ALBERT-LUDWIGS-
UNIVERSITÄT FREIBURG

Algorithm Theory

13 Text Search - Knuth, Morris, Pratt, Boyer, Moore

Christian Schindelhauer

Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Rechnernetze und Telematik
Wintersemester 2007/08

CoNe Freiburg

IIF
INSTITUT FÜR INFORMATIK FREIBURG

Text Search

- **Scenarios**
- **Static texts**
  - Literature databases
  - Library systems
  - Gene databases
  - World Wide Web
- **Dynamic texts**
  - Text editors
  - Symbol manipulators

Text Search

Data type **string**

- array of character
- file of character
- list of character

Operations: (T, P of type **string**)

length: length ()

i-**th character** : T [i]

concatenation: cat (T, P) T.P

Problem Definition

Given:

Text $t_1\, t_2 \,....\, t_n \quad \in \Sigma^n$

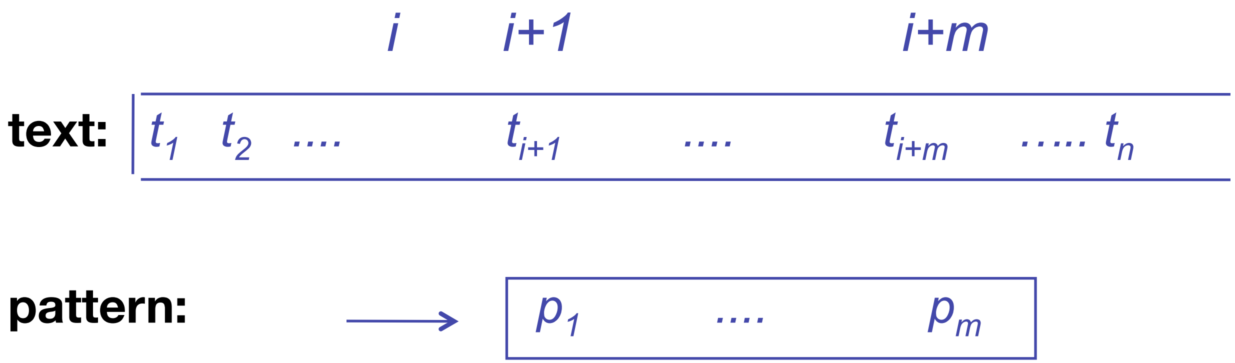
pattern $p_1 p_2 \,...\, p_m \quad \in \Sigma^m$

Goal:

Find one or all occurences of the pattern in the text, i.e. positions i ($0 \leq i \leq n - m$) such that

$$p_1 = t_{i+1}$$
$$p_2 = t_{i+2}$$
$$\vdots$$
$$p_m = t_{i+m}$$

Problem Definition



Possible Running Time :

1. **Worst Solution:**

   # possible alignments: $n - m + 1$

   # pattern positions: m

→ O(n m)

2. **Best possible solution:**

   At least one comparison per m consecutive text positions:

→ $\Omega\,(\,m + n/m\,)$

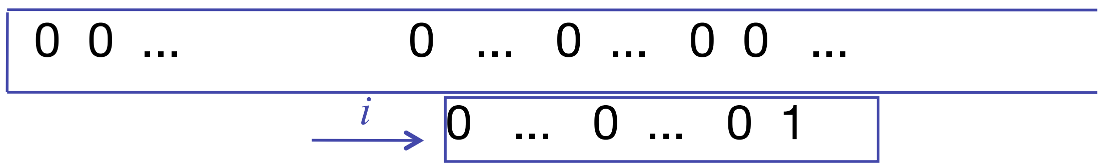
Naive Method

For each possible position $0 \leq i \leq n - m$ check at most m characters pairs. Whenever a mismatch occurs, shift to the next position

```
textsearchbf  := proc (T : : string, P : : string)
# Input:   text T, pattern P
# Output:  list L of positions i, at which P occurs in T
    n := length (T); m := length (P);
    L := [];
    for i from 0 to n-m do
        j := 1;
        while j ≤ m and T[i+j] = P[j]
                do j := j+1 od;
        if j = m+1 then L := [L [] , i] fi;
    od;
    RETURN (L)
end;
```

Naive Method

Running time:



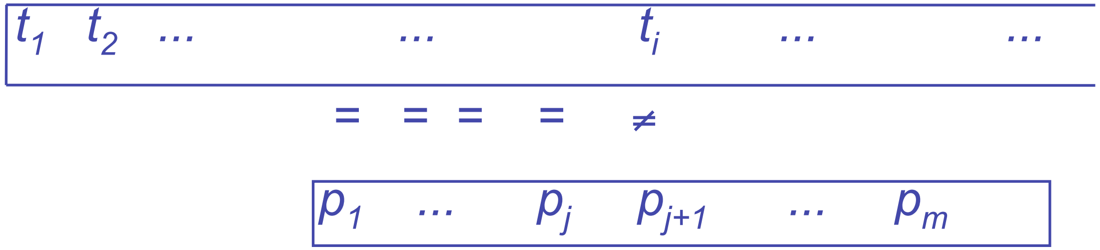
Worst Case: $\Omega(m\, n)$

For real world examples, mismatches occur usually very often.

→ Running time ~ $c\, n$

Knuth-Morris-Pratt Algorithm (KMP)

Let t_i and p_{j+1} be the characters to be compared:



If, for a certain alignment, the first mismatch occurs for characters t_i and p_{j+1} then

- the last j characters compared in T equal the first j characters of P
- $t_i \neq p_{j+1}$

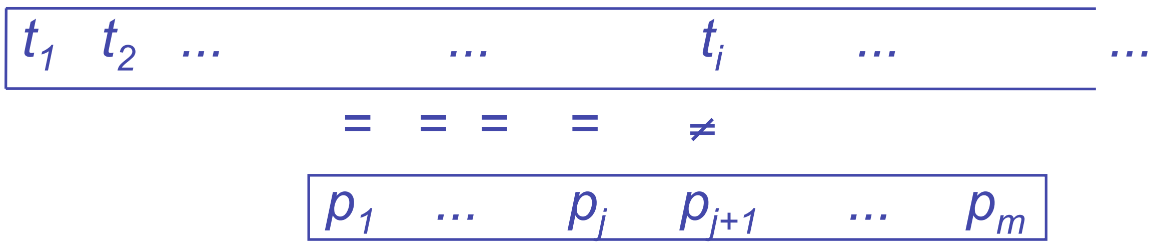
Knuth-Morris-Pratt Algorithm (KMP)

Idea:

Find $j' = \text{next}[j] < j$ such that t_i can then be compared to $p_{j'+1}$

Find maximum $j' < j$ such that $P_{1 \ldots j'} = P_{j-j'+1 \ldots j}$.

Find the longest prefix of P which is a proper suffix of $P_{1 \ldots j}$.



Knuth-Morris-Pratt Algorithm (KMP)

Example for determining next[j]:

t_1 t_2 ... 01011 01011 0 ...

01011 01011 1

01011 01011 1

next[j] = length of the longest prefix of P which is a proper suffix of $P_{1 \ldots j}$

Knuth-Morris-Pratt Algorithm (KMP)

⇒ **for P = 0101101011, 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
		0							
		0	1						
					0				
					0	1			
					0	1	0		
					0	1	0	1	
					0	1	0	1	1

Knuth-Morris-Pratt Algorithm (KMP)

```
KMP := proc (T : : string, P : : string)
# Input: text T, pattern P
# Output:  list L of positions i at which P occurs in T
     n := length (T); m := length(P);
     L :=  []; next := KMPnext(P);
     j  :=  0;
     for i from 1 to n do
             while j>0 and T[i] <> P[j+1] do j := next [j] od;
             if T[i]  = P[j+1]  then j := j+1 fi;
             if j = m then L := [L[] , i-m] ;
                                   j  := next [j]
             fi;
      od;
      RETURN (L);
end;
```

Knuth-Morris-Pratt Algorithm (KMP)

Pattern: abrakadabra, next = [0,0,0,1,0,1,0,1,2,3,4]

```
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
```

next[11] = 4

```
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
```

next[4] = 1

Knuth-Morris-Pratt Algorithm (KMP)

```
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
```

next [4] = 1

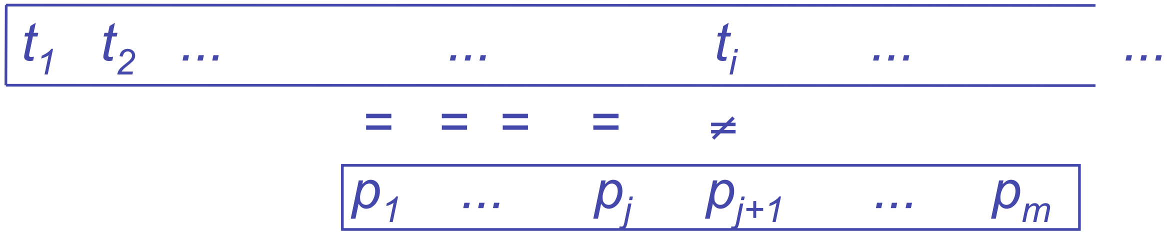
```
a b r a k a d a b r a b r a b a b r a k ...
                          - | |
                          a b r a k
```

next[2] = 0

```
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
```

Knuth-Morris-Pratt Algorithm (KMP)

Correctness:



When starting the for-loop:

$P_{1\ldots j} = T_{\text{i-j...i-1}}$ und $j \neq m$

if $j = 0$: we are located at the first character of P P

if $j \neq 0$: P can be shifted while j>0 and $t_i \neq p_{j+1}$

Knuth-Morris-Pratt Algorithm (KMP)

If $T[i] = P[j+1]$, j and i can be increased (at the end of the loop)

If P has been compared completely ($j = m$) an occurrence of P in T has been found and we can shift to the next position.

Knuth-Morris-Pratt Algorithm (KMP)

Running time:

- text pointer *i* will be never decreased
- text pointer *i* and pattern pointer *j* are always incremented together: next[j] < *j*;
  j can be decreased only as many times as it has been increased

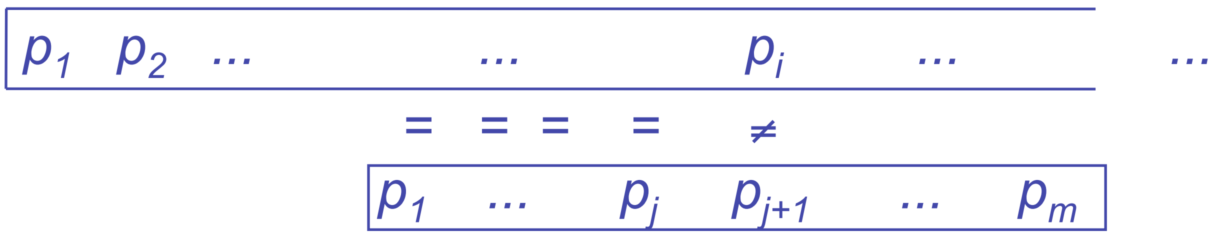
If the next-array is known, KMP runs in time O(n).

Computation of Array *next*

next[i] = length of the longest prefix of P which is a proper suffix of $P_{1 \ldots i}$

next[1] = 0

Let *next*[i-1] = j:



Computation of Array *next*

Consider the following two cases

1) $p_i = p_{j+1}$ → next[i] = $j + 1$

2) $p_i \neq p_{j+1}$ → replace j by next[j] until $p_i = p_{j+1}$ or $j = 0$.

if $p_i = p_{j+1}$ then next[i] = $j + 1$ otherwise next[i] = 0.

Computation of Array *next*

```
KMPnext := proc (P : : string)
#Input    :  pattern P
#Output :   next-array for P
    m := length (P);
    next := array (1..m);
    next [1] := 0;
    j := 0;
    for i from 2 to m do
        while j > 0 and P[i] <> P[j+1]
            do j := next [j] od;
        if P[i] = P[j+1] then j := j+1 fi;
        next [i] := j
    od;
    RETURN (next);
end;
```

Run Time of KMP

The KMP-algorithm runs in time O(n+m).

Can text search be any faster?

The Boyer-Moore-Algorithm (BM)

Idea: For any alignment of the pattern with the text, scan the characters from right to left rather than from left to right

Example:

```
he said abrakadabut but
  |  |  |  |  |  |  |||
but  |  |  |  |  |  |||
   but  |  |  |  |  |||
      but  |  |  |  |||
         but  |  |  |||
            but  |  |||
               but  |||
                  but||
                    but
```

Large jumps and few comparisons

Desired running time: O(m+n/m)

BM – Last-Occurrence Function

For $c \in \Sigma$ and pattern P let

$\delta(c)$:= index of the right-most occurrence of c in P

$$= \max \{j \mid p_j = c\}$$

$$= \begin{cases} 0 & \text{if } c \notin P \\ j & \text{if } c = p \text{ and } c \neq p, \text{ for } j < k \leq m \end{cases}$$

How hard is it to compute all values of δ?

Let $|\Sigma| = l$: Running time: $O(m+l)$

BM – Last-Occurrence Function

Let

c = the character causing the mismatch

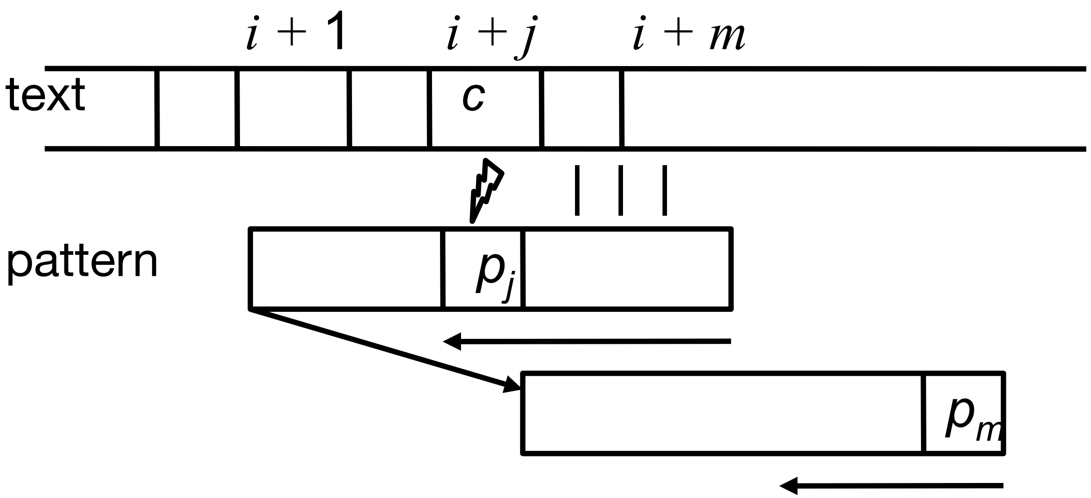
j = the index of the current character in the pattern ($c \neq p_j$)

BM – Last-Occurrence Function

Computation of the pattern shift

Case 1 *c* does not occur in the pattern *P* ($\delta(c) = 0$)

Shift the pattern *j* characters to the right

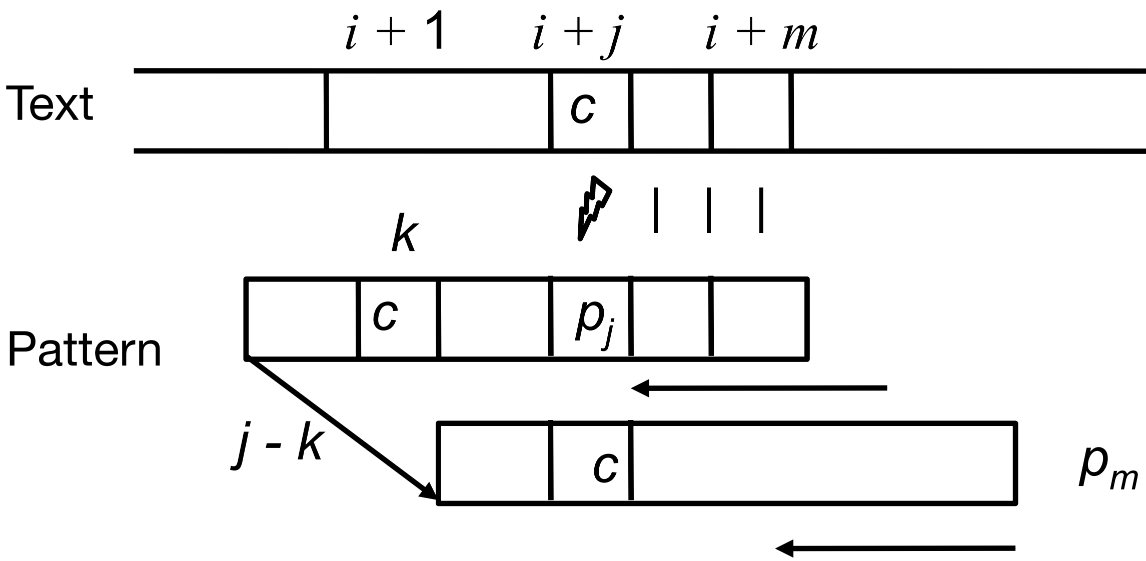


$$\Delta(i) = j$$

BM – Last-Occurrence Function

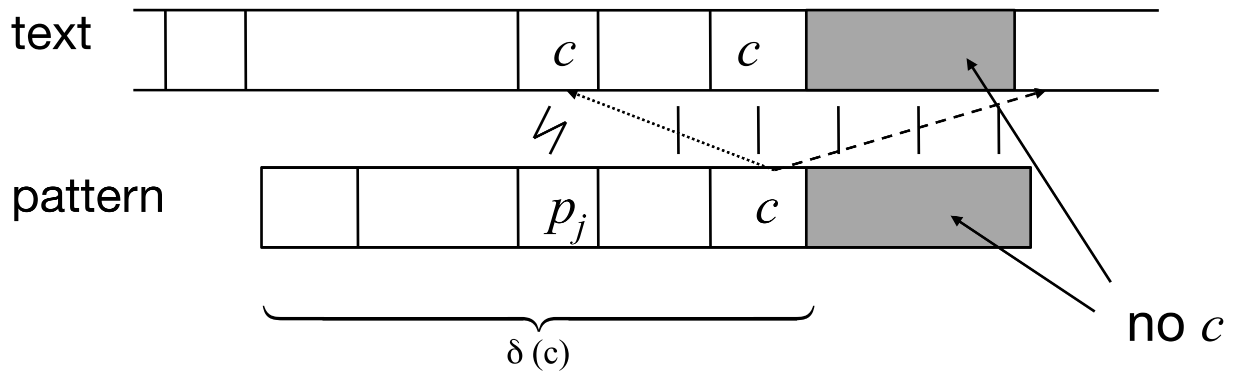
Case 2 *c* occurs in the pattern. ($\delta(c) \neq 0$)

Shift the pattern to the right until the rightmost c in the pattern is aligned with a potential c in the text.



BM – Last-Occurrence Function

Case 2 a: $\delta(c) > j$

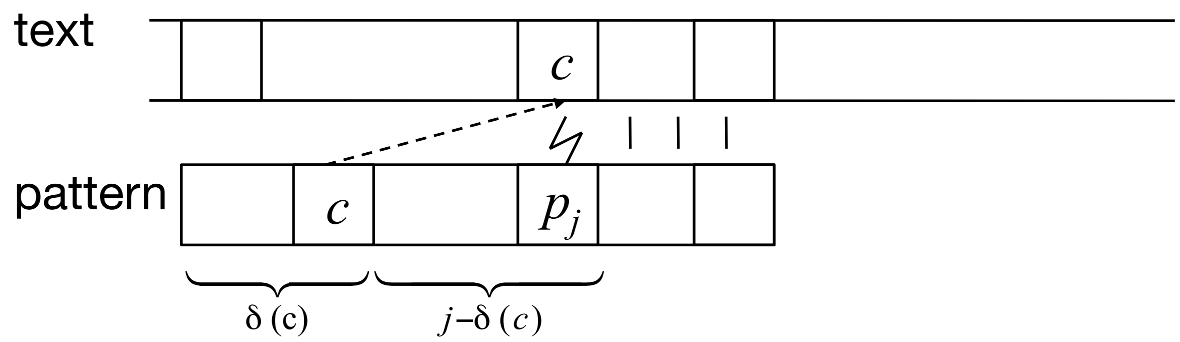


Shift the rightmost c in the pattern to a potential c in the text

shift by $\Delta(i) = m - \delta(c) + 1$

BM – Last-Occurrence Function

Case 2 b: $\delta(c) < j$



Shift the rightmost c in the pattern to c in the text

shift by $\Delta(i) = j - \delta(c)$

Boyer-Moore Algorithm
Version 1

Algorithm *BM-search1*

Input: text *T* and pattern *P*

Output: all positions of *P* in *T*

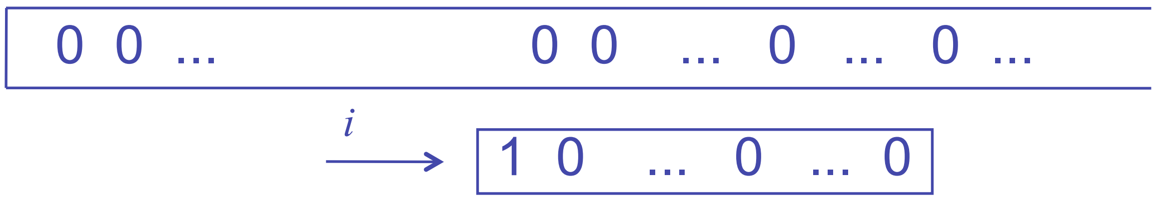
```
n := length(T); m := length(P)
compute δ
i := 0
while i ≤ n – m do
    j := m
    while j > 0 and P[j] = T[i + j] do
        j := j – 1
    end while;
    if j = 0
    then output position i
          i := i + 1
    else if δ(T[i + j]) > j
        then i := i + m + 1 - δ[T[i + j]]
        else i := i + j - δ[T[i + j]]
end while;
```

Boyer-Moore Algorithm
Version 1

Analysis:

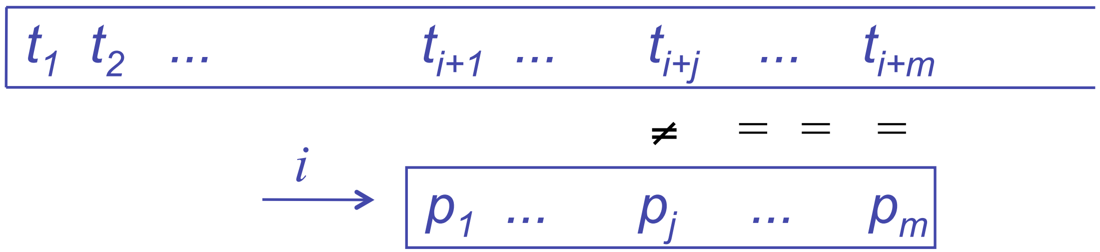
Desired running time : O(m + n/m)

Worst-case running time: $\Omega(n\ m)$



Match Heuristic

Use the information collected before a mismatches $p_j \neq t_{i+j}$ occurs.



$gsf[j]$ = position of the end of the next occurrence of the suffix $P_{j+1 \dots m}$ from the right that is not preceded by character P_j (good suffix function)

Possible shift: $\gamma[j] = m - gsf[j]$

Example for Computing *gsf*

gsf[*j*] = position of the end of the closest occurrence of the suffix $P_{j+1 \ldots m}$ from the right that is not preceded by character P_j

Pattern: banana

gsf[j]	inspected suffix	forbidden character	Other matches	Position
gsf[5]	**a**	**n**	**banana**	**2**
gsf[4]	**na**	**a**	***** bana na**	**0**
gsf[3]	**ana**	**n**	**banana**	**4**
gsf[2]	**nana**	**a**	**banana**	**0**
gsf[1]	**anana**	**b**	**banana**	**0**
gsf[0]	**banana**	ε	**banana**	**0**

Example for Computing *gsf*

$\Rightarrow$ gsf (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
```

Boyer-Moore Algorithm
Version 2

Algorithm *BM-search2*
Input: text *T* and pattern *P*
Output: shift for all occurrences of *P* in *T*

```
n := length(T); m := length(P)
compute δ and γ
i := 0
while i ≤ n – m do
    j := m
    while j > 0 and P[j] = T[i + j] do
        j := j – 1
    end while;
    if j = 0
      then output position i
           i := i + γ[0]
      else i := i + max(γ[j], j - δ[T[i + j]])
end while;
```

ALBERT-LUDWIGS-UNIVERSITÄT FREIBURG

Algorithm Theory

13 Text Search - Knuth, Morris, Pratt, Boyer, Moore

Christian Schindelhauer

Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Rechnernetze und Telematik
Wintersemester 2007/08

CoNe Freiburg

IIF INSTITUT FÜR INFORMATIK FREIBURG