Theory I
Algorithm Design and Analysis

(10 - Text search, part 1)

Prof. Dr. Th. Ottmann

Text search

Different scenarios:

Dynamic texts

- Text editors
- Symbol manipulators

Static texts

- Literature databases
- Library systems
- Gene databases
- World Wide Web

Text search

Data type **string**:

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

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

Length:	length ()
i-**th character**:	*T* [*i*]
concatenation:	cat (*T*, *P*) *T*.*P*

Problem definition

Input:

Text $t_1\ t_2 \ldots\ t_n \quad \in \Sigma^n$

Pattern $p_1 p_2 \ldots\ p_m \quad \in \Sigma^m$

Goal:

Find one or all occurrences of the pattern in the text, i.e. shifts i ($0 \le i \le n - m$) such that

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

Problem definition

i $i+1$ $i+m$

Text: t_1 t_2 .... t_{i+1} .... t_{i+m} ..... t_n

Pattern: → p_1 .... p_m

Estimation of cost (time) :

1. # possible shifts: $n - m + 1$ # pattern positions: m
   → $O(n \cdot m)$

2. At least 1 comparison per m consecutive text positions:
   → $\Omega(m + n/m)$

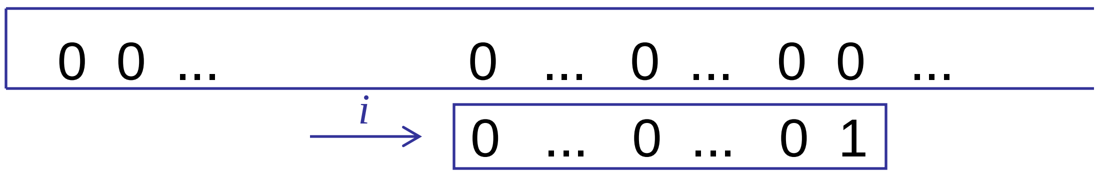
Naïve approach

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

```
textsearchbf  := proc (T : : string, P : : string)
#  Input:   Text T und Muster P
# Output:  List L of shifts i, at which P occurs in T
     n := length (T); m := length (P);
     L := [];
     for i from 0 to n-m {
              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;
     }
     RETURN (L)
end;
```

Naïve approach

Cost estimation (time):



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

In practice: mismatch often occurs very early

→ running time ~ $c \cdot n$

Method of Knuth-Morris-Pratt (KMP)

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

$t_1 \quad t_2 \quad \ldots \qquad \ldots \qquad t_i \qquad \ldots \qquad \ldots$

$= \quad = \quad = \quad = \qquad \neq$

$p_1 \quad \ldots \quad p_j \quad p_{j+1} \quad \ldots \quad p_m$

If, at a shift, the first mismatch occurs at

t_i and p_{j+1}, then:

- The last j characters inspected in T equal the first j characters in P.
- $t_i \neq p_{j+1}$

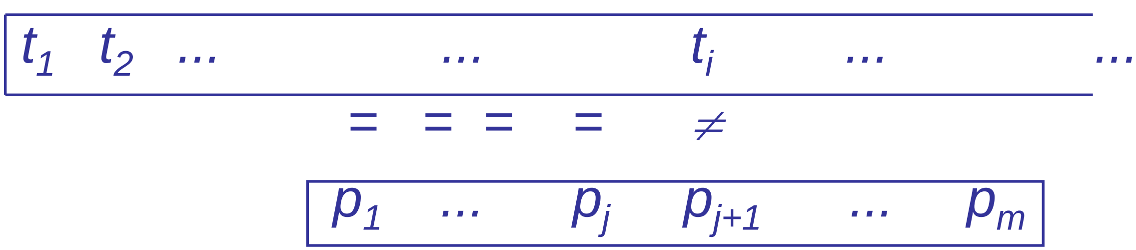
Method of Knuth-Morris-Pratt (KMP)

Idea:

Determine $j' = next[j] < j$ such that t_i can then be compared with $p_{j'+1}$.

Determine $j' < j$ such that $P_{1...j'} = P_{j-j'+1...j}$.

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



Method of Knuth-Morris-Pratt (KMP)

Example for determining *next*[*j*]:

```
t1  t2  ... 01011   01011   0              ...
            01011   01011   1
                    01011   01011   1
```

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

Method of Knuth-Morris-Pratt (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

Method of Knuth-Morris-Pratt (KMP)

```
KMP := proc (T : : string, P : : string)
# Input: text T and pattern P
# Output: list L of shifts 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;
```

Method of Knuth-Morris-Pratt (KMP)

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

```
a b r a c a d a b r a b r a b a b r a c ...
| | | | | | | | | | |
a b r a c a d a b r a
```

next[11] = 4

```
a b r a c a d a b r a b r a b a b r a c ...
              - - - - ∤
              a b r a c
```

next[4] = 1

Method of Knuth-Morris-Pratt (KMP)

```
a b r a c a d a b r a b r a b a b r a c ...
                        - | | | ∤
                        a b r a c
```
next [4] = 1

```
a b r a c a d a b r a b r a b a b r a c ...
                              - | ∤
                              a b r a c
```
next [2] = 0

```
a b r a c a d a b r a b r a b a b r a c ...
                                  | | | | |
                                  a b r a c
```

Method of Knuth-Morris-Pratt (KMP)

Correctness:

$t_1 \quad t_2 \quad \ldots \qquad \ldots \qquad t_i \qquad \ldots \qquad \ldots$

$= \quad = \quad = \quad = \qquad \neq$

$p_1 \quad \ldots \quad p_j \quad p_{j+1} \quad \ldots \quad p_m$

Situation at start of the for-loop:

$P_{1\ldots j} = T_{i-j\ldots i-1}$ and $j \neq m$

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

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

Method of Knuth-Morris-Pratt (KMP)

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

When P has been compared completely ($j = m$), a position was found, and we can shift.

Method of Knuth-Morris-Pratt (KMP)

Time complexity:

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

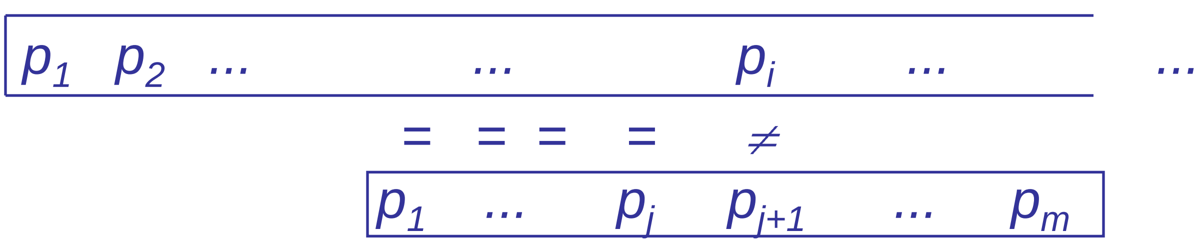
The KMP algorithm can be carried out in time *O*(*n*),
if the *next*-array is known.

Computing the *next*-array

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

next[1] = 0

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



Computing the *next*-array

Consider two cases:

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

2) $p_i \neq p_{j+1} \rightarrow$ replace j by $next[j]$, until $p_i = p_{j+1}$ or $j = 0$.
If $p_i = p_{j+1}$, we can set $next[i] = j + 1$,
otherwise $next[i] = 0$.

Computing the *next*-array

```
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;
```

Running time of KMP

The KMP algorithm can be carried out in time $O(n + m)$.

Can text search be even faster?

Method of Boyer-Moore (BM)

Idea: Align the pattern from left to right, but compare the characters from right to left.

Example:

```
e r   s a g t e   a b r a k a d a b r a   a b e r
      ≠
a b e r

e r   s a g t e   a b r a k a d a b r a   a b e r
              ≠
        a b e r
```

Method of Boyer-Moore (BM)

```
e r   s a g t e   a b r a k a d a b r a   a b e r
                ≠
          a b e r

e r   s a g t e   a b r a k a d a b r a   a b e r
                      ≠
                a b e r

e r   s a g t e   a b r a k a d a b r a   a b e r
                            ≠
                      a b e r
```

Method of Boyer-Moore (BM)

```
e r   s a g t e   a b r a k a d a b r a   a b e r
                                    ≠
                              a b e r

e r   s a g t e   a b r a k a d a b r a   a b e r
                                             ≠
                                       a b e r

e r   s a g t e   a b r a k a d a b r a   a b e r
                                          | | | |
                                          a b e r
```

Large jumps: few comparisons

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

BM – Heuristic of occurrence

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

$\delta(c)$:= index of the first occurrence of c in P from the right

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

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

What is the cost for computing all δ-values?

Let $|\Sigma| = l$:

BM – Heuristic of occurrence

Let

c = the character causing the mismatch

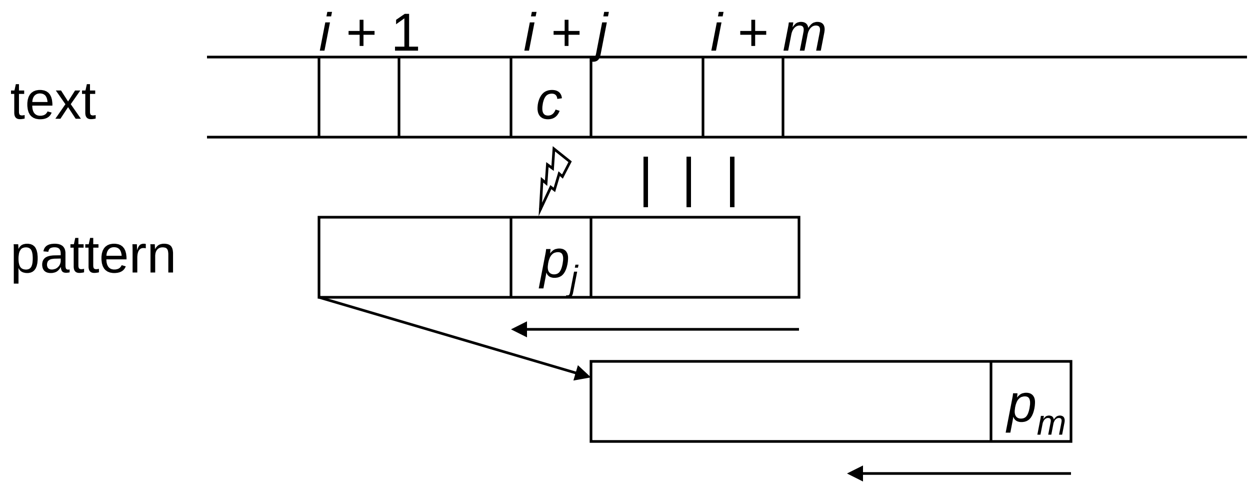
j = index of the current character in the pattern ($c \neq p_j$)

BM – Heuristic of occurrence

Computation of the pattern shift

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

Shift the pattern to the right by *j* characters

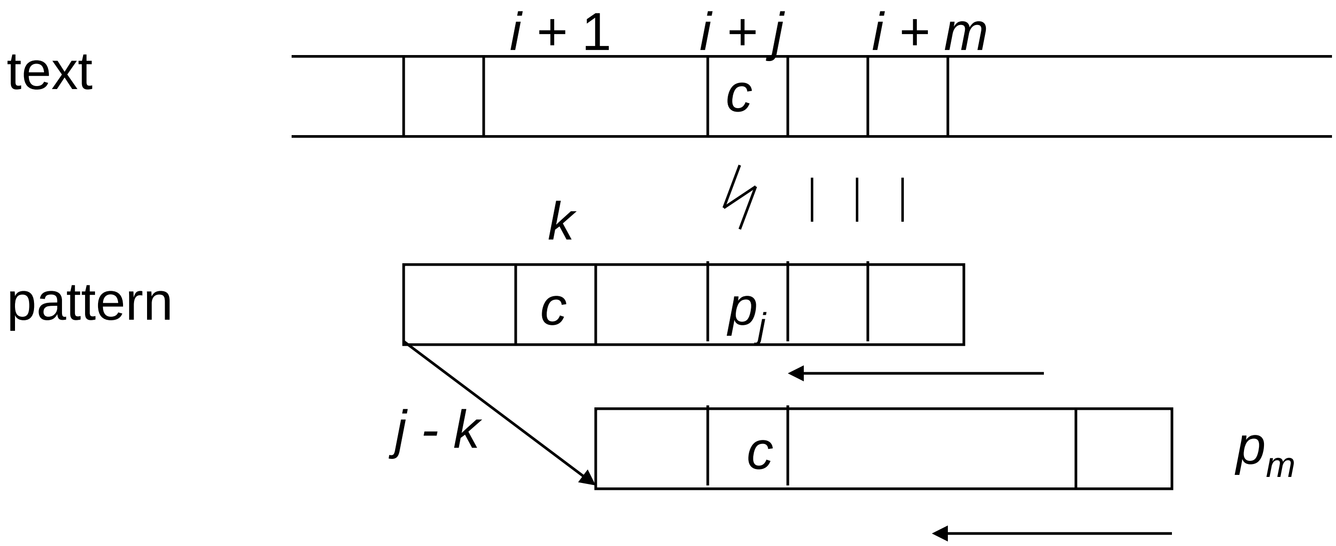


$$\Delta(i) = j$$

BM – Heuristic of occurrence

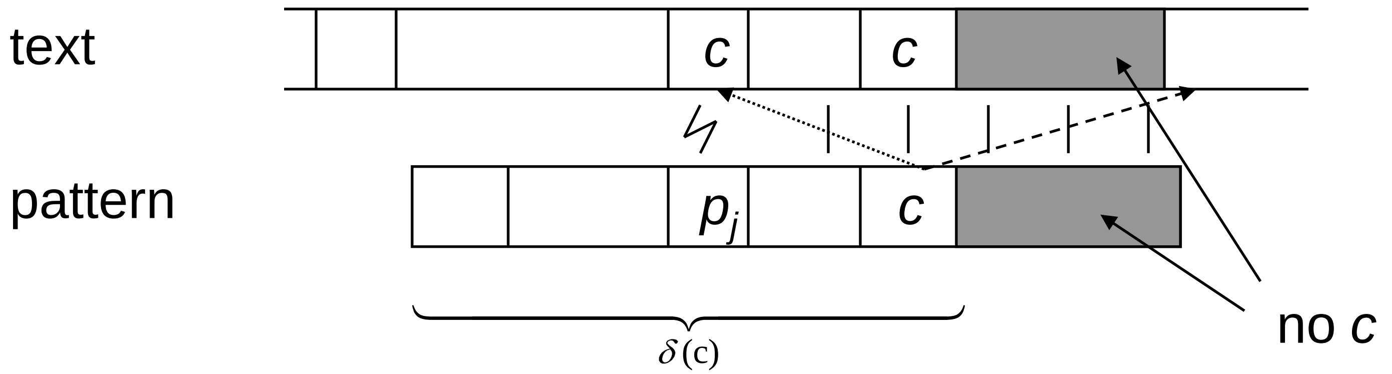
Case 2 *c* occurs in the pattern. ($\delta(c) \neq 0$)

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



BM – Heuristic of occurrence

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

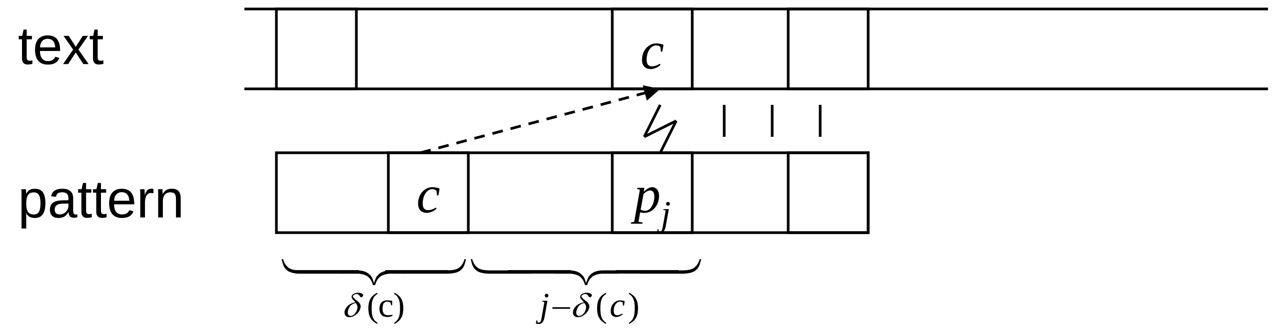


Shift of the rightmost *c* in the pattern to a potential *c* in the text.

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

BM – Heuristic of occurrence

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



Shift of the rightmost *c* in the pattern to *c* in the text:

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

BM algorithm (1st version)

Algorithm BM-search1

Input: Text T and pattern P

Output: Shifts for all occurrences 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;
```

BM algorithm (1st version)

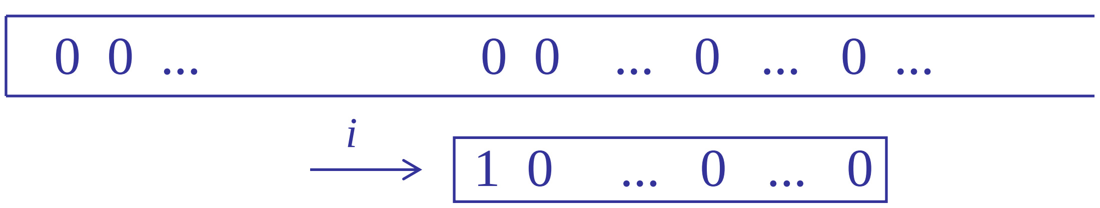
```
  if j = 0
     then output shift i
          i := i + 1
  else if δ(T[i + j]) > j
     then i := i + m + 1 - δ[T[i + j]]
     else i := i + j - δ[T[i + j]]
end while;
```

BM algorithm (1st version)

Analysis:

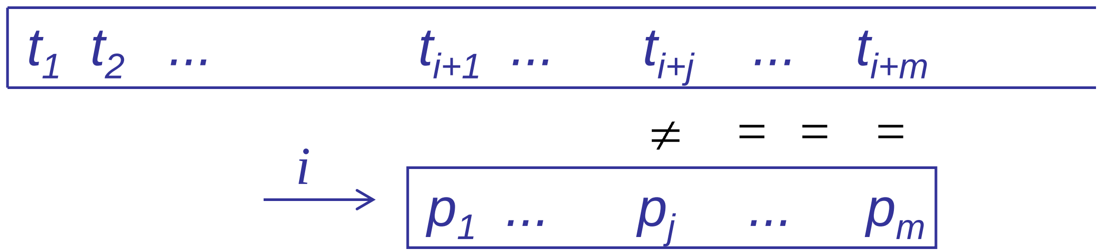
desired running time : $c(m + n/m)$

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



Match heuristic

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



wrw[*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 .

Possible shift: $\gamma[j] = m - wrw[j]$ (wrw[j] >0)

Example for computing *wrw*

wrw[*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

wrw[j]	inspected suffix	forbidden character	further occurrence	posit-ion
wrw[5]	a	n	banana	2
wrw[4]	na	a	*** bana na	0
wrw[3]	ana	n	banana	4
wrw[2]	nana	a	banana	0
wrw[1]	anana	b	banana	0
wrw[0]	banana	ε	banana	0

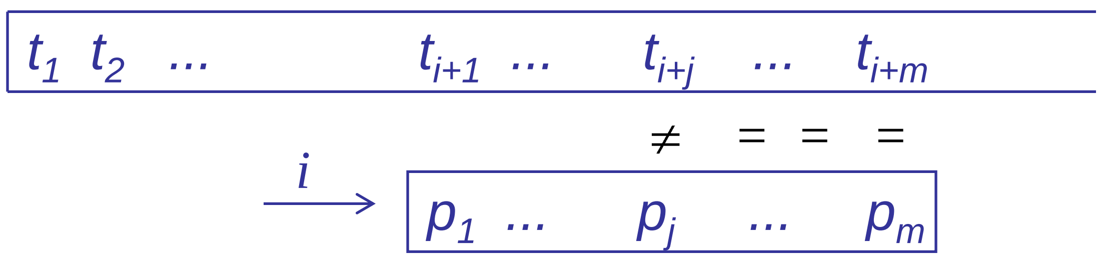
Example for computing *wrw*

$\Rightarrow$ *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
```

Match heuristic

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



wrw[*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 .

Possible shift: $\gamma[j] = m - wrw[j]$ (wrw[j] >0)

$\gamma[j] = ??$ (wrw[j] =0)

BM algorithm (2nd version)

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;
```

BM algorithm (2nd version)

```
     if j = 0
       then output shift i
              i := i + γ[0]
       else i := i + max(γ[j], j - δ[T[i + j]])
 end while;
```