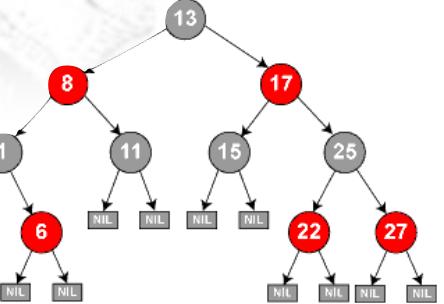
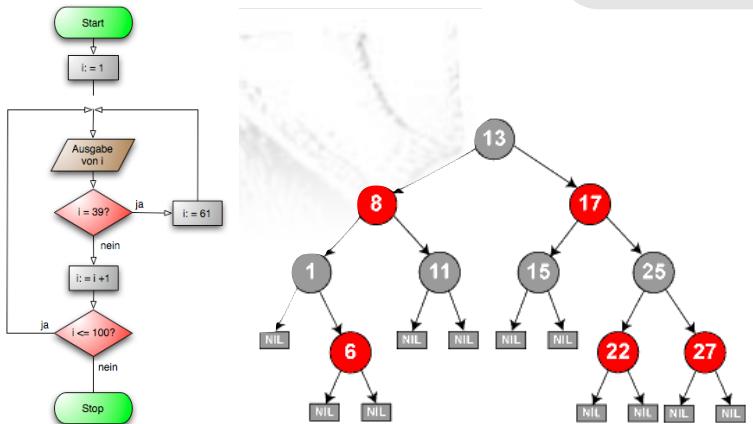




Algorithmen und Datenstrukturen



Version 1.26 vom 20. Mai 2021

VI. Suchen in Texten

6. Suchen in Texten

- 6.1. Einführung
- 6.2. Direkte Mustersuche
- 6.3. Das Verfahren von Rabin und Karp
- 6.4. Suche mit endlichen Automaten
- 6.5. Das Verfahren von Knuth, Morris und Pratt
- 6.6. Das Verfahren von Boyer und Moore

VI. Suchen in Texten

6. Suchen in Texten

6.1. Einführung

- 6.2. Direkte Mustersuche
- 6.3. Das Verfahren von Rabin und Karp
- 6.4. Suche mit endlichen Automaten
- 6.5. Das Verfahren von Knuth, Morris und Pratt
- 6.6. Das Verfahren von Boyer und Moore

Grundbegriffe (Wiederholung)

- Σ eine endliches **Alphabet** (z.B. $\Sigma = \{a, \dots, z\}$)
- Σ^* **Menge aller Wörter** über Σ
- $|w|$ **Wortlänge** von $w \in \Sigma^*$
- ε das **leere Wort** ($|w| = 0$)
- $w \circ v$: **Konkatenation** von w und v . Es ist $|w \circ v| = |w| + |v|$
- $v \sqsubset w$: v ist **Präfix** von w
- $v \sqsupset w$: v ist **Suffix** von w
- $v \sqsupset_i w$ v ist **ist Suffix der Länge i von w**

Wörter als String

- Wörter betrachten wir fortan auch als Zeichenfelder und nennen diese auch **String**

D String / Teilstring

Sei $w = a_0 \cdots a_{n-1}$ ein Wort über einem Alphabet Σ , dann ist der zu w gehörige **String** ein n -dimensionales Feld T mit $T[i] = a_i$ ($0 \leq i \leq n - 1$) und wir schreiben kurz: $T = w$.

Für $i \in \{0, \dots, n - 1\}$ nennen wir

$$T[i, l] = \begin{cases} a_i \cdots a_{i+l-1} & \text{falls } 1 \leq l \leq (n - i) \\ \varepsilon & \text{sonst} \end{cases}$$

den **Teilstring** von T der Länge l , beginnend bei Index i .

Formale Problembeschreibung

D String-Matching Problem

Seien Σ ein endliches Alphabet, $t \in \Sigma^*$ ein Text und $p \in \Sigma^*$ ein Suchmuster mit $n = |t| \geq |p| = m$. Ferner seien T und P Strings mit $T = t$ und $P = p$.

Wir sagen **P taucht mit Versatz s in T auf**, wenn gilt:

$$T[s, m] = P$$

(mit $0 \leq s \leq n - m$).

Wenn P mit Versatz s in T auftaucht, so nennen wir s einen **gültigen Versatz**.

Das **String-Matching Problem** besteht nun darin, jeden gültigen Versatz zu ermitteln, mit dem P in T auftritt.

Formale Problembeschreibung

D String-Matching Problem

Seien Σ ein endliches Alphabet, $t \in \Sigma^*$ ein Text und $p \in \Sigma^*$ ein Suchmuster mit $n = |t| \geq |p| = m$. Ferner seien T und P Strings mit $T = t$ und $P = p$.

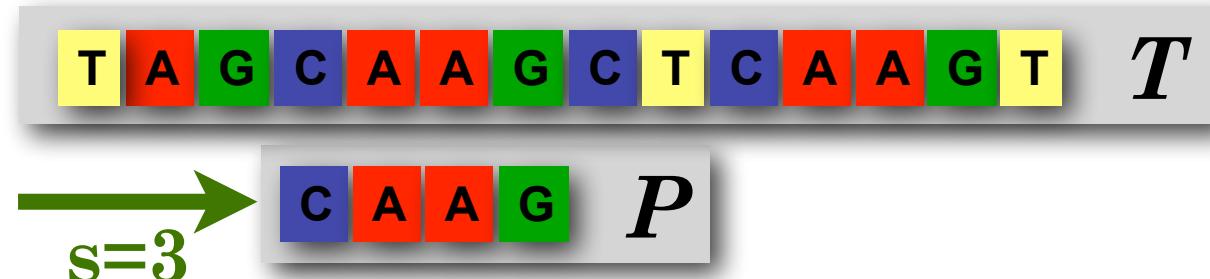
Wir sagen **P taucht mit Versatz s in T auf**, wenn gilt:

$$T[s, m] = P$$

(mit $0 \leq s \leq n - m$).

Wenn P mit Versatz s in T auftaucht, so nennen wir s einen **gültigen Versatz**.

Das **String-Matching Problem** besteht nun darin, jeden gültigen Versatz zu ermitteln, mit dem P in T auftritt.



VI. Suchen in Texten

6. Suchen in Texten

6.1. Einführung

6.2. Direkte Mustersuche

6.3. Das Verfahren von Rabin und Karp

6.4. Suche mit endlichen Automaten

6.5. Das Verfahren von Knuth, Morris und Pratt

6.6. Das Verfahren von Boyer und Moore

Direkte Mustersuche

- Prüfe an allen Positionen in t , ob das Muster p enthalten ist

Direkte Mustersuche

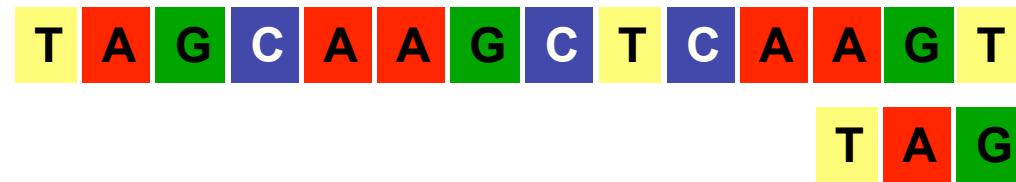
- Prüfe an allen Positionen in t , ob das Muster p enthalten ist



T A G C A A G C T C A A G T

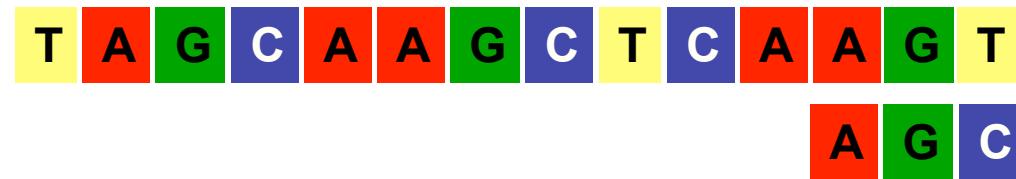
Direkte Mustersuche

- Prüfe an allen Positionen in t , ob das Muster p enthalten ist



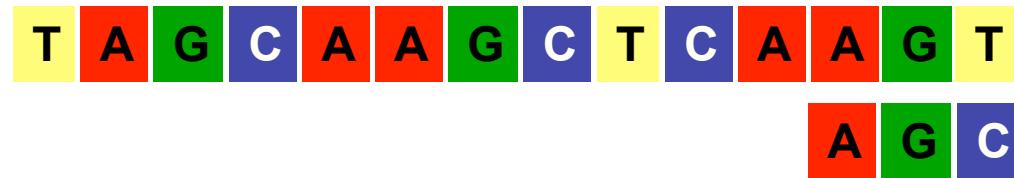
Direkte Mustersuche

- Prüfe an allen Positionen in t , ob das Muster p enthalten ist



Direkte Mustersuche

- Prüfe an allen Positionen in t , ob das Muster p enthalten ist

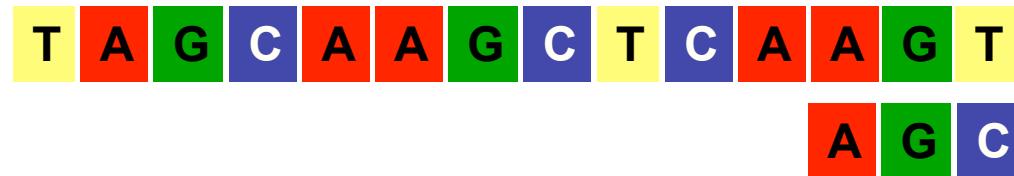


- Implementierung in C++

```
void stringMatch(vector<int>& result, string t, string p) {  
    result.clear();  
    if (t.size() < p.size()) return;  
    int n=t.size(), m=p.size();  
    bool found;  
    for (int i=0 ; i<n-m+1 ; i++) { // i: Startposition Muster  
        found = true;  
        for (int k=0 ; k<m ; k++) { // k: Position im Muster  
            if (t[i+k] != p[k]) { found = false; break; }  
        }  
        if (found) result.push_back(i);  
    }  
}
```

Direkte Mustersuche

- Prüfe an allen Positionen in t , ob das Muster p enthalten ist



- Implementierung in C++

```
void stringMatch(vector<int>& result, string t, string p) {  
    result.clear();  
    if (t.size() < p.size()) return;  
    int n=t.size(), m=p.size();  
    bool found;  
    for (int i=0 ; i<n-m+1 ; i++) { // i: Startposition Muster  
        found = true;  
        for (int k=0 ; k<m ; k++) { // k: Position im Muster  
            if (t[i+k] != p[k]) { found = false; break; }  
        }  
        if (found) result.push_back(i);  
    }  
}
```

$O((N-M+1) \cdot M)$ Vergleiche

VI. Suchen in Texten

6. Suchen in Texten

6.1. Einführung

6.2. Direkte Mustersuche

6.3. Das Verfahren von Rabin und Karp

6.4. Suche mit endlichen Automaten

6.5. Das Verfahren von Knuth, Morris und Pratt

6.6. Das Verfahren von Boyer und Moore

Idee des Algorithmus von Rabin und Karp

- Ordne dem Muster $P[0, m]$ eine **eineindeutige** Dezimalzahl p zu

Idee des Algorithmus von Rabin und Karp

- Ordne dem Muster $P[0, m]$ eine **eineindeutige** Dezimalzahl p zu
- Analog: ordne jedem Teilstring

$$T[s, m] \quad (0 \leq s \leq n-m)$$

eine eineindeutige Dezimalzahl t_s zu

Idee des Algorithmus von Rabin und Karp

- Ordne dem Muster $P[0, m]$ eine **eineindeutige** Dezimalzahl p zu
- Analog: ordne jedem Teilstring

$$T[s, m] \quad (0 \leq s \leq n-m)$$

eine eineindeutige Dezimalzahl t_s zu

- Offenbar gilt

$$P[0, m] = T[s, m] \quad \Leftrightarrow \quad p = t_s$$

also: s ist gültiger Versatz, wenn $p = t_s$.

Idee des Algorithmus von Rabin und Karp

- Ordne dem Muster $P[0, m]$ eine **eineindeutige** Dezimalzahl p zu
- Analog: ordne jedem Teilstring

$$T[s, m] \quad (0 \leq s \leq n-m)$$

eine eineindeutige Dezimalzahl t_s zu

- Offenbar gilt

$$P[0, m] = T[s, m] \quad \Leftrightarrow \quad p = t_s$$

also: s ist gültiger Versatz, wenn $p = t_s$.

- Sofern wir p in Zeit $\Theta(m)$ und **alle** (!) t_s in Zeit $\Theta(n-m+1)$ berechnen können, ist das String-Matching-Problem lösbar mit Zeitkomplexität

$$\Theta(m) + \Theta(n-m+1) = \Theta(n)$$

Berechnung der Wortgewichte

- p kann in $\Theta(m)$ berechnet werden; z.B. mit **Horners Regel**:

Berechnung der Wortgewichte

- p kann in $\Theta(m)$ berechnet werden; z.B. mit **Horners Regel**:

$$p = P[m-1] + 10 \cdot (P[m-2] + 10 \cdot (P[m-3] + \dots + 10 \cdot (P[1] + 10 \cdot P[0]) \dots))$$

Berechnung der Wortgewichte

- p kann in $\Theta(m)$ berechnet werden; z.B. mit **Horners Regel**:

$$p = P[m-1] + 10 \cdot (P[m-2] + 10 \cdot (P[m-3] + \dots + 10 \cdot (P[1] + 10 \cdot P[0]) \dots))$$

- Der Wert t_0 kann analog in $\Theta(m)$ berechnet werden

Berechnung der Wortgewichte

- p kann in $\Theta(m)$ berechnet werden; z.B. mit **Horners Regel**:

$$p = P[m-1] + 10 \cdot (P[m-2] + 10 \cdot (P[m-3] + \dots + 10 \cdot (P[1] + 10 \cdot P[0]) \dots))$$

- Der Wert t_0 kann analog in $\Theta(m)$ berechnet werden
- Es besteht nun folgender Zusammenhang zwischen t_s und t_{s+1} :

$$t_{s+1} = 10 \cdot (t_s - 10^{m-1} \cdot T[s]) + T[s+m-1]$$

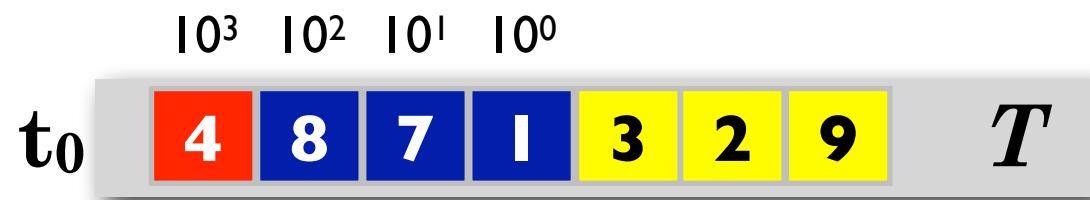
Berechnung der Wortgewichte

- p kann in $\Theta(m)$ berechnet werden; z.B. mit **Horners Regel**:

$$p = P[m-1] + 10 \cdot (P[m-2] + 10 \cdot (P[m-3] + \dots + 10 \cdot (P[1] + 10 \cdot P[0]) \dots))$$

- Der Wert t_0 kann analog in $\Theta(m)$ berechnet werden
- Es besteht nun folgender Zusammenhang zwischen t_s und t_{s+1} :

$$t_{s+1} = 10 \cdot (t_s - 10^{m-1} \cdot T[s]) + T[s+m-1]$$



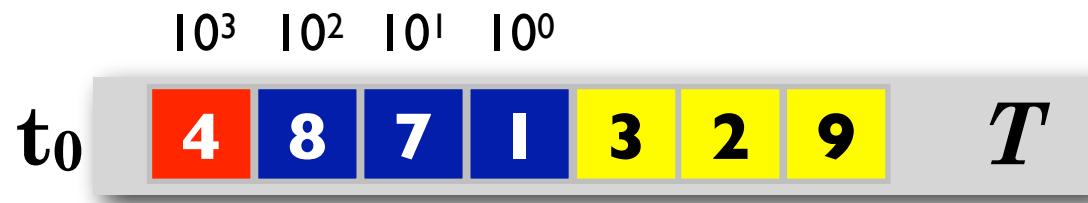
Berechnung der Wortgewichte

- p kann in $\Theta(m)$ berechnet werden; z.B. mit **Horners Regel**:

$$p = P[m-1] + 10 \cdot (P[m-2] + 10 \cdot (P[m-3] + \cdots + 10 \cdot (P[1] + 10 \cdot P[0]) \cdots))$$

- Der Wert t_0 kann analog in $\Theta(m)$ berechnet werden
 - Es besteht nun folgender Zusammenhang zwischen t_s und t_{s+1} :

$$t_{s+1} = 10 \cdot (t_s - 10^{m-1} \cdot T[s]) + T[s+m-1]$$



Optimierung des Verfahrens

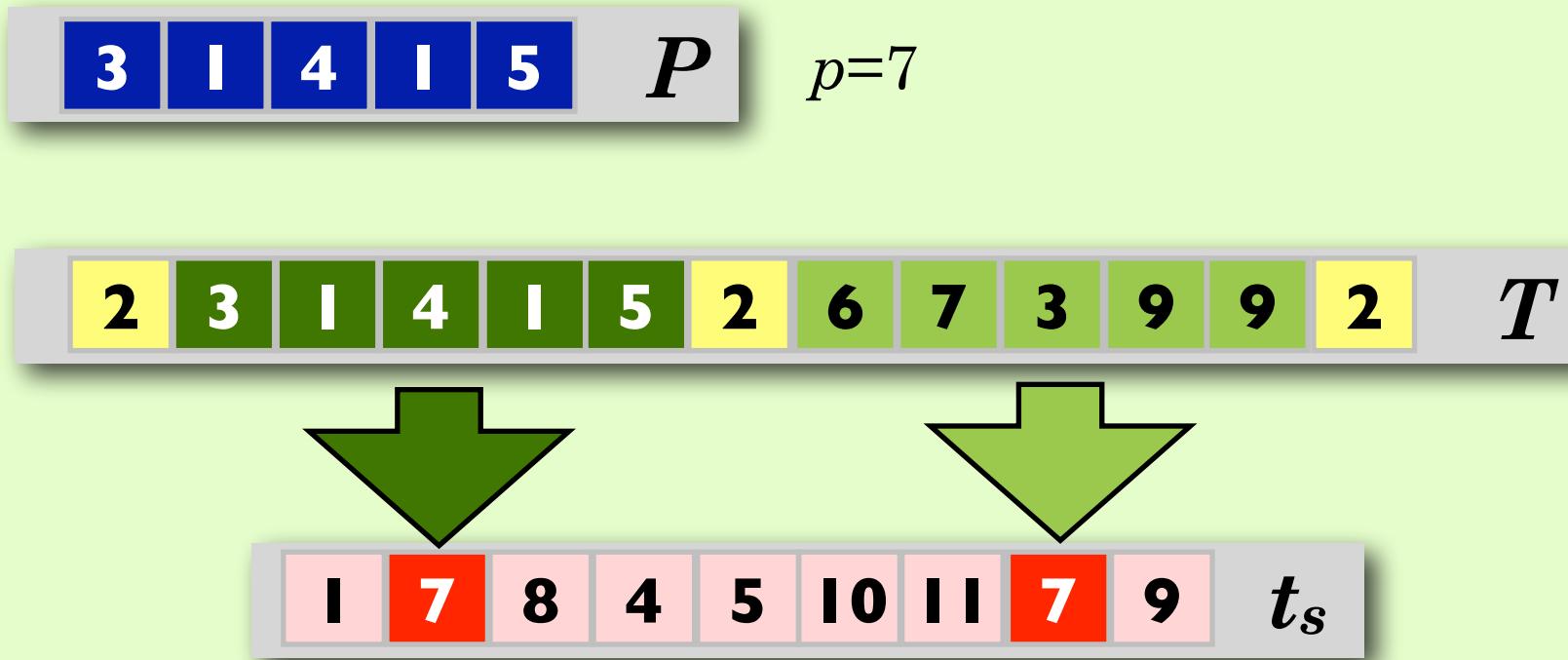
- **Problem:** Die Werte von p und t_s werden zu groß!
 - Annahme, dass t_{s+1} in konstanter Zeit berechnet werden kann zu optimistisch!
- **Lösung:** modulo q rechnen, wobei
 - q eine Primzahl ist
 - $10q$ in ein Maschinenwort passt;
genauer: $d \cdot q$ passt in ein Maschinenwort, wobei $d=|\Sigma|$
 - Dann: Berechnung von t_{s+1} in konstanter Zeit machbar!
 - $t_{s+1} \neq p \Rightarrow T[s, m] \neq P$ gilt zwar immer noch, aber
 - $t_{s+1} = p \Rightarrow T[s, m] = P$ gilt nicht mehr - exakter Vergleich notwendig!



Beispiel: Rabin-Karp Verfahren

B String-Matching mit Rabin-Karp

Es seien $d=10$ und $q=13$



VI. Suchen in Texten

6. Suchen in Texten

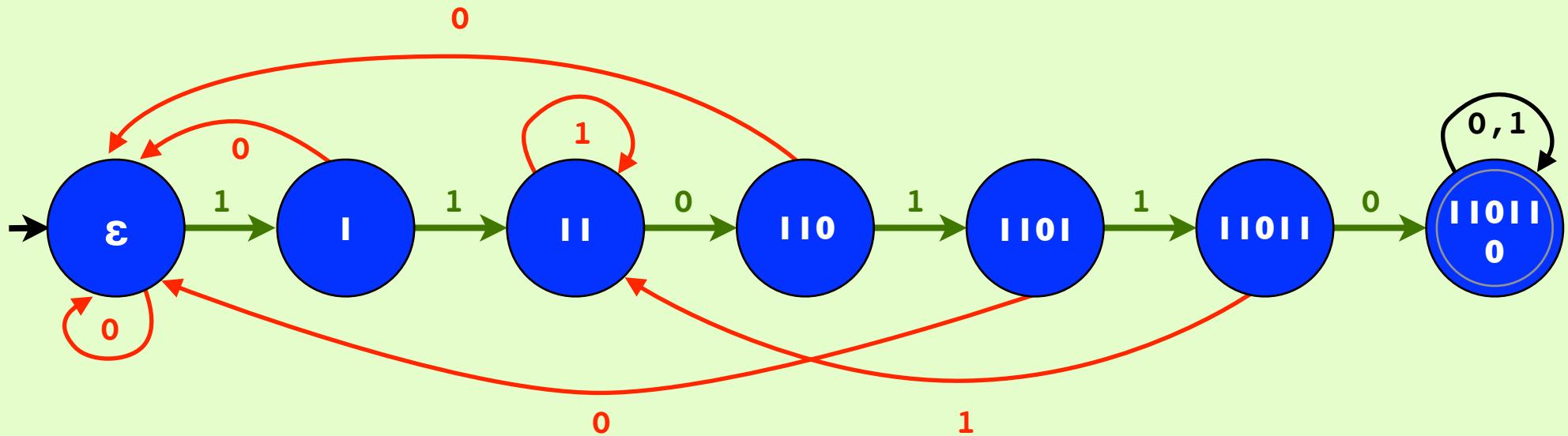
- 6.1. Einführung
- 6.2. Direkte Mustersuche
- 6.3. Das Verfahren von Rabin und Karp
- 6.4. Suche mit endlichen Automaten**
- 6.5. Das Verfahren von Knuth, Morris und Pratt
- 6.6. Das Verfahren von Boyer und Moore

Einführung

- **Aus der Theoretischen Informatik: Wortproblem!**
 - **Ausgangspunkt:** Sprache $L = \{w \in \Sigma^* \mid w \text{ enthält Muster } p\}$
 - **Gegeben:** Text $t \in \Sigma^*$
 - **Frage:** gilt $t \in L$ (also: enthält t das Muster p)?
 - **Lösung:** deterministische endlichen Automaten (DEA)!

B DEA

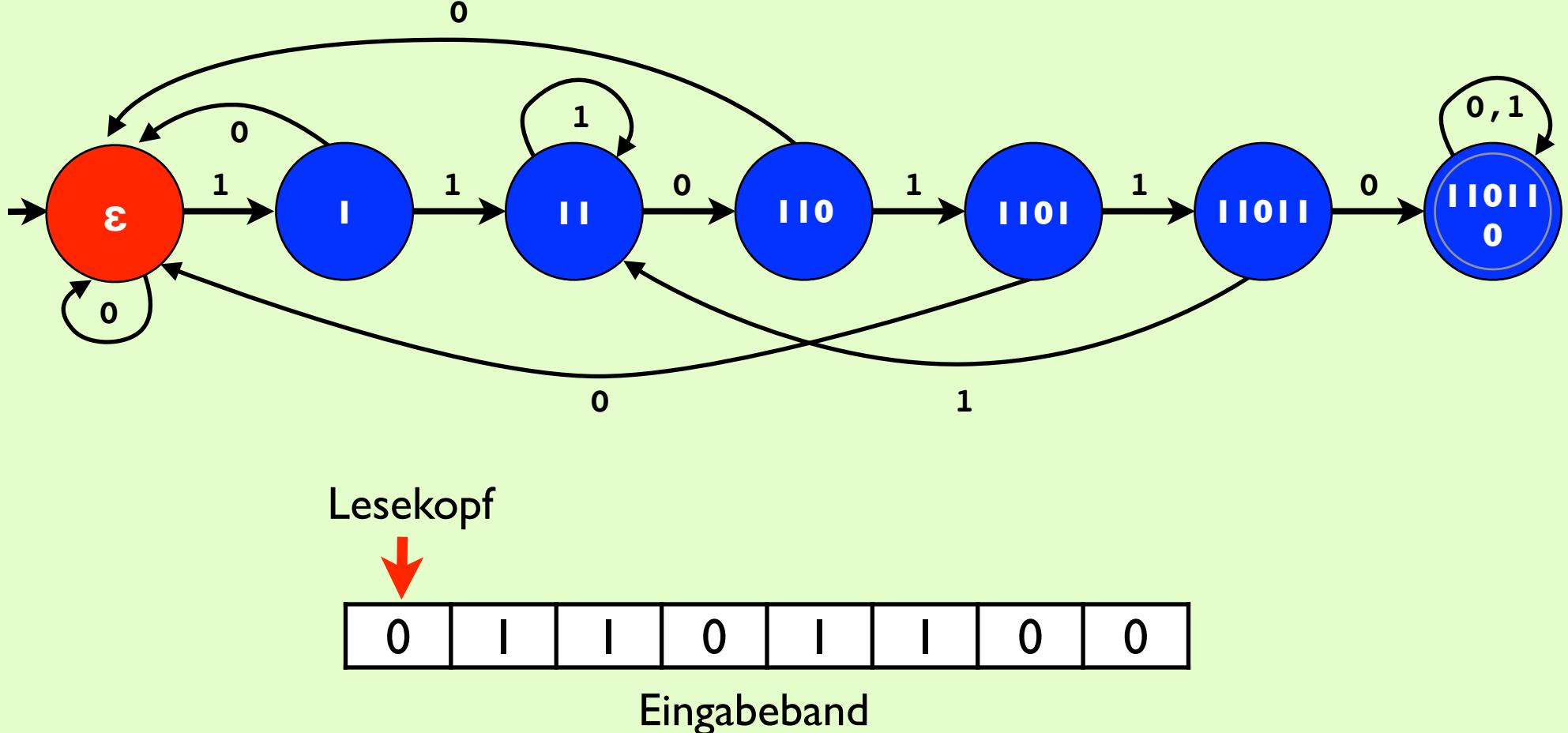
$L = \{w \in \Sigma_{\text{Bool}}^* \mid w \text{ enthält } 110110\}$



DEAs als „Suchmaschinen“

B DEA

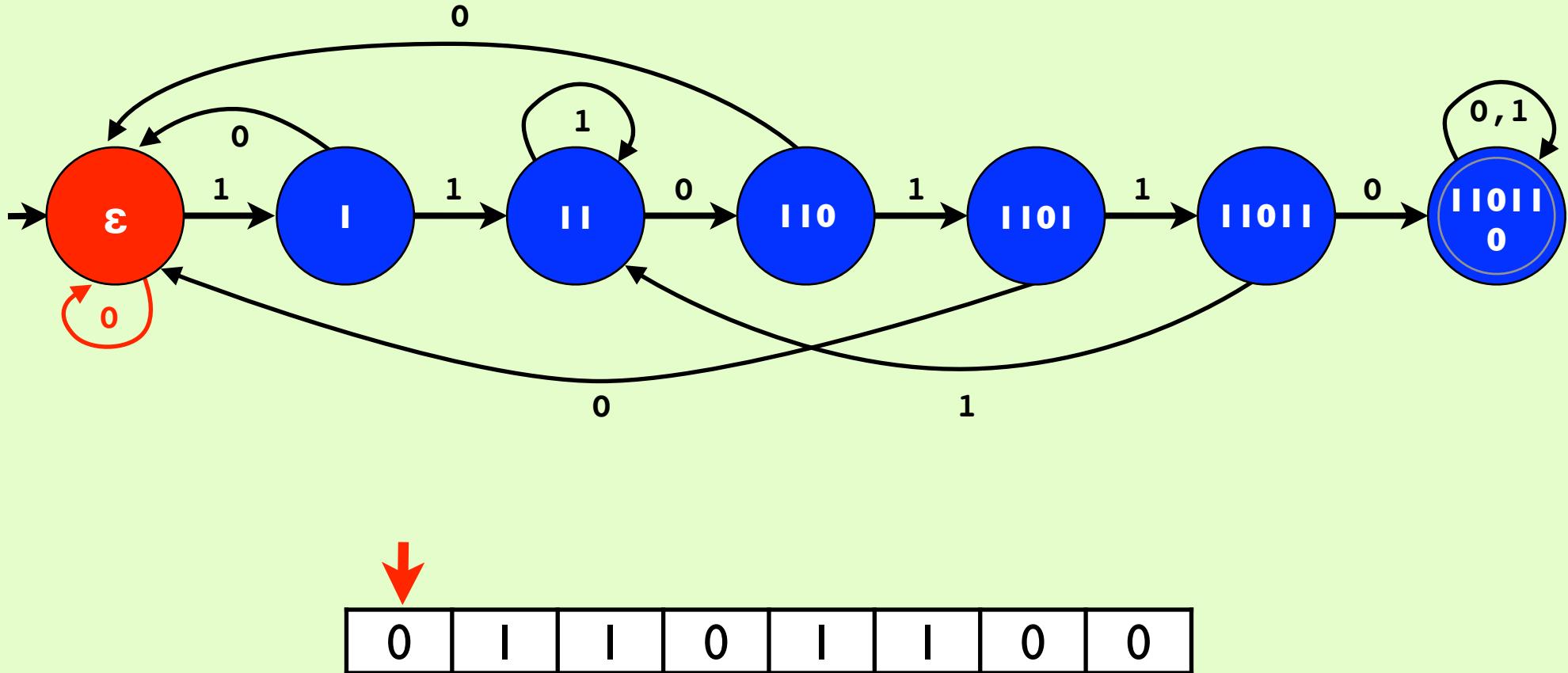
$$L = \{w \in \Sigma_{\text{Bool}}^* \mid w \text{ enthält } 110110\}$$



DEAs als „Suchmaschinen“

B DEA

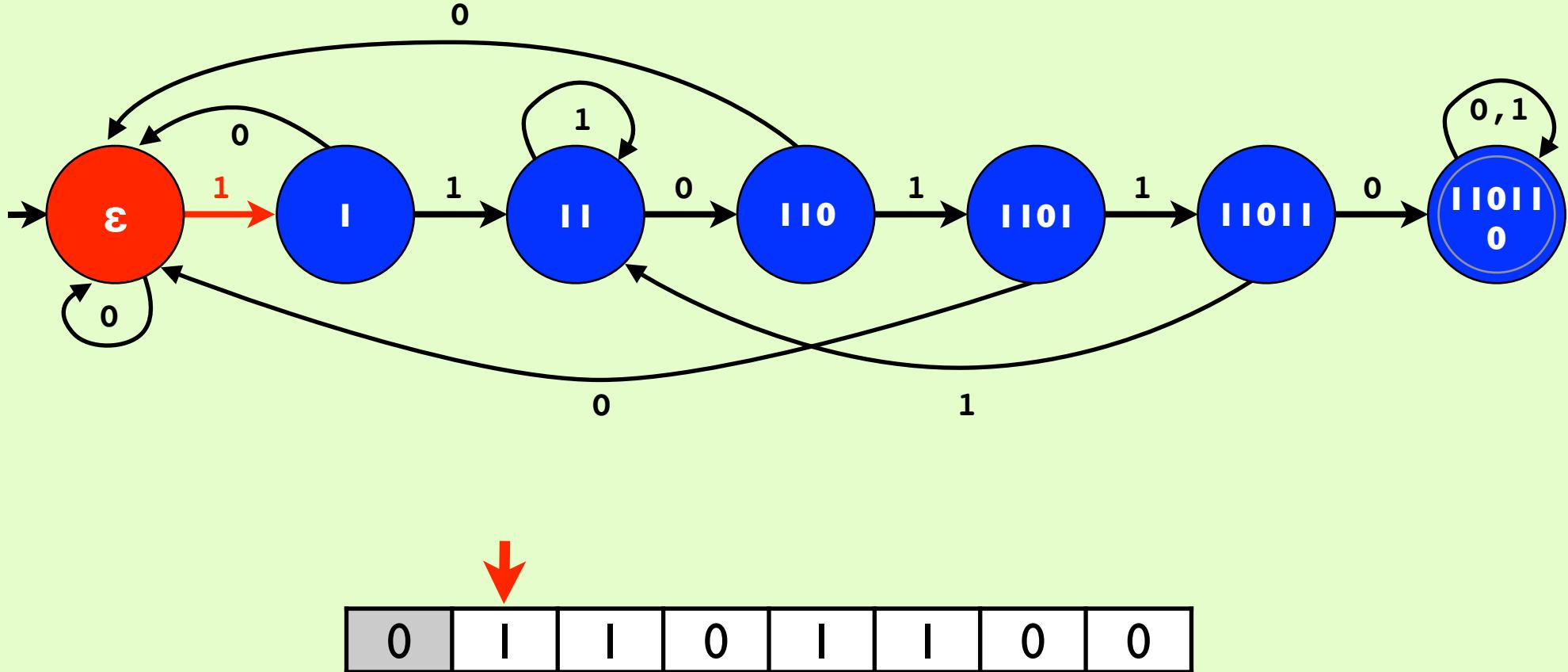
$$L = \{w \in \Sigma_{\text{Bool}}^* \mid w \text{ enthält } 110110\}$$



DEAs als „Suchmaschinen“

B DEA

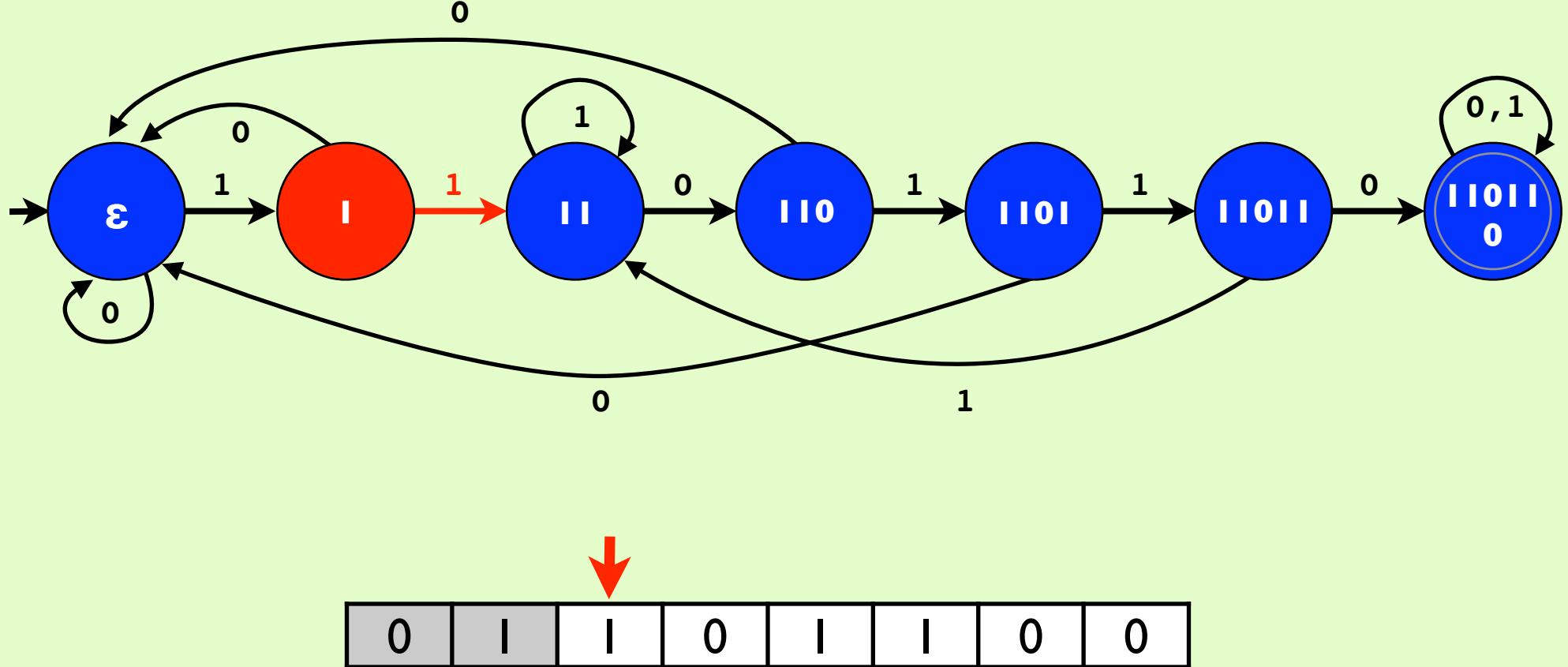
$$L = \{w \in \Sigma_{\text{Bool}}^* \mid w \text{ enthält } 110110\}$$



DEAs als „Suchmaschinen“

B DEA

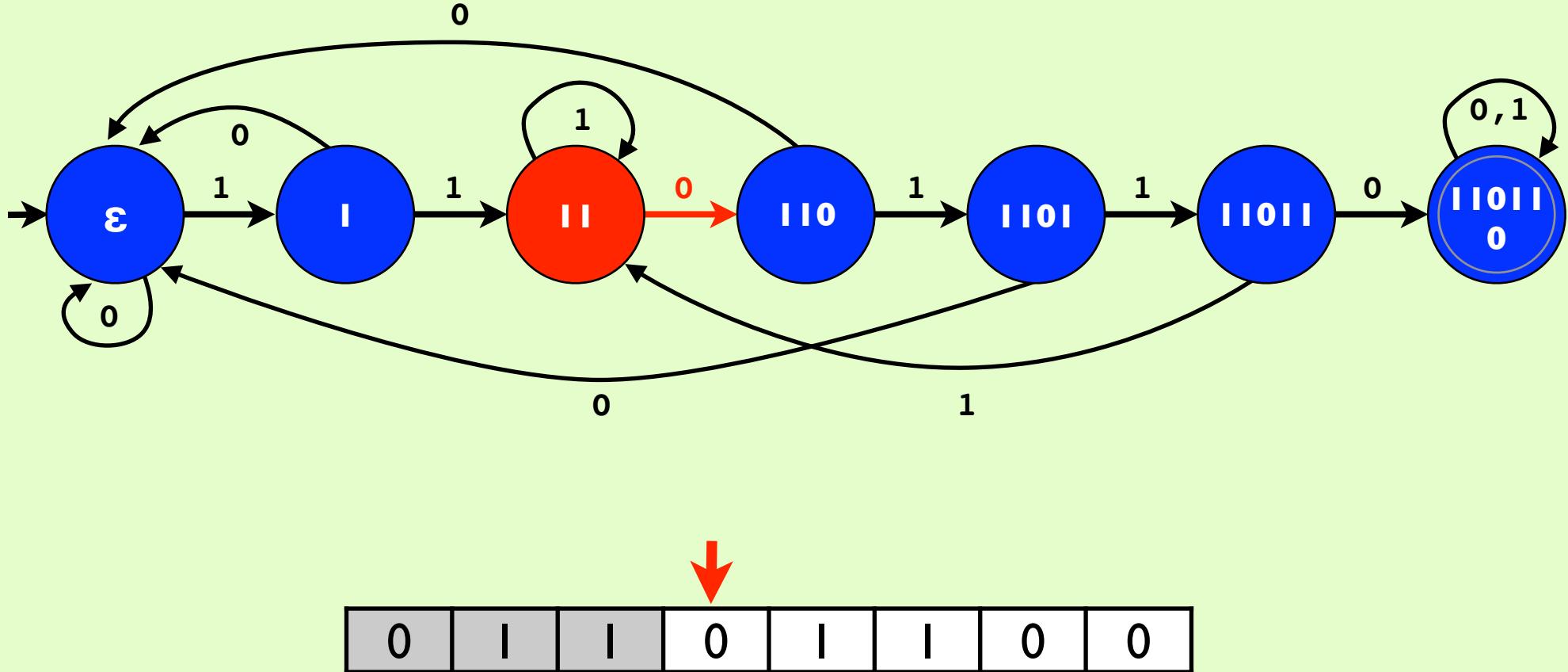
$$L = \{w \in \Sigma_{\text{Bool}}^* \mid w \text{ enthält } 110110\}$$



DEAs als „Suchmaschinen“

B DEA

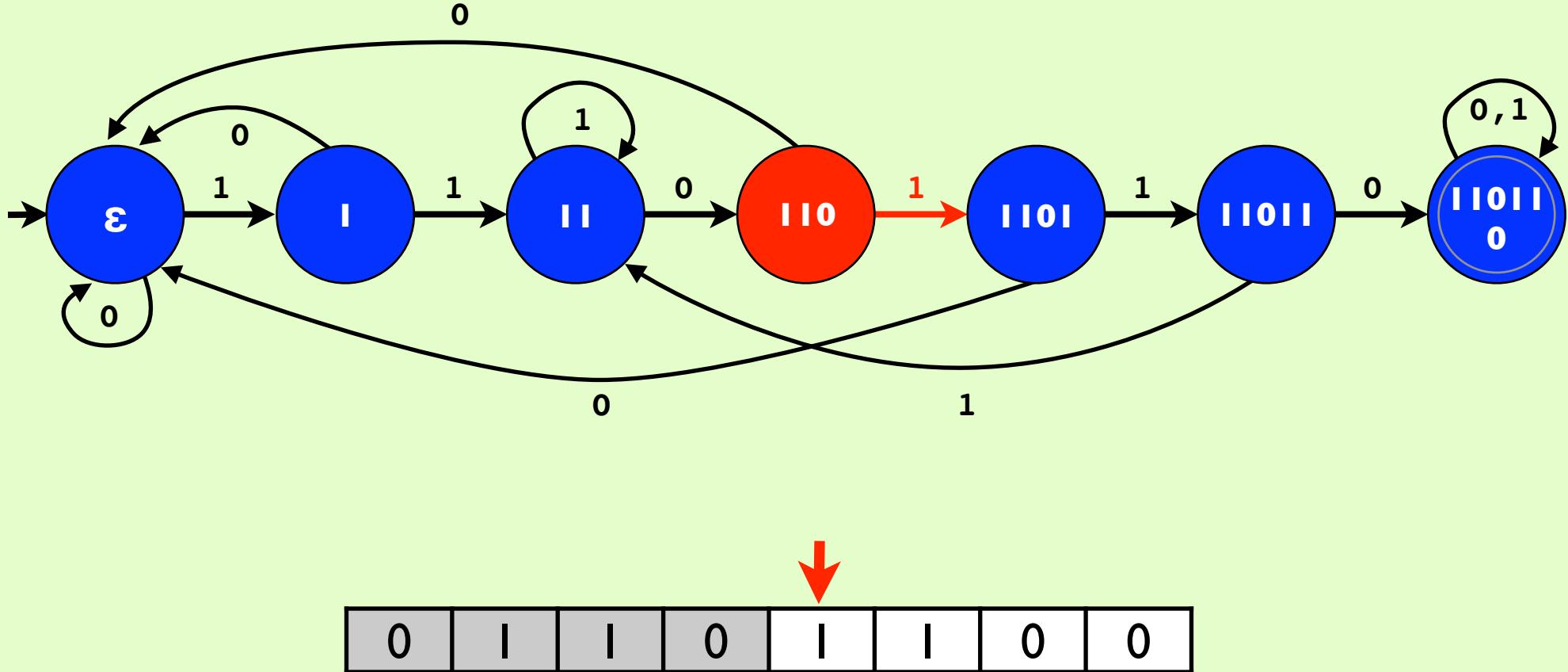
$$L = \{w \in \Sigma_{\text{Bool}}^* \mid w \text{ enthält } 110110\}$$



DEAs als „Suchmaschinen“

B DEA

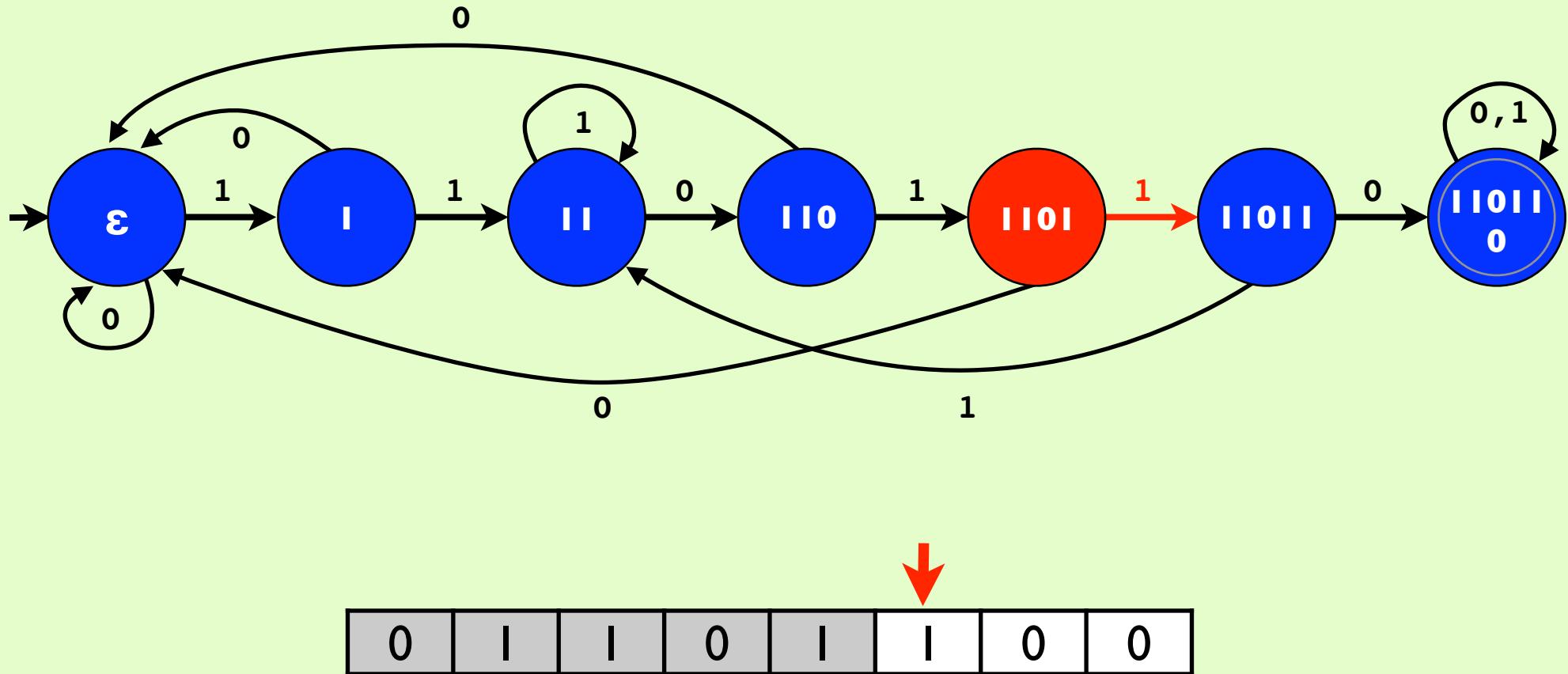
$$L = \{w \in \Sigma_{\text{Bool}}^* \mid w \text{ enthält } 110110\}$$



DEAs als „Suchmaschinen“

B DEA

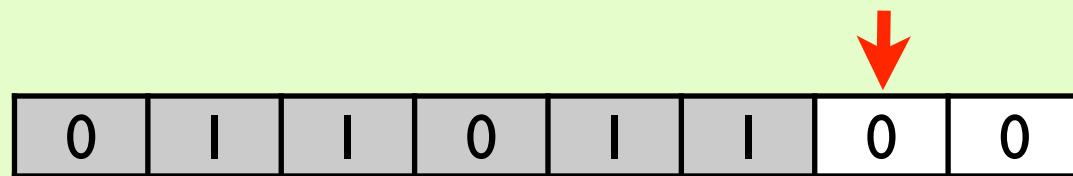
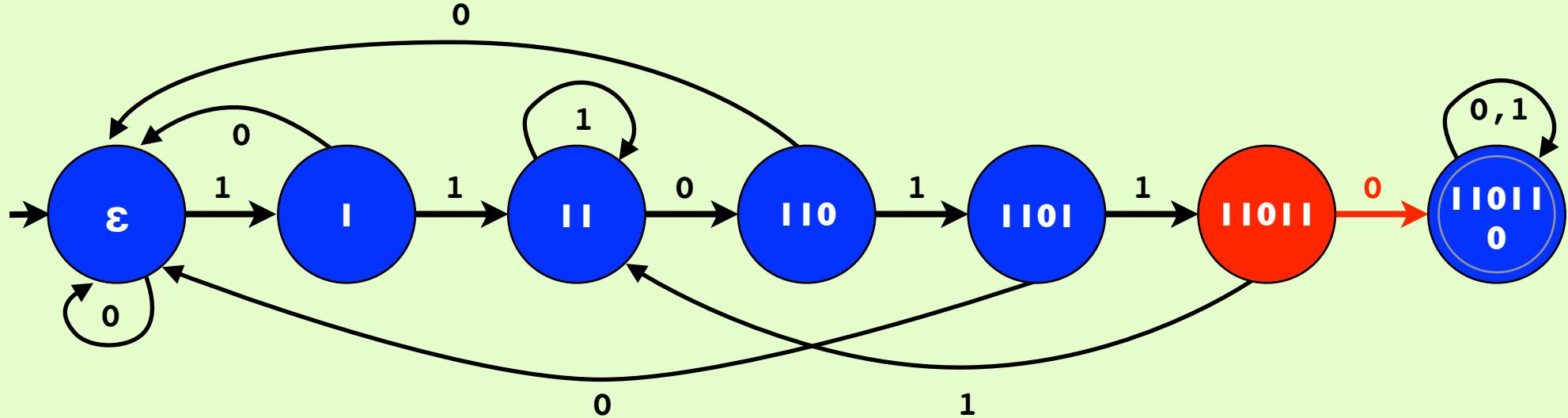
$$L = \{w \in \Sigma_{\text{Bool}}^* \mid w \text{ enthält } 110110\}$$



DEAs als „Suchmaschinen“

B DEA

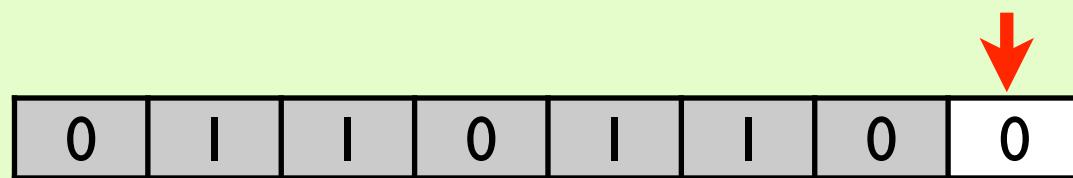
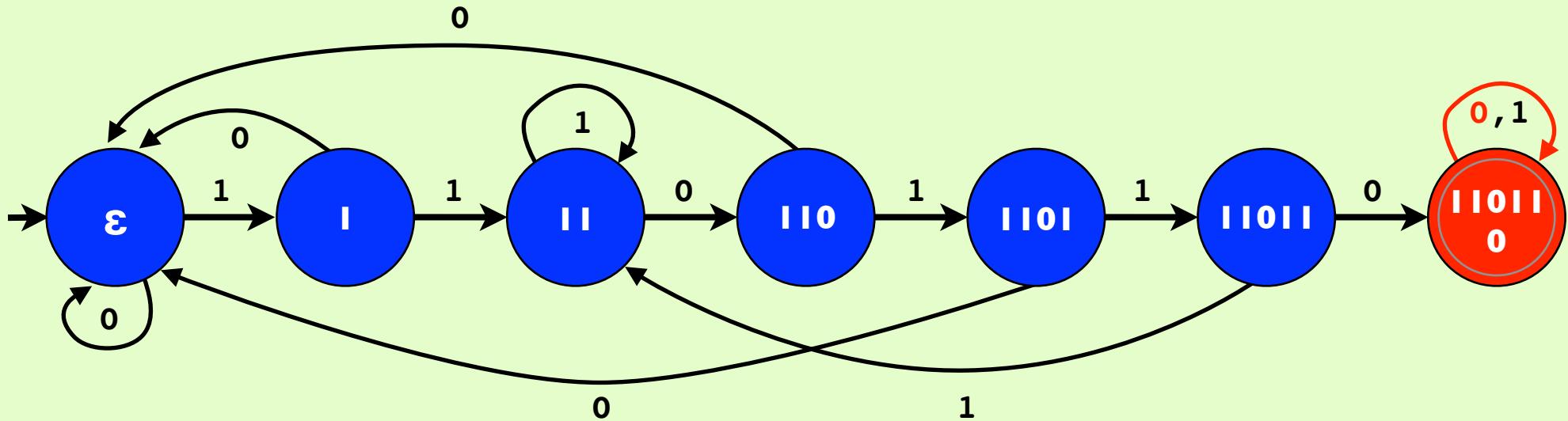
$$L = \{w \in \Sigma_{\text{Bool}}^* \mid w \text{ enthält } 110110\}$$



DEAs als „Suchmaschinen“

B DEA

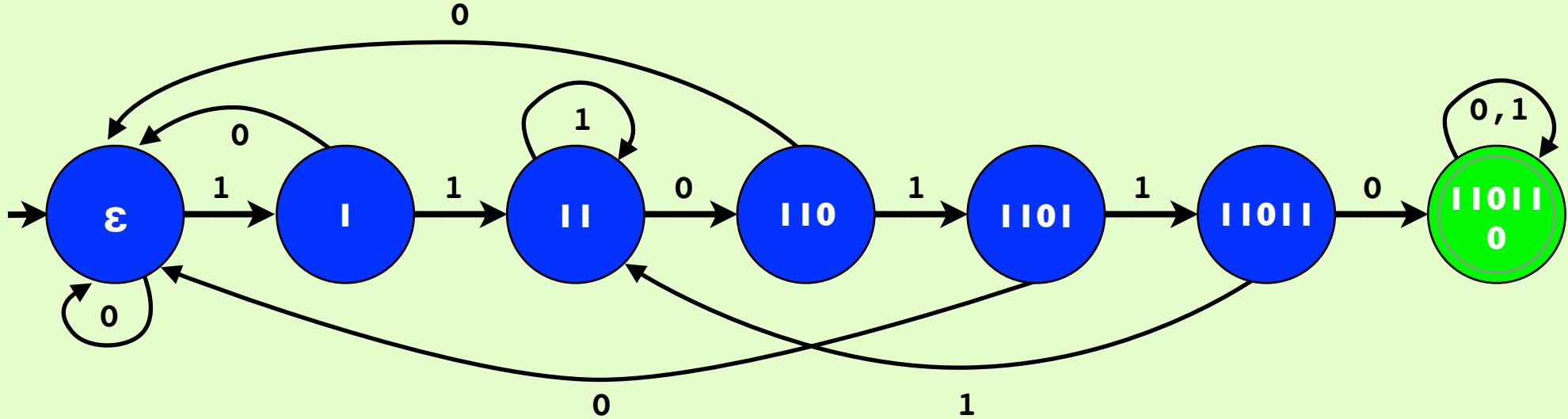
$$L = \{w \in \Sigma_{\text{Bool}}^* \mid w \text{ enthält } 110110\}$$



DEAs als „Suchmaschinen“

B DEA

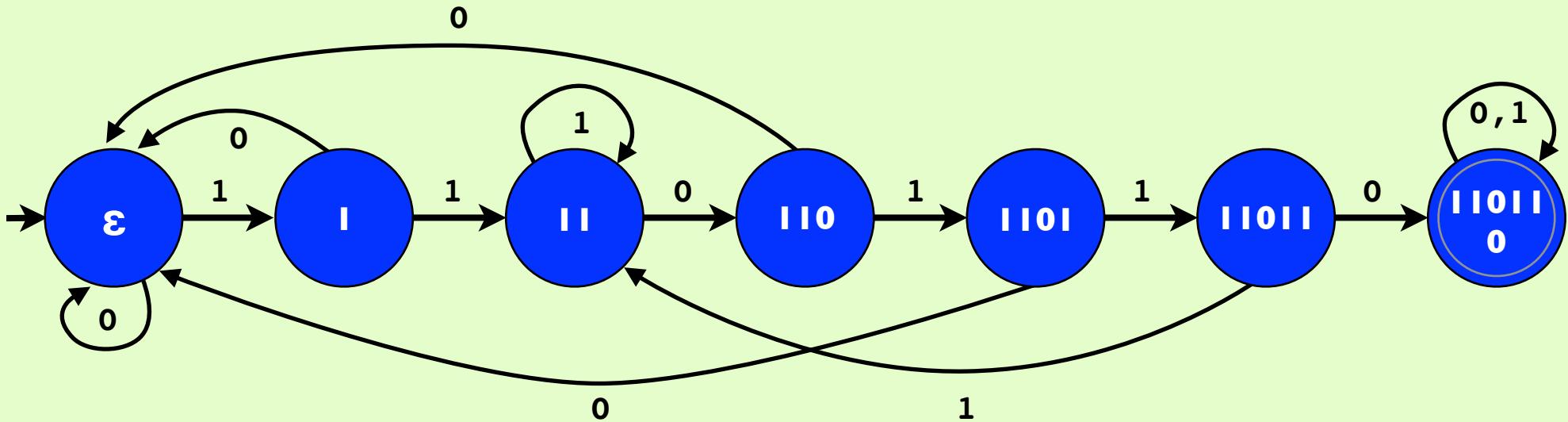
$$L = \{w \in \Sigma_{\text{Bool}}^* \mid w \text{ enthält } 110110\}$$



DEAs als „Suchmaschinen“ - Überlegungen

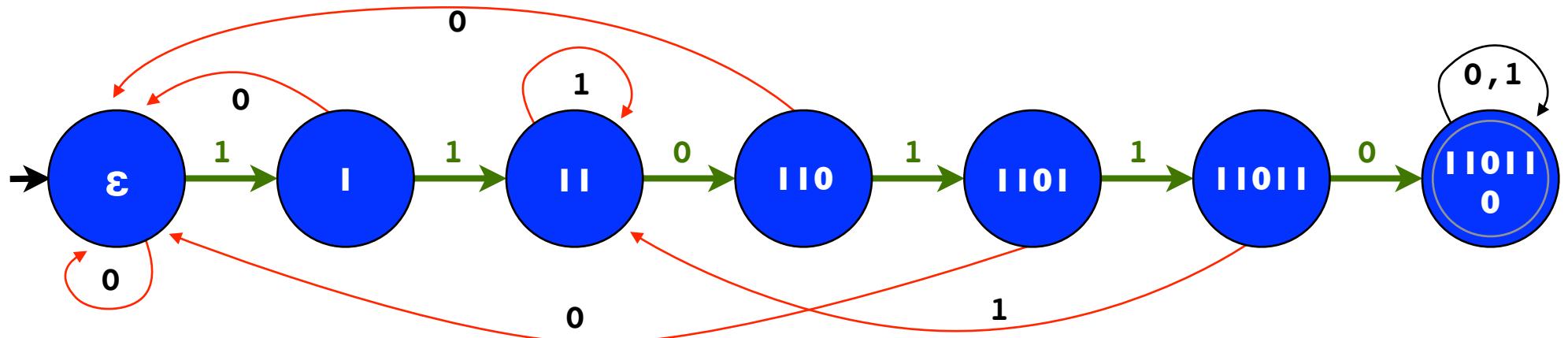
B DEA

$$L = \{w \in \Sigma_{\text{Bool}}^* \mid w \text{ enthält } 110110\}$$



- Automat liest jedes Zeichen des Textes genau einmal!
- Suche nach jedem gültigen Versatz demnach mit $O(n)$
- **Problem 1:** wie konstruiert man Automaten systematisch?
- **Problem 2:** Größe des Automaten abhängig vom Alphabet; daher: Konstruktion u.U. aufwendig!

DEA - Systematische Konstruktion



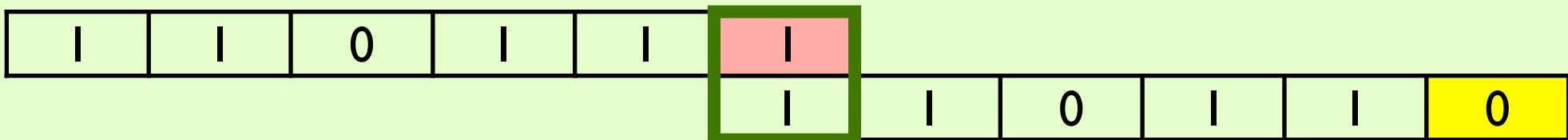
- Zustände des DEAs entsprechen bereits erkannten Musteranteil (Präfixe des Musters)
- Falls ein unerwartetes Zeichen an Position i kommt (rote Kanten):

Finde das längste Suffix der bereits gelesenen Eingabe $T[0, i]$, welches Präfix des Suchmusters ist - finde $P[0, j]$ mit

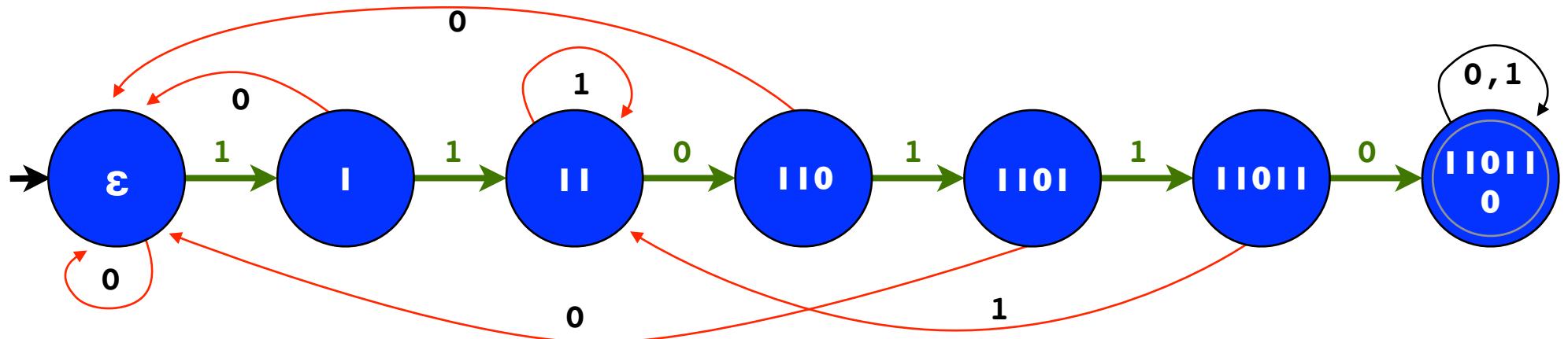
$$j = \max \{ k \in \{0, \dots, m-1\} \mid P[0, k] \sqsupseteq T[0, i] \}$$

B DEA

Mismatch nach Lesen von III0111:



DEA - Systematische Konstruktion



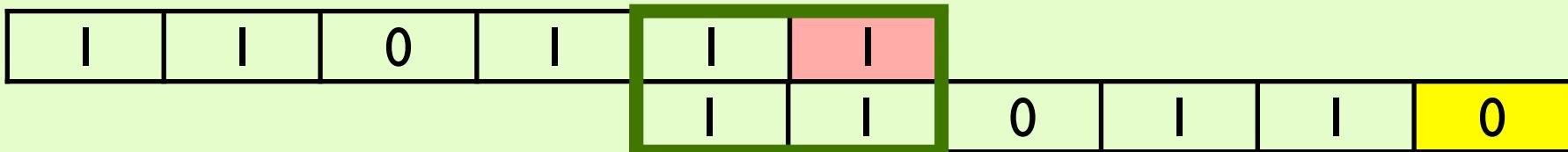
- Zustände des DEAs entsprechen bereits erkannten Musteranteil (Präfixe des Musters)
- Falls ein unerwartetes Zeichen an Position i kommt (rote Kanten):

Finde das längste Suffix der bereits gelesenen Eingabe $T[0, i]$, welches Präfix des Suchmusters ist - finde $P[0, j]$ mit

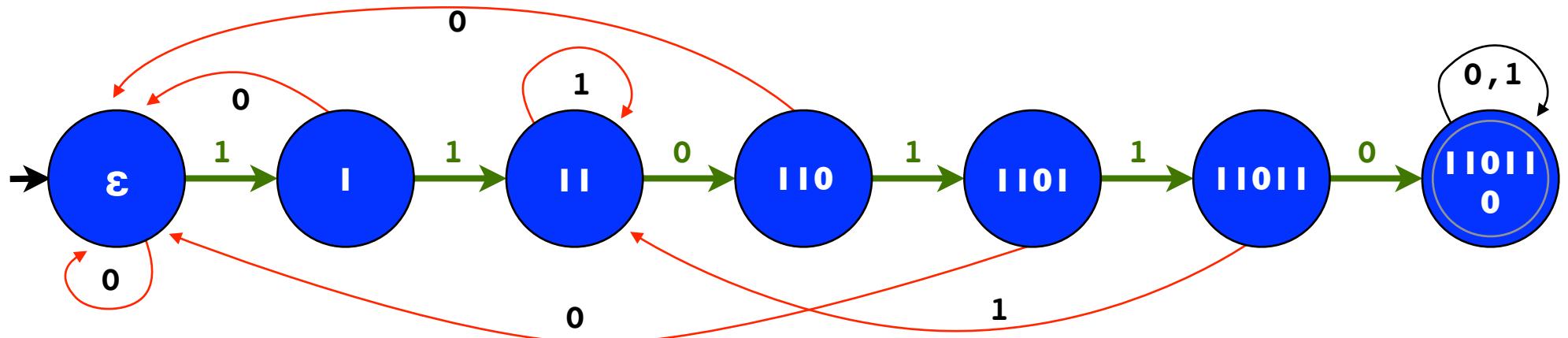
$$j = \max \{ k \in \{0, \dots, m-1\} \mid P[0, k] \sqsupseteq T[0, i] \}$$

B DEA

Mismatch nach Lesen von III0111:



DEA - Systematische Konstruktion



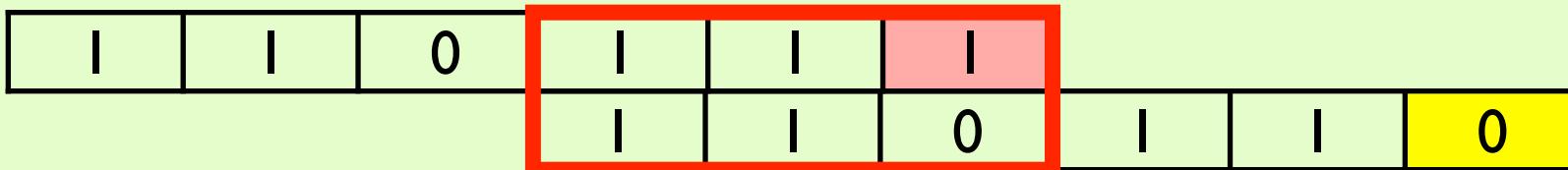
- Zustände des DEAs entsprechen bereits erkannten Musteranteil (Präfixe des Musters)
- Falls ein unerwartetes Zeichen an Position i kommt (rote Kanten):

Finde das längste Suffix der bereits gelesenen Eingabe $T[0, i]$, welches Präfix des Suchmusters ist - finde $P[0, j]$ mit

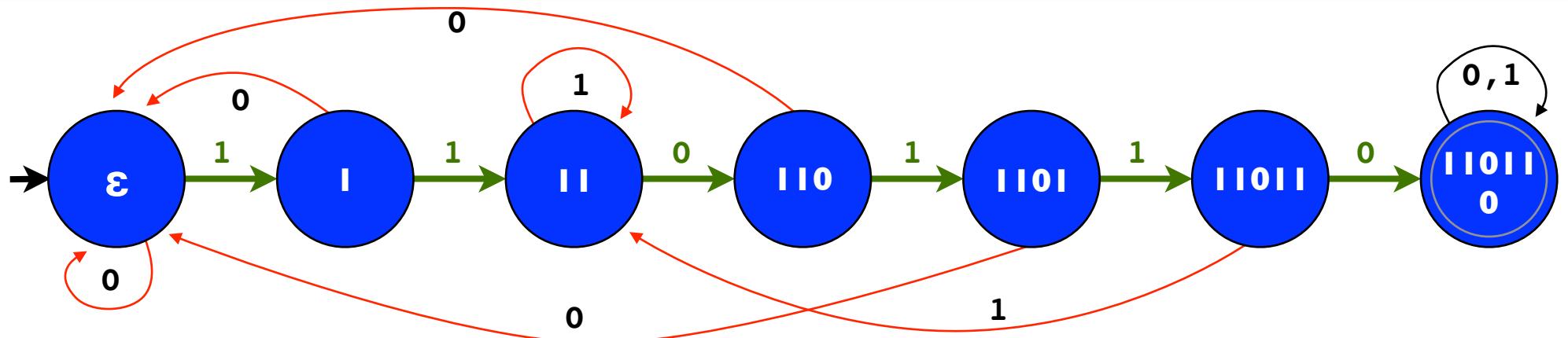
$$j = \max \{k \in \{0, \dots, m-1\} \mid P[0, k] \sqsupseteq T[0, i]\}$$

B DEA

Mismatch nach Lesen von III0111:



DEA - Systematische Konstruktion



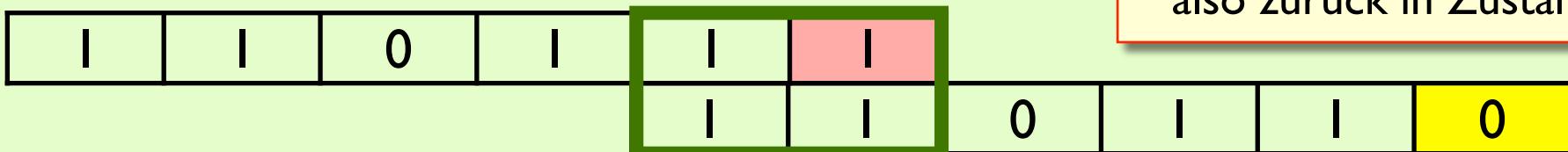
- Zustände des DEAs entsprechen bereits erkannten Musteranteil (Präfixe des Musters)
- Falls ein unerwartetes Zeichen an Position i kommt (rote Kanten):

Finde das längste Suffix der bereits gelesenen Eingabe $T[0, i]$, welches Präfix des Suchmusters ist - finde $P[0, j]$ mit

$$j = \max \{k \in \{0, \dots, m-1\} \mid P[0, k] \sqsupseteq T[0, i]\}$$

B DEA

Mismatch nach Lesen von III0III:



II ist dieses größte Suffix,
also zurück in Zustand II

Die Suffix-Funktion zu einem Muster P

- **Basierend auf unserer Beobachtung definieren wir folgende Hilfsfunktion:**

D Die Suffixfunktion zu einem Muster P

Sei Σ ein Alphabet und $P = p = a_0 \cdots a_{m-1}$ ein Muster ($p \in \Sigma^*$), dann ist die **Suffixfunktion zu P** $\sigma_p : \Sigma^* \rightarrow \Sigma^*$ definiert durch

$$\begin{aligned}\sigma_p(w) &= P[0, j] \text{ mit } j = \max \{k \in \{0, \dots, m-1\} \mid P[0, k] \sqsupseteq w\}. \\ &\quad (* \text{ Längstes Präfix von } p, \text{ das Suffix von } w \text{ ist } *)\end{aligned}$$

- **Diese Funktion induziert die Transitionsfunktion δ des endlichen Automaten**
(es gilt $\delta(q, a) = \sigma_p(qa)$)
- **Beachte: Zustände des Automaten sind Wörter**
(alle Präfixe des Musters)
- **Anmerkung: $\sigma_p(p)=p$**

Formale Beschreibung des DEA

D String-Matching-Automat

Sei Σ ein Alphabet und $P = p = a_0 \cdots a_{m-1}$ ein Muster ($p \in \Sigma^*$), dann ist der **String-Matching Automat zu P** ein deterministischer endlicher Automat $M = (Q, \Sigma, \delta, q_0, F)$ mit

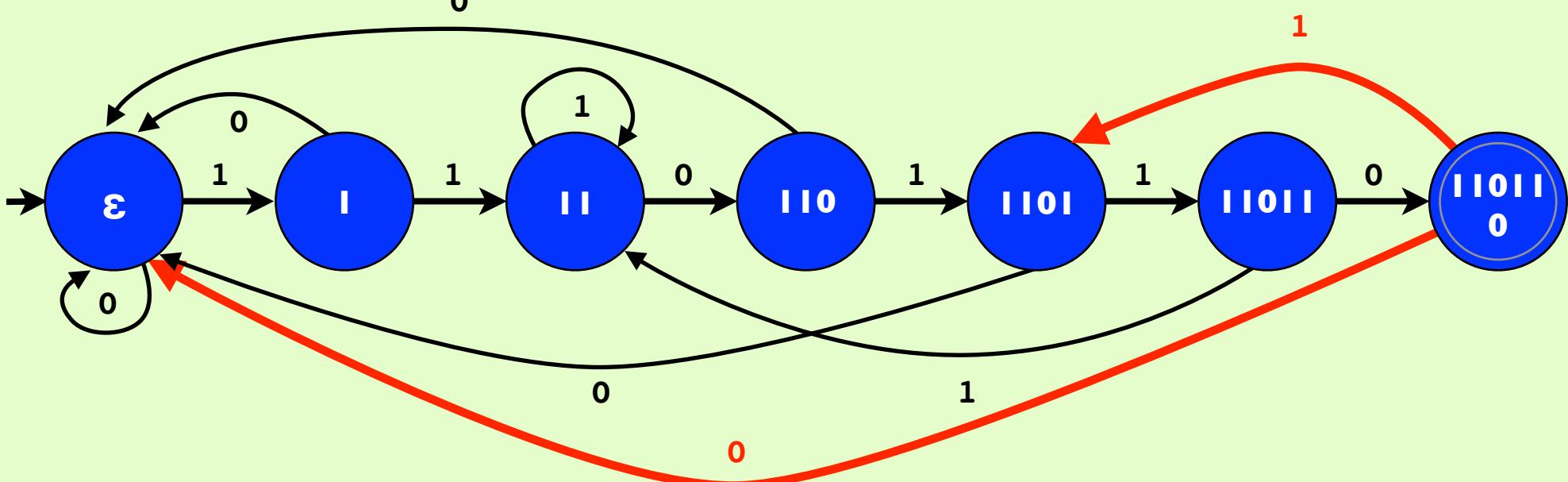
- $Q = \bigcup_{0 \leq i \leq m} P[0, i]$ (* Die Menge aller Präfixe von p *)
- $q_0 = \varepsilon$
- $F = \{p\}$
- $\delta(q, a) = \begin{cases} qa & \text{falls } qa \in Q \text{ (* } qa \text{ ist Präfix von } p \text{ *)} \\ \sigma_p(qa) & \text{sonst} \end{cases}$

- Zustände des Automaten sind Wörter!
- Fallunterscheidung bei eigentlich δ nicht erforderlich. Es ist $\delta(q, a) = \sigma_p(qa)$
- Automat bleibt nicht im akzeptierenden Zustand stehen (siehe nächste Folie)
 - Vorstellung: Automat „sendet Signal“, wenn akzeptierender Zustand erreicht (Muster gefunden)

DEAs als „Suchmaschinen“ - Überlegungen

B Tatsächlicher konstruierter DEA:

$$L = \{w \in \Sigma_{\text{Bool}}^* \mid w \text{ enthält } 110110\}$$



- **Idee:** Automat „feuert“ bei Eintritt in akzeptierenden Zustand ein Ereignis und rechnet weiter

Eigenschaften von σ_p

S Eigenschaften der Suffix-Funktion zu p

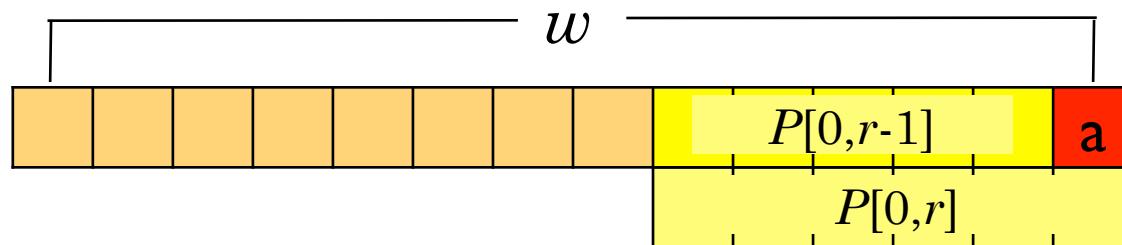
Sei Σ ein Alphabet und $p \in \Sigma^*$ ein Muster. σ_p , die Suffix-Funktion zu p , erfüllt folgende Eigenschaften

1. $|\sigma_p(wa)| \leq |\sigma_p(w)| + 1$
2. $q = \sigma_p(w) \Rightarrow \sigma_p(wa) = \sigma_p(qa)$

wobei $w, q \in \Sigma^*$ und $a \in \Sigma$.

Beweis:

1. Sei $r = |\sigma_p(wa)|$. Gilt $r = 0$, so gilt die Behauptung trivialerweise, denn $|\varepsilon| = 0 \leq |\sigma_p(w)| + 1$, da $|\sigma_p(w)|$ immer positiv ist. Sei $r > 0$, dann folgt aus der Definition von σ_p , dass $P[0, r] \sqsupseteq wa$ - folglich ist $P[0, r-1] \sqsupseteq w$. Damit ist $r-1 \leq |\sigma_p(w)|$, denn $\sigma_p(w)$ liefert das größte Suffix von w . Schließlich erhalten wir $|\sigma_p(wa)| = r \leq \sigma_p(w) + 1$.



Eigenschaften von σ_p

S Eigenschaften der Suffix-Funktion zu p

Sei Σ ein Alphabet und $p \in \Sigma^*$ ein Muster. σ_p , die Suffix-Funktion zu p , erfüllt folgende Eigenschaften

1. $|\sigma_p(wa)| \leq |\sigma_p(w)| + 1$
2. $q = \sigma_p(w) \Rightarrow \sigma_p(wa) = \sigma_p(qa)$

wobei $w, q \in \Sigma^*$ und $a \in \Sigma$.

Beweis:

1. Trivial: Das längste Suffix von wa , welches Präfix von p ist, ist höchstens ein Zeichen länger als das längste Suffix von w , das Präfix von p ist.
2. Wenn q längstes Suffix von w ist, welches Präfix von p ist, dann sind die längsten Suffixe von wa und qa , die gleichzeitig Präfix von p sind, identisch.

Aus $q = \sigma_p(w)$ folgt $q \sqsupseteq w$. Trivialerweise gilt dann auch $qa \sqsupseteq wa$. Sei $u = \sigma_p(wa)$ dann ist $|u| \leq |q| + 1$; denn es gilt $|\sigma_p(wa)| \leq |\sigma_p(w)| + 1$. Nun haben wir:

$$qa \sqsupseteq wa \quad \wedge \quad u \sqsupseteq wa \quad \wedge \quad |u| \leq |wa|$$

was $u \sqsupseteq qa$ impliziert. Damit folgt einerseits $|u| \leq |\sigma_p(qa)|$ (es könnte noch ein größeres Suffix geben, das Präfix von p ist) bzw. $|\sigma_p(wa)| \leq |\sigma_p(qa)|$ und wegen $qa \sqsupseteq wa$ andererseits auch $|\sigma_p(qa)| \geq |\sigma_p(wa)|$ - also ist $\sigma_p(qa) = \sigma_p(wa)$, da es nur ein Suffix einer bestimmten Länge geben kann.

Korrektheit von M

Sei $M = (Q, \Sigma, \delta, q_0, F)$ ein deterministischer endlicher Automat. Wie üblich, definieren wir die kanonische Fortsetzung von $\delta : Q \times \Sigma \rightarrow Q$ auf Wörter als $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$ wie folgt:

- $\hat{\delta}(q, \varepsilon) = q$
- $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$

für $q \in Q$, $w \in \Sigma^*$ und $a \in \Sigma$.

S Korrektheit von M

Sei $M = (Q, \Sigma, \delta, q_0, F)$ ein String-Matching-Automat für p . Dann gilt:

$$\hat{\delta}(q_0, w) = \sigma_p(w)$$

Beweis: Induktion über die Wortlänge von w .

Induktionsanfang: $\hat{\delta}(q_0, \varepsilon) = q_0 = \sigma_p(\varepsilon)$.

Induktionsschritt. Sei $u = wa$ und gelte $\hat{\delta}(q_0, w) = a_1 \cdots a_n = q = \sigma_p(w)$.

$$\begin{aligned}\hat{\delta}(q_0, wa) &= \delta(\hat{\delta}(q_0, w), a) && (* \text{ Definition von } \hat{\delta} *) \\ &= \delta(q, a) && (* \text{ Definition von } q *) \\ &= \sigma_p(qa) && (* \text{ Definition von } \delta *) \\ &= \sigma_p(wa) && (* \text{ Satz über Eigenschaften von } \sigma_p *)\end{aligned}$$

Mit $q, w \in \Sigma^*$, $a, a_1, \dots, a_n \in \Sigma$ und $q_0 = \varepsilon \in Q$.

Berechnung der Transitionsfunktion (C++)

- Die „naive“ Berechnung der Transitionsfunktion ist recht aufwendig:
- **Problem:** Abhangigkeit von der Groe des Alphabets!

Berechnung der Transitionsfunktion (C++)

- Die „naive“ Berechnung der Transitionsfunktion ist recht aufwendig:

```
typedef pair<unsigned, char> domain; // Definitionsbereich der Transitionsfunktion
                                         // Beachte: Zustand entspricht Wortlänge!
map<domain, unsigned> calcDelta(const string& alphabet, const string& p) {
    map<domain, unsigned> delta;
    int m = p.length();                      // Länge des Musters
    for (int ql=0 ; ql<=m ; ql++) {          // Alle Präfixe des Musters (incl. epsilon)
        string q = p.substr(0,ql);            // Präfix als String
        for (unsigned int c=0 ; c<alphabet.length() ; c++) { // Alle Zeichen des Alphabets
            string qc = q+alphabet[c];         // prospektiver Eingabestring
            // k - Länge des Präfixes des Musters, welches gleichzeitig
            // Suffix von qc sein soll. Größtmögliche k ist entweder |q|+1
            // Zeichen lang, oder, falls q=m, gerade m Zeichen.
            int k = ql+1 > m ? m+1 : ql+2; // (+1 wegen anschließendem Dekrement)
            do {
                k=k-1; // Präfix verkürzen
            } while ( p.substr(0,k) != qc.substr(ql-k+1,k) ); // bis es Suffix von qc ist
            delta[domain(ql,alphabet[c])] = k;
        }
    }
    return delta;
}
```

- **Problem:** Abhängigkeit von der Größe des Alphabets!

Berechnung der Transitionsfunktion (C++)

- Die „naive“ Berechnung der Transitionsfunktion ist recht aufwendig:

```
typedef pair<unsigned, char> domain; // Definitionsbereich der Transitionsfunktion
                                         // Beachte: Zustand entspricht Wortlänge!
map<domain, unsigned> calcDelta(const string& alphabet, const string& p) {
    map<domain, unsigned> delta;
    int m = p.length();                      // Länge des Musters
    for (int ql=0 ; ql<=m ; ql++) {          // Alle Präfixe des Musters (incl. epsilon)
        string q = p.substr(0,ql);            // Präfix als String
        for (unsigned int c=0 ; c<alphabet.length() ; c++) { // Alle Zeichen des Alphabets
            string qc = q+alphabet[c];         // prospektiver Eingabestring
            // k - Länge des Präfixes des Musters, welches gleichzeitig
            // Suffix von qc sein soll. Größtmögliche k ist entweder |q|+1
            // Zeichen lang, oder, falls q=m, gerade m Zeichen.
            int k = ql+1 > m ? m+1 : ql+2; // (+1 wegen anschließendem Dekrement)
            do {
                k=k-1; // Präfix verkürzen
            } while ( p.substr(0,k) != qc.substr(ql-k+1,k) ); // bis es Suffix von qc ist
            delta[domain(ql,alphabet[c])] = k;
        }
    }
    return delta;
}
```

Initialisierungsaufwand: $O(m^3 \cdot |\Sigma|)$

- Problem: Abhängigkeit von der Größe des Alphabets!

Simulation des DEA (C++)

- Das Ausführen des DEA lässt sich nun leicht kodieren:
- Anwendung:

Simulation des DEA (C++)

- Das Ausführen des DEA lässt sich nun leicht kodieren:

```
void stringMatchDEA(vector<unsigned>& result,      // Versätze
                     map<domain,unsigned>& delta, // Transitionsfunktion
                     const string& t,           // Text
                     const string& p) { // Muster
    unsigned state=0; // Wir starten beim leeren Wort (hat Länge 0)
    result.clear(); // Ergebnisvektor zurücksetzen
    for (unsigned i=0 ; i<t.length() ; ++i) { // Automat liest jeden Buchstaben
        // Neuen Zustand mithilfe der Transitionsfunktion ermitteln
        state = delta[domain(state,t[i])];
        if (state == p.length()) { // Ist es der akzeptierende Zustand?
            // ja: Index merken, an dem Muster gefunden wurde
            result.push_back(i-p.length()+1);
        }
    }
}
```

- Anwendung:

```
map<domain,unsigned> delta=calcDelta("bla","bla");
vector<unsigned> positions;

stringMatchDEA(positions,delta,"blablablablaaabla","bla");
for (unsigned i=0 ; i<positions.size() ; i++)
    cout << positions[i] << " ";
```

Simulation des DEA (C++)

- Das Ausführen des DEA lässt sich nun leicht kodieren:

```
void stringMatchDEA(vector<unsigned>& result,      // Versätze
                     map<domain,unsigned>& delta, // Transitionsfunktion
                     const string& t,           // Text
                     const string& p) { // Muster
    unsigned state=0; // Wir starten beim leeren Wort (hat Länge 0)
    result.clear(); // Ergebnisvektor zurücksetzen
    for (unsigned i=0 ; i<t.length() ; ++i) { // Automat liest jeden Buchstaben
        // Neuen Zustand mithilfe der Transitionsfunktion ermitteln
        state = delta[domain(state,t[i])];
        if (state == p.length()) { // Ist es der akzeptierende Zustand?
            // ja: Index merken, an dem Muster gefunden wurde
            result.push_back(i-p.length()+1);
        }
    }
}
```

Komplexität:

$$O(n + m^3 \cdot |\Sigma|)$$

- Anwendung:

```
map<domain,unsigned> delta=calcDelta("bla","bla");
vector<unsigned> positions;

stringMatchDEA(positions,delta,"blablablablaaaabla","bla");
for (unsigned i=0 ; i<positions.size() ; i++)
    cout << positions[i] << " ";
```

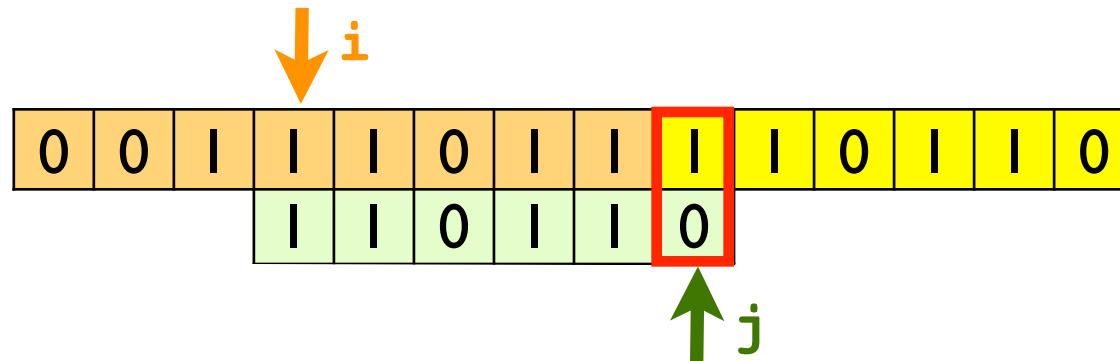
VI. Suchen in Texten

6. Suchen in Texten

- 6.1. Einführung
- 6.2. Direkte Mustersuche
- 6.3. Das Verfahren von Rabin und Karp
- 6.4. Suche mit endlichen Automaten
- 6.5. Das Verfahren von Knuth, Morris und Pratt**
- 6.6. Das Verfahren von Boyer und Moore

Grundidee des KMP-Vefahrens

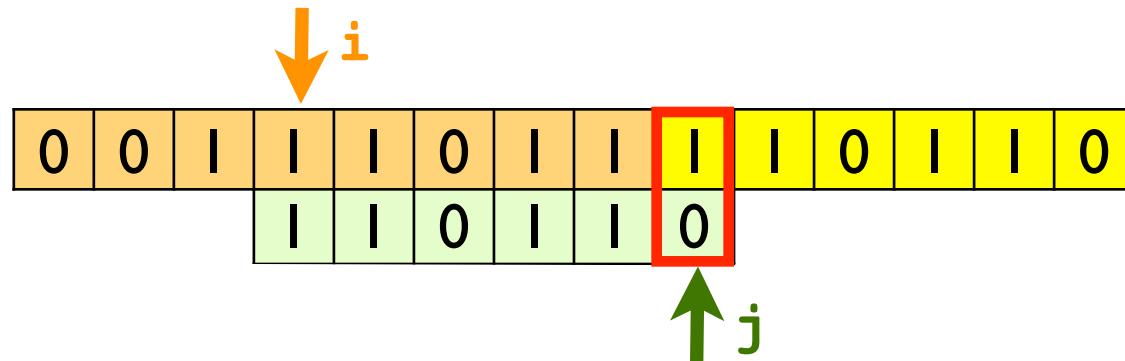
- Betrachten wir die folgende Situation:



- Es kommt zu einem **Mismatch** an Musterposition $j=5$

Grundidee des KMP-Vefahrens

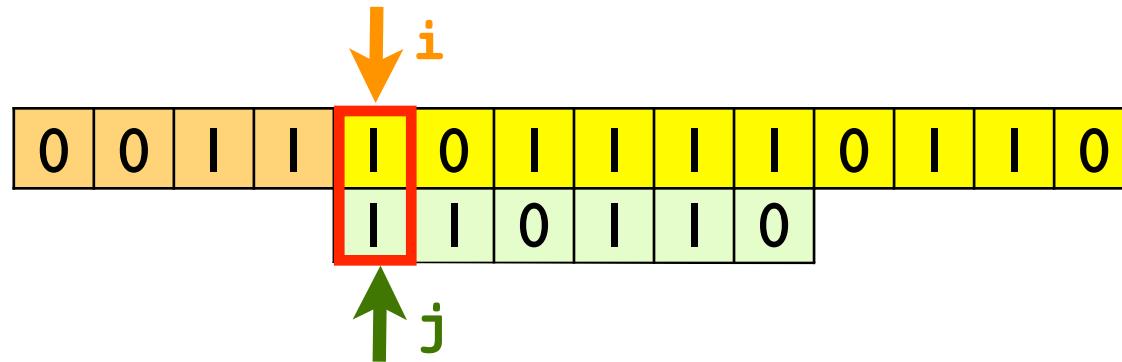
- Betrachten wir die folgende Situation:



- Es kommt zu einem **Mismatch an Musterposition $j=5$**
 - Im **naiven Algorithmus** (Direkte Mustersuche) würde das Muster jetzt um eine Stelle nach rechts geschoben - j würde auf 0 gesetzt und i inkrementiert

Grundidee des KMP-Vefahrens

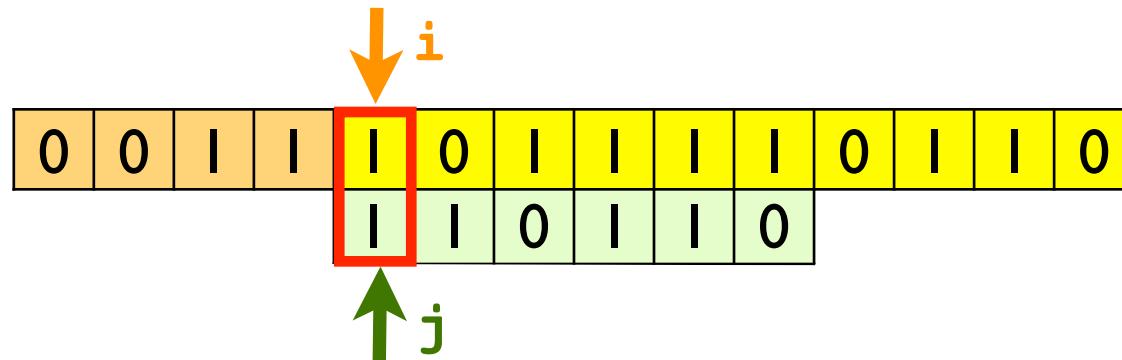
- Betrachten wir die folgende Situation:



- Es kommt zu einem **Mismatch** an Musterposition $j=5$
 - Im **naiven Algorithmus** (Direkte Mustersuche) würde das Muster jetzt um **eine** Stelle nach rechts geschoben - j würde auf 0 gesetzt und i inkrementiert

Grundidee des KMP-Vefahrens

- Betrachten wir die folgende Situation:



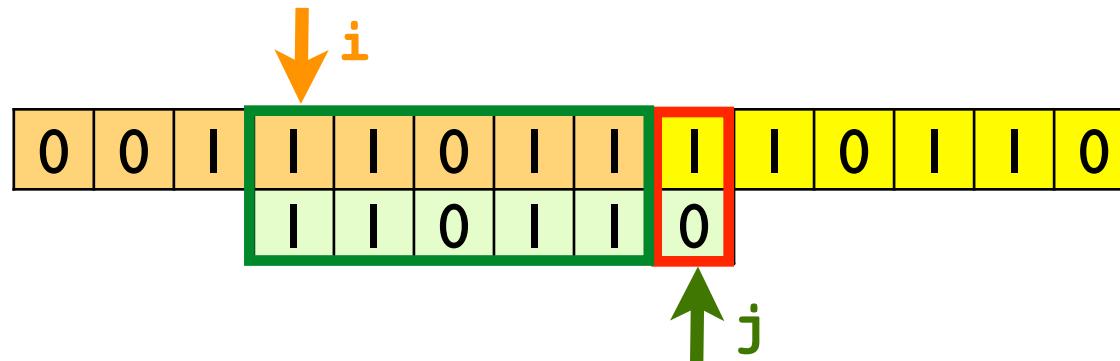
- Es kommt zu einem **Mismatch** an Musterposition $j=5$
 - Im **naiven Algorithmus** (Direkte Mustersuche) würde das Muster jetzt um **eine** Stelle nach rechts geschoben - j würde auf 0 gesetzt und i inkrementiert

Das geht besser !

Man kann das Muster in dieser Situation direkt
um **drei Stellen**
nach rechts bewegen

Grundidee des KMP-Vefahrens

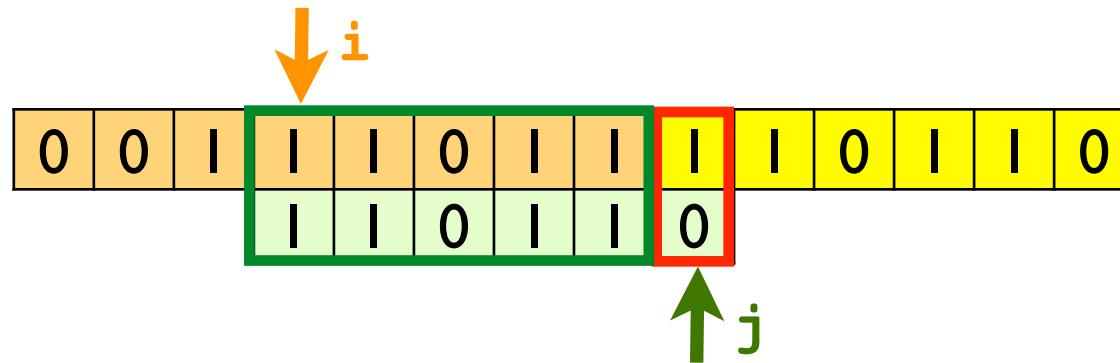
- Betrachten wir noch einmal die folgende Situation:



- Es kommt zu einem **Mismatch** an Musterposition $j=5$
 - Wir wissen aber, dass das Muster **bis zur Stelle $j-1$** im Text war
das Muster um **eine** oder **zwei** Stellen nach rechts zu setzen ist offenbar wenig sinnvoll

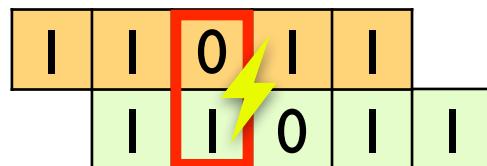
Grundidee des KMP-Vefahrens

- Betrachten wir noch einmal die folgende Situation:



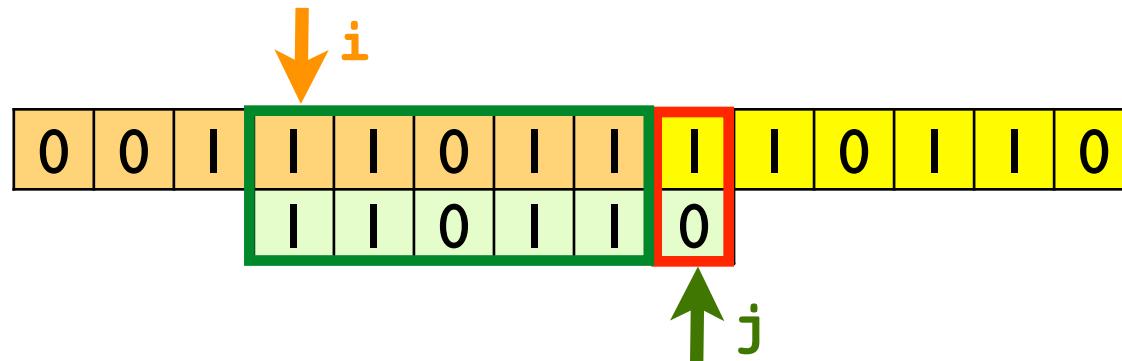
- Es kommt zu einem **Mismatch** an Musterposition $j=5$

- Wir wissen aber, dass das Muster **bis zur Stelle $j-1$** im Text war
das Muster um **eine** oder **zwei** Stellen nach rechts zu setzen ist offenbar wenig sinnvoll
- Betrachten wir dazu nur den bereits übereinstimmenden Part:



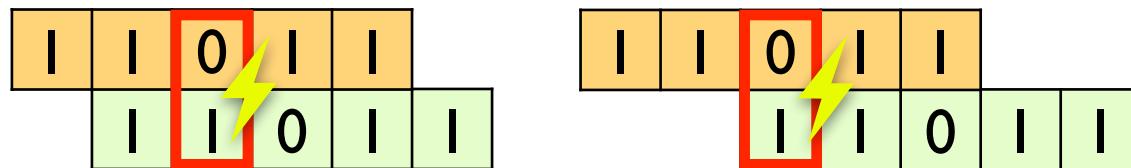
Grundidee des KMP-Vefahrens

- Betrachten wir noch einmal die folgende Situation:



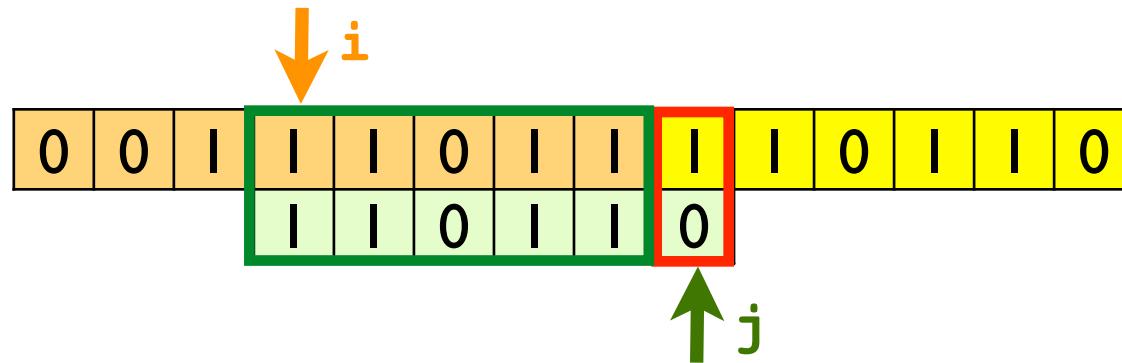
- Es kommt zu einem **Mismatch** an Musterposition $j=5$

- Wir wissen aber, dass das Muster **bis zur Stelle $j-1$** im Text war
das Muster um **eine** oder **zwei** Stellen nach rechts zu setzen ist offenbar wenig sinnvoll
- Betrachten wir dazu nur den bereits übereinstimmenden Part:



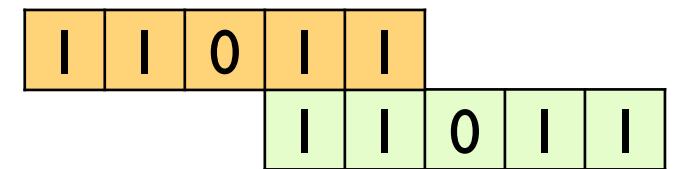
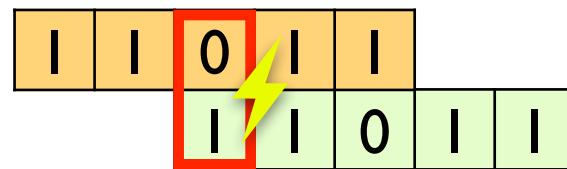
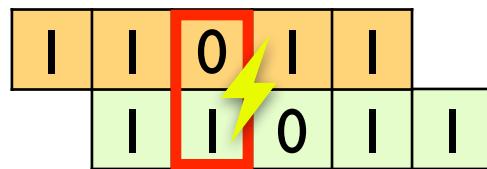
Grundidee des KMP-Vefahrens

- Betrachten wir noch einmal die folgende Situation:



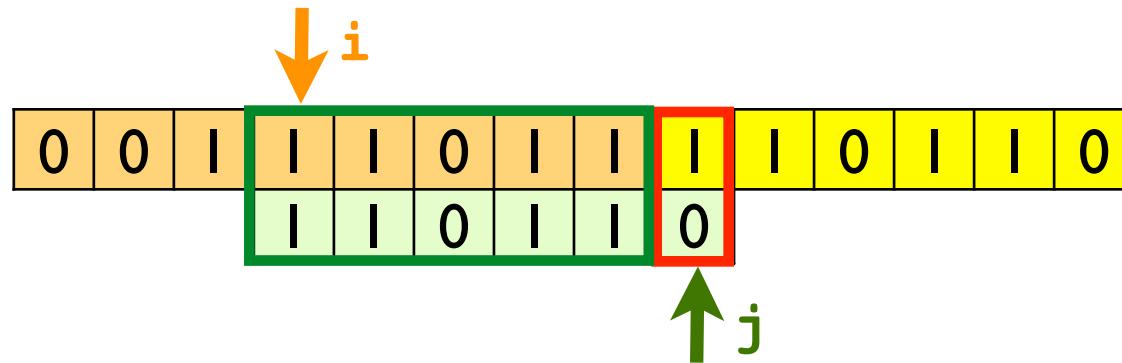
- Es kommt zu einem **Mismatch** an Musterposition $j=5$

- Wir wissen aber, dass das Muster **bis zur Stelle $j-1$** im Text war
das Muster um **eine** oder **zwei** Stellen nach rechts zu setzen ist offenbar wenig sinnvoll
- Betrachten wir dazu nur den bereits übereinstimmenden Part:



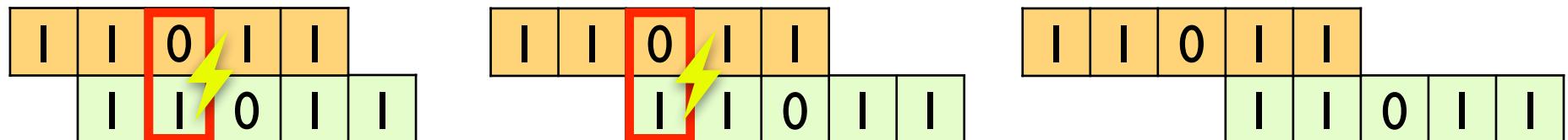
Grundidee des KMP-Vefahrens

- Betrachten wir noch einmal die folgende Situation:



- Es kommt zu einem **Mismatch** an Musterposition $j=5$

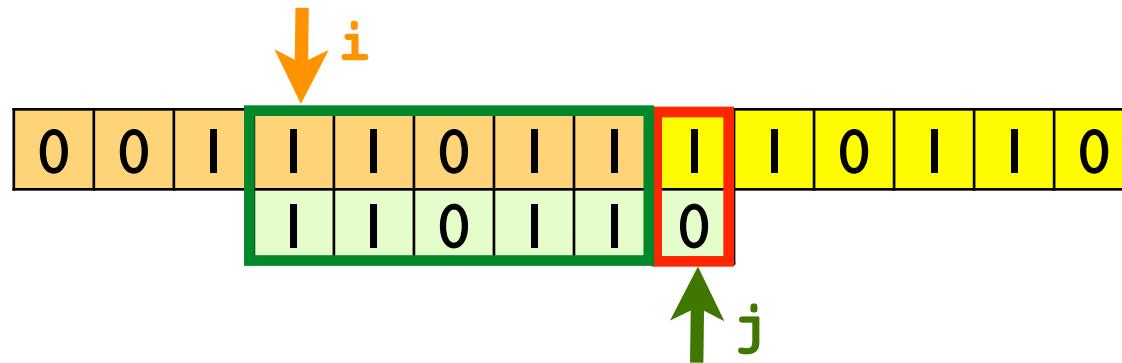
- Wir wissen aber, dass das Muster **bis zur Stelle $j-1$** im Text war
das Muster um **eine** oder **zwei** Stellen nach rechts zu setzen ist offenbar wenig sinnvoll
- Betrachten wir dazu nur den bereits übereinstimmenden Part:



- Gesucht:** größtes Präfix von $P[0,j]$, das gleichzeitig Suffix von $P[0,j]$ und von $P[0,j]$ verschieden ist - ein solches Teilwort heißt **maximaler Rand**

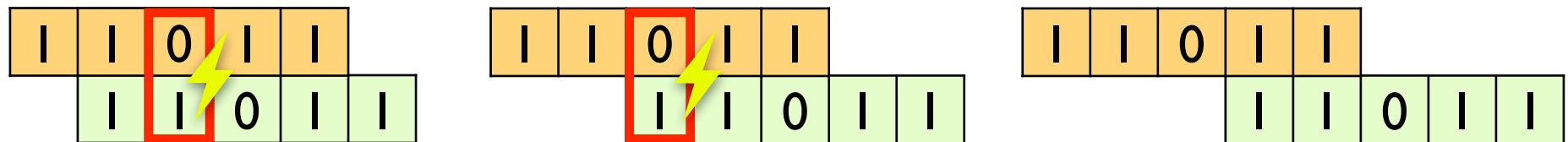
Grundidee des KMP-Vefahrens

- Betrachten wir noch einmal die folgende Situation:



- Es kommt zu einem **Mismatch** an Musterposition $j=5$

- Wir wissen aber, dass das Muster **bis zur Stelle $j-1$** im Text war
das Muster um **eine** oder **zwei** Stellen nach rechts zu setzen ist offenbar wenig sinnvoll
- Betrachten wir dazu nur den bereits übereinstimmenden Part:

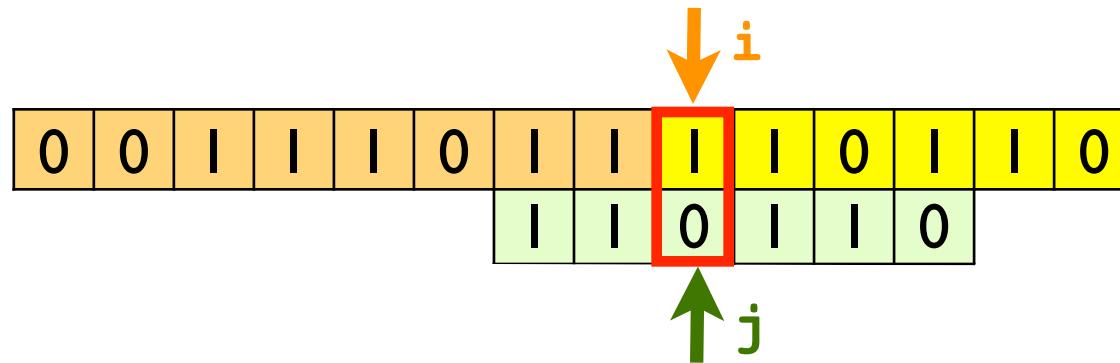


- Gesucht:** größtes Präfix von $P[0,j]$, das gleichzeitig Suffix von $P[0,j]$ und von $P[0,j]$ verschieden ist - ein solches Teilwort heißt **maximaler Rand**

Für Suche nach diesem maximalem Rand brauchen wir nur das Muster!

Grundidee des KMP-Vefahrens

- Betrachten wir noch einmal die folgende Situation:

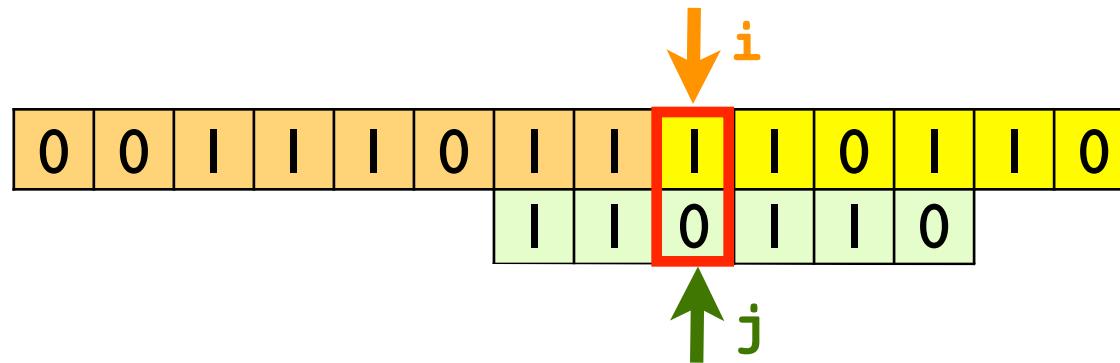


- Es kommt zu einem **Mismatch** an Musterposition $j=5$

- Wir wissen aber, dass das Muster **bis zur Stelle $j-1$** im Text war
das Muster um **eine** oder **zwei** Stellen nach rechts zu setzen ist offenbar wenig sinnvoll
- Die nächste sinnvolle Musterposition erreicht man durch Bewegen des
Musters um 3 Schritte nach rechts
bzw. wenn man den Musterzeiger j auf 2 setzt

Grundidee des KMP-Vefahrens

- Betrachten wir noch einmal die folgende Situation:



- Es kommt zu einem **Mismatch** an Musterposition $j=5$

- Wir wissen aber, dass das Muster **bis zur Stelle $j-1$** im Text war
das Muster um **eine** oder **zwei** Stellen nach rechts zu setzen ist offenbar wenig sinnvoll
- Die nächste sinnvolle Musterposition erreicht man durch Bewegen des
Musters um 3 Schritte nach rechts
bzw. wenn man den Musterzeiger j auf 2 setzt
- Der Index i in den Text bleibt unverändert

Beispiel: Ablauf der Suche bisher

- Insgesamt würde die Suche jetzt so ablaufen



Beispiel: Ablauf der Suche bisher

- Insgesamt würde die Suche jetzt so ablaufen



- Startwert von j

Beispiel: Ablauf der Suche bisher

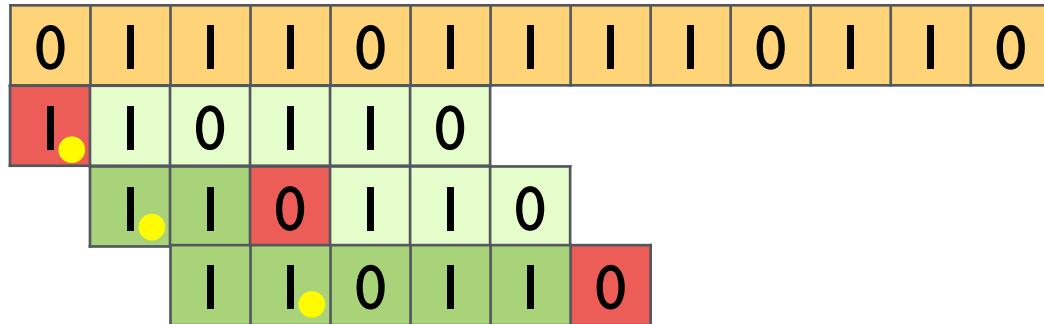
- Insgesamt würde die Suche jetzt so ablaufen

0	I	I	I	0	I	I	I	I	0	I	I	0
I	I	0	I	I	0							
I	I	0	I	I	I	0						

- Startwert von j

Beispiel: Ablauf der Suche bisher

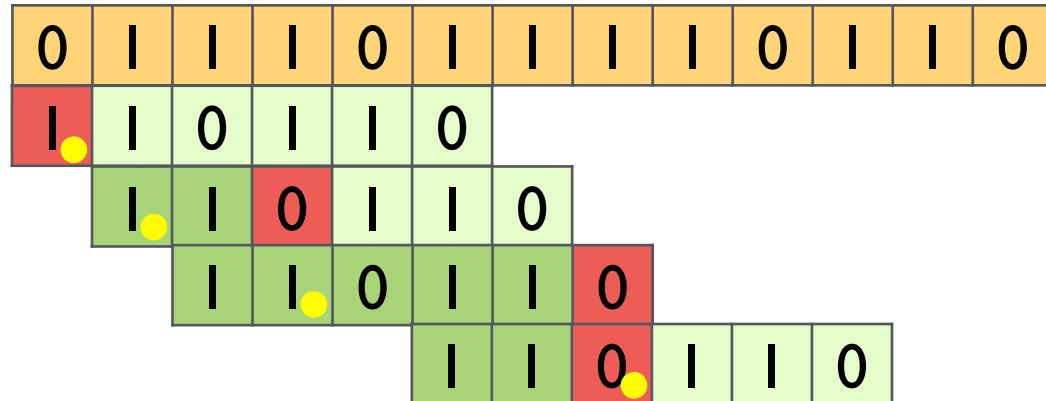
- Insgesamt würde die Suche jetzt so ablaufen



- Startwert von j

Beispiel: Ablauf der Suche bisher

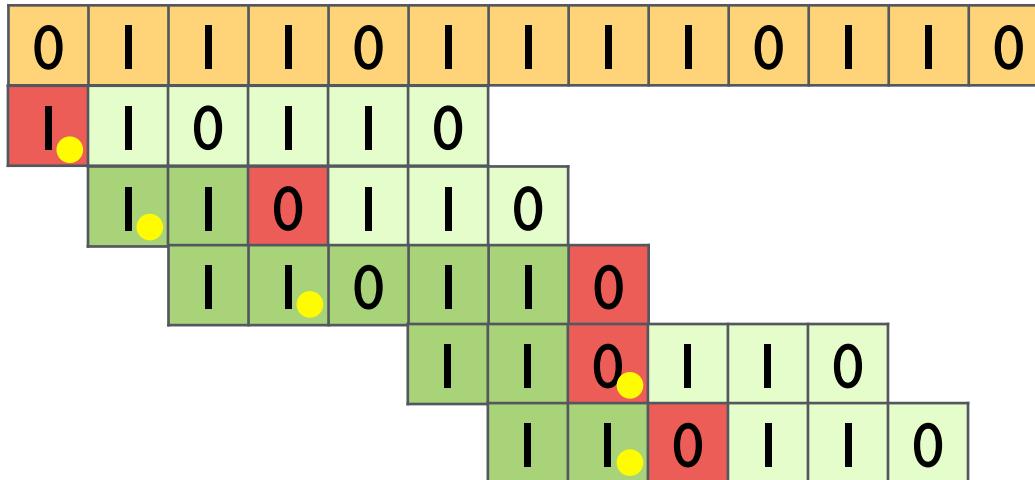
- Insgesamt würde die Suche jetzt so ablaufen



- Startwert von j

Beispiel: Ablauf der Suche bisher

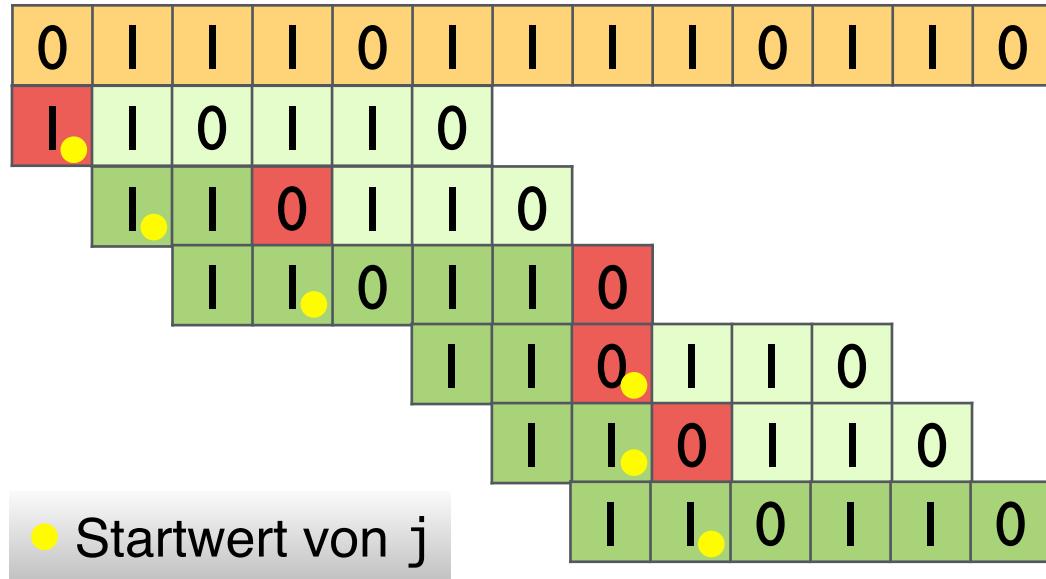
- Insgesamt würde die Suche jetzt so ablaufen



- Startwert von j

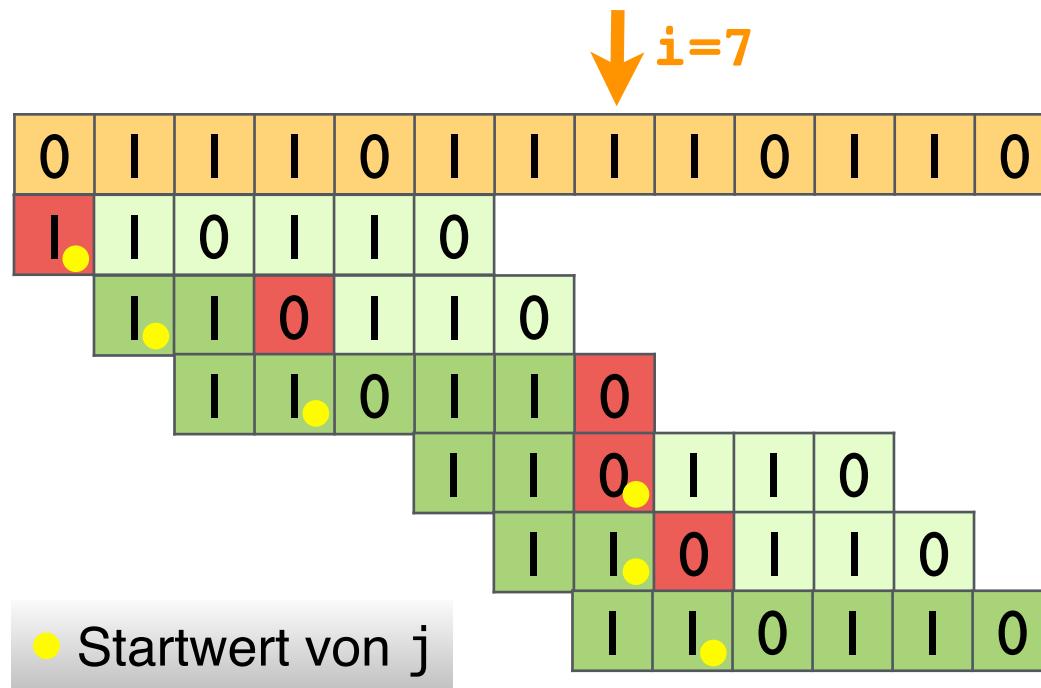
Beispiel: Ablauf der Suche bisher

- Insgesamt würde die Suche jetzt so ablaufen



Beispiel: Ablauf der Suche bisher

- Insgesamt würde die Suche jetzt so ablaufen

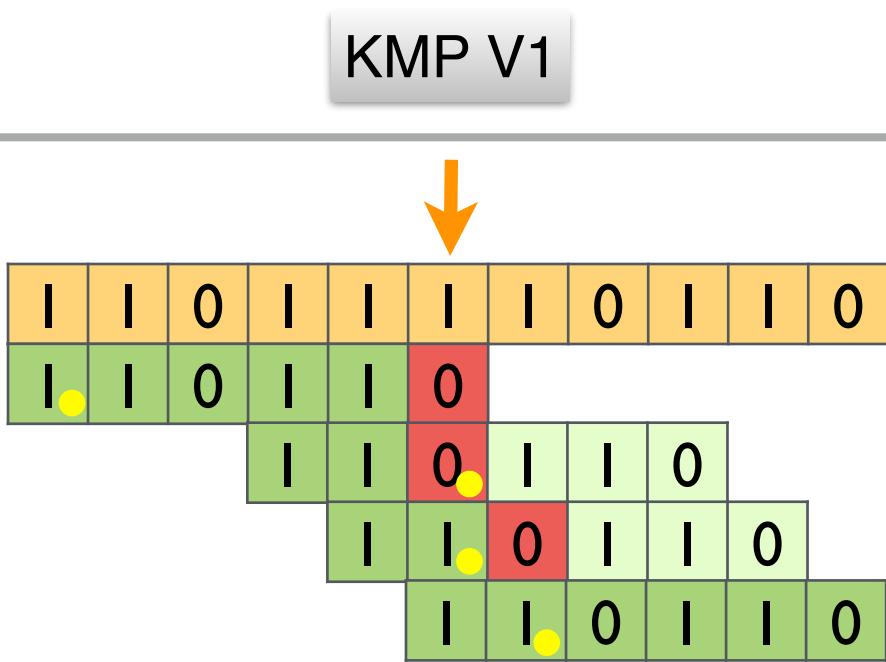


- Beachte

- Konzeptionell werden Zeichen des Textes **nach Verschieben des Musters** wiederholt betrachtet (z.B. Indexposition 7)
- **Grund:** wir lassen im Gegensatz zum endlichen Automaten das Zeichen, welches den Mismatch verursacht hat, ausser acht.

DEA vs. KMP V1

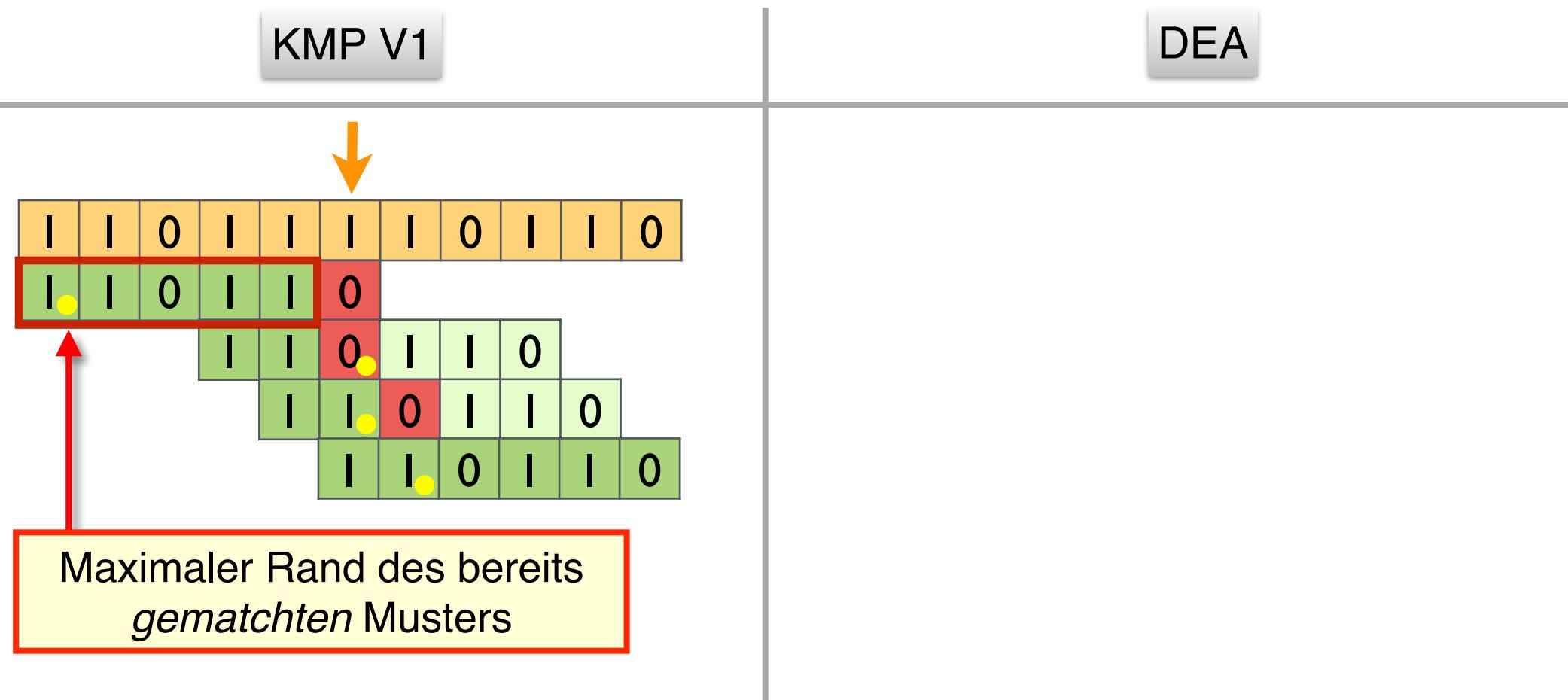
- Versatz wird unterschiedlich bestimmt:



DEA

DEA vs. KMP V1

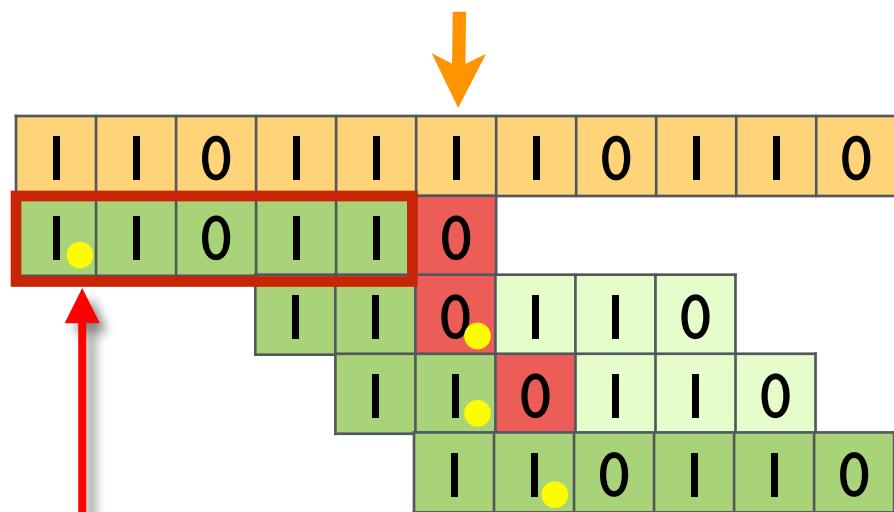
- Versatz wird unterschiedlich bestimmt:



DEA vs. KMP V1

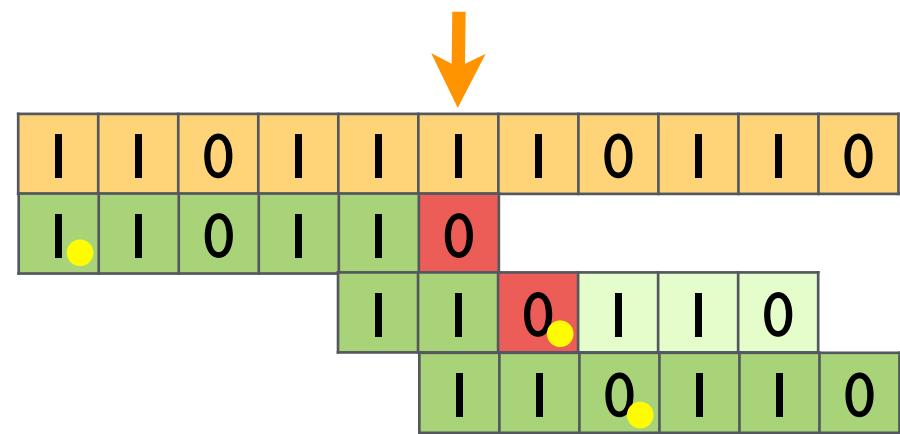
- Versatz wird unterschiedlich bestimmt:

KMP V1



Maximaler Rand des bereits
gematchten Musters

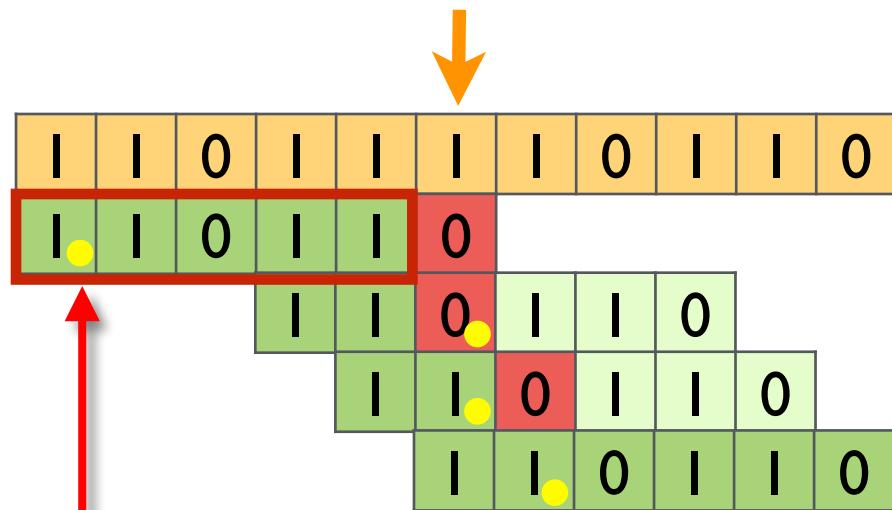
DEA



DEA vs. KMP V1

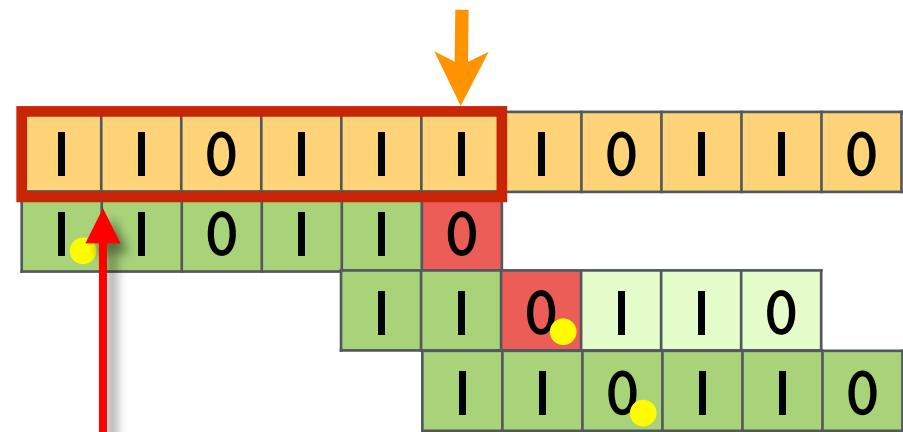
- Versatz wird unterschiedlich bestimmt:

KMP V1



Maximaler Rand des bereits
gematchten Musters

DEA

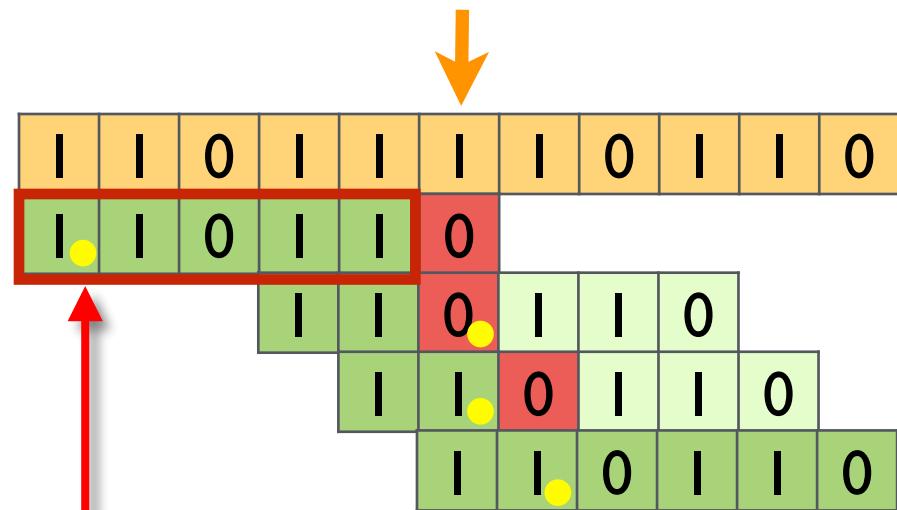


Maximaler Rand der
gelesenen Eingabe
genauer: Aktueller Zustand + zuletzt
gelesenes Zeichen

DEA vs. KMP V1

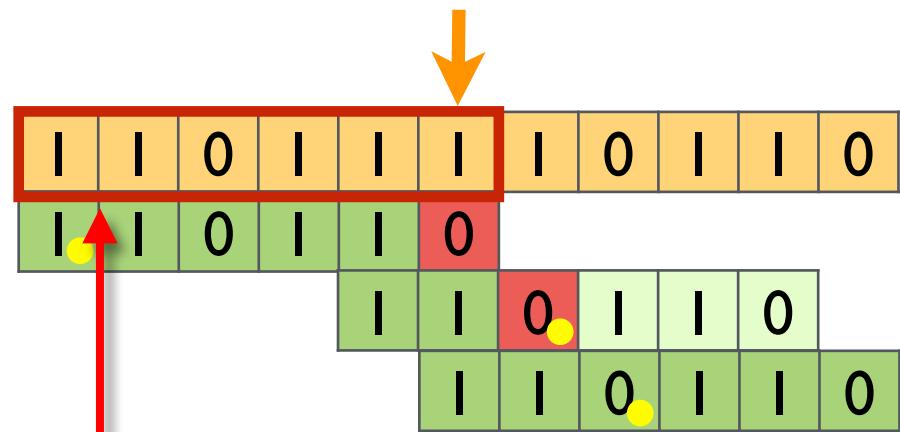
- **Versatz wird unterschiedlich bestimmt:**

KMP V1



Maximaler Rand des bereits
gematchten Musters

DEA



Maximaler Rand der
gelesenen Eingabe
genauer: Aktueller Zustand + zuletzt
gelesenes Zeichen

- **DEA berücksichtigt das zuletzt gelesene Zeichen**
 - **Problem:** das kennen wir aber erst, wenn wir den Text kennen!

- Benutze hierzu folgende Hilfsfunktion:

D Die Präfixfunktion zu einem Muster P

Seien Σ ein Alphabet und $P = p = a_0 \cdots a_{m-1}$ ein Muster ($p \in \Sigma^*$), dann ist die **Präfixfunktion zu P** $\pi_p : \Sigma^* \rightarrow \Sigma^*$ definiert durch

$$\pi_p(w) = \begin{cases} \varepsilon & \text{falls } w = \varepsilon \\ P[0, j] \text{ mit } j = \max \{k \in \{0, \dots, m-1\} \mid P[0, j] \sqsupseteq w\} & \text{sonst} \end{cases}$$

- Beachte:

- $\pi_p(p)$ ist gleichzeitig Präfix und Suffix von p - $\pi_p(p)$ liefert **Rand!**
- $\pi_p(p) \neq p$ - $\pi_p(p)$ liefert **echten Rand!**
- $\pi_p(p)$ hat maximale Länge - $\pi_p(p)$ liefert **maximalen Rand!**

Eigenschaften der Präfix-Funktion zu $p - 1$

- Durch die wiederholte Anwendung der Präfixfunktion auf ein Muster p , erhält man **alle Ränder** von p (Präfixe von p , die gleichzeitig Suffixe von p sind):

B Iteration der Präfix-Funktion - Beobachtungen

Wir betrachten das Muster $p = 101101$. Die Menge aller Präfixe von p ist

$$\text{Prefixes}(p) = \{\varepsilon, 1, 10, 101, 1011, 10110, 101101\}$$

Die Menge aller Suffixe von p ist

$$\text{Suffixes}(p) = \{\varepsilon, 1, 01, 101, 1101, 01101, 101101\}$$

Demnach ist $\text{Prefixes}(p) \cap \text{Suffixes}(p) = \{\varepsilon, 1, 101, 101101\}$. Wiederholte Anwendung von π_p auf p liefert:

$$\begin{aligned}\pi_p(p) &= 101 \\ \pi_p(\pi_p(p)) &= 1 \\ \pi_p(\pi_p(\pi_p(p))) &= \varepsilon\end{aligned}$$

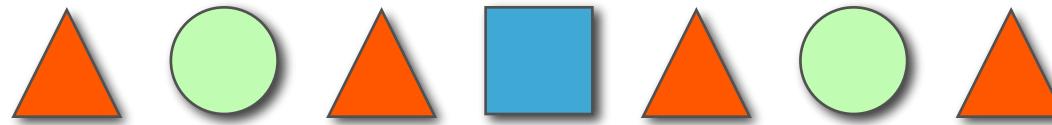
Also gilt offenbar

$$\text{Prefixes}(p) \cap \text{Suffixes}(p) - \{p\} = \bigcup_{i \in \mathbb{N}} \{\pi_p^i(p)\} \quad \text{und ferner: } \pi_p^{i+1}(p) \subset \pi_p^i(p)$$

Beachte: Ab einem bestimmten Index t gilt für alle $j \in \mathbb{N}$: $\pi_p^t(p) = \pi_p^{t+j}(p) = \varepsilon$ - mit diesem Abbruchkriterium lässt sich die unendliche Vereinigung leicht berechnen.

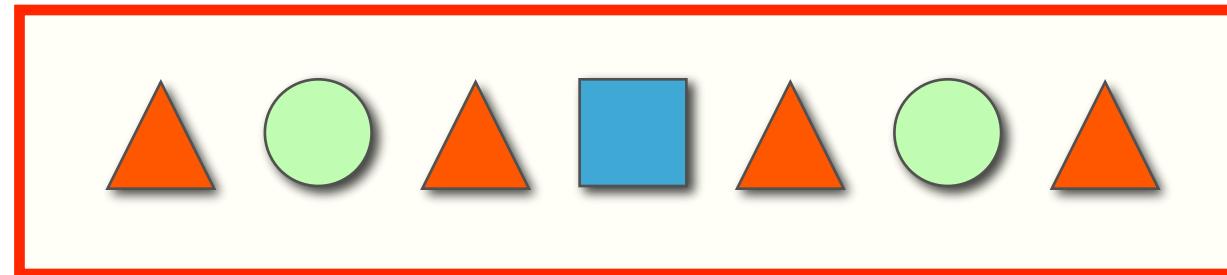
Eigenschaften der Präfix-Funktion zu $p - 2$

- Wiederholte Anwendung der Präfixfunktion listet alle Ränder auf



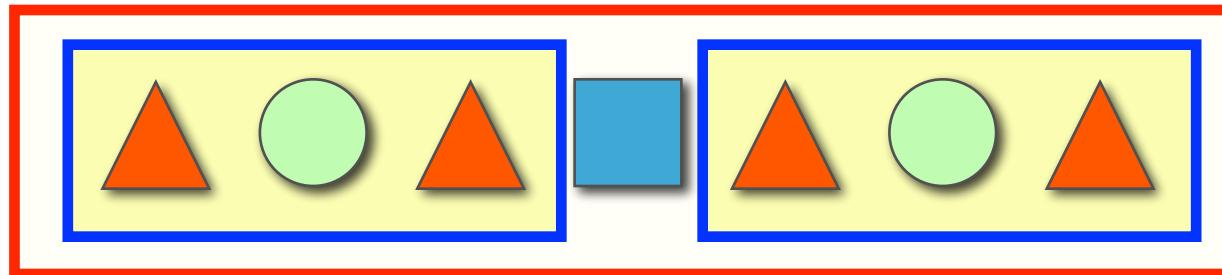
Eigenschaften der Präfix-Funktion zu $p - 2$

- Wiederholte Anwendung der Präfixfunktion listet alle Ränder auf



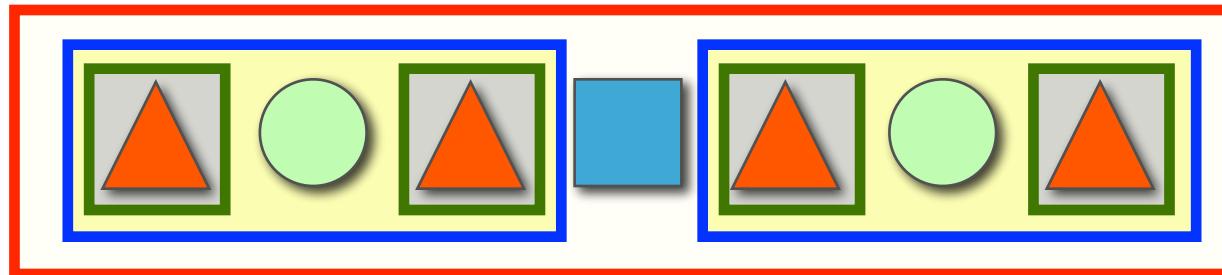
Eigenschaften der Präfix-Funktion zu $p - 2$

- Wiederholte Anwendung der Präfixfunktion listet alle Ränder auf



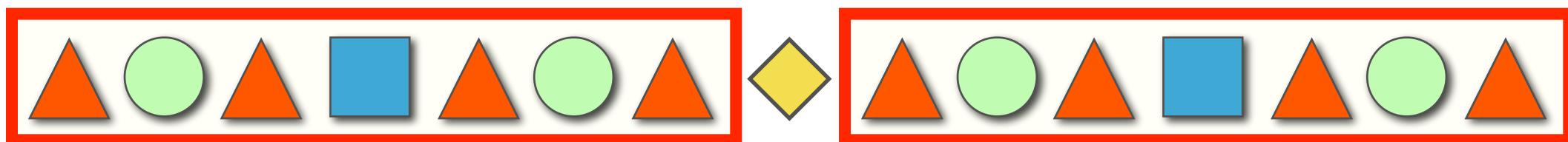
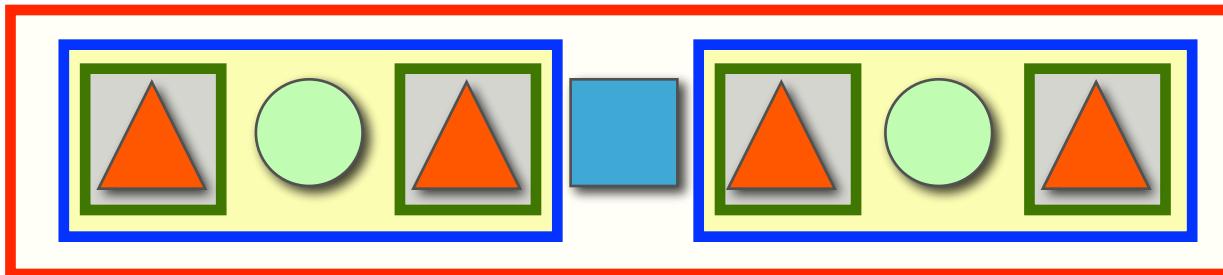
Eigenschaften der Präfix-Funktion zu $p - 2$

- Wiederholte Anwendung der Präfixfunktion listet alle Ränder auf



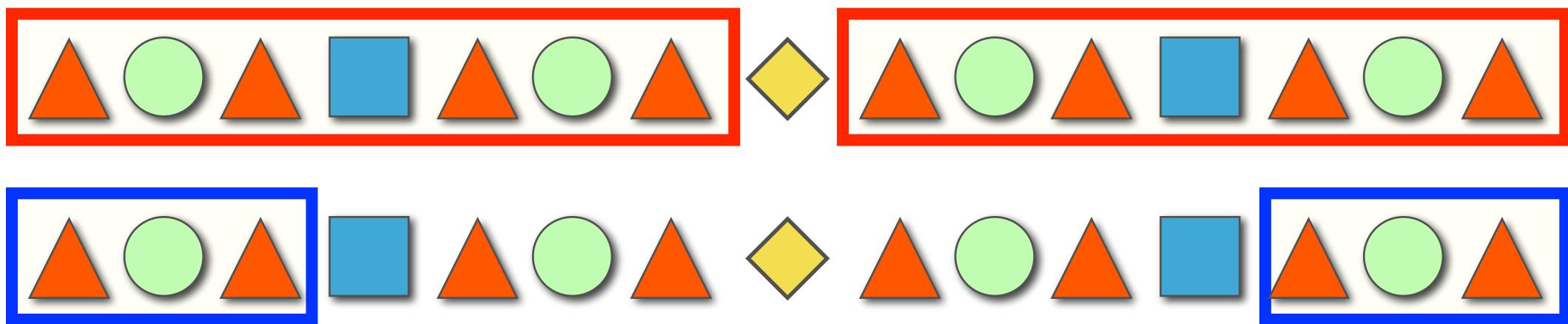
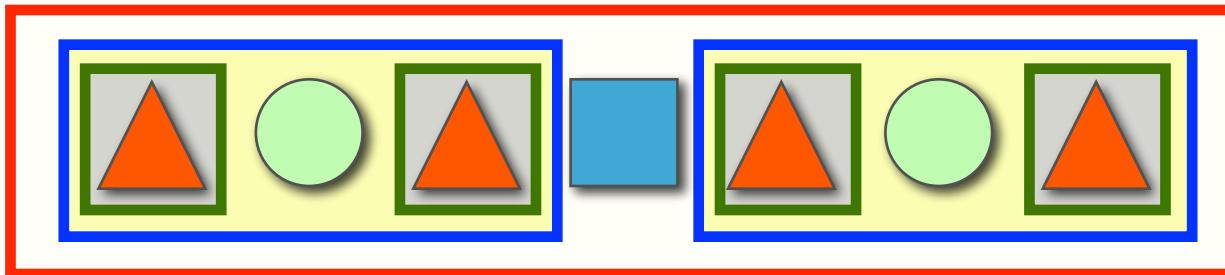
Eigenschaften der Präfix-Funktion zu $p - 2$

- Wiederholte Anwendung der Präfixfunktion listet alle Ränder auf



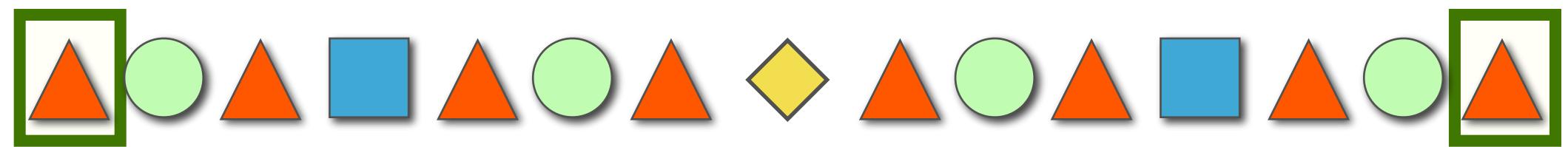
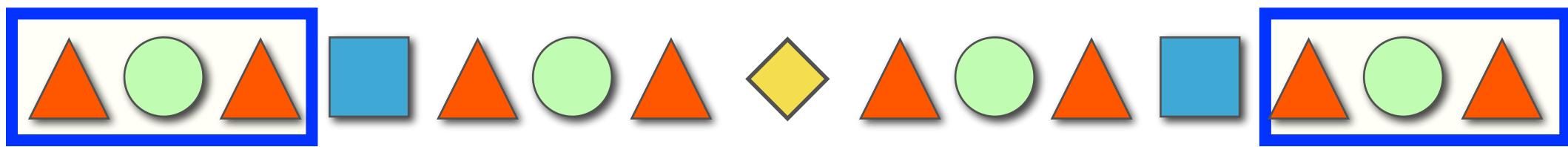
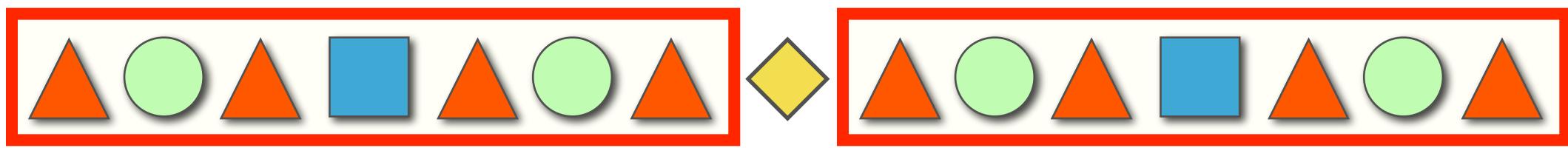
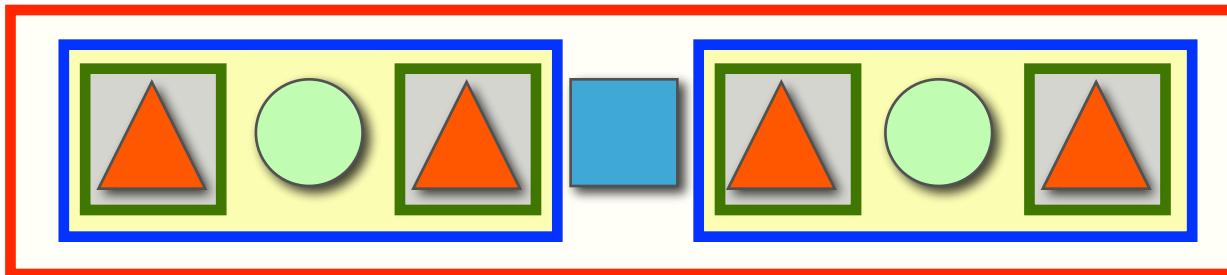
Eigenschaften der Präfix-Funktion zu $p - 2$

- Wiederholte Anwendung der Präfixfunktion listet alle Ränder auf



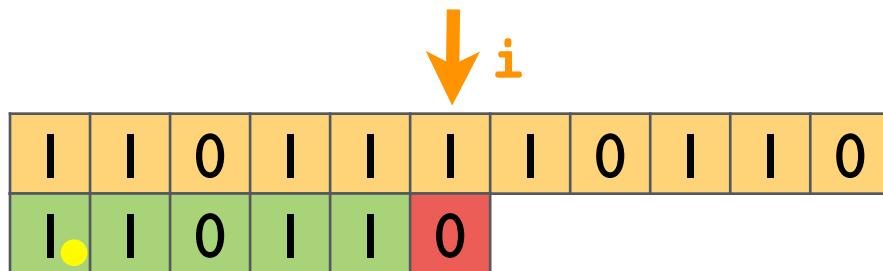
Eigenschaften der Präfix-Funktion zu $p - 2$

- Wiederholte Anwendung der Präfixfunktion listet alle Ränder auf

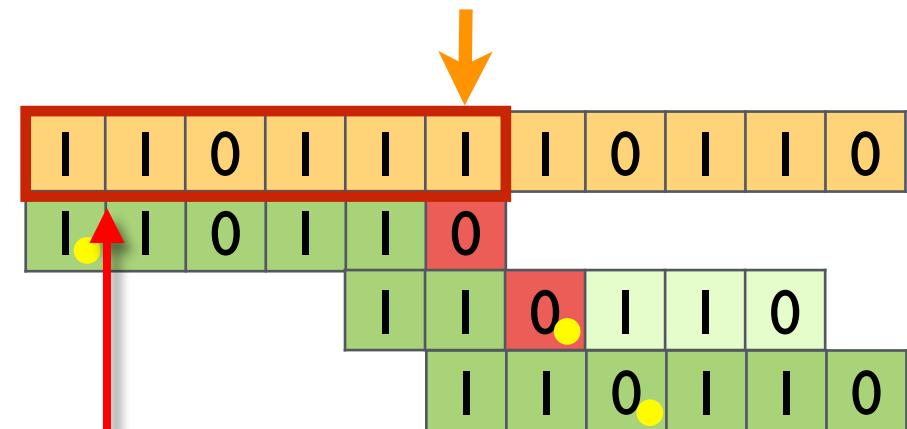


KMP V2

KMP V2



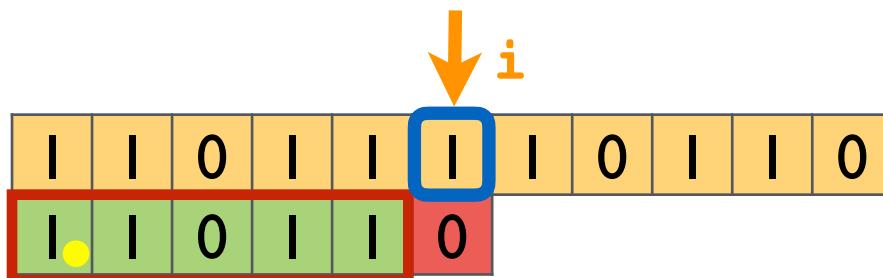
DEA



Maximaler Rand der
gelesenen Eingabe
genauer: Aktueller Zustand + zuletzt
gelesenes Zeichen

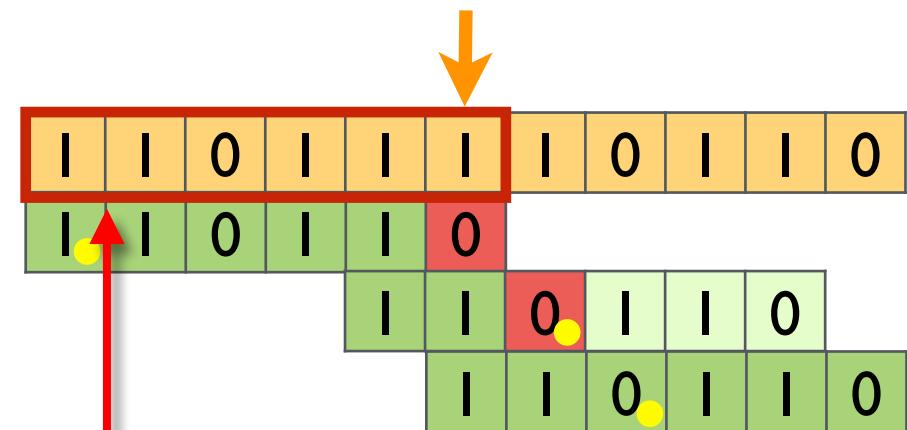
KMP V2

KMP V2



Suche grössten Rand $R=P[0,k]$ von $P[0,j]$, so dass $P[k] = T[i]$

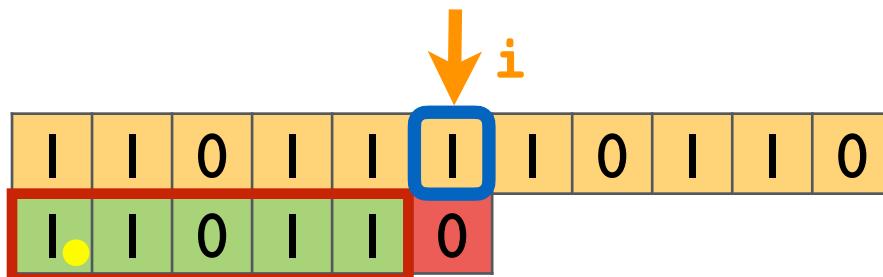
DEA



Maximaler Rand der
gelesenen Eingabe
genauer: Aktueller Zustand + zuletzt
gelesenes Zeichen

KMP V2

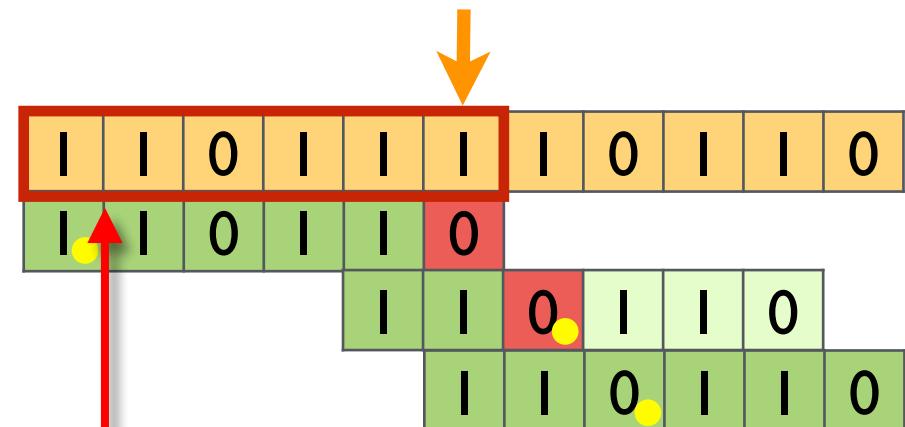
KMP V2



Suche grössten Rand $R=P[0,k]$ von $P[0,j]$, so dass $P[k] = T[i]$

$k=2$  $0=P[2] \neq T[i]=1$

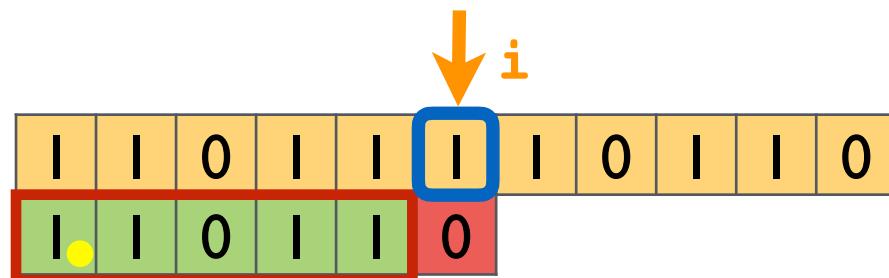
DEA



Maximaler Rand der gelesenen Eingabe
genauer: Aktueller Zustand + zuletzt gelesenes Zeichen

KMP V2

KMP V2

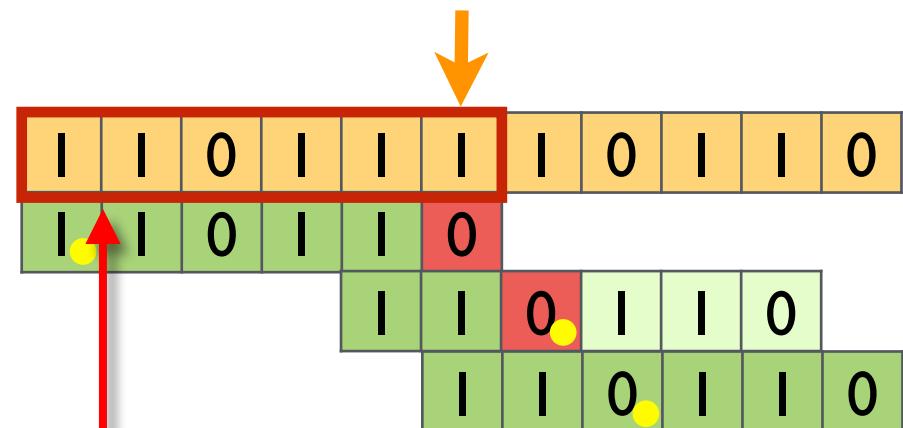


Suche grössten Rand $R=P[0,k]$
von $P[0,j]$, so dass $\textcolor{blue}{P[k]} = \textcolor{red}{T[i]}$

$k=2$ | | 0 | | 0 = $P[2]$ ≠ $T[i] = 1$

$k=1$ | 1 | 0 | 1 | 1 $1=P[1] = T[i]=1$

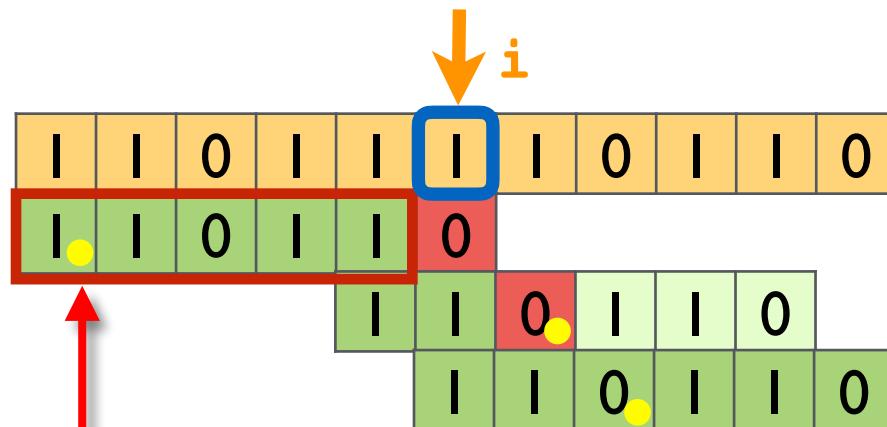
DEA



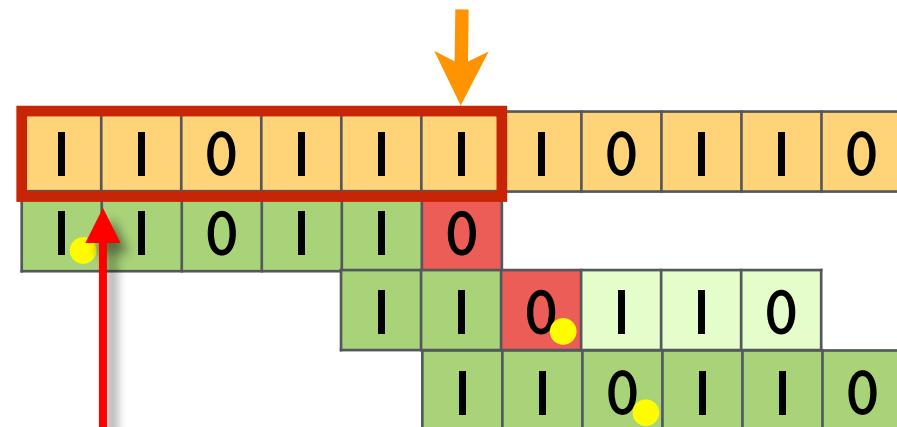
Maximaler Rand der
gelesenen Eingabe
genauer: Aktueller Zustand + zuletzt
gelesenes Zeichen

KMP V2

KMP V2



Suche grössten Rand $R=P[0,k]$
von $P[0,j]$, so dass $P[k] = T[i]$



Maximaler Rand der
gelesenen Eingabe
genauer: Aktueller Zustand + zuletzt
gelesenes Zeichen

$k=2$ 0= $P[2]$ \neq $T[i]=1$

$k=1$ 1= $P[1]$ = $T[i]=1$

Aufgabe: Bestimme maximale Ränder aller Präfixe von P

- **Wir suchen ein “inkrementelles” Verfahren**
 - kann man aus dem Rand von $P[0,q-1]$ den Rand von $P[0,q]$ gewinnen?

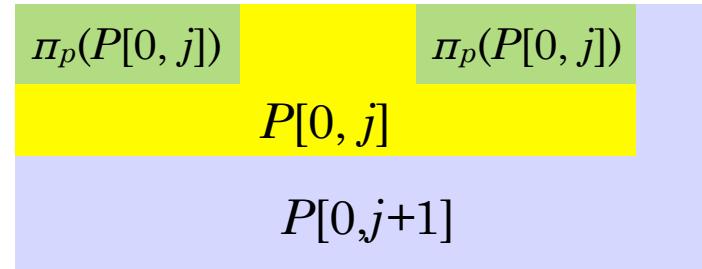
Eigenschaften der Präfix-Funktion zu p - 3

- $\pi_p(P[0, j])$ sei der **maximalen Rand** von $P[0, j]$
maximales Präfix von p (ungleich p), welches Suffix von $P[0..j]$ ist



Eigenschaften der Präfix-Funktion zu p - 3

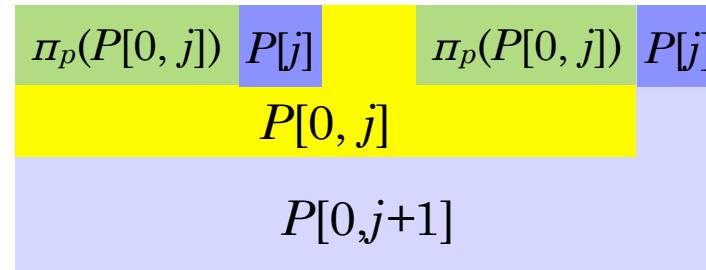
- $\pi_p(P[0, j])$ sei der **maximalen Rand** von $P[0, j]$
maximales Präfix von p (ungleich p), welches Suffix von $P[0:j]$ ist



- Ist der maximale Rand für $P[0, j+1]$ **nicht das leere Wort**, so gilt
$$\pi_p(P[0, j + 1]) = \pi_p(P[0, j] \cdot P[j]) = \pi_p(P[0, j]) \cdot P[j]$$

Eigenschaften der Präfix-Funktion zu p - 3

- $\pi_p(P[0, j])$ sei der **maximalen Rand** von $P[0, j]$
maximales Präfix von p (ungleich p), welches Suffix von $P[0..j]$ ist



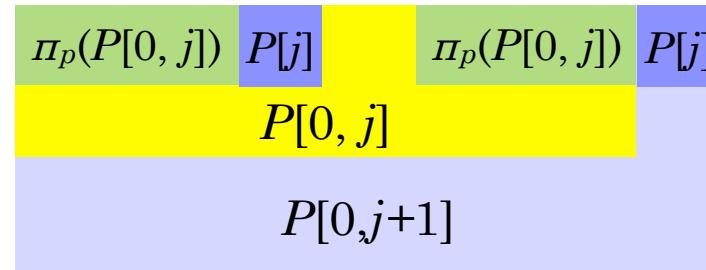
- Ist der maximale Rand für $P[0, j+1]$ **nicht das leere Wort**, so gilt

$$\pi_p(P[0, j + 1]) = \pi_p(P[0, j] \cdot P[j]) = \pi_p(P[0, j]) \cdot P[j]$$

- der maximale Rand von $P[0, j+1]$ ist dann also eine Verlängerung des maximalen Randes von $P[0, j]$

Eigenschaften der Präfix-Funktion zu p - 3

- $\pi_p(P[0, j])$ sei der **maximalen Rand** von $P[0, j]$
maximales Präfix von p (ungleich p), welches Suffix von $P[0..j]$ ist



- Ist der maximale Rand für $P[0, j+1]$ **nicht das leere Wort**, so gilt

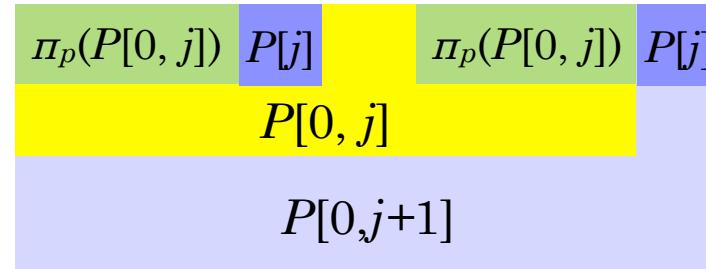
$$\pi_p(P[0, j + 1]) = \pi_p(P[0, j] \cdot P[j]) = \pi_p(P[0, j]) \cdot P[j]$$

- der maximale Rand von $P[0, j+1]$ ist dann also eine Verlängerung des maximalen Randes von $P[0, j]$
- Insbesondere folgt für diesen Fall

$$\pi_p(P[0, j]) \sqsubset \pi_p(P[0, j + 1])$$

Eigenschaften der Präfix-Funktion zu p - 3

- $\pi_p(P[0, j])$ sei der **maximalen Rand** von $P[0, j]$
maximales Präfix von p (ungleich p), welches Suffix von $P[0..j]$ ist



- Ist der maximale Rand für $P[0, j+1]$ **nicht das leere Wort**, so gilt
$$\pi_p(P[0, j + 1]) = \pi_p(P[0, j] \cdot P[j]) = \pi_p(P[0, j]) \cdot P[j]$$
 - der maximale Rand von $P[0, j+1]$ ist dann also eine Verlängerung des maximalen Randes von $P[0, j]$
 - Insbesondere folgt für diesen Fall
$$\pi_p(P[0, j]) \sqsubset \pi_p(P[0, j + 1])$$
- mit dieser Beobachtung können wir π_p induktiv definieren und effizient berechnen!

Effiziente Berechnung von $\pi_p(P[0, q])$

Effiziente Berechnung von $\pi_p(P[0, q])$

- **Induktionsanfang** (aus Definition von π_p)

$$\pi_p(P[0, 0] = \varepsilon) = \varepsilon$$

Effiziente Berechnung von $\pi_p(P[0, q])$

- **Induktionsanfang** (aus Definition von π_p)

$$\pi_p(P[0, 0] = \varepsilon) = \varepsilon$$

- **Induktionsschritt. Es sei**

$$\pi_p(P[0, q]) = P[0, k]$$



Effiziente Berechnung von $\pi_p(P[0,q])$

- **Induktionsanfang** (aus Definition von π_p)

$$\pi_p(P[0,0] = \varepsilon) = \varepsilon$$

- **Induktionsschritt. Es sei**

$$\pi_p(P[0,q]) = P[0,k]$$



- **Um $\pi_p(P[0, q+1])$ zu berechnen unterscheiden wir zwei Fälle:**
 - $P[k] = P[q]$
 - Dann haben wir mit $P[0,k+1]$ den größten Rand von $P[0,q+1]$ bereits gefunden
 - $P[k] \neq P[q]$
 - Dann ist der größte Rand von $P[0,q+1]$ der größte Rand von $P[0,k]$, der, erweitert um $P[q]$, Rand von $P[0,q+1]$ ist. (Beispiel - s.n.F.)
 - Wir suchen also das kleinste i , so dass

$$P [|\pi_p^i(\pi_p(P[0,k]))|] = P[q]$$

Effiziente Berechnung von π_p

B Berechnung von $\pi_p(P[0,24])$

Ausgangspunkt: Wir haben den maximalen Rand von $P[0,23]$ bestimmt:

1 0 0 1 0 1 1 0 0 1 0 1 1 0 0 1 0 1 1 0 0 1 0 0 1 0 0

$$\pi_p(P[0,23]) = P[0,11] = \mathbf{1 0 0 1 0 1 1 0 0 1 0}$$

Effiziente Berechnung von π_p

B Berechnung von $\pi_p(P[0,24])$

Ausgangspunkt: Wir haben den maximalen Rand von $P[0,23]$ bestimmt:

1 0 0 1 0 1 1 0 0 1 0 1 1 0 0 1 0 1 1 0 0 1 0 0 1 0 0

$$\pi_p(P[0,23]) = P[0,11] = \text{10010110010}$$

Wir suchen den maximalen Rand von $P[0,24]$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
1	0	0	1	0	1	1	0	0	1	0	1	1	0	0	1	0	1	1	0	0	1	0	0

Effiziente Berechnung von π_p

B Berechnung von $\pi_p(P[0,24])$

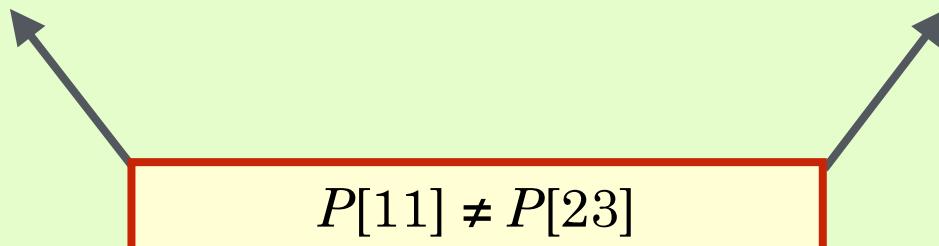
Ausgangspunkt: Wir haben den maximalen Rand von $P[0,23]$ bestimmt:

1 0 0 1 0 1 1 0 0 1 0 1 1 0 0 1 0 1 1 0 0 1 0 0

$$\pi_p(P[0,23]) = P[0,11] = \text{10010110010}$$

Wir suchen den maximalen Rand von $P[0,24]$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
1	0	0	1	0	1	1	0	0	1	0	1	1	0	0	1	0	1	1	0	0	1	0	0



Effiziente Berechnung von π_p

B Berechnung von $\pi_p(P[0,24])$

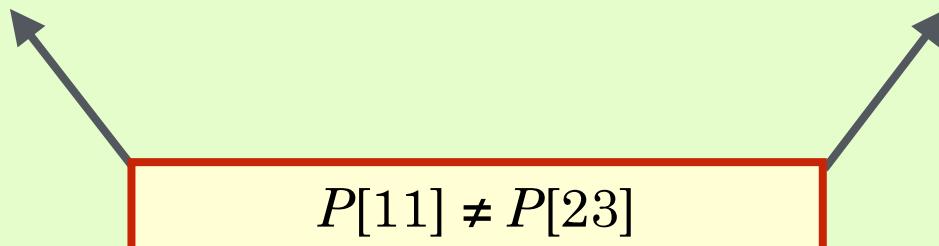
Ausgangspunkt: Wir haben den maximalen Rand von $P[0,23]$ bestimmt:

10010110010110010100100

$$\pi_p(P[0,23]) = P[0,11] = \textcolor{blue}{10010110010}$$

Wir suchen den maximalen Rand von $P[0,24]$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
1	0	0	1	0	1	1	0	0	1	0	1	1	0	0	1	0	1	1	0	0	1	0	0



Wir fahren daher mit dem nächst kürzeren Rand fort. Der ergibt sich aus

$$\pi_p^{-1}(\pi_p(P[0,23])) = \pi_p(P[0,11]) = \pi_p(\textcolor{blue}{10010110010}) = \textcolor{blue}{10010}$$

Effiziente Berechnung von π_p

B Berechnung von $\pi_p(P[0,24])$

Ausgangspunkt: Wir haben den maximalen Rand von $P[0,23]$ bestimmt:

1 0 0 1 0 1 1 0 0 1 0 1 1 0 0 1 0 1 1 0 0 1 0 0

$$\pi_p(P[0,23]) = P[0,11] = \text{10010110010}$$

Wir suchen den maximalen Rand von $P[0,24]$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
1	0	0	1	0	1	1	0	0	1	0	1	1	0	0	1	0	1	1	0	0	1	0	0

Effiziente Berechnung von π_p

B Berechnung von $\pi_p(P[0,24])$

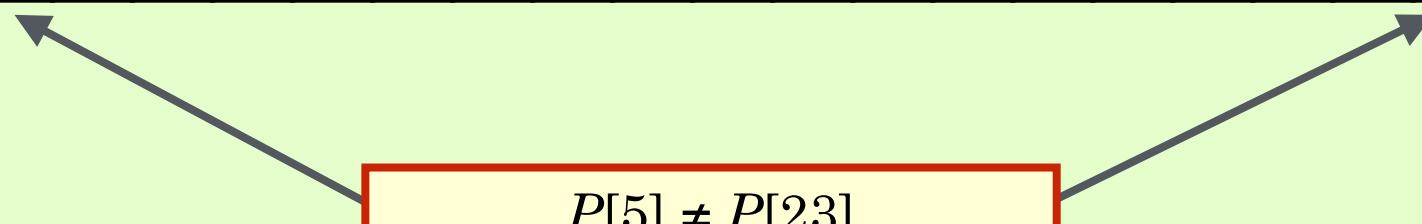
Ausgangspunkt: Wir haben den maximalen Rand von $P[0,23]$ bestimmt:

1 0 0 1 0 1 1 0 0 1 0 1 1 0 0 1 0 1 1 0 0 1 0 0

$$\pi_p(P[0,23]) = P[0,11] = \text{10010110010}$$

Wir suchen den maximalen Rand von $P[0,24]$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
1	0	0	1	0	1	1	0	0	1	0	1	1	0	0	1	0	1	1	0	0	1	0	0



Effiziente Berechnung von π_p

B Berechnung von $\pi_p(P[0,24])$

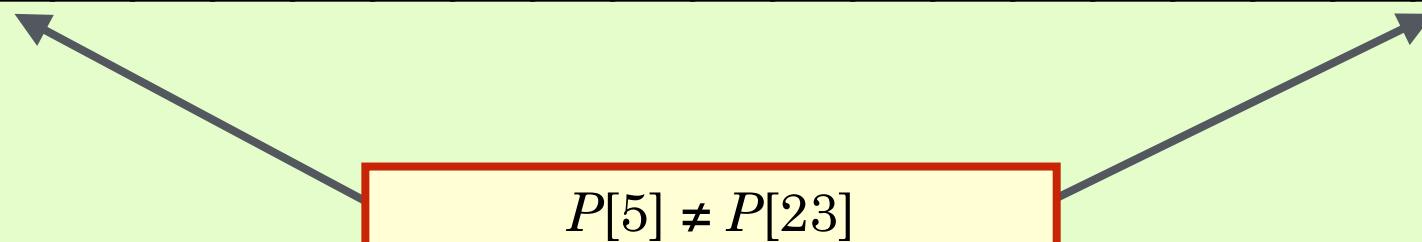
Ausgangspunkt: Wir haben den maximalen Rand von $P[0,23]$ bestimmt:

1 0 0 1 0 1 1 0 0 1 0 1 1 0 0 1 0 1 1 0 0 1 0 0

$$\pi_p(P[0,23]) = P[0,11] = \text{10010110010}$$

Wir suchen den maximalen Rand von $P[0,24]$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
1	0	0	1	0	1	1	0	0	1	0	1	1	0	0	1	0	1	1	0	0	1	0	0



Wir fahren daher mit dem nächst kürzeren Rand fort. Der ergibt sich aus

$$\pi_p^2(\pi_p(P[0,23])) = \pi_p^2(P[0,11]) = \pi_p^2(\text{10010110010}) = \pi_p(\text{10010}) = \text{10}$$

Effiziente Berechnung von π_p

B Berechnung von $\pi_p(P[0,24])$

Ausgangspunkt: Wir haben den maximalen Rand von $P[0,23]$ bestimmt:

1 0 0 1 0 1 1 0 0 1 0 1 1 0 0 1 0 1 1 0 0 1 0 0

$$\pi_p(P[0,23]) = P[0,11] = \text{10010110010}$$

Wir suchen den maximalen Rand von $P[0,24]$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
1	0	0	1	0	1	1	0	0	1	0	1	1	0	0	1	0	1	1	0	0	1	0	0

Effiziente Berechnung von π_p

B Berechnung von $\pi_p(P[0,24])$

Ausgangspunkt: Wir haben den maximalen Rand von $P[0,23]$ bestimmt:

1 0 0 1 0 1 1 0 0 1 0 1 1 0 0 1 0 1 1 0 0 1 0 0

$$\pi_p(P[0,23]) = P[0,11] = \text{10010110010}$$

Wir suchen den maximalen Rand von $P[0,24]$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
1	0	0	1	0	1	1	0	0	1	0	1	1	0	0	1	0	1	1	0	0	1	0	0

$P[2] = P[23]$

Effiziente Berechnung von π_p

B Berechnung von $\pi_p(P[0,24])$

Ausgangspunkt: Wir haben den maximalen Rand von $P[0,23]$ bestimmt:

1 0 0 1 0 1 1 0 0 1 0 1 1 0 0 1 0 1 1 0 0 1 0 0

$$\pi_p(P[0,23]) = P[0,11] = \text{10010110010}$$

Wir suchen den maximalen Rand von $P[0,24]$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
1	0	0	1	0	1	1	0	0	1	0	1	1	0	0	1	0	1	1	0	0	1	0	0



Damit haben wir den längsten Rand von $P[0,24]$ gefunden - er ist **100**

1 0 0 1 0 1 1 0 0 1 0 1 1 0 0 1 0 0

Effiziente Berechnung von π_p (C++)

- Wir rechnen die **Längen der Ränder** aus
die eigentlichen Ränder sind für uns nicht wichtig

$$\text{pi}[q] = |\pi_p(P[0, q])|$$

Effiziente Berechnung von π_p (C++)

- Wir rechnen die **Längen der Ränder** aus
die eigentlichen Ränder sind für uns nicht wichtig

$$\text{pi}[q] = |\pi_p(P[0, q])|$$

```
vector<int> calcPi(const string& p) {
    vector<int> pi;
    pi.push_back(0); // epsilon
    pi.push_back(0); // P[0,1]
    int k=0; // k: Länge des größten gefundenen Randes für P[0,q]
    // q: Index des Zeichens, um das verlängert wird
    for (unsigned q=1 ; q<p.length() ; q++) {
        if (p[k] == p[q]) // Können wir Rand verlängern?
            k++;
        else { // Nein: längsten Rand in P[1..k] suchen,
                // den wir erweitern können
            do {
                k = pi[k];
            } while ((k > 0) && (p[k] != p[q]))
            if (p[k] == p[q])
                k++; // +1, da Rand ja erweitert wird
        }
        pi.push_back(k); // pi[q]=k: Länge des Randes von P[0,q] merken
    }
    return pi;
}
```

Effiziente Berechnung von π_p (C++)

- Wir rechnen die **Längen der Ränder** aus
die eigentlichen Ränder sind für uns nicht wichtig

$$\text{pi}[q] = |\pi_p(P[0, q])|$$

```
vector<int> calcPi(const string& p) {
    vector<int> pi;
    pi.push_back(0); // epsilon
    pi.push_back(0); // P[0,1]
    int k=0; // k: Länge des größten gefundenen Randes für P[0,q]
    // q: Index des Zeichens, um das verlängert wird
    for (unsigned q=1 ; q<p.length() ; q++) {
        if (p[k] == p[q]) // Können wir Rand verlängern?
            k++;
        else { // Nein: längsten Rand in P[1..k] suchen,
                // den wir erweitern können
            do {
                k = pi[k];
            } while ((k > 0) && (p[k] != p[q]))
            if (p[k] == p[q])
                k++; // +1, da Rand ja erweitert wird
        }
        pi.push_back(k); // pi[q]=k: Länge des Randes von P[0,q] merken
    }
    return pi;
}
```

Initialisierungsaufwand: $O(m)$

KMP-Algorithmus (C++)

KMP-Algorithmus (C++)

```
void stringMatchKMP(vector<int>& result, string t, string p) {  
    vector<int> pi=calcPi(p);  
    int q=0; // q: Länge des Randes  
    for (unsigned i=0 ; i<t.length() ; ++i) { // Gesamten Text durchsuchen  
        if (p[q] == t[i]) { // Übereinstimmung an aktueller Position  
            q++; // Ja: nächste Stelle untersuchen  
        }  
        else { // Nein: Versatz berechnen  
            // passend bisher war war P[0,q]  
            do {  
                q = pi[q];  
            } while ((q > 0) && (p[q] != t[i]))  
            if (p[q]==t[i])  
                q=q+1;  
        }  
        if (q == p.length()) { // Musterende erreicht?  
            result.push_back(i-q+1); // Ja: Versatz merken  
            q=pi[q]; // Nächste Musterposition berechnen  
        }  
    }  
}
```

```
stringMatchKMP(positions,"blablablablaaabla","bla");  
for (int i=0 ; i<positions.size() ; i++)  
    cout << positions[i] << " ";
```

KMP-Algorithmus (C++)

```
void stringMatchKMP(vector<int>& result, string t, string p) {  
    vector<int> pi=calcPi(p);  
    int q=0; // q: Länge des Randes  
    for (unsigned i=0 ; i<t.length() ; ++i) { // Gesamten Text durchsuchen  
        if (p[q] == t[i]) { // Übereinstimmung an aktueller Position  
            q++; // Ja: nächste Stelle untersuchen  
        }  
        else { // Nein: Versatz berechnen  
            // passend bisher war war P[0,q]  
            do {  
                q = pi[q];  
            } while ((q > 0) && (p[q] != t[i]))  
            if (p[q]==t[i])  
                q=q+1;  
        }  
        if (q == p.length()) { // Musterende erreicht?  
            result.push_back(i-q+1); // Ja: Versatz merken  
            q=pi[q]; // Nächste Musterposition berechnen  
        }  
    }  
}
```

Gesamtaufwand: $O(m+n)$

```
stringMatchKMP(positions,"blablablablaaabla","bla");  
for (int i=0 ; i<positions.size() ; i++)  
    cout << positions[i] << " ";
```

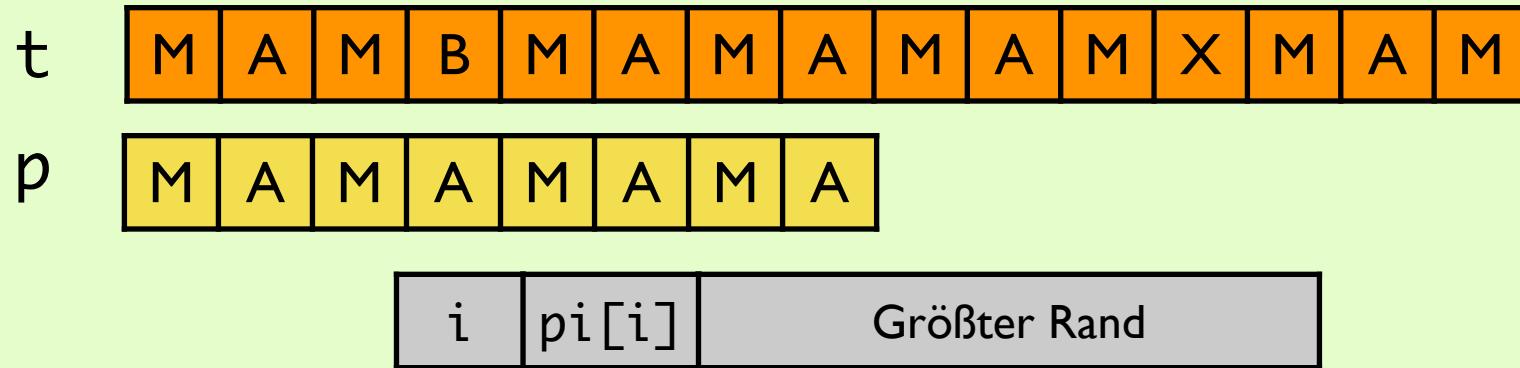
Anwendung des KMP: Beispiel

B Anwendung des KMP-Algorithmus

t	M	A	M	B	M	A	M	A	M	A	M	X	M	A	M
p	M	A	M	A	M	A	M	A	M	A	M				

Anwendung des KMP: Beispiel

B Anwendung des KMP-Algorithmus



**Berechnung
von π :**

Anwendung des KMP: Beispiel

B Anwendung des KMP-Algorithmus

t M A M B M A M A M A M X M A M

p M A M A M A M A

i	pi[i]	Größter Rand
0	0	

**Berechnung
von pi:**

Anwendung des KMP: Beispiel

B Anwendung des KMP-Algorithmus

t M A M B M A M A M A M X M A M

p M A M A M A M A

i	pi[i]	Größter Rand
0	0	
1	0	M

**Berechnung
von pi:**

Anwendung des KMP: Beispiel

B Anwendung des KMP-Algorithmus

t M A M B M A M A M A M X M A M

p M A M A M A M A

i	pi[i]	Größter Rand
0	0	
1	0	M
2	0	M A

**Berechnung
von pi:**

Anwendung des KMP: Beispiel

B Anwendung des KMP-Algorithmus

t

M	A	M	B	M	A	M	A	M	A	M	X	M	A	M
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

p

M	A	M	A	M	A	M	A
---	---	---	---	---	---	---	---

**Berechnung
von π :**

i	$\pi[i]$	Größter Rand
0	0	
1	0	M
2	0	M A
3	1	M A M

Anwendung des KMP: Beispiel

B Anwendung des KMP-Algorithmus

t M | A | M | B | M | A | M | A | M | A | M | X | M | A | M
p M | A | M | A | M | A | M | A

i	pi[i]	Größter Rand
0	0	
1	0	M
2	0	M A
3	1	M A M
4	2	M A M A

**Berechnung
von pi:**

Anwendung des KMP: Beispiel

B Anwendung des KMP-Algorithmus

t M | A | M | B | M | A | M | A | M | A | M | X | M | A | M
p M | A | M | A | M | A | M | A

i	pi[i]	Größter Rand
0	0	
1	0	M
2	0	M A
3	1	M A M
4	2	M A M A
5	3	M A M A M

**Berechnung
von pi:**

Vorsicht bei
Überlappungen
von Präfix und Suffix!

Anwendung des KMP: Beispiel

B Anwendung des KMP-Algorithmus

t M | A | M | B | M | A | M | A | M | A | M | X | M | A | M
p M | A | M | A | M | A | M | A

i	pi[i]	Größter Rand
0	0	
1	0	M
2	0	M A
3	1	M A M
4	2	M A M A
5	3	M A M A M
6	4	M A M A M A

**Berechnung
von pi:**

Vorsicht bei
Überlappungen
von Präfix und Suffix!

Anwendung des KMP: Beispiel

B Anwendung des KMP-Algorithmus

t M | A | M | B | M | A | M | A | M | A | M | X | M | A | M
p M | A | M | A | M | A | M | A

i	pi[i]	Größter Rand
0	0	
1	0	M
2	0	M A
3	1	M A M
4	2	M A M A
5	3	M A M A M
6	4	M A M A M A
7	5	M A M A M A M

**Berechnung
von pi:**

Vorsicht bei
Überlappungen
von Präfix und Suffix!

Anwendung des KMP: Beispiel

B Anwendung des KMP-Algorithmus

t M | A | M | B | M | A | M | A | M | A | M | X | M | A | M
p M | A | M | A | M | A | M | A

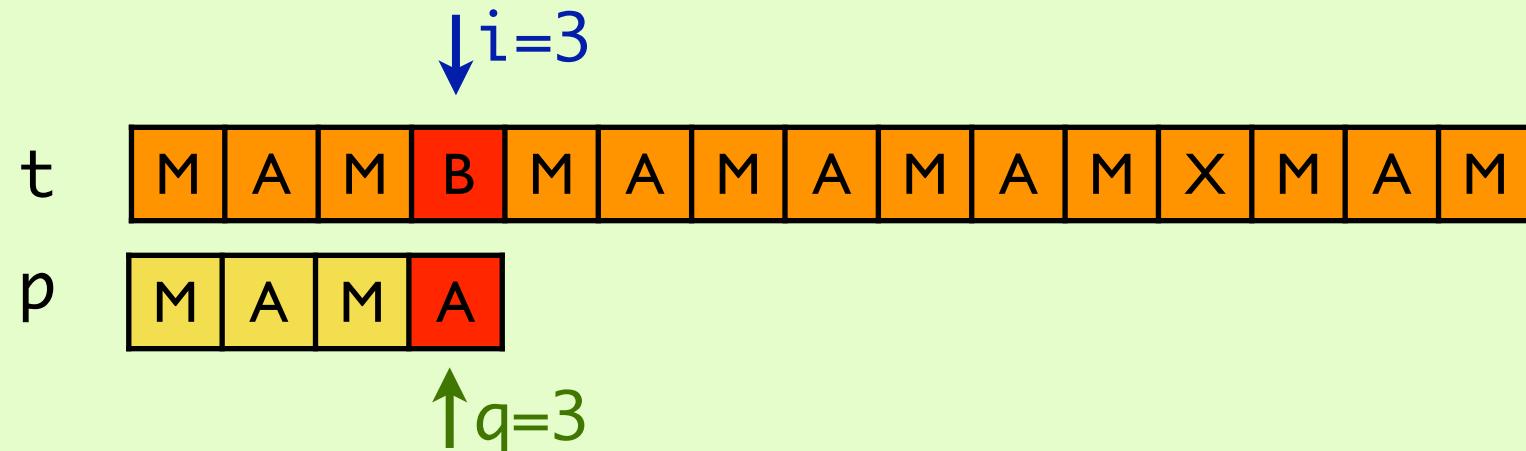
i	pi[i]	Größter Rand
0	0	
1	0	M
2	0	M A
3	1	M A M
4	2	M A M A
5	3	M A M A M
6	4	M A M A M A
7	5	M A M A M A M
8	6	M A M A M A M A

**Berechnung
von pi:**

Vorsicht bei
Überlappungen
von Präfix und Suffix!

Anwendung des KMP: Beispiel

B Anwendung des KMP-Algorithmus

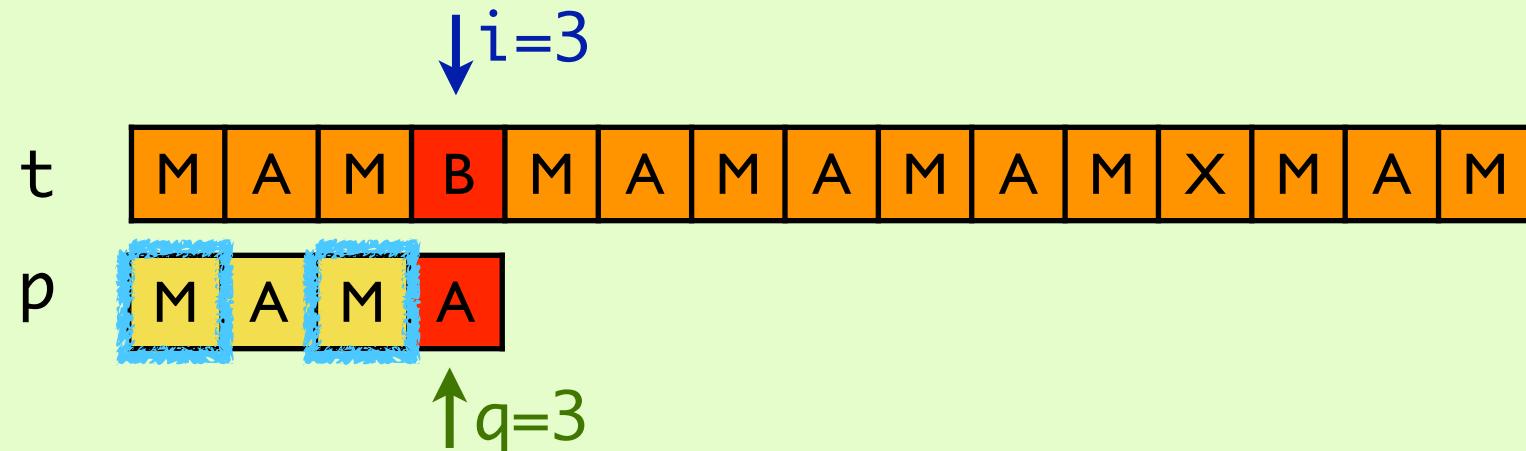


q	$\text{pi}[q]$	Größter Rand
0	0	
1	0	M
2	0	M A
3	1	M A M
4	2	M A M A
5	3	M A M A M
6	4	M A M A M A
7	5	M A M A M A M
8	6	M A M A M A M A

```
do {  
    q = pi[q];  
} while ((q > 0) && (p[q] != t[i]))  
  
if (p[q]==t[i])  
    q=q+1;
```

Anwendung des KMP: Beispiel

B Anwendung des KMP-Algorithmus

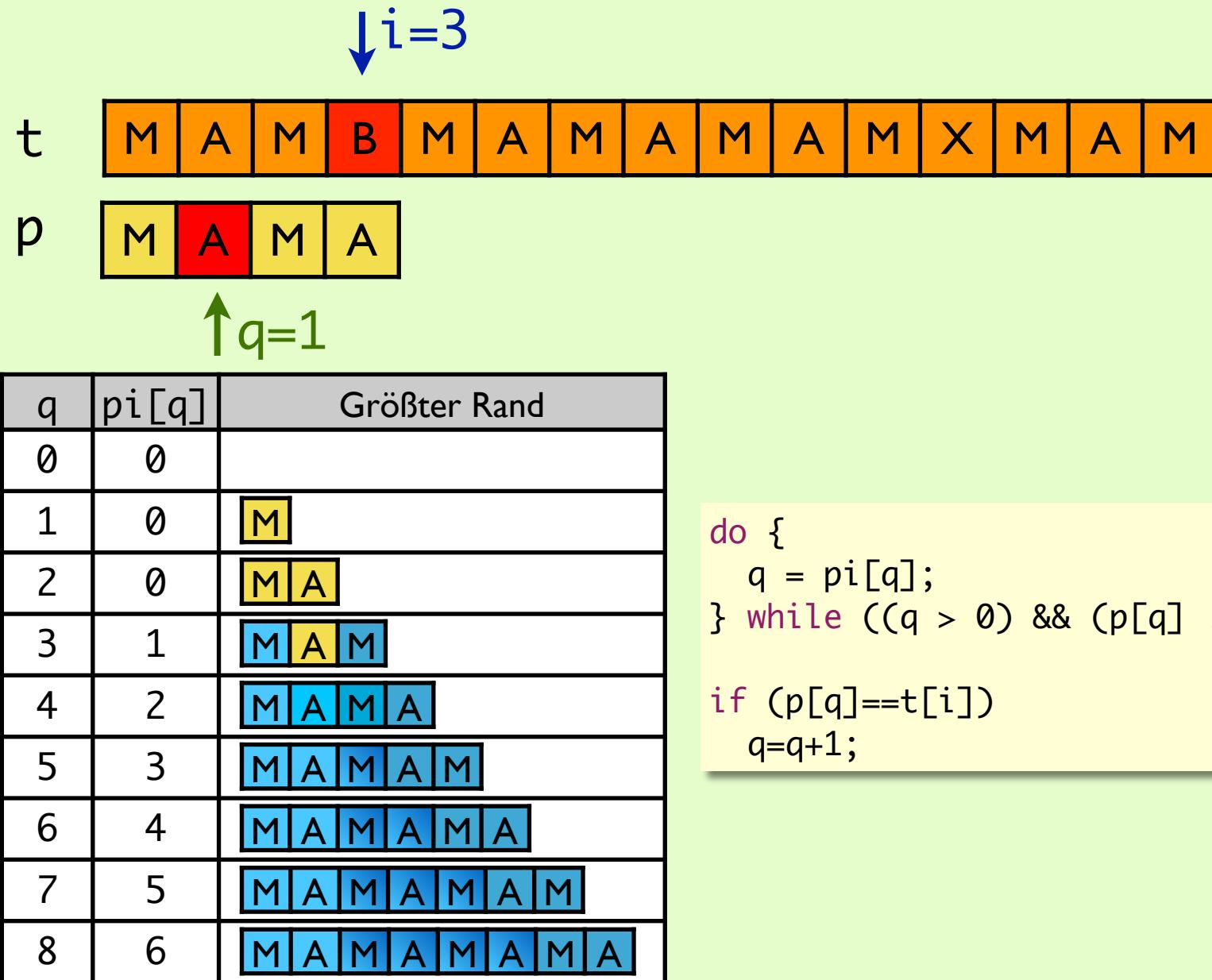


q	$\text{pi}[q]$	Größter Rand
0	0	
1	0	M
2	0	MA
3	1	MAM
4	2	MAMA
5	3	MAMAM
6	4	MAMAMA
7	5	MAMAMAM
8	6	MAMAMAMA

```
do {  
    q = pi[q];  
} while ((q > 0) && (p[q] != t[i]))  
  
if (p[q]==t[i])  
    q=q+1;
```

Anwendung des KMP: Beispiel

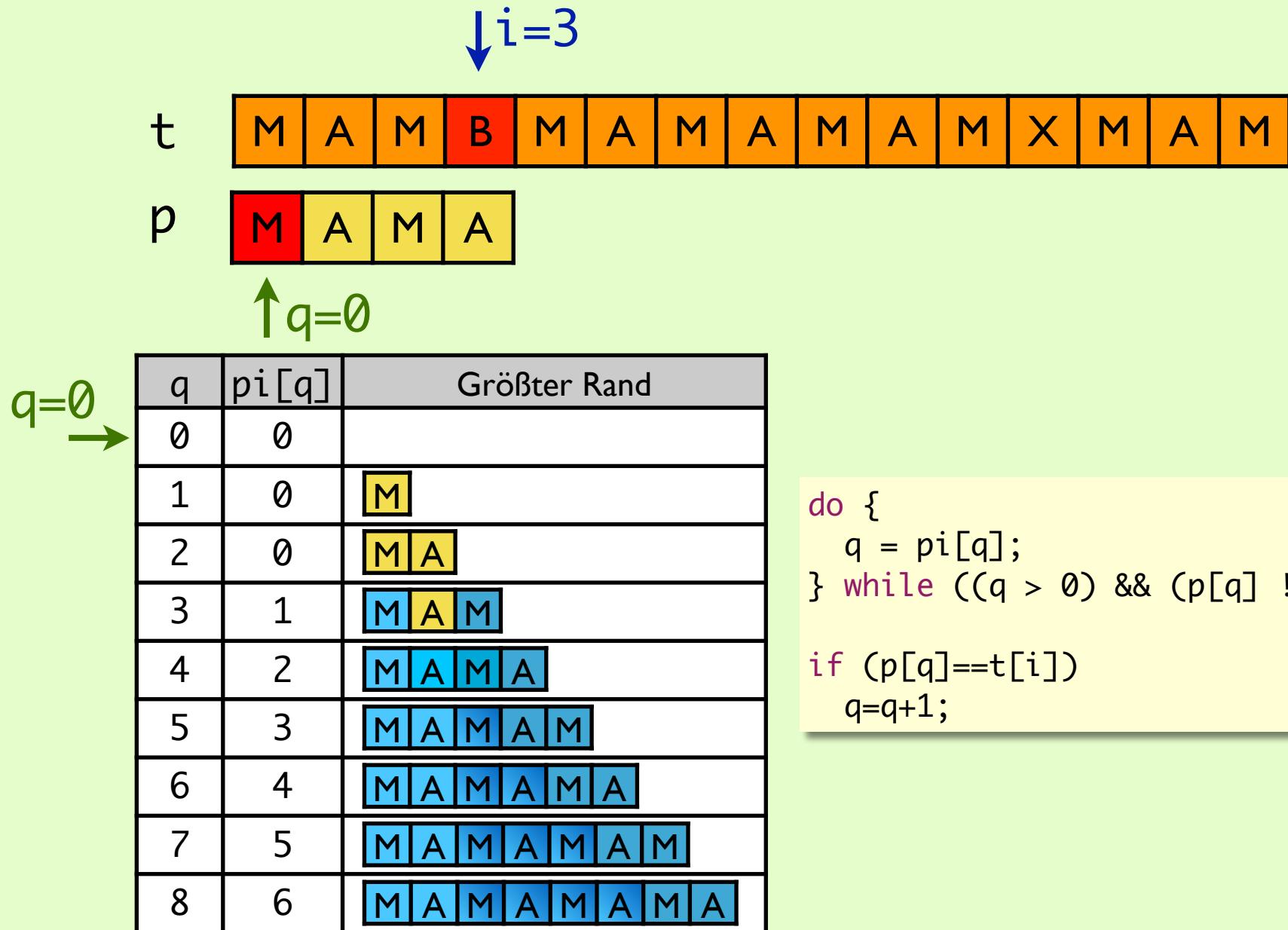
B Anwendung des KMP-Algorithmus



```
do {  
    q = pi[q];  
} while ((q > 0) && (p[q] != t[i]))  
  
if (p[q]==t[i])  
    q=q+1;
```

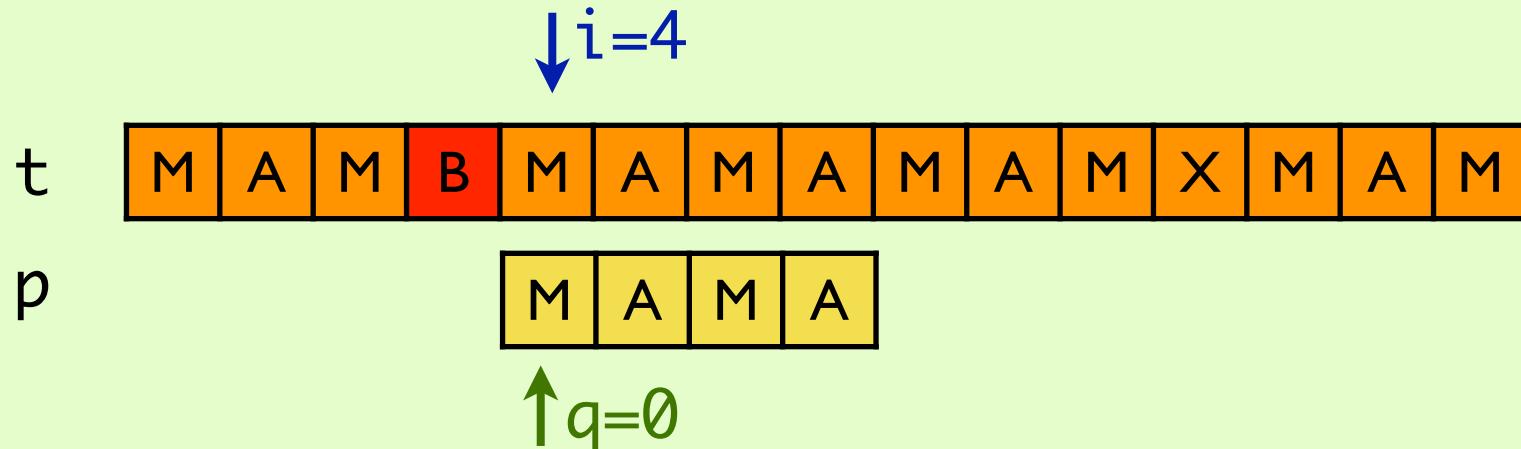
Anwendung des KMP: Beispiel

B Anwendung des KMP-Algorithmus



Anwendung des KMP: Beispiel

B Anwendung des KMP-Algorithmus



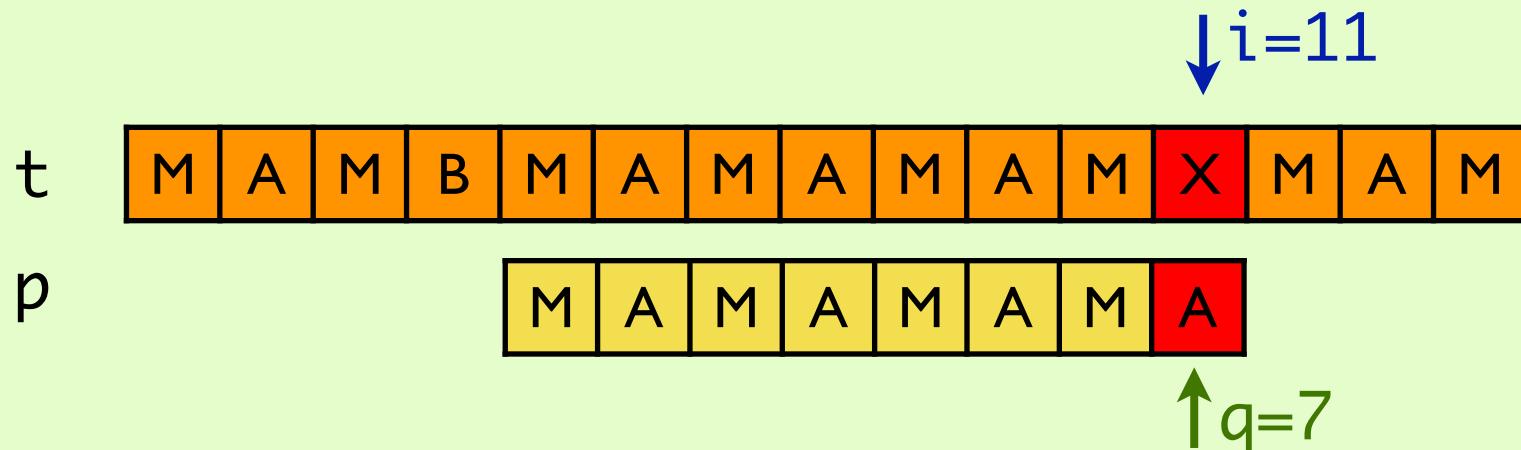
$q=0 \rightarrow$

q	pi[q]	Größter Rand
0	0	
1	0	M
2	0	MA
3	1	MAM
4	2	MAMA
5	3	MAMAM
6	4	MAMAMA
7	5	MAMAMAM
8	6	MAMAMAMA

```
do {  
    q = pi[q];  
} while ((q > 0) && (p[q] != t[i]))  
  
if (p[q]==t[i])  
    q=q+1;
```

Anwendung des KMP: Beispiel 2

B Anwendung des KMP-Algorithmus



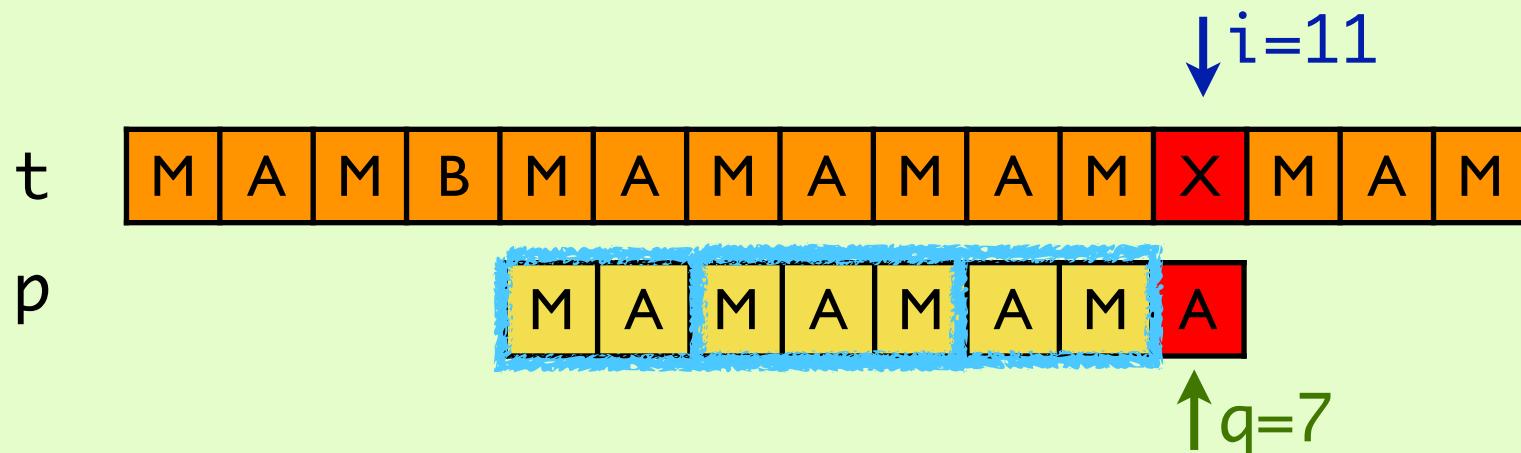
q	$\pi[q]$	Größter Rand
0	0	
1	0	M
2	0	MA
3	1	MAM
4	2	MAMA
5	3	MAMAM
6	4	MAMAMA
7	5	MAMAMAM
8	6	MAMAMAMA

$q=7 \rightarrow$

```
do {  
    q = pi[q];  
} while ((q > 0) && (p[q] != t[i]))  
  
if (p[q]==t[i])  
    q=q+1;
```

Anwendung des KMP: Beispiel 2

B Anwendung des KMP-Algorithmus



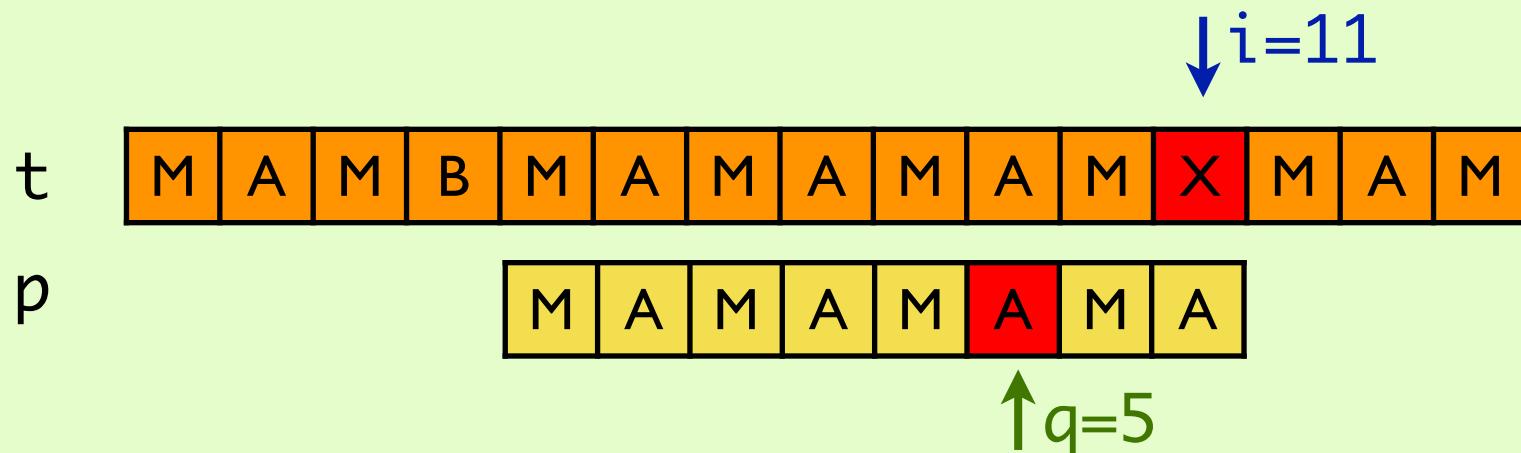
q	$\pi[q]$	Größter Rand
0	0	
1	0	M
2	0	MA
3	1	MAM
4	2	MAMA
5	3	MAMAM
6	4	MAMAMA
7	5	MAMAMAMA
8	6	MAMAMAMAMA

q=7
→

```
do {  
    q = pi[q];  
} while ((q > 0) && (p[q] != t[i]))  
  
if (p[q]==t[i])  
    q=q+1;
```

Anwendung des KMP: Beispiel 2

B Anwendung des KMP-Algorithmus



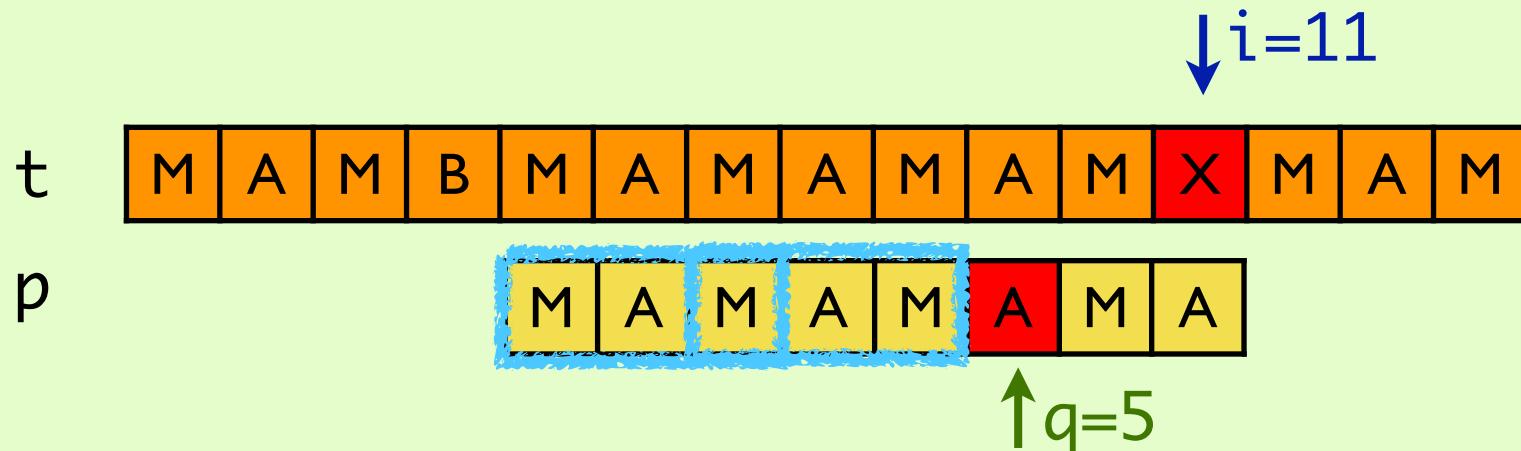
q	pi[q]	Größter Rand
0	0	
1	0	M
2	0	MA
3	1	MAM
4	2	MAMA
5	3	MAMAM
6	4	MAMAMA
7	5	MAMAMAM
8	6	MAMAMAMA

$q=5 \rightarrow$

```
do {  
    q = pi[q];  
} while ((q > 0) && (p[q] != t[i]))  
  
if (p[q]==t[i])  
    q=q+1;
```

Anwendung des KMP: Beispiel 2

B Anwendung des KMP-Algorithmus

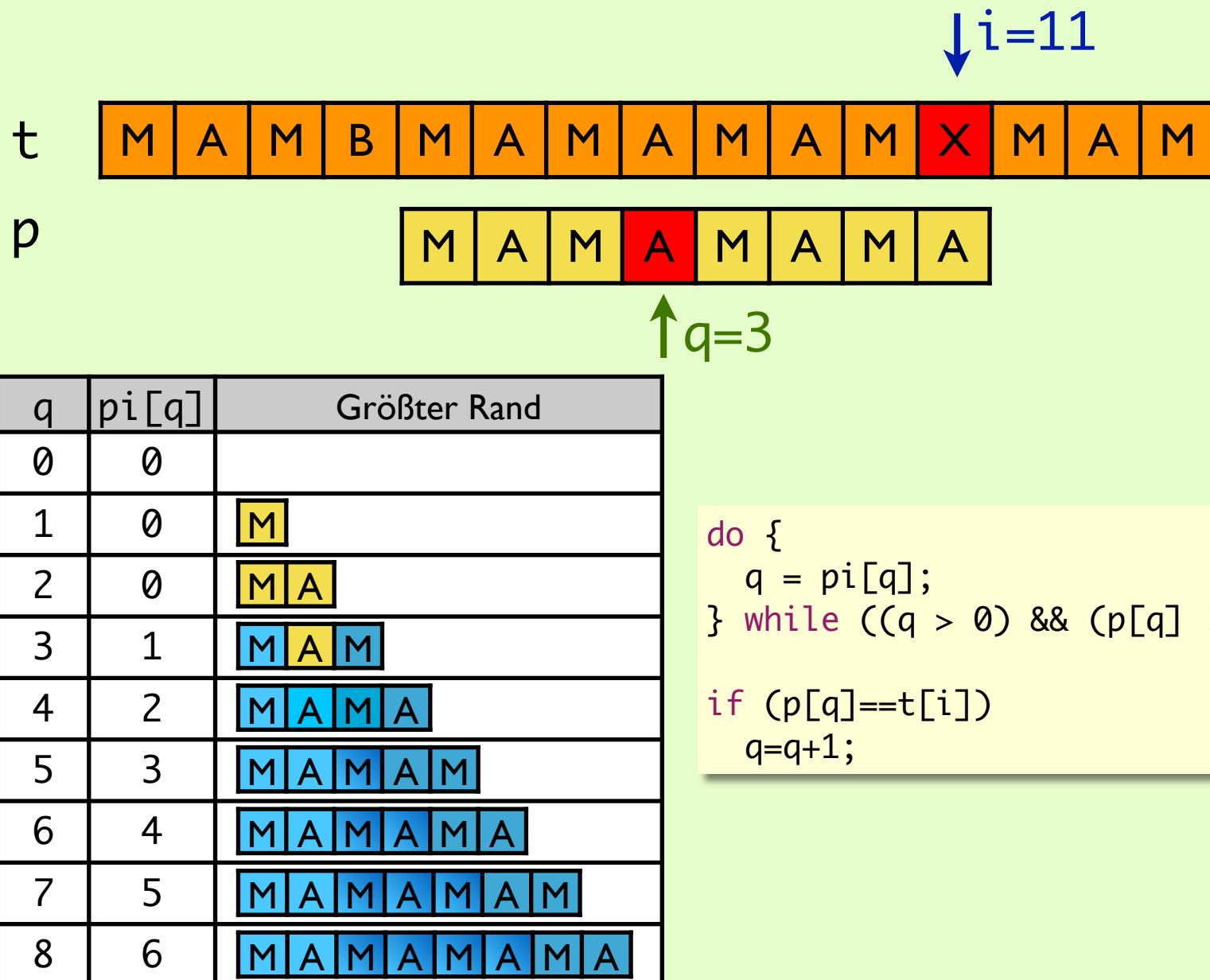


q	$\pi[q]$	Größter Rand
0	0	
1	0	M
2	0	MA
3	1	MAM
4	2	MAMA
5	3	MAMAM
6	4	MAMAMA
7	5	MAMAMAMA
8	6	MAMAMAMAMA

```
do {  
    q = pi[q];  
} while ((q > 0) && (p[q] != t[i]))  
  
if (p[q]==t[i])  
    q=q+1;
```

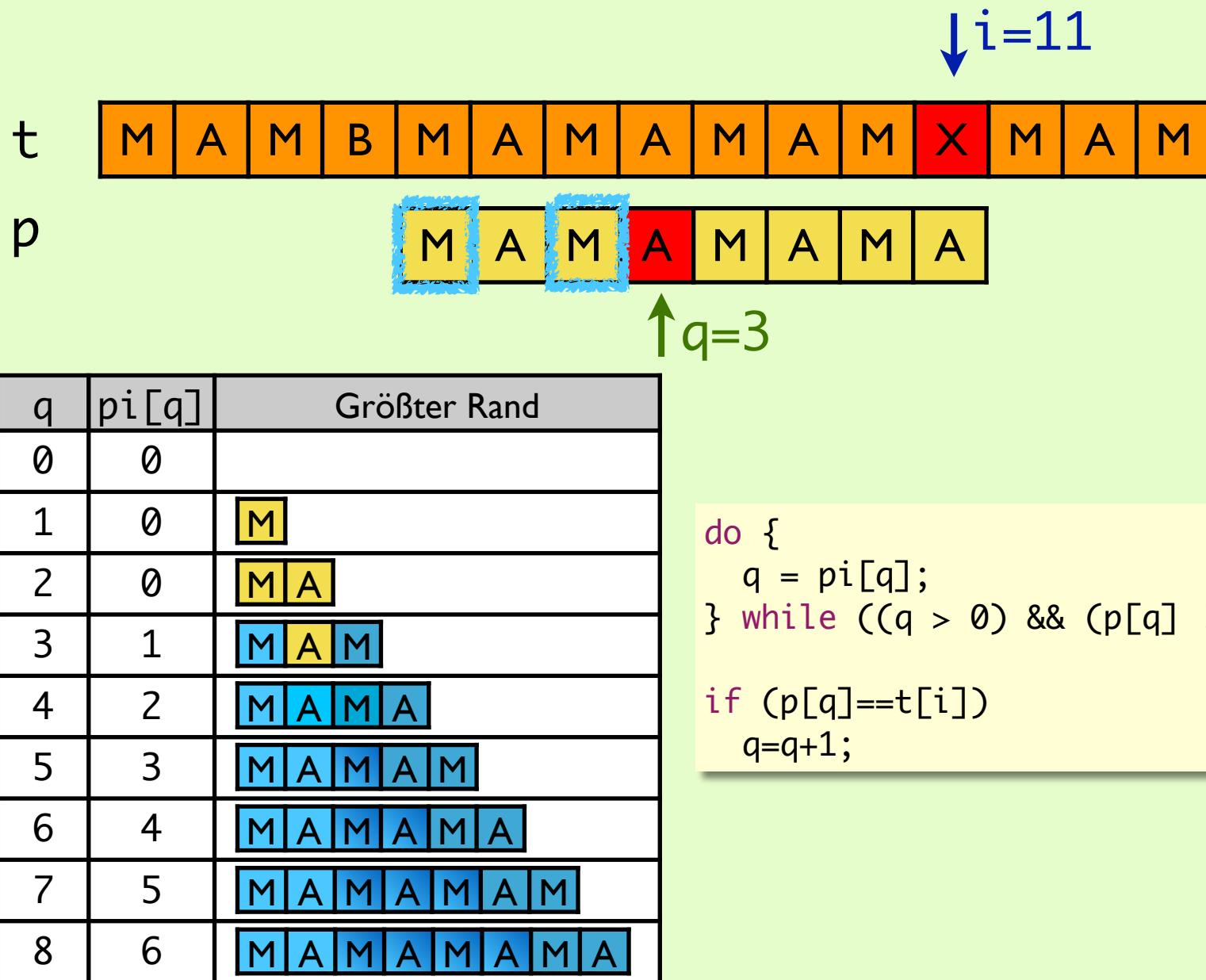
Anwendung des KMP: Beispiel 2

B Anwendung des KMP-Algorithmus



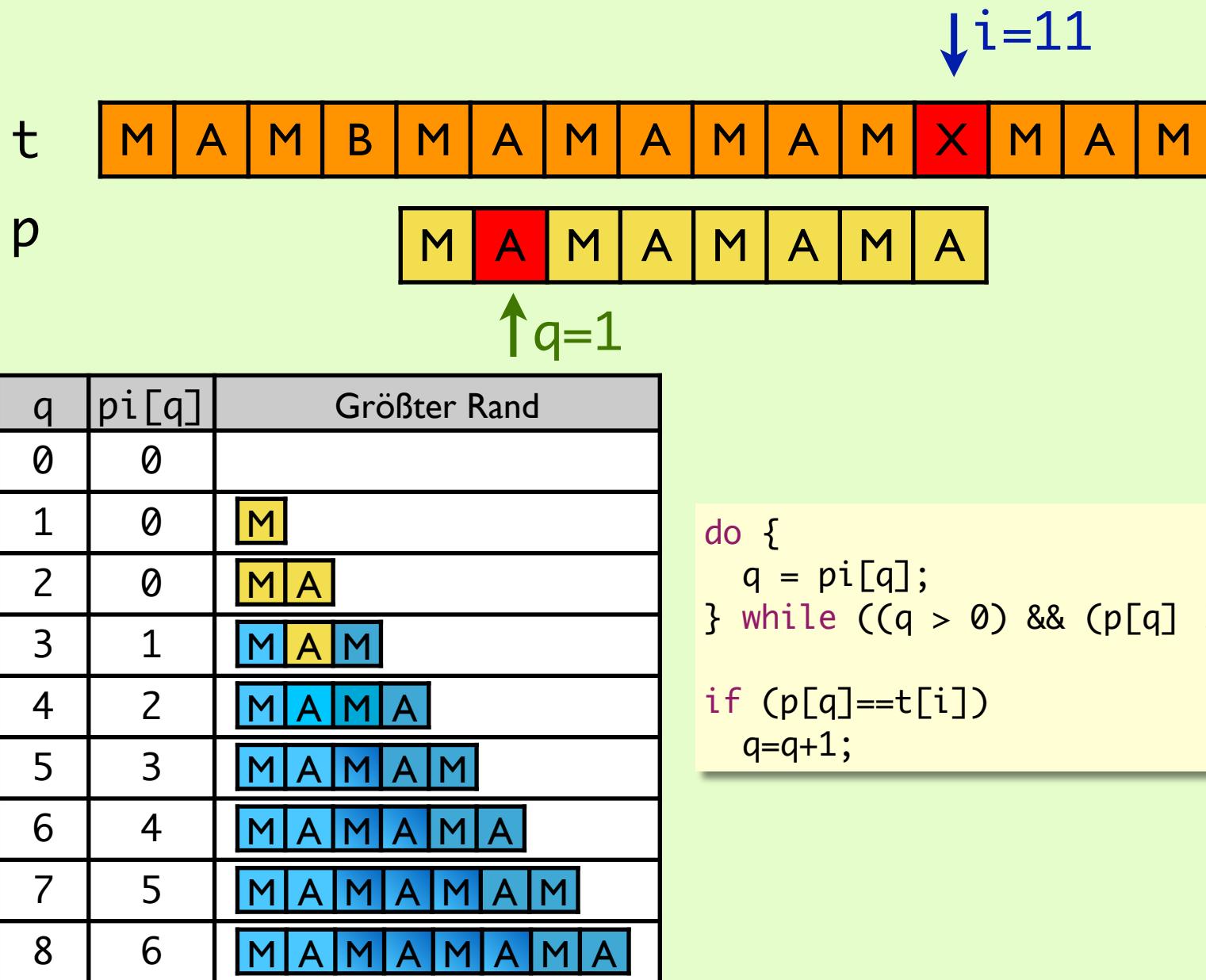
Anwendung des KMP: Beispiel 2

B Anwendung des KMP-Algorithmus



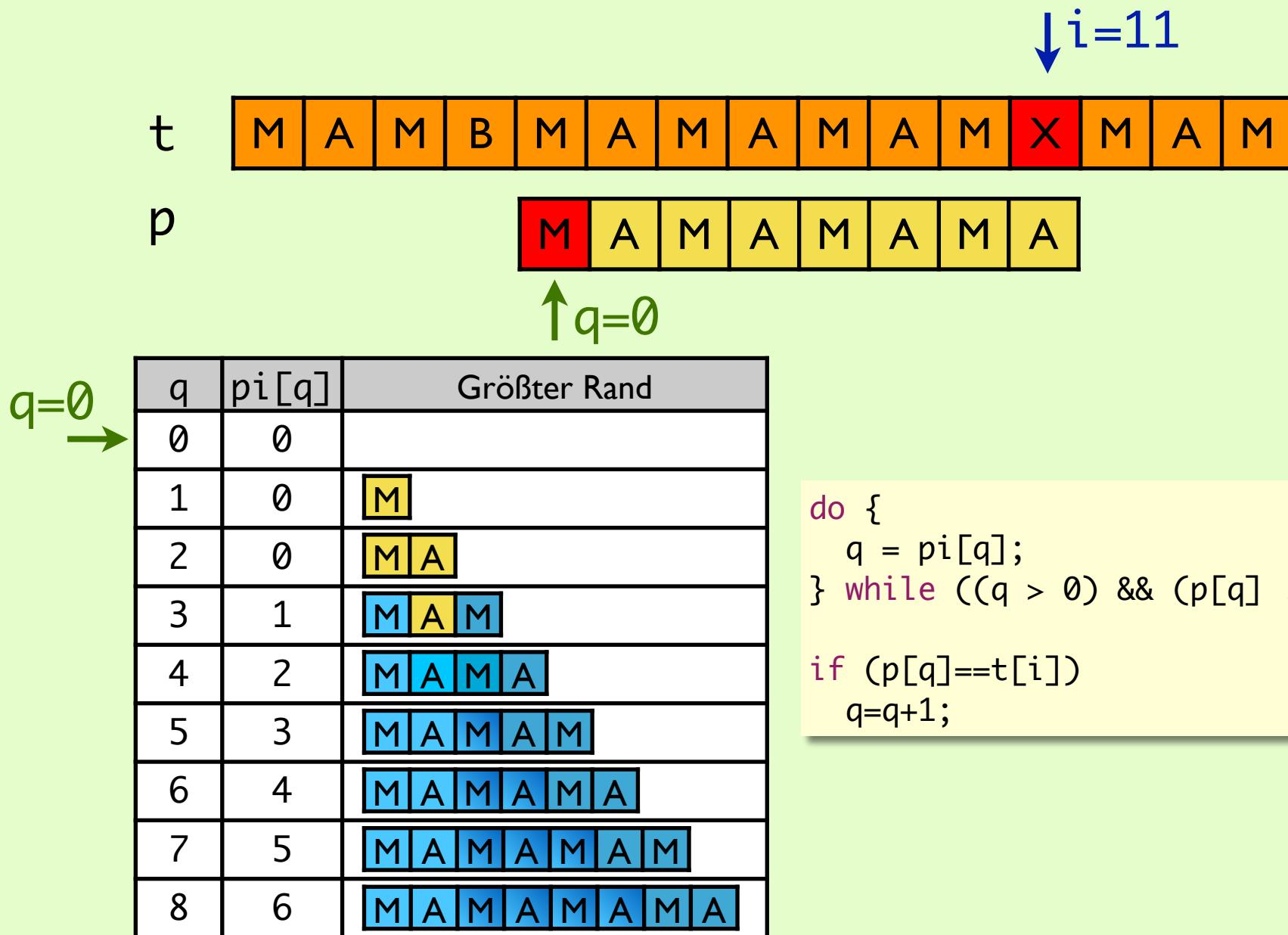
Anwendung des KMP: Beispiel 2

B Anwendung des KMP-Algorithmus



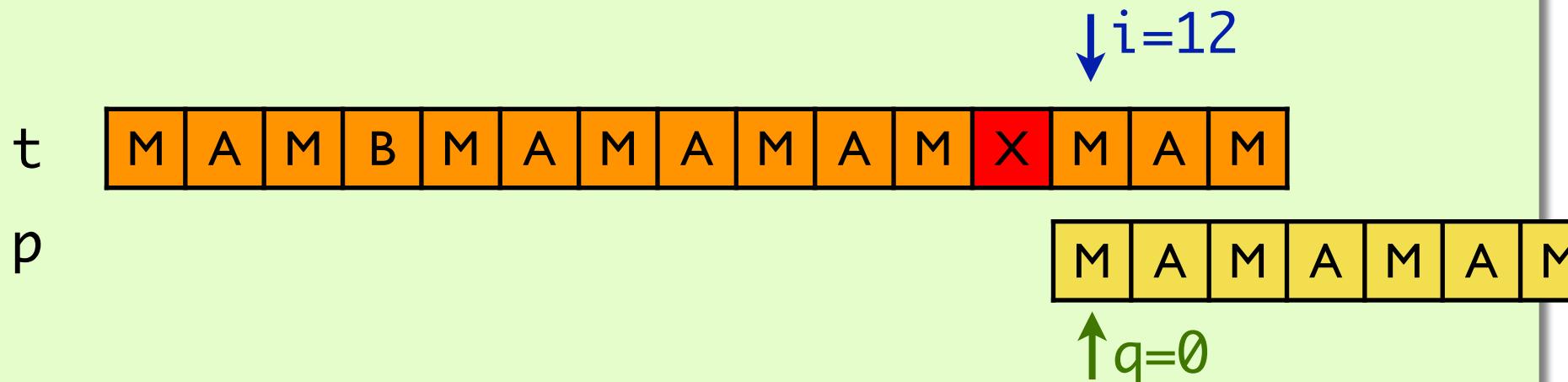
Anwendung des KMP: Beispiel 2

B Anwendung des KMP-Algorithmus



Anwendung des KMP: Beispiel 2

B Anwendung des KMP-Algorithmus



$q=0 \rightarrow$

q	$pi[q]$	Größter Rand
0	0	
1	0	M
2	0	MA
3	1	MAM
4	2	MAMA
5	3	MAMAM
6	4	MAMAMA
7	5	MAMAMAM
8	6	MAMAMAMA

```
do {  
    q = pi[q];  
} while ((q > 0) && (p[q] != t[i]))  
  
if (p[q]==t[i])  
    q=q+1;
```

VI. Suchen in Texten

6. Suchen in Texten

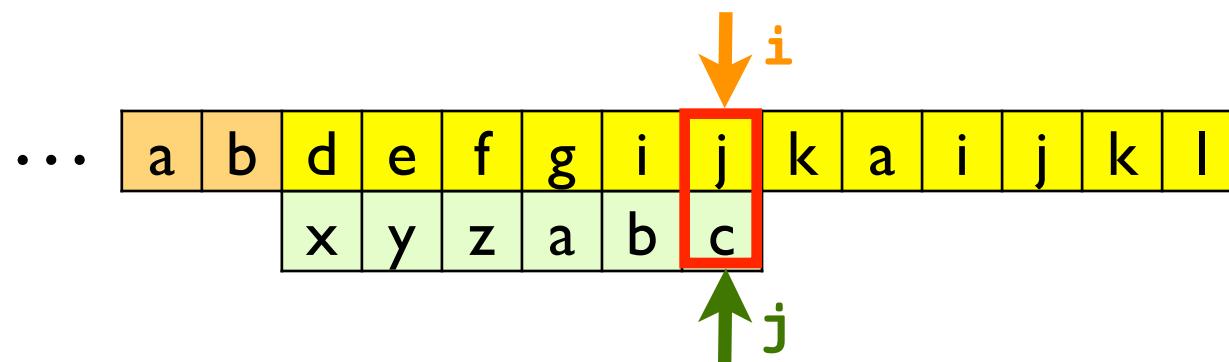
- 6.1. Einführung
- 6.2. Direkte Mustersuche
- 6.3. Das Verfahren von Rabin und Karp
- 6.4. Suche mit endlichen Automaten
- 6.5. Das Verfahren von Knuth, Morris und Pratt
- 6.6. Das Verfahren von Boyer und Moore**

Die Grundidee des Boyer-Moore Verfahrens

- **Muster wird von rechts nach links verglichen**
(nicht von links nach rechts, wie bei den bisher vorgestellten Verfahren!)
- **Dabei kommen zwei Heuristiken (Strategien) zum Einsatz:**
 - die ***Bad Character-Heuristik***
 - und als Ergänzung ggf. die ***Good Suffix-Heuristik***
- **Wir skizzieren hier nur die Idee des Boyer-Moore Verfahrens**

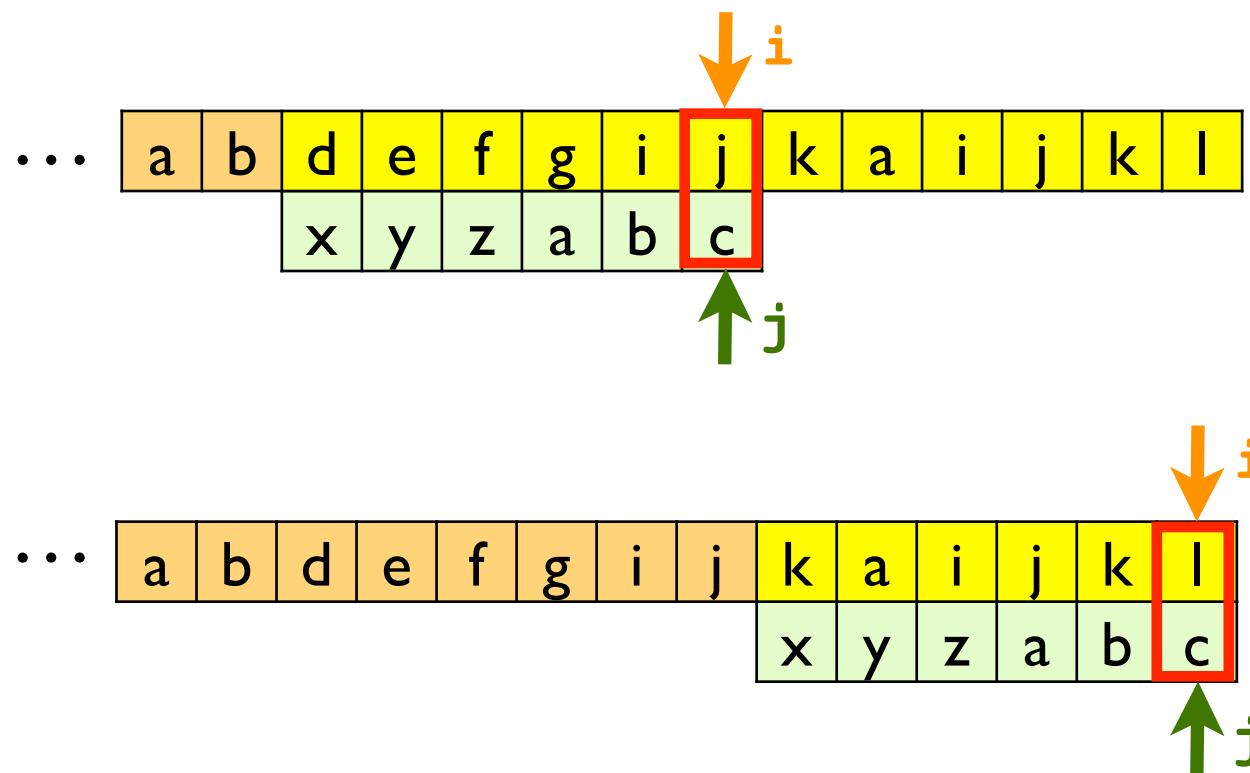
Die *Bad Character*-Heuristik - 1. Fall

- Behandelt den Fall, dass **erstes vergleichenes Zeichen** (letztes Zeichen im Muster) **nicht mit Text übereinstimmt**:
- Zwei Fälle sind denkbar
 1. Letztes Zeichen des Textes taucht nicht im Muster auf. Dann kann das Muster um Musterlänge m Positionen nach rechts verschoben werden:



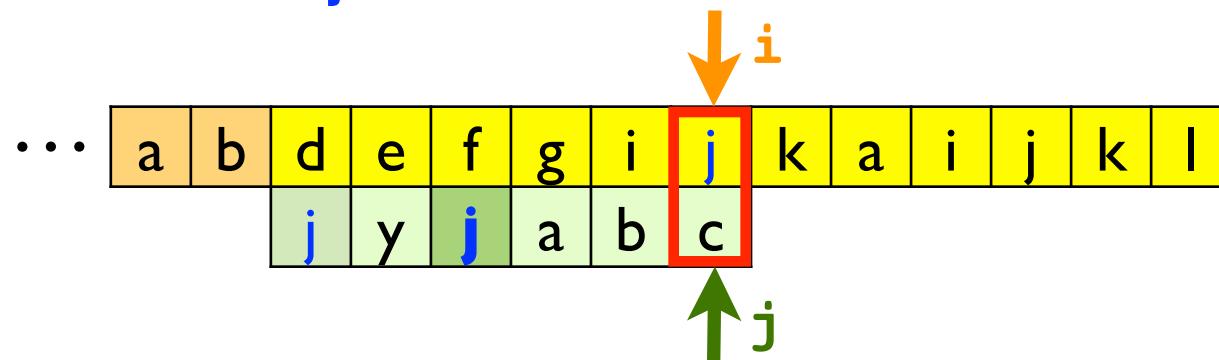
Die *Bad Character*-Heuristik - 1. Fall

- Behandelt den Fall, dass **erstes vergleichenes Zeichen** (letztes Zeichen im Muster) **nicht mit Text übereinstimmt**:
- Zwei Fälle sind denkbar
 1. Letztes Zeichen des Textes taucht nicht im Muster auf. Dann kann das Muster um Musterlänge m Positionen nach rechts verschoben werden:



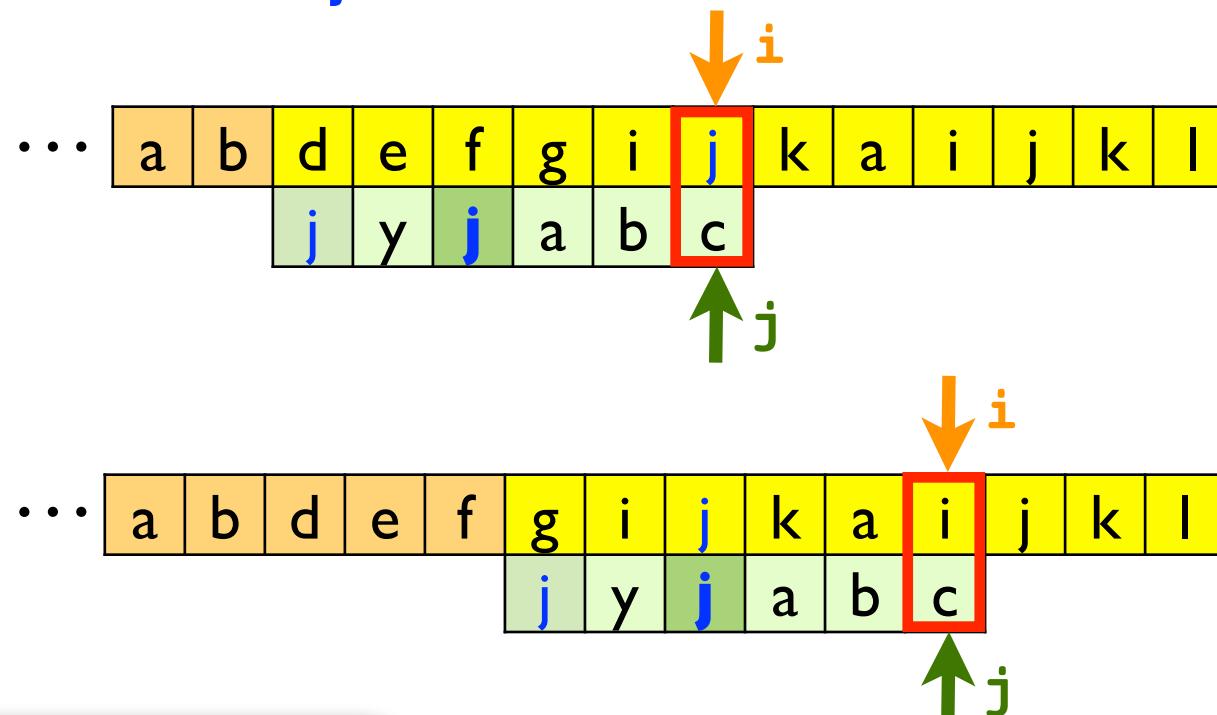
Die *Bad Character*-Heuristik - 2. Fall

- Behandelt den Fall, dass **erstes vergleichenes Zeichen** (letztes Zeichen im Muster) **nicht mit Text übereinstimmt**:
- Zwei Fälle sind denkbar
 1. Letztes Zeichen des Textes taucht nicht im Muster auf. Dann kann das Muster um Musterlänge m Positionen nach rechts verschoben werden.
 2. Das letzte Zeichen des Textes (hier **j**) taucht im Muster auf. Dann kann das Muster um p Positionen nach rechts verschoben werden, wobei p der Abstand des letzten **j** im Muster zum Musterende ist:



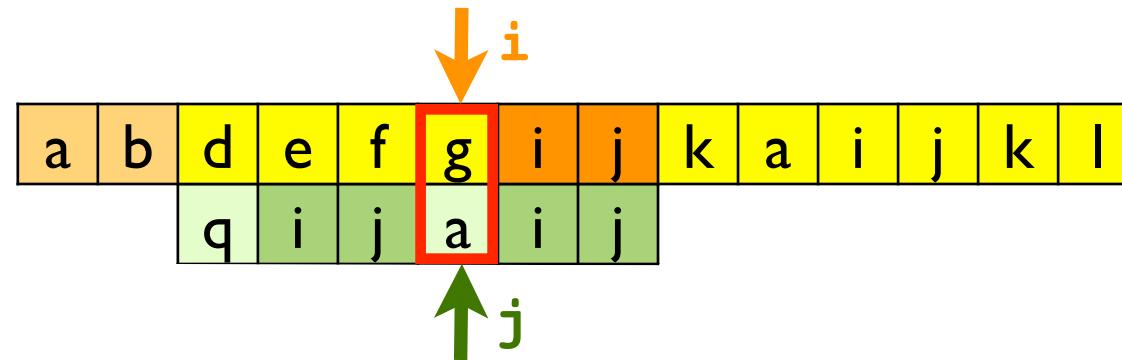
Die *Bad Character*-Heuristik - 2. Fall

- Behandelt den Fall, dass **erstes vergleichenes Zeichen** (letztes Zeichen im Muster) **nicht mit Text übereinstimmt**:
- Zwei Fälle sind denkbar
 1. Letztes Zeichen des Textes taucht nicht im Muster auf. Dann kann das Muster um Musterlänge m Positionen nach rechts verschoben werden.
 2. Das letzte Zeichen des Textes (hier **j**) taucht im Muster auf. Dann kann das Muster um p Positionen nach rechts verschoben werden, wobei p der Abstand des letzten **j** im Muster zum Musterende ist:



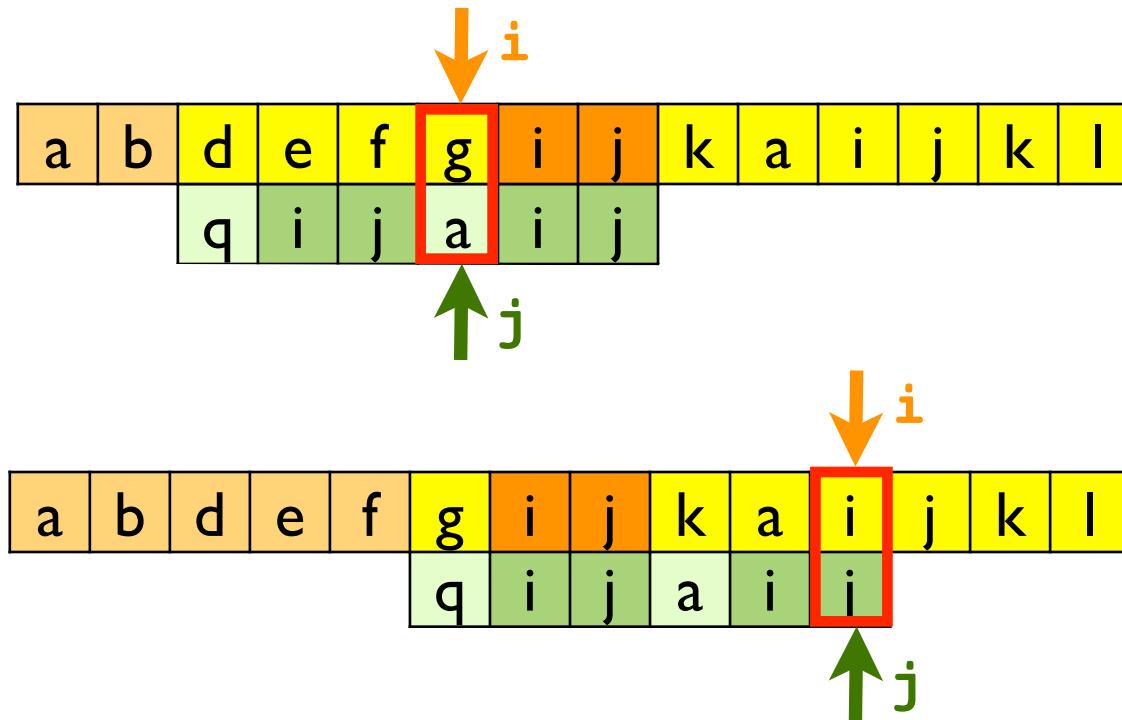
Die *Good Suffix*-Heuristik 1. Fall

- Hier wird die Selbstwiederholung von Mustern genutzt
- Drei Fälle sind denkbar
 - 1. Kommt das Suffix des bereits übereinstimmenden Suchmusters wiederholt im Suchmuster vor, schiebt man das Muster entsprechend weit nach rechts



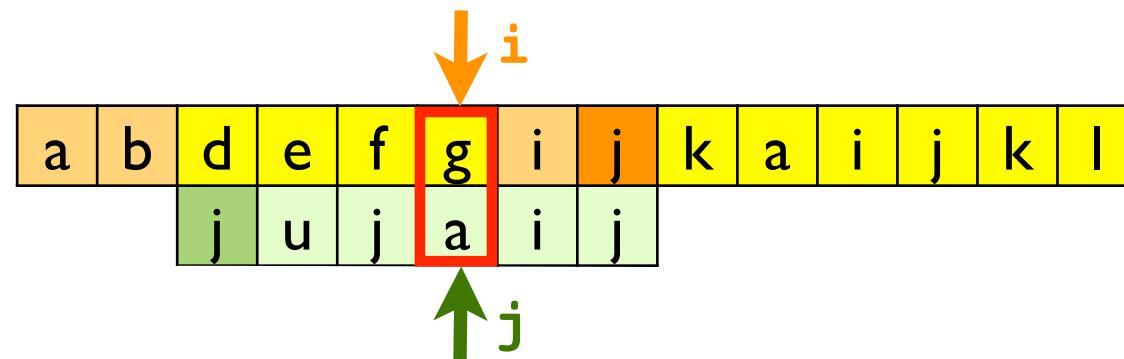
Die *Good Suffix*-Heuristik 1. Fall

- Hier wird die Selbstwiederholung von Mustern genutzt
- Drei Fälle sind denkbar
 - 1. Kommt das Suffix des bereits übereinstimmenden Suchmusters wiederholt im Suchmuster vor, schiebt man das Muster entsprechend weit nach rechts



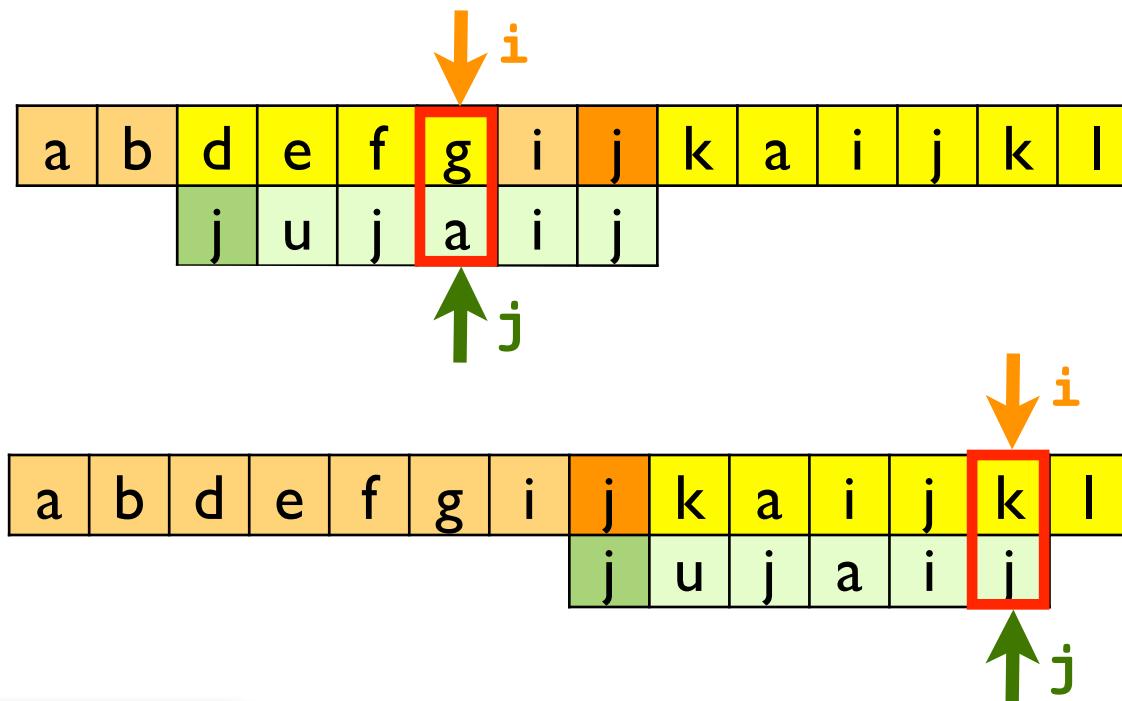
Die *Good Suffix*-Heuristik 2. Fall

- Hier wird die Selbstwiederholung von Mustern genutzt
- Drei Fälle sind denkbar
 1. Kommt das Suffix des bereits übereinstimmenden Suchmusters wiederholt im Suchmuster vor, schiebt man das Muster entsprechend weit nach rechts
 2. Man kann auch den Fall ausnutzen dass ein echtes Suffix des bereits übereinstimmenden Muster, gleichzeitig Präfix des Musters ist (maximaler Rand des Musters, welcher Suffix des bereits übereinstimmenden Teils ist)



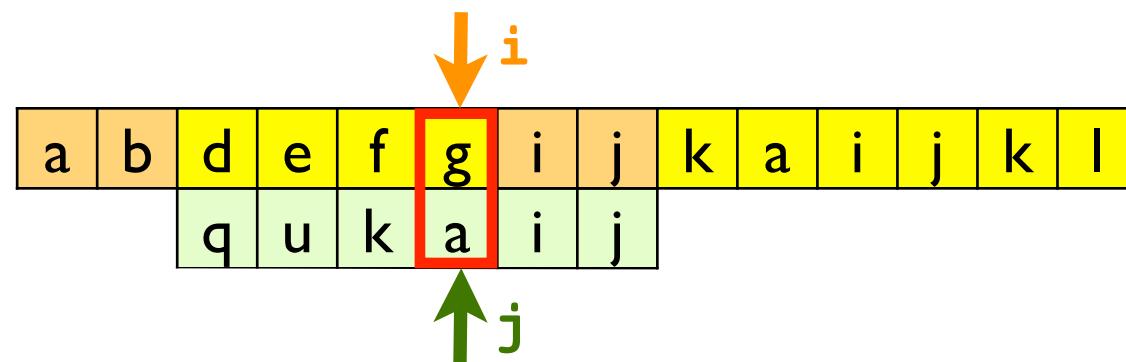
Die *Good Suffix*-Heuristik 2. Fall

- Hier wird die Selbstwiederholung von Mustern genutzt
- Drei Fälle sind denkbar
 1. Kommt das Suffix des bereits übereinstimmenden Suchmusters wiederholt im Suchmuster vor, schiebt man das Muster entsprechend weit nach rechts
 2. Man kann auch den Fall ausnutzen dass ein echtes Suffix des bereits übereinstimmenden Muster, gleichzeitig Präfix des Musters ist (maximaler Rand des Musters, welcher Suffix des bereits übereinstimmenden Teils ist)



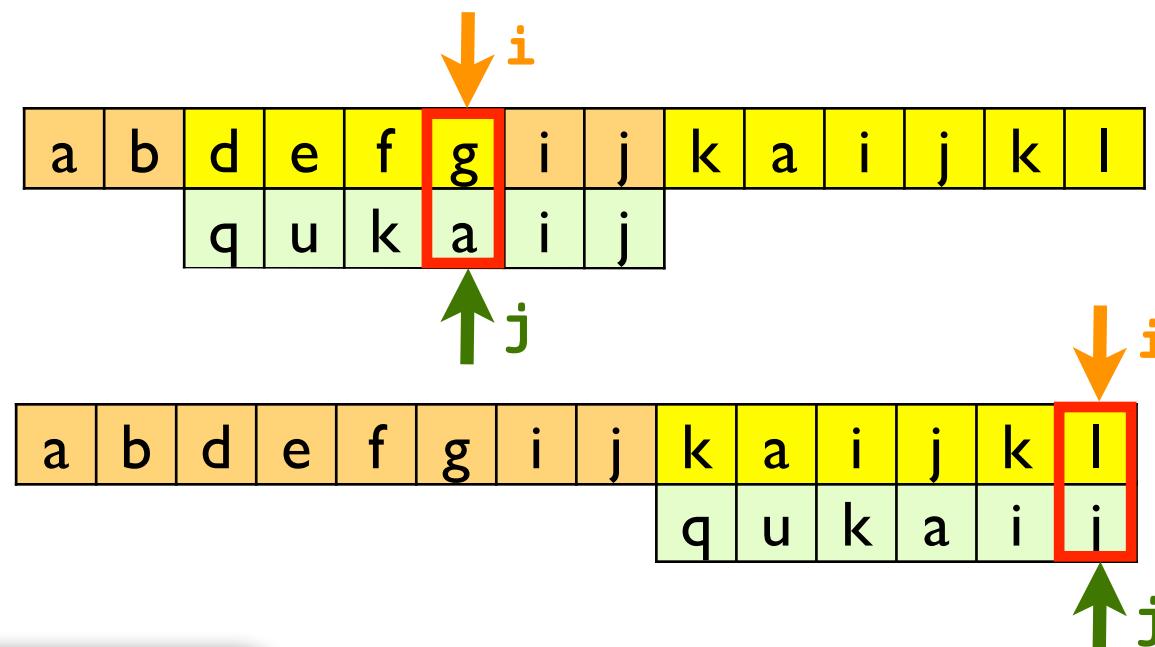
Die *Good Suffix*-Heuristik 2. Fall

- Hier wird die Selbstwiederholung von Mustern genutzt
 - Drei Fälle sind denkbar
 1. Kommt das Suffix des bereits übereinstimmenden Suchmusters wiederholt im Suchmuster vor, schiebt man das Muster entsprechend weit nach rechts
 2. Man kann auch den Fall ausnutzen dass ein echtes Suffix des bereits übereinstimmenden Muster, gleichzeitig Präfix des Musters ist (maximaler Rand des Musters, welcher Suffix des bereits übereinstimmenden Teils ist)
 3. Andernfalls kann man das Muster um m Stellen nach rechts bewegen



Die *Good Suffix*-Heuristik 2. Fall

- Hier wird die Selbstwiederholung von Mustern genutzt
- Drei Fälle sind denkbar
 1. Kommt das Suffix des bereits übereinstimmenden Suchmusters wiederholt im Suchmuster vor, schiebt man das Muster entsprechend weit nach rechts
 2. Man kann auch den Fall ausnutzen dass ein echtes Suffix des bereits übereinstimmenden Muster, gleichzeitig Präfix des Musters ist (maximaler Rand des Musters, welcher Suffix des bereits übereinstimmenden Teils ist)
 3. Andernfalls kann man das Muster um m Stellen nach rechts bewegen



Kombination der Heuristiken

- **Ermittle Verschiebung mit beiden Heuristiken und wähle den größeren**
- **Gewinn der *Good Suffix*-Heuristik bei großem Alphabet eher gering**
 - Wahrscheinlichkeit der Wiederholung eines Teilmusters eher klein
 - zusätzlicher Aufwand zur Berechnung der Ränder
- **Aber: Vorteil bei Suche in binären Daten**
 - Hier wird *Bad Character*-Heuristik keinen großen Versatz erlauben, aber grössere Wahrscheinlichkeit wiederholender Teilmuster