

SS09  
Effiziente Algorithmen  
5. Kapitel:  
Text-Such-Algorithmen

Martin Dietzfelbinger

Juli 2009

Kapitel 5: Textalgorithmen

**Problem:** Textsuche

**Gegeben:** Alphabet  $\Sigma$

„**Text**“  $S = s_1 \dots s_n \in \Sigma^*$  in Array  $S[1..n]$ .

„**Muster**“  $P = p_1 \dots p_m \in \Sigma^*$  in Array  $P[1..m]$ .

**Aufgabe:**

Finde erstes (oder: alle) Vorkommen von  $S$  in  $P$ ,  
d.h. kleinstes (oder: alle)  $\ell$  mit  $P = s_\ell \dots s_{\ell+m-1}$ , falls  
möglich.

*Beispiel:*

$P = \text{NADEL}$

$S = \text{IM HEUHAUFEN DIE NADEL FINDEN}$

Ausgabe:  $\{17\}$

$S = \text{IM NADELHAUFEN DIE NADEL FINDEN}$

Ausgabe:  $\{4, 20\}$

$S = \text{IM WALD DEN BAUM FINDEN}$

Ausgabe:  $\emptyset$

Der naive Algorithmus:

Man vergleicht  $P$  Buchstabe für Buchstabe mit  $s_\ell \dots s_{\ell+m-1}$ ,  
für jedes  $\ell \in \{1, \dots, n - m + 1\}$ .

$A \leftarrow \emptyset;$

**for** 1 **from** 1 **to**  $n - m + 1$  **do**

$j \leftarrow 1;$

$k \leftarrow 1;$

**while**  $k \leq m$  **and**  $s_j = p_k$  **do**  $k++$ ;  $j++$ ;

**if**  $k = m + 1$  **then**  $A \leftarrow A \cup \{1\}$ ;

**return**  $A$ .

Laufzeit:  $O((n-m+1)m)$  im schlechtesten Fall. Oft geringer.

Schlechtester Fall:  $S = a^{n-1}b$ ,  $P = a^{m-1}b$ .

## 5.1. Wortproblem für reguläre Sprachen

**Variation:** Man sucht nach einem von mehreren Wörtern.

Oder: Das Suchwort enthält unbestimmte „Wildcard-Buchstaben“.

Anwendung: Stringsuche mit Editor. „grep“/„sed“ in UNIX.

**Allgemein:** Gegeben ist ein **regulärer Ausdruck**  $r$  über  $\Sigma$ .

Beispiel für konkrete Formulierung:

Gegeben  $S = s_1 \dots s_n$  und  $r$ , finde alle Stellen in  $S$ , an denen ein Teilwort **endet**, das zu  $r$  „passt“, d. h. zu  $L(r)$  gehört.

$$r = N \cdot D + N \cdot [A - Z] \cdot D.$$

$S = \text{IM NADELHAUFEN DIE NADEL FINDEN}$

Ergebnis:  $\{6, 22, 29\}$ .

## Definition 5.1.1.

(**reguläre Ausdrücke**  $r$  und zugeordnete Sprachen  $L(r)$ )

- $\emptyset, \varepsilon, a$  (für  $a \in \Sigma$ ) sind reguläre Ausdrücke mit  $L(\emptyset) = \emptyset, L(\varepsilon) = \{\varepsilon\}, L(a) = \{a\}$ .
- Sind  $r, r_1$  und  $r_2$  reguläre Ausdrücke, dann sind auch  $(r_1 + r_2), (r_1 \cdot r_2)$  und  $(r^*)$  reguläre Ausdrücke mit  $L((r_1 + r_2)) = L(r_1) \cup L(r_2),$   
 $L((r_1 \cdot r_2)) = L(r_1)L(r_2) = \{z \mid z = xy, x \in L(r_1), y \in L(r_2)\},$   
 $L(r^*) = L(r)^* = \{w_1 \dots w_k \mid w_1, \dots, w_k \in L(r)\}.$

$|r| :=$  Anzahl der Zeichen in  $r$  ohne die runden Klammern.

Details: AFS-Vorlesung.

*Beispiel:* Die Menge der Wörter über dem Alphabet  $\{A, \dots, Z\}$ , die mit A beginnen und mindestens drei Buchstaben haben:

$L(r)$  mit

$$r = ((A \cdot \Sigma) \cdot (\Sigma \cdot (\Sigma^*))) \text{ mit } \Sigma = (A + (B + (\dots + Z))).$$

### Textsuche mit regulären Ausdrücken:

Eingabe ist  $S = s_1 \dots s_n$  und ein regulärer Ausdruck  $r$ .

Aufgabe: Suche (eine/die erste/alle) Stellen in  $S$ , an der/denen ein Wort aus  $L(r)$  **endet**.

**Ausgabe:** das erste oder alle  $i$  mit:  $\exists \ell \leq i: s_\ell \dots s_i \in L(r)$ .

### Vereinfachung:

„**Uniformes Präfixproblem**“ für reguläre Sprachen.

„uniform“:  $r$  ist Teil der Eingabe.

**Eingabe:**  $S = s_1 \dots s_n \in \Sigma^n$ , regulärer Ausdruck  $r$ .

**Ausgabe:**  $\{i \mid s_1 \dots s_i \in L(r)\}$ .

„Präfix  $s_1 \dots s_i$  gehört zu  $L(r)$ .“

Wir behandeln einen Algorithmus für das uniforme Präfixproblem für reguläre Sprachen.

(Eine Variante dieses Algorithmus löst auch das **Textsuchproblem**, siehe unten.)

### Satz (aus AFS)

$r$  regulärer Ausdruck über  $\Sigma$

$\Rightarrow$  es existiert  $\varepsilon$ -NFA  $M_r$  mit  $L(r) = L_{M_r}$ .

*Beweis* durch Induktion über den Aufbau von  $r$ .

(Gleichzeitig **Konstruktion** des  $\varepsilon$ -NFA  $M_r$ .)

Wir zeigen:

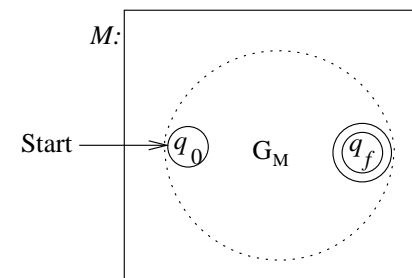
Es gibt einen  $\varepsilon$ -NFA

$$M_r = (Q, \Sigma, q_0, \{q_f\}, \delta), \text{ mit } q_0 \neq q_f,$$

der  $L(r) = L_{M_r}$  erfüllt.

Genau ein akzeptierender Zustand  $q_f$ !

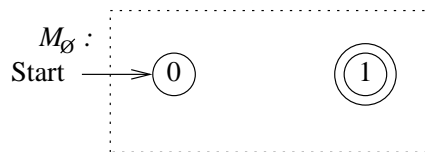
Schema:



### (i) Induktionsanfang

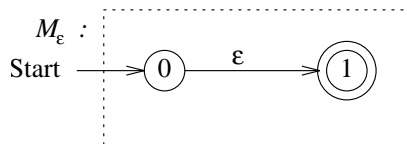
$r = \emptyset$ :

$G_{M_\emptyset}$  hat keine Kante



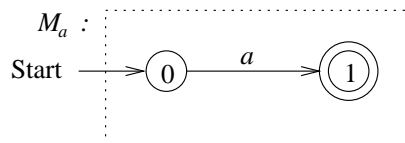
$r = \varepsilon$ :

$G_{M_\varepsilon}$  hat  $\varepsilon$ -Kante



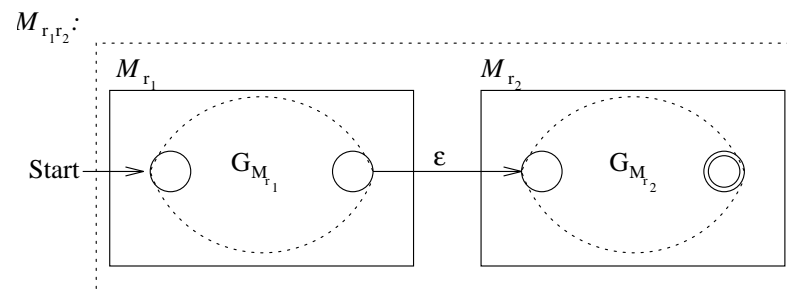
$r = a$ :

$G_{M_a}$  hat  $a$ -Kante

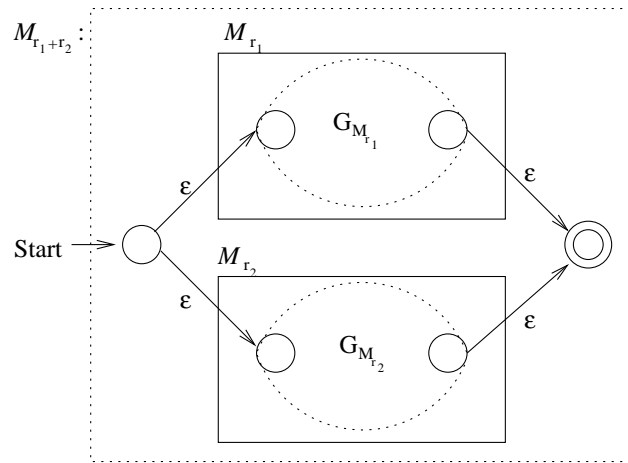


### (ii) Induktionsschritt — 3 Fälle

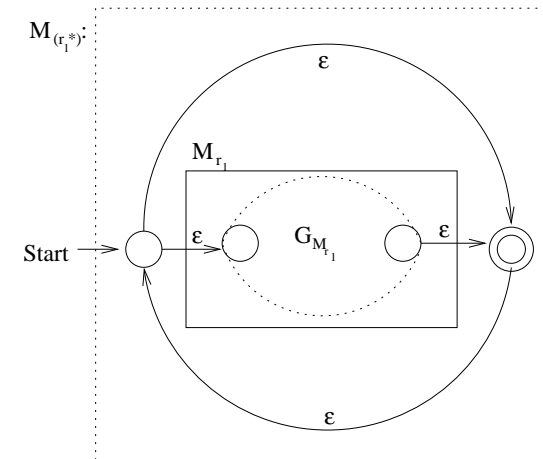
$r = (r_1 r_2)$ : Aus  $M_{r_1}$ ,  $M_{r_2}$  baue  $M_r$ :



$r = (r_1 + r_2)$ : Aus  $M_{r_1}, M_{r_2}$  baue  $M_r$ :



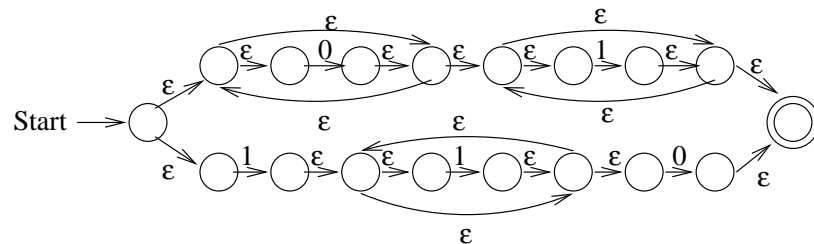
$r = (r_1^*)$ : Aus  $M_{r_1}$  baue  $M_r$ :



Beispiel:

$$0^*1^* + 11^*0$$

liefert:



Knotenzahl:

$$\leq 2 \cdot \# (\text{Symbole in } 0^* \cdot 1^* + 1 \cdot 1^* \cdot 0) = 2 \cdot 12 = 24.$$

Eigenschaften:

1. Die Konstruktion benötigt Zeit  $O(|r|)$ .
2.  $L(r) = L_{M_r}$ .
3.  $M_r$  hat höchstens  $2|r|$  Zustände (Knoten).
4. Knoten in  $M_r$  haben Ingrad und Ausgrad höchstens 2.  
(Adjazenzlisten in Tabelle zu speichern!)
5.  $M_r$  hat genau einen Anfangszustand (mit Ingrad  $\leq 1$ ) und genau einen Endzustand (mit Ausgrad 0).

Es genügt also, das **Präfixproblem** für  $\varepsilon$ -NFAs  $M$  mit maximalem In-/Ausgrad 2 zu lösen!

**Naiv: Potenzmengenkonstruktion** transformiert den  $\varepsilon$ -NFA  $M_r$  in einen äquivalenten DFA  $M$ .

Mit  $M$  kann dann in Zeit  $O(n)$  die Eingabe  $S = s_1 \dots s_n$  auf die Eigenschaft  $s_1 \dots s_i \in L_M = L(r)$  getestet werden.

Nachteil: Laufzeit, Speicherplatz eventuell **exponentiell** in  $|r|$ .

Vertretbar nur wenn  $r$  für viele oder sehr lange Texte benutzt wird.

Viel effizienter: Beibehalten des  $\varepsilon$ -Automaten.

$$M = (Q, \Sigma, q_0, F, \delta).$$

$$S = s_1 \dots s_n.$$

Ansatz analog „Dynamische Programmierung“.

$$R := \{q \in Q \mid \exists \text{Weg in } M \text{ von } q_0 \text{ zu } q \text{ mit Kantenmarkierungen (aneinandergesetzt) } s_1 \dots s_n\}.$$

$$\text{Dann: } S \in L_M \Leftrightarrow R \cap F \neq \emptyset.$$

**Teilprobleme:** Für  $0 \leq i \leq n$  sei

$$R_i := \{q \in Q \mid \exists \text{Weg in } M \text{ von } q_0 \text{ zu } q \text{ mit Kantenmarkierungen (aneinandergesetzt) } s_1 \dots s_i\}.$$

(Bellman-)“Gleichungen“:

$$q \in R_0 \Leftrightarrow \exists \text{Weg von } q_0 \text{ nach } q \text{ aus } \varepsilon\text{-Kanten, mit Länge } \geq 0.$$

$$q \in R_i \Leftrightarrow \exists q' \in R_{i-1} \exists q'' \in Q: \text{Kante } (q', q'') \text{ mit } s_i \text{ markiert und } \exists \text{Weg von } q'' \text{ nach } q \text{ aus } \varepsilon\text{-Kanten, mit Länge } \geq 0.$$

### Abkürzung:

$$\varepsilon\text{-closure}(T) := \{q \in Q \mid \text{es gibt einen Weg aus } \varepsilon\text{-Kanten von einem } q'' \in T \text{ zu } q\}.$$

Wie berechnet man  $\varepsilon\text{-closure}(T)$ ?

**Breitensuche** im Automatengraphen, startend mit  $T$ , nur  $\varepsilon$ -Kanten berücksichtigen.

Zeitaufwand:  $O(|r|)$ .

Iterativer Algorithmus für Textsuche mit

$\varepsilon$ -NFA  $M = (Q, \Sigma, q_0, F, \delta)$ , Inputstring  $S = s_1 \dots s_n$ :

$A \leftarrow \emptyset$ ; (\* sammelt Positionen \*)

$R_0 \leftarrow \varepsilon\text{-closure}(\{q_0\});$

**if**  $R_0 \cap F \neq \emptyset$  **then**  $A \leftarrow A \cup \{0\}$ .

**for**  $i$  **from** 1 **to**  $n$  **do**

$T_i \leftarrow \bigcup_{q' \in R_{i-1}} \delta(q', s_i);$

$R_i \leftarrow \varepsilon\text{-closure}(T_i);$

**if**  $R_i \cap F \neq \emptyset$  **then**  $A \leftarrow A \cup \{i\};$

**return**  $A$

### Satz 5.1.2

Das uniforme Präfixproblem für reguläre Ausdrücke ist in Zeit  $O(n \cdot |r|)$  lösbar; dabei bezeichnet  $n$  die Länge des Wortes  $S$ , und  $r$  ist der betrachtete reguläre Ausdruck.

**Variation** des Algorithmus, die das **Teilwortproblem** mit regulären Ausdrücken löst:

$T_i \leftarrow \bigcup_{q \in T_{i-1} \cup R_0} \delta(q, s_i)$  und teste in jedem Schleifendurchlauf, ob  $R_i \cap F \neq \emptyset$  gilt.

**Effizienzverbesserung:** Zustandsmengen mit Bitvektoren aus etwa  $|\Sigma|/8$  Bytes realisieren.

Eventuell Breitensuchen für kleinere Teilmengen vorberechnen, in Tabelle speichern.

### 5.2 Textsuche nach Rabin-Karp: Randomisiertes Verfahren

$\Sigma \subseteq \{0, \dots, p-1\} = \mathbb{Z}_p$ , wobei  $p$  Primzahl.

**Erinnerung:**  $\mathbb{Z}_p$  mit  $+ \text{ mod } p$  und  $\cdot \text{ mod } p$  ist **Körper**.

Zuerst:  $S = s_1 \dots s_n$ ,  $P = p_1 \dots p_n$ .

Frage: Ist  $S = P$ ?

**Trick:** Algorithmus führt ein Zufallsexperiment aus.

→ “Randomisierte Algorithmen“.

Betrachte **Polynome** über  $\mathbb{Z}_p$ .

$$f_S(x) = (s_1x^{n-1} + s_2x^{n-2} + \dots + s_{n-1}x + s_n) \text{ mod } p$$

$$g_P(x) = (p_1x^{n-1} + p_2x^{n-2} + \dots + p_{n-1}x + p_n) \text{ mod } p$$

$$h(x) = (f_S(x) - g_P(x)) \text{ mod } p.$$

**1. Fall:**  $S = P$ . Dann:  $h(x)$  ist das **Nullpolynom**.

**2. Fall:**  $S \neq P$ . Dann:  $h(x)$  ist **nicht** das Nullpolynom, Grad  $0 \leq \deg(h) < n$ .

$\Rightarrow h(x)$  hat  $< n$  Nullstellen.

$$\Rightarrow \frac{|\{r \in \mathbb{Z}_p \mid f_S(r) = g_P(r)\}|}{p} < \frac{n}{p}.$$

**Randomisierter Identitätstest:**

Wähle  $r \in \mathbb{Z}_p$  **zufällig**.

$a \leftarrow f_S(r)$ ; (Auswertung mit Horner-Schema, Zeit  $\Theta(n)$ )

$b \leftarrow g_P(r)$ ;

**if**  $a = b$  **then return** 0 **else return** 1.

Zeit  $O(n)$ .

Verhalten:  $S = P \Rightarrow$  Ausgabe 0 garantiert.

$S \neq P \Rightarrow \mathbf{Prob}(\text{Ausgabe } 0) < \frac{n}{p}$ .

Wähle z.B.  $p > 2n$ :  $\mathbf{Prob}(\text{Ausgabe } 0) < \frac{1}{2}$ .

$L$ -fache Wiederholung:  $\mathbf{Prob}(\text{Ausgabe } 0) < \frac{1}{2^L}$ .

### Idee für Textsuche:

Für  $i = 1, \dots, n - m + 1$  vergleiche  $s_i \dots s_{i+m-1}$  mit  $p_1 \dots p_m$ .

Kosten:  $O((n - m + 1)m)$ .

Wie der naive Algorithmus. Und Fehler möglich!

### Idee für Textsuche, verbessert:

$a_i = f_{s_i \dots s_{i+m-1}}(r)$ , also

$a_i = (s_i \cdot r^{m-1} + s_{i+1} \cdot r^{m-2} + \dots + s_{i+m-2} \cdot r + s_{i+m-1}) \bmod p$ .

$a_{i+1} = (s_{i+1} \cdot r^{m-1} + s_{i+2} \cdot r^{m-2} + \dots + s_{i+m-1} \cdot r + s_{i+m}) \bmod p$ .

Multiplikation, Subtraktion, Ausklammern:

$a_{i+1} = ((a_i - s_i \cdot r^{m-1}) \cdot r + s_{i+m}) \bmod p$ .

Also:  $a_{i+1}$  aus  $a_i$  in Zeit  $O(1)$  berechenbar, wenn man  $r^{m-1}$  vorberechnet.

### Algorithmus Rabin-Karp( $S[1..n], P[1..m]$ )

**Eingabe:**  $P[1..m]$ : Muster;  $S[1..n]$ : Text; (\* über  $\mathbb{Z}_p$  \*)

**Ausgabe:** Menge  $A \subseteq \{1, \dots, n - m + 1\}$ ; initialisiere:  $A \leftarrow \emptyset$ ;

- (1) wähle  $r$  aus  $\mathbb{Z}_p$  **zufällig**;
- (2)  $z \leftarrow r^{m-1} \bmod p$ ; (\* Zeit  $O(\log m)$  \*)
- (3)  $a_1 \leftarrow (s_1 \cdot r^{m-1} + s_2 \cdot r^{m-2} + \dots + s_{m-1} \cdot r + s_m) \bmod p$ ;
- (4)  $b \leftarrow (p_1 \cdot r^{m-1} + p_2 \cdot r^{m-2} + \dots + p_{m-1} \cdot r + s_m) \bmod p$ ;
- (5) **if**  $a_1 = b$  **then**  $A \leftarrow A \cup \{1\}$ ;
- (6) **for**  $i$  **from** 2 **to**  $n - m + 1$  **do**
- (7)  $a_i \leftarrow ((a_{i-1} - s_{i-1} \cdot z) \cdot r + s_{i+m-1}) \bmod p$
- (8) **if**  $a_i = b$  **then**  $A \leftarrow A \cup \{i\}$ .
- (9) **return**  $A$ .

Man sieht leicht: Statt  $a_1, \dots, a_{n-m+1}$  kann man auch eine Variable  $a$  benutzen. – Dies ergibt:

### Algorithmus Rabin-Karp( $S[1..n], P[1..m]$ )

**Eingabe:**  $P[1..m]$ : Muster;  $S[1..n]$ : Text; (\* über  $\mathbb{Z}_p$  \*)

**Ausgabe:** Menge  $A \subseteq \{1, \dots, n - m + 1\}$ ; initialisiere:  $A \leftarrow \emptyset$ ;

- (1) wähle  $r$  aus  $\mathbb{Z}_p$  **zufällig**;
- (2)  $z \leftarrow r^{m-1} \bmod p$ ; (\* Zeit  $O(\log m)$  \*)
- (3)  $a \leftarrow (s_1 \cdot r^{m-1} + s_2 \cdot r^{m-2} + \dots + s_{m-1} \cdot r + s_m) \bmod p$ ;
- (4)  $b \leftarrow (p_1 \cdot r^{m-1} + p_2 \cdot r^{m-2} + \dots + p_{m-1} \cdot r + s_m) \bmod p$ ;
- (5) **if**  $a = b$  **then**  $A \leftarrow A \cup \{1\}$ ;
- (6) **for**  $i$  **from** 2 **to**  $n - m + 1$  **do**
- (7)  $a \leftarrow ((a - s_{i-1} \cdot z) \cdot r + s_{i+m-1}) \bmod p$
- (8) **if**  $a = b$  **then**  $A \leftarrow A \cup \{i\}$
- (9) **return**  $A$ .

Zeitaufwand:  $O(n + m)$ .

Wahrscheinlichkeits-Analyse:

Sicher werden alle  $i$  mit  $s_i \dots s_{i+m-1} = P$  in A ausgegeben.

Für jedes  $i$  mit  $s_i \dots s_{i+m-1} \neq P$  gilt:

**Prob**( $i$  wird ausgegeben)  $< \frac{m}{p}$ .

**Prob**( $\exists i: i$  wird ausgegeben  $\wedge s_i \dots s_{i+m-1} \neq P$ )  $< \frac{m}{p} \cdot (n - m + 1)$ .

Wähle  $p > 2(n - m + 1)m$ :  $\dots < \frac{1}{2}$ .

Oder: Wähle  $p > n^2 m$ :  $\dots < \frac{1}{n}$ .

(Dieselbe Anzahl von Operationen, mit längeren Zahlen.)

Oder:  $p > 1000nm$ , arbeite mit 3 Zufallszahlen parallel.

Rechenzeit verdreifacht;

**Prob**( $\exists i: i$  wird ausgegeben  $\wedge s_i \dots s_{i+m-1} \neq P$ )  $< \frac{1}{10^9}$ .

### 5.3 Deterministische Textsuche:

#### Algorithmus von Knuth-Morris-Pratt (KMP)

Wir kehren zur (simplen) Ausgangsfrage zurück. Für ein Muster  $P = p_1 \dots p_m$  ist sein Vorkommen in einem String  $S = s_1 \dots s_n$  zu finden.  $S$  und  $P$  sind als Arrays  $S[1 \dots n]$  und  $P[1 \dots m]$  gegeben.

**Grundansatz:** Führe einen Zeiger  $i$  monoton wachsend am String  $S[1 \dots n]$  entlang und finde für jedes  $1 \leq i \leq n$  **das längste Präfix**  $P[1 \dots q + 1]$  von  $P$ , das Suffix  $S[i - q \dots i]$  von  $S[1 \dots i]$  ist. Sollte dies einmal für  $i = m$  der Fall sein, wurde  $P$  in  $S$  lokalisiert.

*Beispiel:* (Ablauf von Intuition gesteuert):

S: ESWAREINMALEINMENSCHDEREINMALEINSRECHNETE

P: EINMALEINS

Man erreicht etwa irgendwann die folgende teilweise Übereinstimmung von  $P[1 \dots q]$  und  $S[i - q \dots i - 1]$ :

		$i$
S: ESWAR	EINMALEIN	MENSCHDEREINMAL...
P:	EINMALEIN	S
	$q$	$q + 1$

Es ist  $S[i] \neq P[q + 1]$ . Man „sieht“, dass man  $q$  ohne weiteres soweit reduzieren kann, dass das Präfix EIN von  $P$  unter das Suffix EIN von  $S[1 \dots i - 1]$  passt:

		$i$
S: ESWAREINMAL	EIN	MENSCHDEREINMAL...
P:	EIN	MALEINS
	$q$	$q + 1$

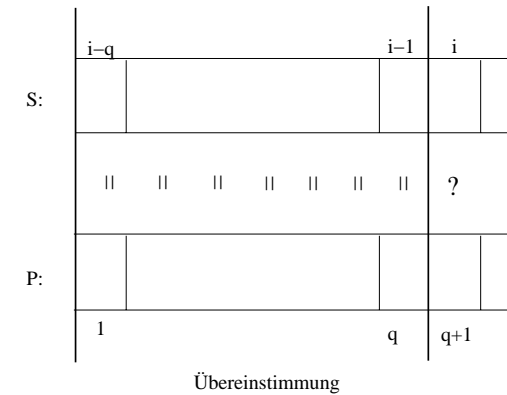
Nun passen die beiden M,  $i$  und  $q$  können also um 1 inkrementiert werden:

		$i$
S: ESWAREINMAL	EINM	ENSCH...
P:	EINM	MALEINS
	$q$	$q + 1$

Jetzt ist wieder  $S[i] \neq P[q+1]$ . Diesmal passt aber kein Präfix von  $P$  auf ein Suffix von E1NM, also ist  $q$  auf 0 zu verringern. Man macht schon hier die „intuitive Beobachtung“, dass für die Verringerung von  $q$  auf einen neuen Wert die Kenntnis von  $S$  unnötig ist.

Wir betrachten folgende allgemeine Zwischensituation.

Man hat eine teilweise Übereinstimmung von Text und Muster identifiziert.



Zwischensituation – „Invariante“: (Für  $0 \leq q < m$ ,  $q < i \leq n$ .)

$$(I_{i,q})(a): \quad p_1 \dots p_q = s_{i-q} \dots s_{i-1}.$$

$p_1 \dots p_q$  ist **Suffix** von  $s_1 \dots s_{i-1}$ .

$$(I_{i,q})(b): \quad \forall r, m > r > q: \quad p_1 \dots p_r p_{r+1} \neq s_{i-r} \dots s_i.$$

$p_1 \dots p_r p_{r+1}$  ist **kein Suffix** von  $s_1 \dots s_i$ .

Man beachte den Extremfall  $q = 0$  (Übereinstimmung von  $S[i]=P[1]$  möglich).

Wenn  $(I_{i,q})(a)(b)$  gelten, gibt es zwei Fälle:

**1. Fall:**  $p_{q+1} = s_i$ . – Zunächst sollten wir prüfen, ob  $q+1 = m$  ist, dann ist nämlich  $P = s_{i-q} \dots s_i$ , und wir sollten  $i - q$  ausgeben, . . .

und (eventuell) die Suche nach weiteren Vorkommen von  $P$  **fortsetzen** (wie: weiter unten).

Falls  $q + 1 < m$ , gilt  $(I_{i+1,q+1})(a)$ .

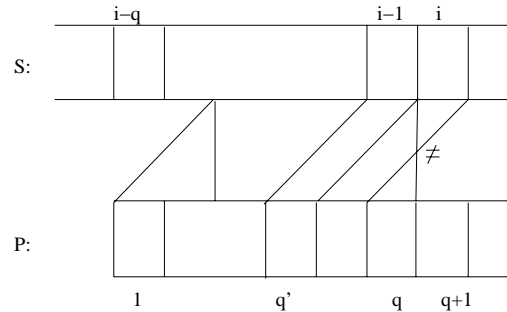
Es gilt aber auch  $(I_{i+1,q+1})(b)$  (Abschwächung von  $(I_{i,q})(b)$ !)

**Daher:**  $i$  und  $q$  inkrementieren, Invariante gilt weiter.

**2. Fall:**  $p_{q+1} \neq s_i$ . – Dann kann an Position  $s_{i-q}$  kein Vorkommen von  $P$  beginnen.

Wir wollen nun das Muster nach rechts verschieben, und mit dem Vergleich fortfahren.

Wie weit? Welche Stellen in  $S$  kommen als Startpunkte von  $P$  in Frage?



$i - q'$ , mit  $q' < q$ , so dass  $p_1 \dots p_{q'} = s_{i-q'} \dots s_{i-1}$ .  
(Dann:  $(I_{i,q'})(a)$ .)

Von allen möglichen solchen  $q'$  sollten wir **das größte** wählen, das entspricht dem **geringstmöglichen Weiterschieben** des Musters im Sinn des naiven Algorithmus.

Dann gilt auch  $(I_{i,q'})(b)$  (für  $r > q$  wegen  $(I_{i,q})(b)$ , für  $r = q$ , weil  $p_{q+1} \neq s_i$ , für  $q' < r < q$  wegen der Maximalität von  $q'$ ).

Wie maximales  $q'$  finden, so dass

$$p_1 \dots p_{q'} = s_{i-q'} \dots s_{i-1}?$$

Beobachte:  $s_{i-q'} \dots s_{i-1} = p_{q-q'+1} \dots p_q$ , wegen  $(I_{i,q})(a)$ .

$$f(q) := \max\{q' < q \mid p_1 \dots p_{q'} \text{ ist Suffix von } p_1 \dots p_q\}.$$

Die Funktion  $f$  („**Fehlerfunktion**“) hängt nur von  $P$  ab, und kann aus  $P$  berechnet werden (wie: später).

$$f: \{1, \dots, m\} \rightarrow \{0, \dots, m - 1\}.$$

Auch wenn  $P = s_{i-q} \dots s_i$  ist (und  $i$  ausgegeben wird), wollen wir das Muster möglichst wenig nach rechts schieben und mit der Suche fortfahren: Maximales  $q'$  ist  $f(m)$ .

*Beispiel:*  $P = \text{EINMALEINS}$ .

Wertverlauf von  $f(q), q = 1, \dots, 10$ :

$q$	1	2	3	4	5	6	7	8	9	10
$P[q]$	E	I	N	M	A	L	E	I	N	S
$f(q)$	0	0	0	0	0	0	1	2	3	0

---

Angenommen, wir hätten die Funktion  $f$  berechnet.

**Knuth-Morris-Pratt-Suche**( $S[1..n], P[1..m]$ )

**Eingabe:**  $P[1..m]$ : Muster;  $S[1..n]$ : Text;

**Vorberechnet:**  $f[1..m]$ : Fehlerfunktion als Tabelle;

**Ausgabe:** Menge  $A \subseteq \{1, \dots, n - m + 1\}$ ; initialisiert:  $A \leftarrow \emptyset$ ;

```
(1)  q ← 0;
(2)  for i from 1 to n do
(3)    while q > 0 ∧ P[q+1] ≠ S[i] do q ← f[q];
(4)    if P[q+1] = S[i] then q++;
(5)    if q = m then A ∪ {i-m+1}; q ← f[q].
(6)  return A.
```

---

**Korrektheit:**

Man beweist durch **Induktion** über  $i$  (Inhalt von  $i$ ),  
dass  $(I_{i,q})(a)(b)$  eine Schleifeninvariante bilden.

(Die Argumente wurden in der Vorüberlegung genannt!)

Daraus folgt dann sofort, dass (Zeile (5)) nur Indizes in die Menge  $A$  kommen, bei denen ein Vorkommen von  $P$  startet.

Etwas diffiziler: Kein Vorkommen von  $P$  wird übersehen.

(Hierfür benötigt:  $(I_{i,q})(b)$ ).

---

**Laufzeit:** (Achtung, eleganter Trick!)

$q$  startet mit Wert 0.

Die  $i$ -Schleife wird  $n$ -mal durchlaufen; bei jedem Durchlauf wird  $q$  maximal um 1 erhöht.

In jedem Durchlauf der  $while$ -Schleife wird  $q$  echt verringert.

⇒ es kann insgesamt nicht mehr als  $n$  Durchläufe durch die  $while$ -Schleife geben.

⇒ Gesamtlaufzeit für „Suche“ ist  $O(n)$ .

---

Berechnung der Fehlerfunktion:

**Knuth-Morris-Pratt-Preprocessing**( $S[1..n], P[1..m]$ )

**Eingabe:**  $P[1..m]$ : Muster;

**Ausgabe:**  $f[1..m]$ : Fehlerfunktion als Tabelle;

```
(1)  f[1] ← 0;
(2)  q ← 0;
(3)  for i from 2 to m do
(4)    while q > 0 ∧ P[q+1] ≠ P[i] do q ← f[q];
(5)    if P[q+1] = P[i] then q++;
(6)  f[i] ← q.
```

---

Interessant: Um  $f[i]$  zu berechnen, benutzt man  $f[q]$  für  $q < i$ , mit der (induktiven) Annahme, dass diese Werte schon korrekt sind.

*Beispiel:* Tafel.

Um zu zeigen, dass diese Prozedur tatsächlich die Fehlerfunktion  $f$  berechnet, beweist man die folgenden **Invarianten**:

$(I_{i,q}^f)$ (a):  $p_1 \dots p_q = p_{i-q} \dots p_{i-1}$ . (Für  $0 \leq q < m$ ,  $q < i \leq n$ .)

$(I_{i,q}^f)$ (b):  $\forall r, m > r > q: p_1 \dots p_r p_{r+1} \neq p_{i-r} \dots p_i$ .  
 $p_1 \dots p_r p_{r+1}$  ist **kein Suffix** von  $p_1 \dots p_i$ .

Dies sieht man genau wie bei den Überlegungen zu KMP-Suche.

---

Laufzeit für Preprocessing:  $O(m)$ .

(Analyse wie für die Suchprozedur!)

### Satz 5.3.1

**Knuth-Morris-Pratt-Preprocessing** und **Knuth-Morris-Pratt-Suche** zusammen lösen das Pattern-Matching-Problem in Zeit  $O(n + m)$  und Platzbedarf  $O(m)$  (von Ein- und Ausgabe abgesehen).