

## Suche in Texten

- Aufgabe des String-Matching-Algorithmus
  - kinderleicht
- Naiver Algorithmus
  - wie würden Sie es tun?
- Knuth-Morris-Pratt-Algorithmus
  - scharfes Anschauen (= Precomputation) des Musters verbessert die Laufzeit
  - das ist optimal (im worst-case)
- Boyer-Moore-Algorithmus
  - schlechte Zeichen erlauben uns, durch den Text zu springen
  - das ist noch besser als nur optimal (in der Praxis)

G. Zachmann Informatik 2 - SS 06 Preprocessing 14

## Aufgabe

- Gegeben:
  - Text  $T$  der Länge  $n$  über einem endlichen Alphabet  $\Sigma$ 

$T[1]$ 

m	a	n	a	m	a	n	a	p	a	t	i	p	i	t	i	p	i
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

 $T[n]$
  - Muster (Pattern)  $P$  der Länge  $m$  über selbem Alphabet  $\Sigma$ 

$P[1]$ 

p	a	t	i
---	---	---	---

 $P[m]$
- Ausgabe: jedes Vorkommen von  $P$  in  $T$ 

$T[s+1..s+m] = P[1..m]$ 

m	a	n	a	m	a	n	a	p	a	t	i	p	i	t	i	p	i
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

  

←

Shift  $s$

→

p	a	t	i
---	---	---	---
- Definition: als (Mis-)Match wird die (Nicht-)Übereinstimmung von einem Zeichen aus dem Muster mit einem Zeichen im Text bezeichnet

G. Zachmann Informatik 2 - SS 06 Preprocessing 15



## Naïves Verfahren

- für jede mögliche Verschiebung  $0 \leq i \leq n - m$  prüfe maximal  $m$  Zeichenpaare, bei Mismatch beginne mit neuer Verschiebung

```

# Input: Text T und Muster P
# Output: Liste L mit Verschiebungen i,
#         an denen P in T vorkommt
# Bug: Indizierung von T/P beginnt bei 1
def naïve_string_match( T, P ):
    L = []
    for s in range( 0, len(T) - len(P) + 1 ):
        j = 1
        while j <= m and T[s+j] == P[j]:
            j += 1
        if j == m + 1:
            L.append( s )
    return L

```

G. Zachmann Informatik 2 - SS 06 Preprocessing 18

## Beispiel

T=„A string consisting of 37 characters.“  
P=„sting“

A string consisting of 37 characters.

sting  
sting  
sting ...

sting  
sting

→ sting ✓ ...  
s=14

G. Zachmann Informatik 2 - SS 06 Preprocessing 19

## Aufwand

- Naiver Algorithmus:
  - benötigt im worst-case die Laufzeit  $(n - m + 1) \cdot m \in O(n \cdot m)$  falls  $m \ll n$

0	0	...	0	...	0	...	0	0	...
			$\xrightarrow{s}$						
			0	...	0	...	0	1	

- für  $n = c \cdot m$  ist das  $O(n^2)$  bzw.  $O(m^2)$
- in der Praxis oft: Mismatch tritt sehr früh auf  $\rightarrow$  Laufzeit  $\sim c \cdot n$
- Untere Schranke für den Worst-Case:
  - mind. 1 Vergleich pro  $m$  aufeinander folgende Textstellen, mind. jedes Zeichen des Patterns muß 1x betrachtet werden  $\rightarrow \Omega(m + \frac{n}{m})$

G. Zachmann Informatik 2 - SS 06 Preprocessing 20

## Verfahren nach Knuth-Morris-Pratt (KMP)

- Seien  $T_i$  und  $P_{j+1}$  die zu vergleichenden Zeichen:
 

$T_1$	$T_2$	...	...	$T_i$	...	...
			=	=	=	≠
			$P_1$	...	$P_j$	$P_{j+1}$
				...		$P_m$
- Tritt bei einer Verschiebung erstmals ein Mismatch auf bei  $T_i$  und  $P_{j+1}$  dann gilt:
  - die zuletzt verglichenen  $j$  Zeichen in  $T$  stimmen mit den ersten  $j$  Zeichen in  $P$  überein
  - $T_i \neq P_{j+1}$

G. Zachmann Informatik 2 - SS 06 Preprocessing 21

- Methode: bestimme  $j' < j$ , so daß
  - $P[1..j'] = T[i'..i-1] = P[j-j'+1 .. j]$
  - $T_i$  anschließend mit  $P_{j'+1}$  verglichen werden kann

$T_1$	$T_2$	...	...	$T_i$	...	...
=	=	=	≠			
$P_1$	...	$P_j$	$P_{j+1}$	...	$P_m$	
=	=	?				
$P_1$	...	$P_{j'}$	$P_{j'+1}$	...	$P_m$	

- M.a.W.: bestimme den
  - längsten Präfix von  $P$ , der echtes Suffix von  $P[1..j]$  ist**
  - Speichere für jedes  $j$  das entsprechende  $j'$  in  $j' = \text{next}[j]$

G. Zachmann Informatik 2 - SS 06 Preprocessing 22

- Beispiel für die Bestimmung von  $\text{next}[j]$ :
 

$T_1$	$T_2$	...	0 1 0 1 1	0 1 0 1 1	0	...
			0 1 0 1 1	0 1 0 1 1	1	
				0 1 0 1 1	0 1 0 1 1	1
- $\text{next}[j] =$  Länge des längsten Präfixes von  $P$ , das echtes Suffix von  $P[1..j]$  ist

G. Zachmann Informatik 2 - SS 06 Preprocessing 23

■ für  $P = 0101101011$  ist  $next = [0,0,1,2,0,1,2,3,4,5]$

0	1	0	1	1	0	1	0	1	1	← Pattern
		0								}
		0	1							
				0						
				0	1					
				0	1	0				
				0	1	0	1			

Suffixe

G. Zachmann    Informatik 2 - SS 06    Preprocessing    24

### Beispiel

■ Muster: abrakadabra,  $next = [0,0,0,1,0,1,0,1,2,3,4]$

$i$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
	a	b	r	a	k	a	d	a	b	r	a	b	r	a	b	a	b	r	a	k	...
								†	†	†	†	‡			‡	‡					
	a	b	r	a	k	a	d	a	b	r	a	k	r	a	k	a	b	k	a	k	

$j = 11$                        $j = 0 = 2$

$next[11] = 04$

G. Zachmann    Informatik 2 - SS 06    Preprocessing    25

## Implementierung

```

# Input: Text T und Muster P
# Output: Liste L mit Verschiebungen s, an denen P in T vorkommt
def kmp_matcher( T, P ):
    n = len(T)
    m = len(P)
    L = []
    next = comp_next( P )
    j = 0
    for i in range(1,n):
        while j > 0 and T[i] != P[j+1] :
            j = next[j]
        if T[i] == P[j+1]:
            j += 1
        if j == m:
            L.append( i-m )
            j = next[j]
    return L

```

G. Zachmann Informatik 2 - SS 06 Preprocessing 26

## Korrektheit

- Situation am Beginn der while-Schleife:  
 $P[1..j] = T[i-j .. i-1]$  und  $j \neq m$

$T_1$	$T_2$	...	...	$T_i$	...	...
=	=	=	≠			
$P_1$	...	$P_j$	$P_{j+1}$	...	$P_m$	

- falls  $j = 0$ :  $j$  steht vor dem erstem Zeichen von P
- falls  $j > 0$ : P kann verschoben werden, solange  $j > 0$  und  $T_i \neq P_{j+1}$
- ist dann  $T[i] = P[j+1]$ , können  $j$  und  $i$  (am Schleifenende) erhöht werden
- wurde ganz P verglichen ( $j = m$ ), ist eine Stelle gefunden, und es kann verschoben werden

G. Zachmann Informatik 2 - SS 06 Preprocessing 27

## Laufzeit

- Beobachtungen:
  - Textzeiger  $i$  wird nie zurückgesetzt
  - Textzeiger  $i$  und Musterzeiger  $j$  werden stets gemeinsam inkrementiert
  - Für alle  $j$  ist  $\text{next}[j] < j \rightarrow j$  kann, insgesamt über die ganze for-Schleife, nur so oft herabgesetzt werden, wie es heraufgesetzt wurde

```

for i in range(1,n):
    while j > 0 and \
        T[i] != P[j+1]:
        j = next[j]
    if T[i] == P[j+1]:
        j += 1
    if j == m:
        L.append( i-m )
        j = next[j]
return L
        
```

- Fazit: der KMP-Algorithmus kann in Zeit  $O(n)$  ausgeführt werden, wenn das next-Array bekannt ist.

G. Zachmann Informatik 2 - SS 06
Preprocessing 28

## 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:
 

$P_1$	$P_2$	...	...	$P_{j-1}$	$P_j$	...
=	=			=	?	
$P_1$	...	$P_j$	$P_{j+1}$	...	$P_m$	
=		=	?			
$P_1$	...	$P_j$	$P_{j+1}$	...		

sei  $\text{next}[j-1] = j$ :

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

G. Zachmann Informatik 2 - SS 06
Preprocessing 29




```

# Input: Muster P
# Output: next-Array für P
def comp_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 06 Preprocessing 30




## Laufzeit 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 06 Preprocessing 31

## 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 06 Preprocessing 32

## Die Idee

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

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

Beginne Vergleich am Ende

Erstes falsches Zeichen! Großen Sprung machen!

Bingo! Noch einen großen Sprung machen!

Das wars! 10 Zeichen verglichen und fertig!

G. Zachmann Informatik 2 - SS 06 Preprocessing 33

### Die "Bad Character"-Heuristik (Vorkommensheuristik)

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

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

"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

G. Zachmann Informatik 2 - SS 06 Preprocessing 34

### Berechnung der Vorkommensheuristik (Fkt $\lambda$ )

- Für  $c \in \Sigma$  und das Muster  $P$  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 Σ:
    δ[a] = 0
for j in range( 1, m+1 ):
    δ[ P[j] ] = j
return δ
    
```

G. Zachmann Informatik 2 - SS 06 Preprocessing 35

- Im Folgenden seien
  - $c$  = das den Mismatch verursachende Zeichen
  - $j$  = 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  $j = j - \delta(c)$  Positionen nach rechts

G. Zachmann Informatik 2 - SS 06 Preprocessing 36

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

- 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:  
 $\rightarrow$  Verschiebung um  $k = j - \delta(c)$

G. Zachmann Informatik 2 - SS 06 Preprocessing 37

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

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

G. Zachmann Informatik 2 - SS 06 Preprocessing 38

■ **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 06 Preprocessing 39

## 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:  $O(\frac{n}{m})$ 
  - insbesondere bei großen Alphabeten und kurzen Mustern
  - typisch bei Textverarbeitungsprogrammen
- Laufzeit im Worst-Case:  $O(n \cdot m)$ 

0	0	...	0	0	...	0	...	0	...						
$\xrightarrow{i}$															
<table style="border-collapse: collapse; margin: auto;"> <tr> <td style="border: 1px solid black; padding: 2px 10px;">1</td> <td style="border: 1px solid black; padding: 2px 10px;">0</td> <td style="border: 1px solid black; padding: 2px 10px;">...</td> <td style="border: 1px solid black; padding: 2px 10px;">0</td> <td style="border: 1px solid black; padding: 2px 10px;">...</td> <td style="border: 1px solid black; padding: 2px 10px;">0</td> </tr> </table>										1	0	...	0	...	0
1	0	...	0	...	0										
- Gewünschte Laufzeit:  $c \cdot (m + \frac{n}{m})$

G. Zachmann Informatik 2 - SS 06
Preprocessing 40

## 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 06
Preprocessing 41

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

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

- Vorbereitung:  
 $wrw[j] = k :=$  Position, an der das von rechts her **nächste** Vorkommen des Suffixes  $P_{j+1..m}$  endet, dem **nicht** das Zeichen  $P_j$  vorangeht
- Mögliche Verschiebung:  $\gamma[j] := m - wrw[j]$

G. Zachmann Informatik 2 - SS 06 Preprocessing 42

### Beispiel für die wrw-Berechnung

- $wrw[j] =$  Position, an der das von rechts her **nächste** Vorkommen des Suffixes  $P_{j+1..m}$  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>ana</u>	0
4	ana	n	ban <u>ana</u>	4
3	nana	a	ban <u>ana</u>	0
2	anana	b	ban <u>ana</u>	0
1	banana	ε	ban <u>ana</u>	0

G. Zachmann Informatik 2 - SS 06 Preprocessing 43

## 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  $\gamma(j)$  → diesen Fall braucht man nicht mehr auszuprogrammieren

G. Zachmann Informatik 2 - SS 06 Preprocessing 44

## 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:
        gib Verschiebung i aus
        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 06 Preprocessing 45