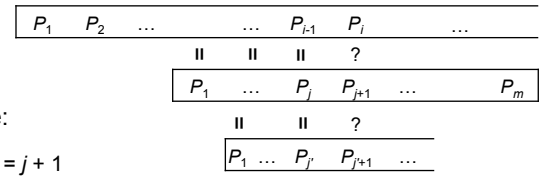
Berechnung des next-Arrays

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



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


```
# 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
```

Die Gesamtlaufzeit von KMP

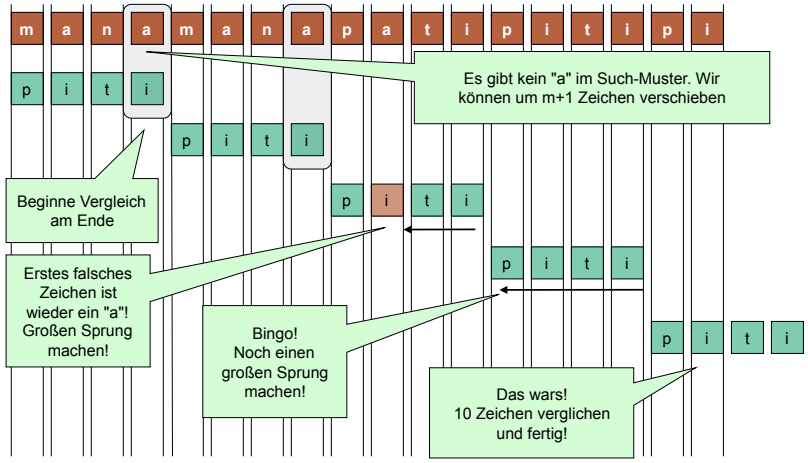
- Satz:
  Der KMP-Algorithmus kann in Zeit $O(n + m)$ ausgeführt werden.
- M.a.W.:
  das String-Matching-Problem kann in Zeit $O(n + m)$ gelöst werden
- Kann die Textsuche noch schneller sein?
  - "nein" im Worst-Case
  - "ja" im Average-Case

Das Verfahren nach Boyer-Moore (BM)

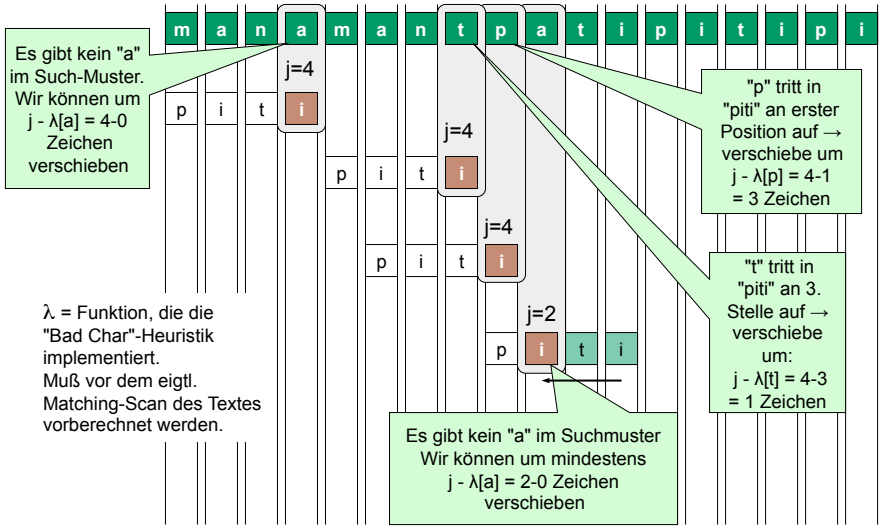
- Gleiche Worst-Case-Laufzeit wie KMP
- Viel bessere Laufzeit in der Praxis
- Basiert auf 2 "Heuristiken"
  - "Bad Character"-Heuristik (Vorkommensheuristik)
  - "Good Suffix"-Heuristik (Match-Heuristik; ähnlich zu KMP)
- Kompletter Algo mit beiden Heuristiken ist etwas knifflig ;-)

Die Idee

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



Die "Bad Character"-Heuristik (Vorkommensheuristik)

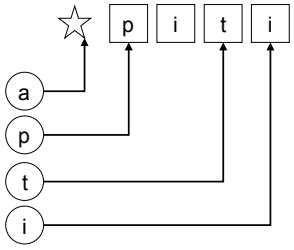


Berechnung der Vorkommensheuristik (die Fkt λ)

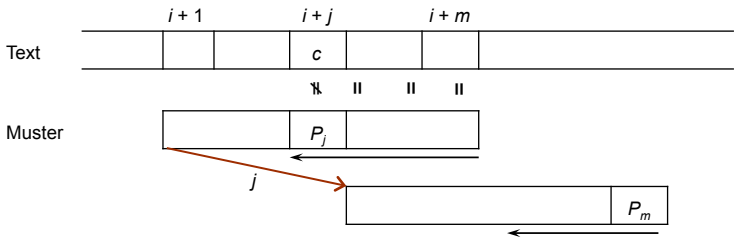
- Für c ∈ Σ und das Pattern *P* definiere

$$\delta(c) := \text{Index des von rechts her ersten Vorkommens von } c \text{ in } P = \begin{cases} 0 & \text{falls } c \notin P \\ \max\{j \mid P[j] = c\} & \text{falls } c \in P \end{cases}$$

```
for a in Σ:
    δ[a] = 0
for j = 1 .. m:
    δ[ P[j] ] = j
return δ
```

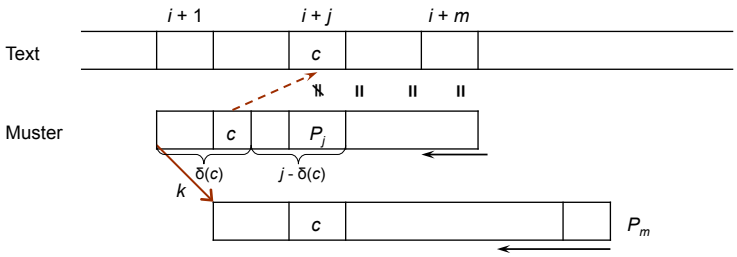


- Im Folgenden seien
  c = das den Mismatch verursachende Zeichen im Text
  j = Index des aktuellen Zeichens im Muster ($c \neq P_j$)
- Fall 1: *c* kommt nicht im Muster P vor → δ(*c*) = 0



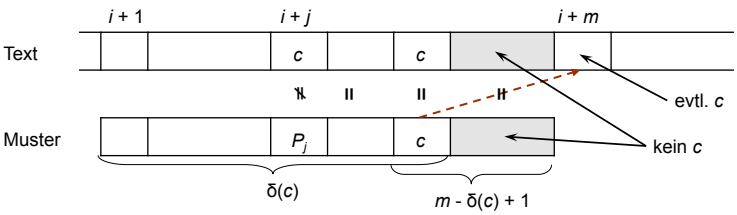
- Fazit: verschiebe das Muster um *j* = *j* - δ(*c*) Positionen nach rechts

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



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

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



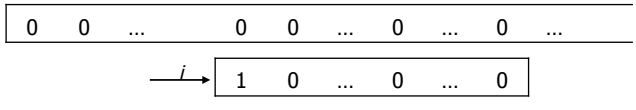
  - Fazit: Verschiebung des "rechtesten" *c* im Muster auf ein potentielles *c* im Text → Verschiebung um $m - \delta(c) + 1$

BM-Algorithmus, 1.Version

```
n = len( T )
m = len( P )
berechne δ
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 = δ( T[i+j] )
        if d > j:
            i += m + 1 - d
        else:
            i += j - d
```


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: $O(n \cdot m)$



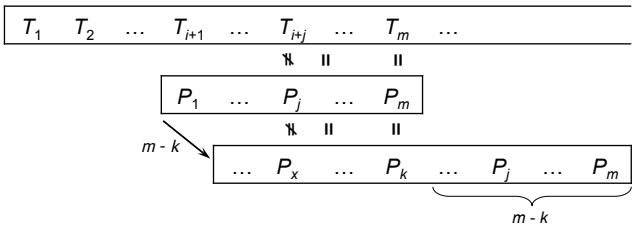
- Gewünschte Laufzeit: $c \cdot \left(m + \frac{n}{m}\right)$

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: $O(\frac{n}{m})$

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

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



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

Beispiel für die Berechnung von wrw

- 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

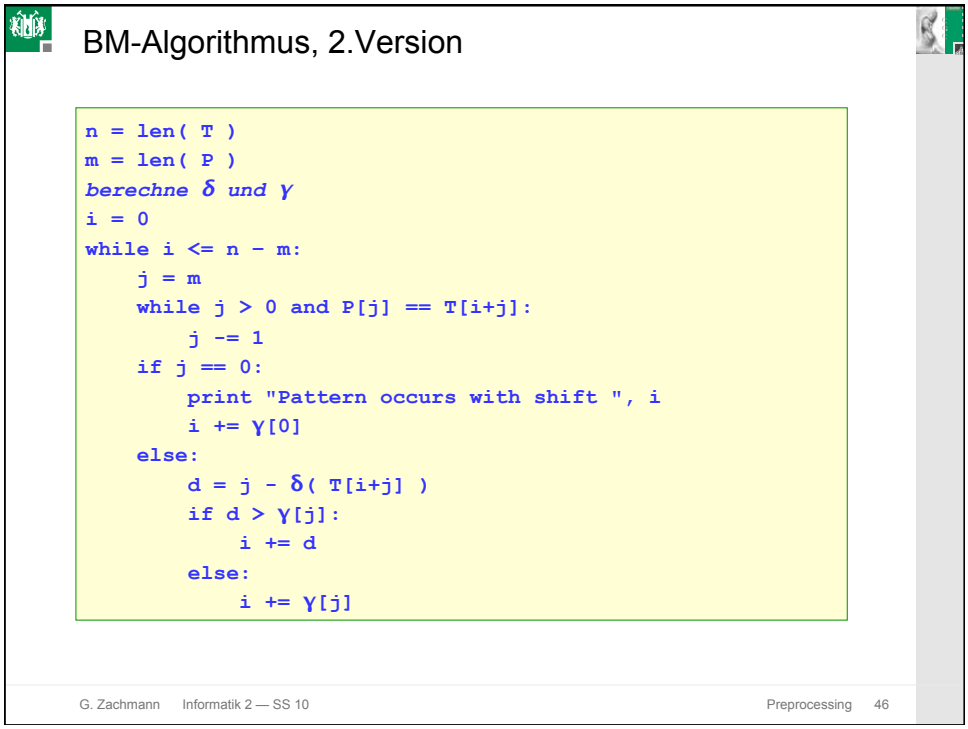
j	betracht. Suffix	verbotenes Zeichen	weiteres Auftreten	wrw[*j*]
6	a	n	banana	2
5	na	a	banana	0
4	ana	n	banana	4
3	nana	a	banana	0
2	anana	b	banana	0
1	banana	ε	banana	0

Beispiel für die Anwedung 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 γ(*j*) → diesen Fall braucht man nicht mehr auszuprogrammieren



BM-Algorithmus, 2.Version

```
n = len( T )
m = len( P )
berechne δ und γ
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 += γ[0]
    else:
        d = j - δ( T[i+j] )
        if d > γ[j]:
            i += d
        else:
            i += γ[j]
```