

Berechnung des next-Arrays

- Erinnerung: $\text{next}[i]$ = Länge des längsten Präfixes von P , das echtes Suffix von $P_{1..i}$ ist
- Initialisierung: $\text{next}[1] = 0$
- Annahme:
sei $\text{next}[i-1] = j$:

P_1	P_2	P_{j-1}	P_j	...
						?
P_1	...	P_j	P_{j+1}	...	P_m	
				?		
P_1	...	P_j	P_{j+1}	...		
- Betrachte zwei Fälle:
 - $P_j = P_{j+1} \Rightarrow \text{next}[i] = j + 1$
 - $P_j \neq P_{j+1} \Rightarrow$ versuche nächst-kleineren Präfix für $P_{1..j}$, d.h., ersetze j durch $j' = \text{next}[j]$, bis $P_j = P_{j'+1}$ oder $j = 0$; falls $P_j = P_{j'+1}$, kann $\text{next}[i] = j'+1$ gesetzt werden, sonst ist $\text{next}[i] = 0$
- Fazit: Algo ist sehr ähnlich zum eigentlichen KMP-Algo von vorhin

G. Zachmann Informatik 2 — SS 10
Preprocessing 30

```

# Input:  Muster P
# Output: next-Array für P
def compute_next( P ):
    m = len( P )
    next = m * [0]
    next[1] = 0
    j = 0
    for i in range( 2, m+1 ):
        while j > 0 and P[i] != P[j+1]:
            j = next[j]
        if P[i] == P[j+1]:
            j += 1
        next[i] = j
    return next

```

G. Zachmann Informatik 2 — SS 10
Preprocessing 31

Die Gesamtlaufzeit von KMP

- **Satz:**
Der KMP-Algorithmus kann in Zeit $O(n + m)$ ausgeführt werden.
- M.a.W.:
das String-Matching-Problem kann in Zeit $O(n + m)$ gelöst werden
- Kann die Textsuche noch schneller sein?
 - "nein" im Worst-Case
 - "ja" im Average-Case

G. Zachmann Informatik 2 — SS 10 Preprocessing 32

Das Verfahren nach Boyer-Moore (BM)

- Gleiche Worst-Case-Laufzeit wie KMP
- Viel bessere Laufzeit in der Praxis
- Basiert auf 2 "Heuristiken"
 - "Bad Character"-Heuristik (Vorkommensheuristik)
 - "Good Suffix"-Heuristik (Match-Heuristik; ähnlich zu KMP)
- Kompletter Algo mit beiden Heuristiken ist etwas knifflig ;-)

G. Zachmann Informatik 2 — SS 10 Preprocessing 33

Die Idee

- Das Pattern von links nach rechts anlegen, aber zeichen-weise von rechts nach links vergleichen

Beginne Vergleich am Ende

Erstes falsches Zeichen ist wieder ein "a"! Großen Sprung machen!

Bingo! Noch einen großen Sprung machen!

Es gibt kein "a" im Such-Muster. Wir können um $m+1$ Zeichen verschieben

Das wars! 10 Zeichen verglichen und fertig!

G. Zachmann Informatik 2 — SS 10 Preprocessing 34

Die "Bad Character"-Heuristik (Vorkommensheuristik)

Es gibt kein "a" im Such-Muster. Wir können um $j - \lambda[a] = 4-0$ Zeichen verschieben

"p" tritt in "piti" an erster Position auf → verschiebe um $j - \lambda[p] = 4-1 = 3$ Zeichen

"t" tritt in "piti" an 3. Stelle auf → verschiebe um: $j - \lambda[t] = 4-3 = 1$ Zeichen

Es gibt kein "a" im Suchmuster Wir können um mindestens $j - \lambda[a] = 2-0$ Zeichen verschieben

λ = Funktion, die die "Bad Char"-Heuristik implementiert. Muß vor dem eigtl. Matching-Scan des Textes vorberechnet werden.

G. Zachmann Informatik 2 — SS 10 Preprocessing 35

Berechnung der Vorkommensheuristik (die Fkt λ)

- Für $c \in \Sigma$ und das Pattern P definiere

$$\delta(c) := \text{Index des von rechts her ersten Vorkommens von } c \text{ in } P$$

$$= \begin{cases} 0 & \text{falls } c \notin P \\ \max \{j \mid P[j] = c\} & \text{falls } c \in P \end{cases}$$

```

for a in  $\Sigma$ :
     $\delta[a] = 0$ 
for j = 1 .. m:
     $\delta[ P[j] ] = j$ 
return  $\delta$ 
            
```

G. Zachmann Informatik 2 — SS 10 Preprocessing 36

- Im Folgenden seien
 - c = das den Mismatch verursachende Zeichen im Text
 - j = Index des aktuellen Zeichens im Muster ($c \neq P_j$)
- Fall 1:** c kommt nicht im Muster P vor $\rightarrow \delta(c) = 0$

- Fazit:** verschiebe das Muster um $j - \delta(c)$ Positionen nach rechts

G. Zachmann Informatik 2 — SS 10 Preprocessing 37

■ **Fall 2a:** c kommt im Muster P vor und $0 < \delta(c) < j$:

Text: $i+1$ $i+j$ $i+m$
 Muster: c P_j P_m
 $\delta(c)$ $j - \delta(c)$ k

■ **Fazit:** verschiebe das Muster soweit nach rechts, daß das "rechtteste" c im Muster über einem potentiellen c im Text liegt
 ■ **Verschiebung des "rechtesten" c im Muster auf c im Text:**
 → Verschiebung um $k = j - \delta(c)$

G. Zachmann Informatik 2 — SS 10 Preprocessing 38

■ **Fall 2b:** c kommt im Muster P vor und $\delta(c) > j$:

Text: $i+1$ $i+j$ $i+m$
 Muster: P_j c P_m
 $\delta(c)$ $m - \delta(c) + 1$ $\text{evtl. } c$ $\text{kein } c$

■ **Fazit:** Verschiebung des "rechtesten" c im Muster auf ein potentielles c im Text → Verschiebung um $m - \delta(c) + 1$

G. Zachmann Informatik 2 — SS 10 Preprocessing 39

BM-Algorithmus, 1.Version

```

n = len( T )
m = len( P )
berechne  $\delta$ 
i = 0
while i <= n - m:
    j = m
    while j > 0 and P[j] == T[i+j]:
        j -= 1
    if j == 0:
        gib Verschiebung i aus
        i += 1
    else:
        d =  $\delta( T[i+j] )$ 
        if d > j:
            i += m + 1 - d
        else:
            i += j - d

```

G. Zachmann Informatik 2 — SS 10 Preprocessing 40

Zusammenfassung bis jetzt und Analyse

- Methode:
 - Vergleiche das Muster von **rechts nach links** mit dem Text und springe bei Nicht-Übereinstimmung möglichst weit nach rechts
 - Insbesondere: springe um die volle Musterlänge, wenn nicht übereinstimmendes Text-Zeichen nicht im Muster vorkommt
- Laufzeit in der Praxis: $O\left(\frac{n}{m}\right)$
 - Insbesondere bei großen Alphabeten und kurzen Mustern
 - Typisch bei Textverarbeitungsprogrammen
- Laufzeit im Worst-Case: $O(n \cdot m)$

0	0	...	0	0	...	0	...	0	...	
			\xrightarrow{i}							
			1	0	...	0	...	0		

- Gewünschte Laufzeit: $c \cdot \left(m + \frac{n}{m}\right)$

G. Zachmann Informatik 2 — SS 10 Preprocessing 41

Verbesserungsansatz

- Bisher verwendete Vorkommensheuristik nutzt nicht das Wissen über die bereits besuchten und übereinstimmenden Zeichen
- Kombination mit Match-Heuristik, ähnlich der des KMP-Algorithmus
- Ausnutzen von Selbstähnlichkeit des Musters
- Verhindern der Worst-Case-Laufzeit
- Eigenschaften
 - Worst-Case-Laufzeit mit Vorberechnung: $O(n + m)$
 - durchschnittliche Laufzeit immer noch: $O(\frac{n}{m})$

G. Zachmann Informatik 2 — SS 10 Preprocessing 42

Die "Good Suffix"-Heuristik (Match-Heuristik)

- Nutze die bis zum Auftreten eines Mismatches $P_j \neq T_{i+j}$ gesammelte Information

The diagram shows a text string $T_1 T_2 \dots T_{i+1} \dots T_{i+j} \dots T_m \dots$ and a pattern $P_1 \dots P_j \dots P_m$. A mismatch is indicated at position j with a crossed-out equals sign. Below, a suffix of length $m-k$ is shown, which matches a prefix of the pattern. The pattern is shifted to the right by $m-k$ positions to align the suffix with the prefix.

- Vorbereitung:
 - Suche das **am weitesten rechts** stehende Vorkommen des Suffixes $P_{j+1\dots m}$, dem **nicht** das Zeichen P_j vorangeht;
 - setze $wrw[j] = k$ = diejenige Position, an der dieses Vorkommen **endet**
 - Mögliche Verschiebung: $\gamma[j] := m - wrw[j]$

G. Zachmann Informatik 2 — SS 10 Preprocessing 43

Beispiel für die Berechnung von wrw

- $wrw[j]$ = Position, an der das von rechts her **nächste** Vorkommen des Suffixes $P_{j+1\dots m}$ endet, dem **nicht** das Zeichen P_j vorangeht
- Muster: banana

j	betracht. Suffix	verbotenes Zeichen	weiteres Auftreten	$wrw[j]$
6	a	n	<u>ban</u> ana	2
5	na	a	ban <u>an</u> a	0
4	ana	n	ban <u>a</u> na	4
3	nana	a	ban <u>an</u> a	0
2	anana	b	<u>ban</u> ana	0
1	banana	ϵ	<u>ban</u> ana	0

G. Zachmann Informatik 2 — SS 10 Preprocessing 44

Beispiel für die Anwendung der wrw-Funktion

- $wrw["banana"] = [0,0,0,4,0,2]$

```

a b a a b a b a n a n a n a n a
  ≠ = = =
    b a n a n a
      b a n a n a
  
```

- Beobachtung: Fall 2b aus der Version 1 produziert nie eine Verschiebung größer als $\psi(j) \rightarrow$ diesen Fall braucht man nicht mehr auszuprogrammieren

G. Zachmann Informatik 2 — SS 10 Preprocessing 45

BM-Algorithmus, 2.Version

```
n = len( T )
m = len( P )
berechne  $\delta$  und  $\gamma$ 
i = 0
while i <= n - m:
    j = m
    while j > 0 and P[j] == T[i+j]:
        j -= 1
    if j == 0:
        print "Pattern occurs with shift ", i
        i +=  $\gamma$ [0]
    else:
        d = j -  $\delta$ ( T[i+j] )
        if d >  $\gamma$ [j]:
            i += d
        else:
            i +=  $\gamma$ [j]
```

G. Zachmann Informatik 2 — SS 10 Preprocessing 46