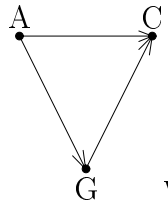
Sei w eine Funktion $(\Sigma \cup \{-\}) \times (\Sigma \cup \{-\}) \to \mathbb{R}$
w heißt Kostenfunktion. Die Kosten für eine Sequenz von Edit-Operationen. $S = o_1 ... o_n$ ist definiert als

$$w(S) = \sum_{i=1}^{n} w(o_i)$$

Die Edit-Distanz von zwei Wörtern $a, b \in \Sigma^*$ ist definiert als

$$d_w(a,b) = min\{w(S) | a \Rightarrow_S b\}$$

- Problem: Es gibt unendlich viele Sequenzen, die a nach b überführen.
- Idee: keine Mehrfach-Ersetzungen



w(A,C) ≤ w(A,G) + w(G,C)

Eine Kostenfunktion w heißt Metrik, falls sie folgende Bedingungen erfüllt:

1. $w(x,y) = 0 \leftrightarrow x = y$
2. $w(x,y) = w(y,x =)$ (symmetrisch)
3. $w(x,z) \leq w(x,y) + w(y,z)$ (Dreiecksungleichung)

gegeben sind 2 Wörter $a, b \in \Sigma^*$
2 andere Wörter sind ein Alignment von (a,b) falls $a^*, b^* \in (\Sigma \cup \{-\})^*$ und:

1. $|a^*| = |b^*|$
2. $\forall_i \neg(a_i^* = - = b_i^*)$
3. $a^*|_\Sigma = a$ und $b^*|_\Sigma = b$

- Beispiel eines möglichen Alignments

a =	A	C	G	G	A	T		
b =	C	C	G	C	T	T		
a^* =	A	C	-	G	G	-	A	T
b^* =	-	C	C	G	C	T	-	T

Die Kosten eines Alignments (a^*, b^*) bei gegebener Kostenfunktion w sind gegeben durch

$$w(a^*, b^*) = \sum_{i=1}^{|a^*|} w(a_i^*, b_i^*)$$

Die Alignment-Distanz von (a, b) ist:

$$D(a,b) = min\{w(a^*, b^*) | (a^*, b^*) \text{Alignment von} (a, b)\}$$

- ! Falls w eine Metrik $\rightarrow$ Edit-Distanz = Alignment-Distanz !
- Vorteil: statt unendlich vieler Edit-Sequenzen, nur endlich viele Alignments
- Aber: immer noch zu viele für direktes Suchen
- Lösung: Dynamische Programmierung
- Idee:
  1. Problem läßt sich auf Teilprobleme zurückführen
  2. Teilprobleme werden in Matrix gespeichert

Seien (a, b) zwei Wörter aus Σ^*. Die Matrix $(D_{ij})_{\substack{0 \le i \le |a| \\ 0 \le j \le |b|}}$ ist definiert durch:

$$D_{ij} = min\{w(u^*, v^*) | (u^*, v^*) \text{Alignment von} (a_1 ... a_i, b_1 ... b_j)\}$$

$\varepsilon \in \Sigma^*$ ist das leere Wort mit $|\varepsilon| = 0$.

- Beispiel:

  a = AT, b = AAGT

  $$\text{w(x,y)} = \begin{cases} 1 & \text{falls } x \neq y, \\ 0 & \text{sonst} \end{cases}$$

		A	A	G	T
	0	1	2	3	4
A	1	0			
T	2				D(a,b)

- Bemerkung: Sei (D_{ij}) Alignmentmatrix von a, b mit |a| = n und |b| = m. Dann ist D(a,b) = $D_{n,m}$

- Beispiel für Aufspaltung von Alignments

$$w\begin{pmatrix} A & C & - & G & G & - & A & T \\ - & C & C & G & C & T & - & T \end{pmatrix} = w\begin{pmatrix} A & C & - & G & G & - & A \\ - & C & C & G & C & T & - \end{pmatrix} + w\begin{pmatrix} T \\ T \end{pmatrix}$$

- Proposition:

Sei (u^*, v^*) ein Alignment von $(a_1...a_i, b_1...b_j)$. Sei $|u^*| = |v^*| = r$. Dann gilt:

$$w(u^*, v^*) = w(u_1^*...u_{r-1}^*, v_1^*...v_{r-1}^*) + w(u_r^*, v_r^*)$$

1. Löschen: Alignment von $(a_1...a_{i-1}, b_1...b_j)$
   ... | a_i $\quad w(u^*, v^*) = D_{i-1,j} + w(u_r^*, v_r^*)$
   ... | -
2. Einfügen: Alignment von $(a_1...a_i, b_1...b_{j-1})$
   ... | - $\quad w(u^*, v^*) = D_{i,j-1} + w(u_r^*, v_r^*)$
   ... | b_j
3. Ersetzen: Alignment von $(a_1...a_{i-1}, b_1...b_{j-1})$
   ... | a_i $\quad w(u^*, v^*) = D_{i-1,j-1} + w(u_r^*, v_r^*)$
   ... | b_j

- Beispiel:

  a = AT, i = 2, $a_1...a_2$ = AT

b = AAGT, j = 3, $b_1 ... b_3$ = AAG

1. Fall:	u^* =	-	-	A	T
	v^* =	A	A	G	-
2. Fall:		u^* =	A	T	-
		v^* =	A	A	G
3. Fall:		u^* =	-	A	T
		v^* =	A	A	G

- Satz:

Sei $(D_{i,j})$ die Alignmentmatrix von (a,b) mit |a| = n und |b| = m. Dann gilt:

$$D_{0,0} = 0$$

$$D_{i,0} = \sum_{k=1}^{i} w(a_k, -)$$

$$D_{0,j} = \sum_{k=1}^{j} w(-, b_k)$$

und für $1 \leq i \leq n, 1 \leq j \leq m$

$$D_{i,j} = min \begin{cases} D_{i-1,j-1} + w(a_i, b_j) \\ D_{i-1,j} + w(a_i, -) \\ D_{i,j-1} + w(-, b_j) \end{cases}$$

Problem: Mit $D_{i,j}$ allein $\Rightarrow$ nur Alignment Distanz, ist aber nicht automatisch bestes Alignment

5.1 Erweiterung: Tracematrix und Traceback

Idee Für bestes Alignment merke für jedes $D_{i,j}$, wo kommt $D_{i,j}$ her-

Beispiel: $D_{2,4}$

		A	A	G	T
	0	1	2	3	4
A	1	0	1	2	3
T	2	1	1	2	2

$$D_{2,4} = min \begin{cases} D_{1,3} + w(T,T) \\ D_{2,3} + w(-,T) \\ D_{1,4} + w(T,-) \end{cases} = D_{1,3} + w(T,T)$$

$$D_{1,3} = min \begin{cases} D_{0,2} + w(A,G) \\ D_{1,2} + w(-,G) \text{(Am besten)} \\ D_{0,3} + w(A,-) \end{cases}$$

$\Rightarrow$ Bestes Alignment:

-	A	-	T
A	A	G	T

Formal:

Die Tracematrix $(tr_{i,j})_{(o \leq i \leq |a|) und (o \leq j \leq b)}$ für Alignmentmatrix $D_{i,j}$ und Wörter (a,b) ist eine Matrix mit Elementen $tr_{i,j} \leq \{\nwarrow, \uparrow, \leftarrow\}$ wobei folgendes gilt:

$tr_{0,0} = \varnothing$

$tr_{0,j} = \{\leftarrow\}$

$tr_{i,0} = \{\uparrow\}$

$\forall_{i,j} \geq 1$

$\nwarrow \epsilon tr_{i,j}$ genau dann wenn, $D_{i,j} = D_{i-1,j-1} + w(a_i, b_i)$

$\uparrow \epsilon tr_{i,j}$ genau dann wenn, $D_{i,j} = D_{i-1,j} + w(a_i, -)$

$\leftarrow \epsilon tr_{i,j}$ genau dann wenn, $D_{i,j} = D_{i,j-1} + w(-, b_j)$

Beispiel: a=AT b=AAGT

		A	A	G	T
	$\{\varnothing\}$	$\{\leftarrow\}$	$\{\leftarrow\}$	$\{\leftarrow\}$	$\{\leftarrow\}$
A	$\{\uparrow\}$	$\{\nwarrow\}$	$\{\nwarrow, \leftarrow\}$	$\{\leftarrow\}$	$\{\leftarrow\}$
T	$\{\uparrow\}$	$\{\uparrow\}$	$\{\leftarrow\}$	$\{\leftarrow, \nwarrow\}$	$\{\nwarrow\}$

Pfad durch $tr_{i,j}$, von rechts unten bis links oben. Notiert von rechts nach links.

t_1 :$\leftarrow\nwarrow\leftarrow\nwarrow$

t_2 :$\nwarrow\leftarrow\leftarrow\nwarrow$

Sei t ein Traceback für(a,b). Das dazugehörige Alignment ist dann wie folgt definiert.

a^*= Ersetze i-tes Vorkommen von $\nwarrow, \uparrow$ durch a_i, jedes $\leftarrow$ durch -(gap)

b^*= Ersetze j-tes Vorkommen von $\nwarrow, \leftarrow$ durch b_j, jedes $\uparrow$ durch gap

für t_1:	für t_2:
$a_1^* = -A-T$	$a_2^* = A--T$
$b_1^* = AAGT$	$b_2^* = AAGT$

5.2 Algorithmus

Mengen-Darstellung:

a,Mengenelemente$\leftrightarrow$Zahlen

$\nwarrow \rightarrow_{def.} 0$

$\uparrow \rightarrow_{def.} 1$

$\leftarrow \rightarrow_{def.} 2$

Teilmengen meiner Gesamtmengen Σ $A \subseteq \Sigma$ werden durch Identitätsfunktion dargestellt.

$\vartheta_A : \Sigma \rightarrow \{0,1\}$

$\vartheta_A(e) = 1$ gdw. $e \epsilon A$

A=$\{\nwarrow, \uparrow\}$

ϑ_A: 012
110

Auf dem Rechner:

TR=int_array[|a|+1][|b|+1] [3](3 dimensionales ARRAY)

- Algorithmus für Needleman-Wunsch-Algorithmus

  – "unschöne" Variante, ohne STACK-Datenstrukturen

```
BEGIN
  n = |A|;   m = |B|;
  D = intarray[n+1][m+1];
  match=0;    left = 1;    up = 2;
  TR = intarray[n+1][m+1][3];
//Annahme: alle Einträge initial 0
//Aufbau der Matrizen
  FOR i = 1 TO n
    D[i][0] = D[i-1][0] + w(A[i],-);
    TR[i][0][up] = 1;
  END FOR
  FOR j = 1 TO m
    D[0][j] = D[0][j-1] + w(-,B[j]);
    TR[0][j][left] = 1;
  END FOR
  FOR i = 1 TO n
    FOR j = 1 TO m
                          { D[i-1][j-1] + w(A[i],B[j])
      D[i][j] = min       { D[i][j-1] + w(-,B[j])
                          { D[i-1][j] + w(A[i],-)
      IF D[i][j] = D[i-1][j-1] + w(A[i],B[j]) THEN
        TR[i][j][match] = 1;
      IF D[i][j] = D[i][j-1] + w(-,B[j]) THEN
        TR[i][j][left] = 1;
      IF D[i][j] = D[i-1][j] + w(A[i],-) THEN
        TR[i][j][up] = 1;
    END FOR
  END FOR
//Berechnung des Traceback
//"unschön" ohne STACK
```

```
    k = 0;
    i = n; j = m;
    TB = intarray[n+m];
    WHILE (i > 0 or j > 0)
      k = k + 1;
      IF TR[i][j][match]=1 THEN
        i = i - 1; j = j - 1;
        TB[n+m-k+1] = match;
      ELSEIF TR[i][j][up]=1 THEN
        i = i - 1; //j unverändert
        TB[n+m-k+1] = up;
      ELSEIF TR[i][j][left]=1 THEN
        j = j - 1; //i unverändert
        TB[n+m-k+1] = match;
      END IF
    END WHILE
  //erster (am weitesten links liegender) Eintrag in TB
    first = n + m - k + 1; i = 1
  //Alignmentzeile für A
    FOR l = 1 TO k
      IF TB[first + l - 1]=left THEN
        PRINT "-";
      ELSE
        PRINT A[i];
      i = i + 1;      END IF
    END FOR
    PRINT NEWLINE; //Alignmentzeile für B
    j = 1;
    FOR l = 1 TO k
      IF TB[first + l - 1]=up THEN
        PRINT "-";
      ELSE
        PRINT B[j];
      j = j + 1;      END IF
    END FOR
  END.
```