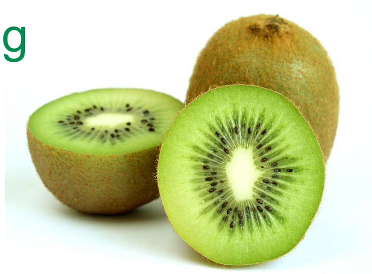
Informatik II
Precomputation / Preprocessing



G. Zachmann
Clausthal University, Germany
zach@in.tu-clausthal.de

Definitionen

- Preconditioning (Vorbehandlung):

  Aufgabe: löse nacheinander mehrere Probleme, die einen gemeinsamen Anteil haben, d.h., die Eingaben sind von der Form

  $$< x_1; y > < x_2; y > < x_3; y > \ \ldots$$

  Lösungsansatz: transformiere zunächst den gemeinsamen Anteil, d.h., berechne

  $$\hat{y} = T(y)$$

  und löse dann die Probleme

  $$< x_1; \hat{y} > < x_2; \hat{y} > < x_3; \hat{y} > \ \ldots$$

  (hoffentlich effizienter)

- Precomputation (Vorberechnung):
  auch wenn die Eingabe nur einmal vorkommt – aber dennoch aus 2 Teilen <*x* ; *y*> besteht – so kann es effizienter sein, zunächst einen Teil zu transformieren, $\hat{y} := T(y)$, und dann die Lösung aus <*x* ; *ŷ*> zu berechnen

$$
\begin{tikzcd}
\langle x ; y \rangle \arrow[d, "B_1"'] \arrow[dr, "\text{Algo A}"] & \\
\langle x ; \hat{y} \rangle \arrow[r, "\text{Algo B}_2"'] & z
\end{tikzcd}
$$

  - Beispiel: dynamischen Text nach einem Wort durchsuchen
- Bemerkung: im Umgangssprachgebrauch wird "*Precomputation*" oft für beide Arten verwendet
- Wesentliche Frage bei Precomputation / Preconditioning ist immer: Wie hoch ist der Aufwand dafür? macht er evtl. den anschließenden Zeitgewinn kaputt?

Beispiele bisher

- Closest Pairs
  - Vorsortieren der Punkte entlang x-Achse
- Huffman-Kodierung:
  - Bestimmung der (relativen) Häufigkeiten der einzelnen Zeichen
- Etc.

Anwendung dieser Technik im Folgenden

- Auswertung eines Polynoms an vielen Stellen; Eingabe ist

$$< x_i ; \underbrace{a_0 , \ldots , a_n}_{\text{"konstant"}} >$$

- Vorgänger-/Nachfolger-Beziehung in einem Baum testen; Eingabe ist

$$< v_1 , v_2 ; \underbrace{\text{Baum}}_{\text{„konstant"}} >$$

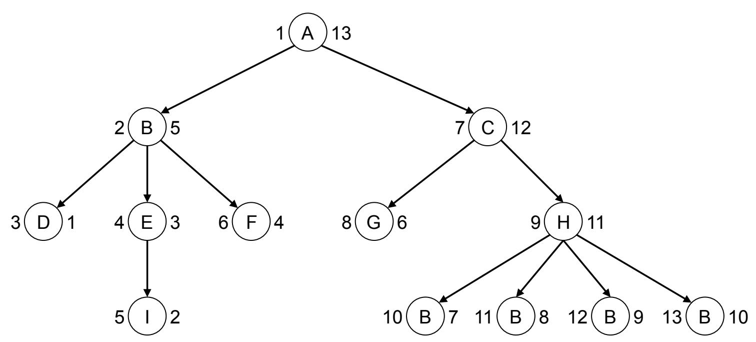
- Gegebenen Text nach einem Wort durchsuchen

Der Vorgänger im Baum

- Problem:
  - Gegeben ist ein Baum
  - Für viele Paare von Knoten (v, w) sollen wir entscheiden, ob v Vorgängerin, d.h. direkte oder indirekte Mutter, von w ist
- Naive Lösung:
  - Traversiere den Baum von w aus bis zur Wurzel und vergleiche mit v
  - Aufwand:
    - $O(n)$ im Worst-Case,
    - Untere Schranke für Worst-Case = $\Omega(\log n)$
- Behauptung: wir können auf dem Baum *Preconditioning* durchführen in Zeit $\Theta(n)$, so daß danach das Problem in Zeit $O(1)$ gelöst werden kann

- Das Verfahren:
  1. Durchlaufe den Baum in *Preorder* und nummeriere dabei die Knoten; bezeichne diese Nummer mit $N_{\text{pre}}(v)$
     - Nummeriere also erst den Knoten selbst mit fortlaufender Nummer, dann nummeriere rekursiv den linken, dann den rechten Teilbaum
  2. Nummeriere analog die Knoten in *Postorder*; Bezeichnung $N_{\text{post}}(v)$
  3. Jetzt gilt für jedes Paar *v*, *w* von Knoten:
     1. $N_{\text{pre}}(v) \leq N_{\text{pre}}(w) \Leftrightarrow$
        v Vorgänger von *w* oder *v* links von *w*
     2. $N_{\text{post}}(v) \geq N_{\text{post}}(w) \Leftrightarrow$
        v Vorgänger von *w* oder *v* rechts von *w*
  4. Zusammen :
     $N_{\text{pre}}(v) \leq N_{\text{pre}}(w) \wedge N_{\text{post}}(v) \geq N_{\text{post}}(w) \Leftrightarrow$
     v ist Vorgänger von *w*

Beispiel



Wiederholte Auswertung eines Polynoms

- Gegeben: Polynom $p(x) = a_n x^n + \cdots + a_1 x + a_0$
- Gesucht ist: $p(x_i)$ für viele $x_1, \ldots, x_n$
- **Verboten** ist:

```
p = a[0] + a[1]*x + a[2]*x*x + ...
```

- **Noch schlimmer ist**!:

```
p =    a[0] + a[1]*x + a[2]*pow(x,2.0)
     + a[3]*pow(x,3.0) ...
```


- Einfache Methode: transformiere p in das sog. Horner-Schema

$$p(x) = ((\cdots((a_n x + a_{n-1})x + a_{n-2})\cdots)x + a_1) + a_0$$

```
p = a[n]
for j in range( n-1,-1,-1 ):
    p = p*x + a[j]
```

- Reicht für mittellange Polynome völlig aus
- Aufwand: n Multiplikationen / n Additionen

Auswertung an äquidistanten Stellen

- Definition: Vorwärts-Differenzen

$$\Delta^1 p(x) := p(x+h) - p(x)$$
$$\Delta^i p(x) := \Delta^{i-1} p(x+h) - \Delta^{i-1} p(x)$$
$$\Delta^0 p(x) := p(x)$$

- Behauptung: $\Delta^i p(x)$ ist ein Polynom vom Grad n-i
- Beweis:

$$\begin{aligned}\Delta^1 p(x) &= p(x+h) - p(x) \\ &= a_0 + \cdots + a_n (x+h)^n \qquad - a_0 - \cdots - a_n x^n \\ &= a_0 + \cdots + a_n \sum_{i=0}^{n} \binom{n}{i} x^i h^{n-i} \quad - a_0 - \cdots - a_n x^n \\ &= a'_0 + \cdots + a'_{n-1} x^{n-1}\end{aligned}$$

  - Rest per Induktion

- Korollar:

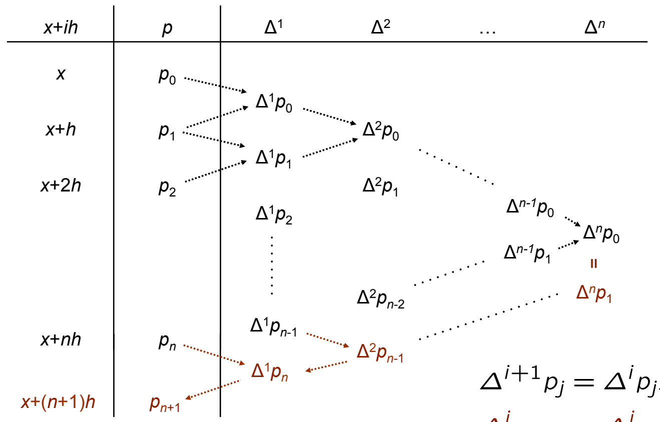
$$\Delta^n p(x + jh) \equiv \text{const}$$

- Bemerkung: auf dieser Beobachtung basiert das ganze Verfahren!
- Bemerkung: $\Delta^i p(x)$ hängt ab von $p(x)$, $p(x+h)$, …, $p(x+ih)$
- Beispiel:

$$\begin{aligned}\Delta^2 p(x) &= \Delta^1 p(x+h) - \Delta^1 p(x) \\ &= \left(\Delta^0 p(x+h+h) - \Delta^0 p(x+h)\right) \\ &\quad - \left(\Delta^0 p(x+h) - \Delta^0 p(x)\right) \\ &= p(x+2h) - 2p(x+h) + p(x)\end{aligned}$$

Das Verfahren

- Erstelle eine Vorwärtsdifferenzen-Pyramide:
  - Vereinfachende Schreibweise $\Delta^i p(x + jh) =: \Delta^i p_j$
  - Die Pyramide:



$$\Delta^{i+1} p_j = \Delta^i p_{j+1} - \Delta^i p_j$$

$$\Delta^i p_{j+1} = \Delta^i p_j + \Delta^{i+1} p_j$$

- Algo zur Berechnung von vielen äquidistanten Werten von *p*:
  1. Initialisieren die Pyramide mit den Punkten *x*, …, *x* + *nh*
  2. Berechne den nächsten Punkt an der Stelle *x* + (*n*+1)*h* durch Fortsetzen der Pyramide um eine Zeile am unteren Ende
- Aufwand: für jeden neuen Punkt braucht man nur *n* Additionen (keine Multiplikation!)
- Bemerkungen:
  - Man muß immer nur die unterste Zeile der Pyramide speichern
  - Rundungsfehler akkumulieren sich! → Pyramide ab und zu neu aufsetzen
- Durch Umformung kann man ein Polynom vom Grad *n* mit $\frac{n}{2}$ Multiplikationen an beliebigen Stellen auswerten (s. Knuth)

Suchen in Texten (*String Matching*)

- 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
- Boyer-Moore-Algorithmus
  - "Schlechte" Zeichen erlauben uns, größere Sprünge im Text zu machen
  - Das ist noch besser als "nur" worst-case optimal (in der Praxis)

Die Aufgabe

- Gegeben:
  - Text *T* der Länge *n* über einem endlichen Alphabet Σ:

    T[1] … *T*[*n*]

    | m | a | n | a | m | a | n | a | p | a | t | i | p | i | t | i | p | i |

  - Muster (Pattern) *P* der Länge *m* über dem selben Alphabet Σ:

    P[1] … P[m]

    | p | a | t | i |

- Ausgabe: jedes Vorkommen von *P* in *T*:

  T[1+s,..,m+s] = 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

- Beispiel: suche alle Vorkommen des Strings ananas in

  anasanamsanamananasnsamansnamsananasamsnamanananasana

  anasanamsanamananasnsamansnamsananasamsnamanananasana

  s+1 = 14   31   45

Verschiedene Szenarios

- Statische Texte (später)
  - Literaturdatenbanken
  - Bibliothekssysteme
  - Gen-Datenbanken
  - WWW-Verzeichnisse
- Dynamische Texte (im folgenden)
  - Texteditoren (Word & Co, Powerpoint, ASCII-Editor, IDE, ...)
- Achtung: im folgenden nehmen wir meistens an, daß Indizes ab 1 laufen — Vorsicht bei einer realen Implementierung!

Das naive Verfahren

- Für jede mögliche Verschiebung $0 \le i \le n - m$ prüfe maximal m Zeichenpaare, bei Mismatch beginne mit nächster 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
```


Beispiel

```
T="A string consisting of 37 characters."
P="sting"

A string consisting of 37 characters.
sting
 sting
  sting   …
            sting
             sting
              sting  ✓  …
```

s=14

Aufwand

- Worst-case Beispiel:

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

- Fazit: benötigt im worst-case die Laufzeit

$$(n-m+1)\cdot m \in O(n\cdot m)$$

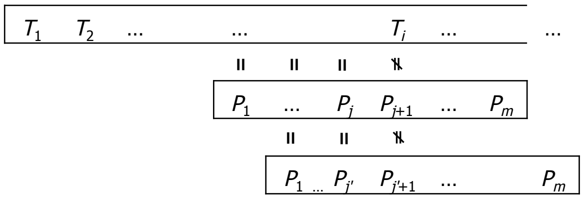
  - Für $m \in O(n)$ ist das $O(n^2)$
  - In der Praxis oft tritt oft ein Mismatch sehr früh auf $\rightarrow$ Laufzeit ~ $c \cdot n$
- Untere Schranke für die Textsuche im Worst-Case:
  - Mind. 1 Vergleich pro m aufeinander folgende Textzeichen
  - Mind. jedes Zeichen des Patterns muß 1x betrachtet werden
  - $\rightarrow \Omega\left(m+\frac{n}{m}\right)$

Das Verfahren nach Knuth-Morris-Pratt (KMP)

- Tritt bei einer Verschiebung erstmals bei T_i und P_{j+1} ein Mismatch auf, dann gilt:
  - Die zuletzt verglichenen j Zeichen in T stimmen mit den ersten j Zeichen in P überein
  - $T_i \neq P_{j+1}$

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

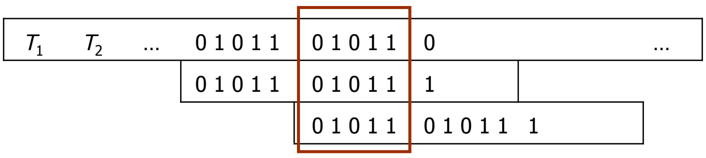
- Die Idee: bestimme $j' < j$, so daß
  - $P[1..j'] = P[j-j'+1..j] = T[i'..i-1]$



  - Vorteil: man kann sofort $P_{j'+1}$ mit T_i vergleichen
- 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' = next[j]

- Beispiel für die Bestimmung von next[j]:



- Für P = 0101101011 ist next = [0,0,1,2,0,1,2,3,4,5]

1	2	3	4	5	6	7	8	9	10	
0	1	0	1	1	0	1	0	1	1	← Pattern
		0								Suffixe
		0	1							
					0					
					0	1				
					0	1	0			
					0	1	0	1		
					0	1	0	1	1	

- next[*j*] =
Länge des längsten Präfixes von *P*, das echtes Suffix von P[1..*j*] ist

Beispiel zum Vorgehen beim Matching

- Pattern: 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 b	r	a	k b	r a	a b	k	a	k	
											j = 11 *j* = 4				*j* = 0 *j* = 2						

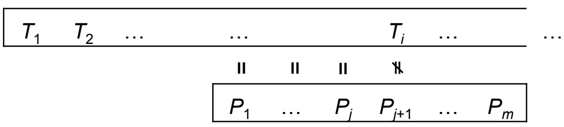
next[4] = 1 next[2] = 0

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 = compute_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
```


Korrektheit des Algorithmus

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



- Falls $j = 0 \rightarrow j$ steht vor dem erstem Zeichen von P (d.h., es gab noch keinen Vergleich zwischen P_1 und T_i
- Falls $j > 0 \rightarrow$ 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

Laufzeit

- Beobachtungen:
  - Textzeiger i wird nie zurückgesetzt
  - Textzeiger i und Musterzeiger j werden stets gemeinsam inkrementiert
  - Es gilt

    $$\forall j : \text{next}[j] < j$$

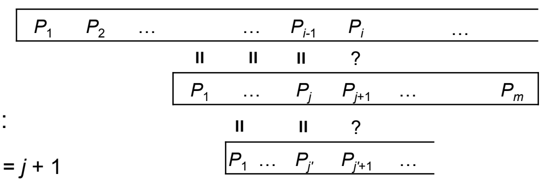
    $\rightarrow$ j kann, insgesamt über die ganze for-Schleife, nur so oft herabgesetzt werden, wie es heraufgesetzt wurde, also höchstens n Mal

```
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 schon berechnet ist

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 $P_{1..j}$, 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

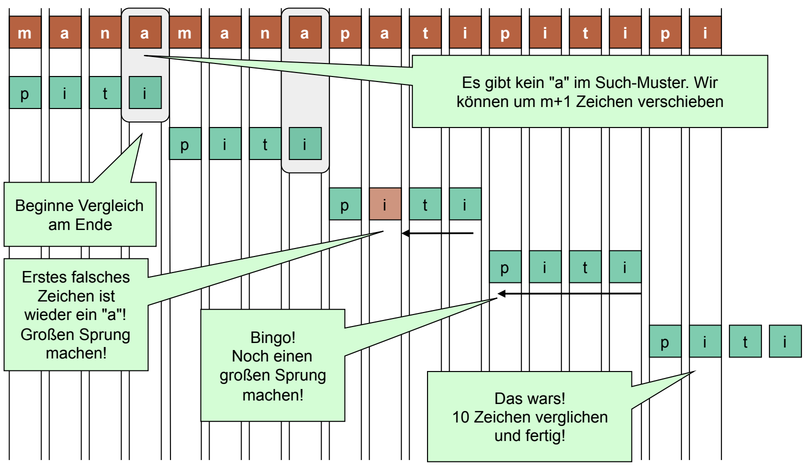
- Satz:
  Der KMP-Algorithmus kann in Zeit $O(n + m)$ ausgeführt werden.
- M.a.W.:
  das String-Matching-Problem kann in Zeit $O(n + m)$ gelöst werden
- Kann die Textsuche noch schneller sein?
  - "nein" im Worst-Case
  - "ja" im Average-Case

Das Verfahren nach Boyer-Moore (BM)

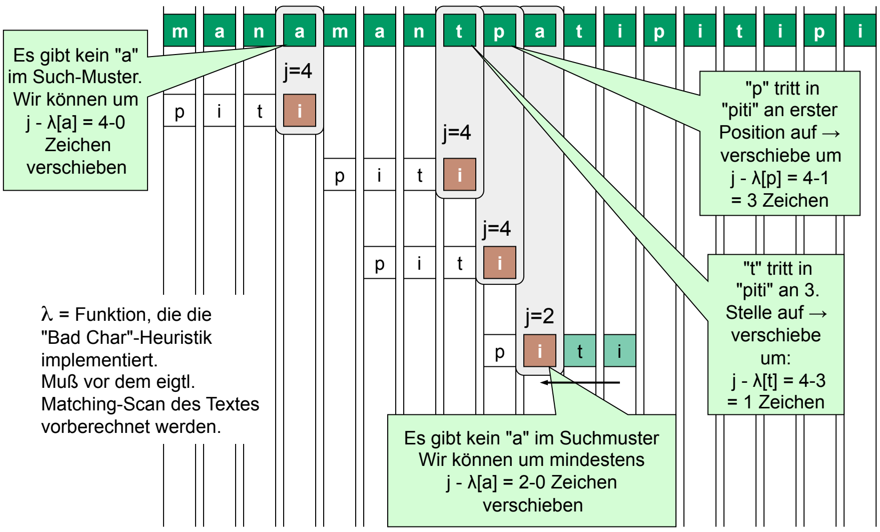
- Gleiche Worst-Case-Laufzeit wie KMP
- Viel bessere Laufzeit in der Praxis
- Basiert auf 2 "Heuristiken"
  - "Bad Character"-Heuristik (Vorkommensheuristik)
  - "Good Suffix"-Heuristik (Match-Heuristik; ähnlich zu KMP)
- Kompletter Algo mit beiden Heuristiken ist etwas knifflig ;-)

Die Idee

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



Die "Bad Character"-Heuristik (Vorkommensheuristik)

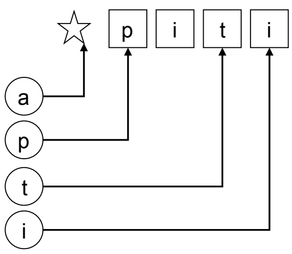


Berechnung der Vorkommensheuristik (die Fkt λ)

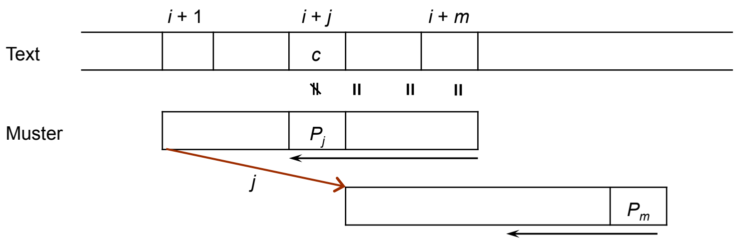
- Für $c \in \Sigma$ und das Pattern P definiere

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

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

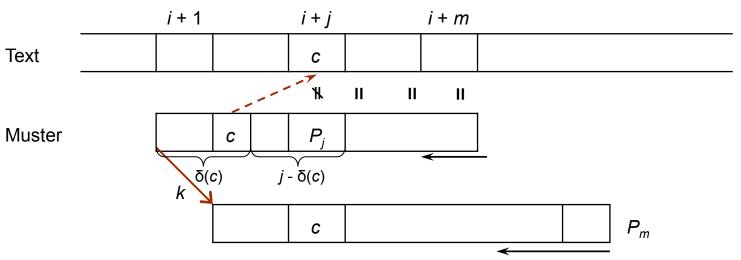


- Im Folgenden seien
  c = das den Mismatch verursachende Zeichen im Text
  j = Index des aktuellen Zeichens im Muster ($c \neq P_j$)
- Fall 1: c kommt nicht im Muster P vor $\rightarrow$ $\delta(c) = 0$



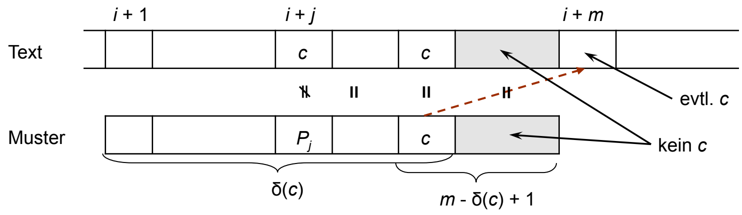
- Fazit: verschiebe das Muster um $j = j - \delta(c)$ Positionen nach rechts

- 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:
  $\rightarrow$ 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 $\rightarrow$ 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: $O(n \cdot m)$

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

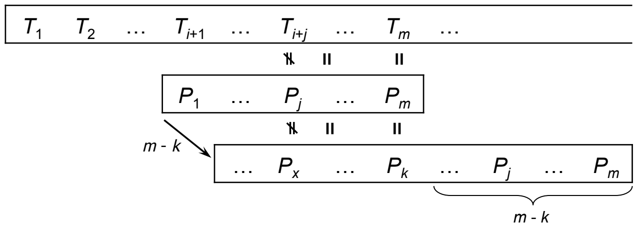
- Gewünschte Laufzeit: $c \cdot \left(m + \frac{n}{m}\right)$

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

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 $P_{j+1\ldots m}$ 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
```

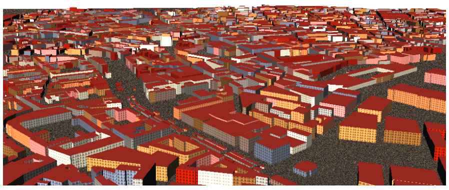
- Beobachtung: Fall 2b aus der Version 1 produziert nie eine Verschiebung größer als $\gamma(j)$ → diesen Fall braucht man nicht mehr auszuprogrammieren

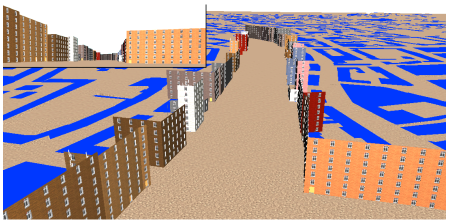
BM-Algorithmus, 2.Version

```
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]
```

Anwendung: Visibility Computation

- Gegeben: großes graphisches Modell (z.B. eine Stadt)
- Aufgabe: nur diejenigen Polygone zeichnen, die vom aktuellen Viewpoint aus sichtbar sind
- Idee: Precomputation
  - Unterteile den Raum in Zellen (z.B. Gitter)
  - Berechne vorab für jede Zelle, welche Polygone der Szene sichtbar sind





Anwendung: vorberechneter Lichtaustausch

- Aufgabe: für ein bestimmtes Objekt und bestimmte Beleuchtung *schnell* berechnen, wieviel Licht an jedem Punkt auf der Oberfläche ankommt
- Beobachtung: *welchen* Teil der Umgebung ein bestimmter Punkt auf der Oberfläche "sieht", hängt nur von der Geometrie ab → vorberechnen und für "jeden" Punkt speichern
- Laufzeit: für jeden Punkt der Oberfläche direkt das Licht aus der sichtbaren Umgebung "einsammeln"