Algorithms Theory

15 – Text search

P.D. Dr. Alexander Souza

Winter term 11/12

Text search

Various 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

Given:

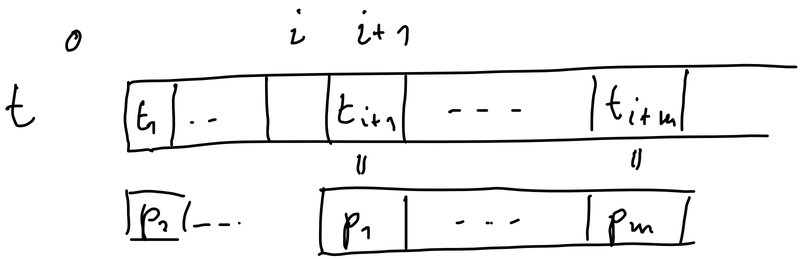
text $t_1\, t_2 \ldots t_n \in \Sigma^n$

pattern $p_1 p_2 \ldots p_m \in \Sigma^m$

Goal:

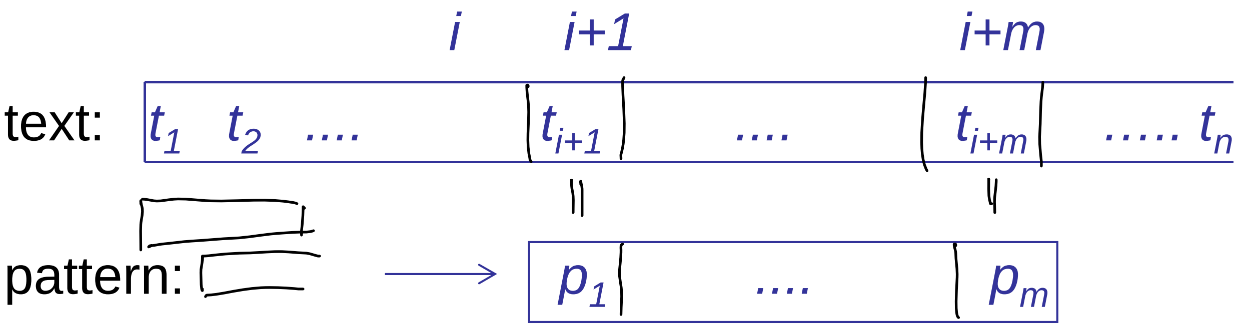
Find one or all occurrences of the pattern in the text, i.e. positions i ($0 \le i \le n - m$) such that

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



complete match.

Problem definition



Running time:

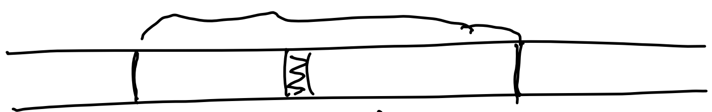
1. \# possible alignments: $n - m + 1$, # pattern positions: m

   → O($n \cdot m$)

2. At least 1 comparison per m consecutive text positions:

   → $\Omega\,(\,m + n/m\,)$

   reading pattern



≥ 1 comparison for any correct algorithm
otherwise pattern could be missed.

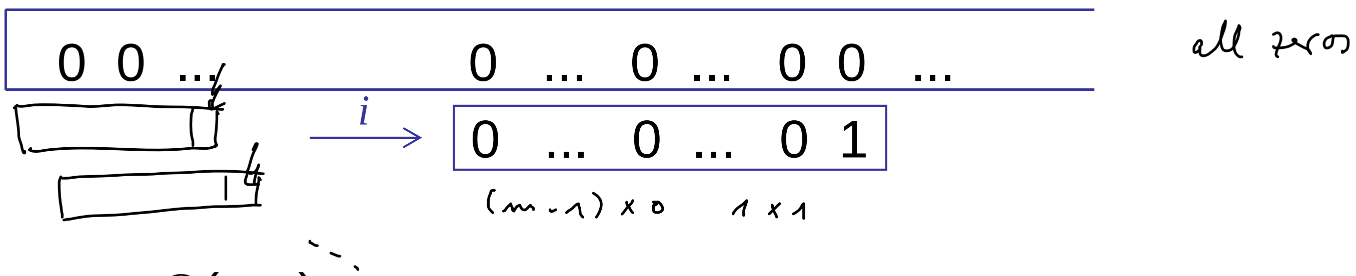
Naive method

For each possible position $0 \leq i \leq n - m$, check at most m character 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: $\Omega(m \cdot n)$

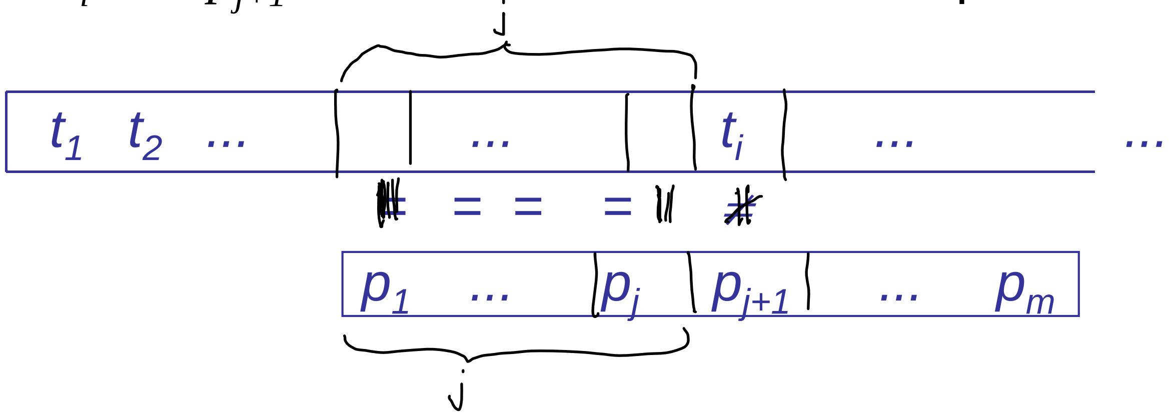
In practice, a mismatch usually occurs very early.

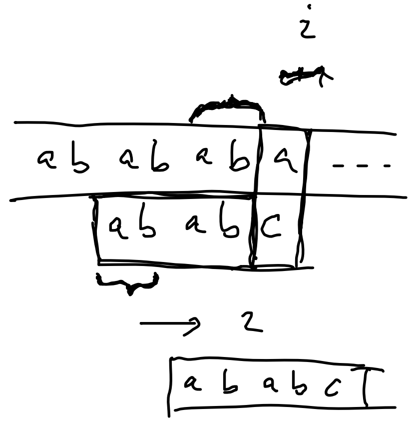
→ running time ~ $c\, n$ in practice.

The Knuth-Morris-Pratt algorithm (KMP)

We want to find the largest prefix in the pattern which is also a proper suffix.

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
- $t_i \neq p_{j+1}$

Naive: shift pattern only one position ahead

Idea: Do not necessarily start from scratch, but try to shift the pattern by more than just position. How far?

The Knuth-Morris-Pratt algorithm (KMP)

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

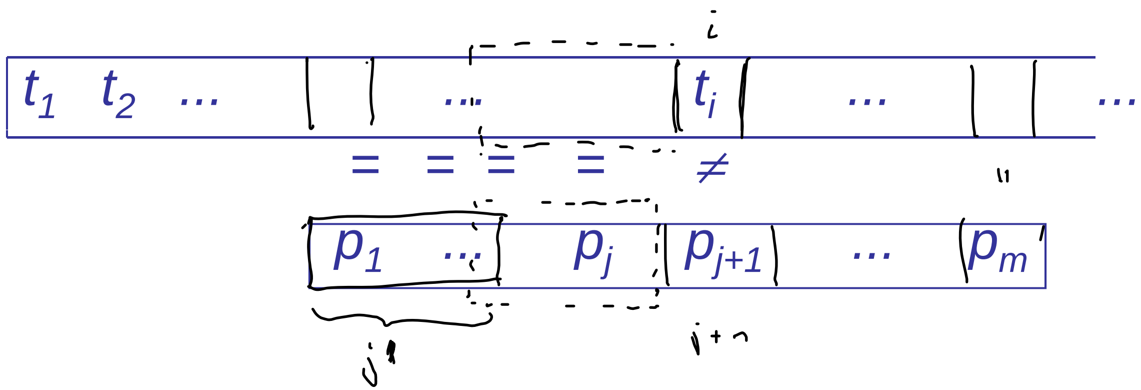
array to be constructed

Idea:

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

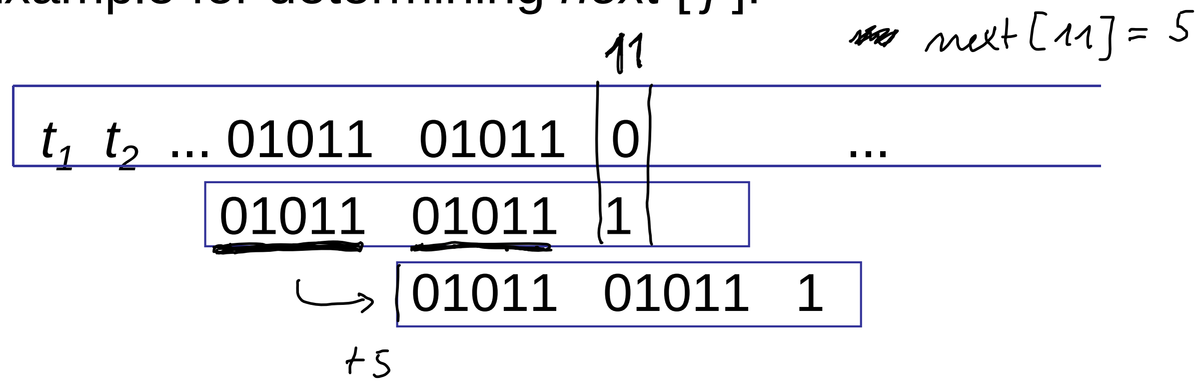
Find greatest $j' < j$ such that $P_{1 \ldots j'} = P_{j-j'+1 \ldots j}$.

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



The Knuth-Morris-Pratt algorithm (KMP)

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



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

The 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

The 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;                                              next[j] < j
    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;
  O(1)    if j = m then L := [ L[ ] , i-m ] ;
                          j := next [ j ]
          fi;
     od;
     RETURN (L);
end;
```

The 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
next [4] = 1

The 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

The Knuth-Morris-Pratt algorithm (KMP)

Correctness:

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

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

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

When starting the for-loop:

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

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

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

The 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.

The Knuth-Morris-Pratt algorithm (KMP)

Running time:

- the 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

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

$O(n + m)$

Computation of the *next*-array

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

next [1] = 0

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

p_1 p_2 ... ... p_i ... ...

= = = = $\neq$

p_1 ... p_j p_{j+1} ... p_m

Computation of the *next*-array

Consider 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 $p_i = p_{j+1}$, set *next* [*i*] = *j* + 1, otherwise *next* [*i*] = 0.

Computation of 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 runs in $O(n + m)$ time.

Can text search be realized even 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:

```
h e   s a i d   a b r a k a d a b r a   b u t
    ≠
b u t

h e   s a i d   a b r a k a d a b r a   b u t
          ≠
      b u t
```

The Boyer-Moore algorithm (BM)

```
he said abrakadabra but
        ≠
      but

he said abrakadabra but
           ≠
         but

he said abrakadabra but
              ≠
            but
```

The Boyer-Moore algorithm (BM)

```
he said abrakadabra but
                 ∤
               but

he said abrakadabra but
                  ∤
                but

he said abrakadabra but
                     ∤
                   but

he said abrakadabra but
                    |||
                    but
```

Large jumps:
few comparisons

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

BM: last-occurrence function

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

$\delta[c]$:= index of the right-most occurrence of c in P

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

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

What is the cost for computing all δ-values?

Let $|\Sigma| = l$:

BM: last-occurrence function

Let

c = the character causing the mismatch

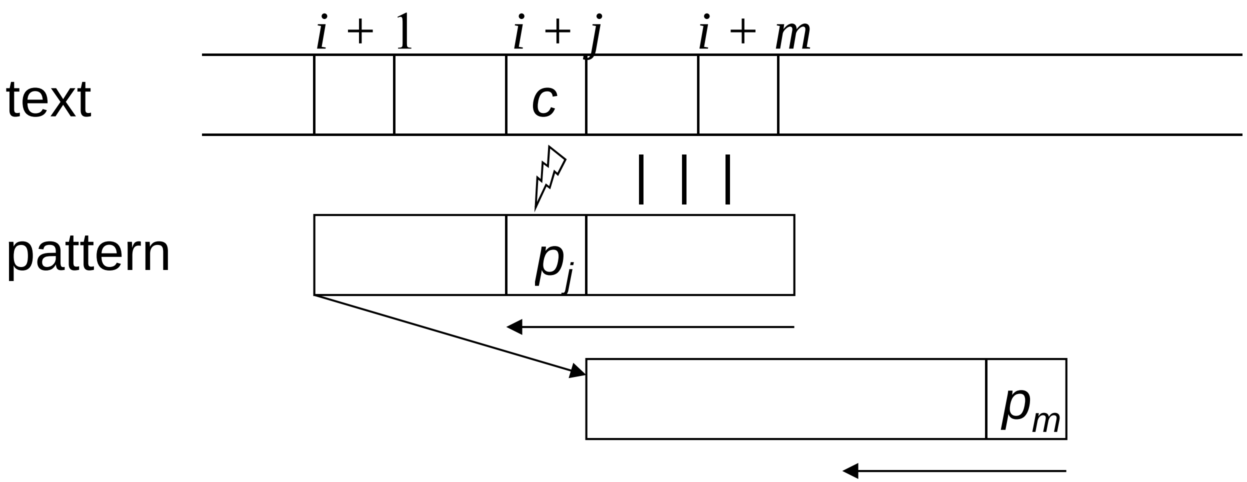
j = the index of the current character in the pattern ($c \neq p_j$)

BM: last-occurrence function

Computation of the pattern shift

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

Shift the pattern *j* characters to the right.

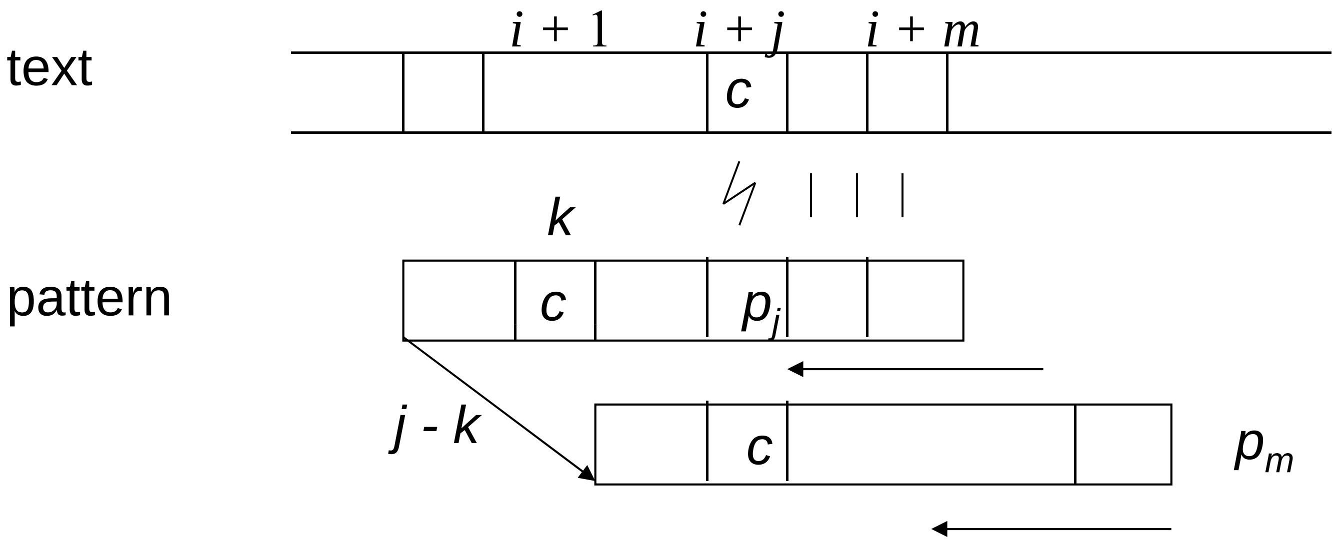


$$\Delta[i] = j$$

BM: last-occurrence function

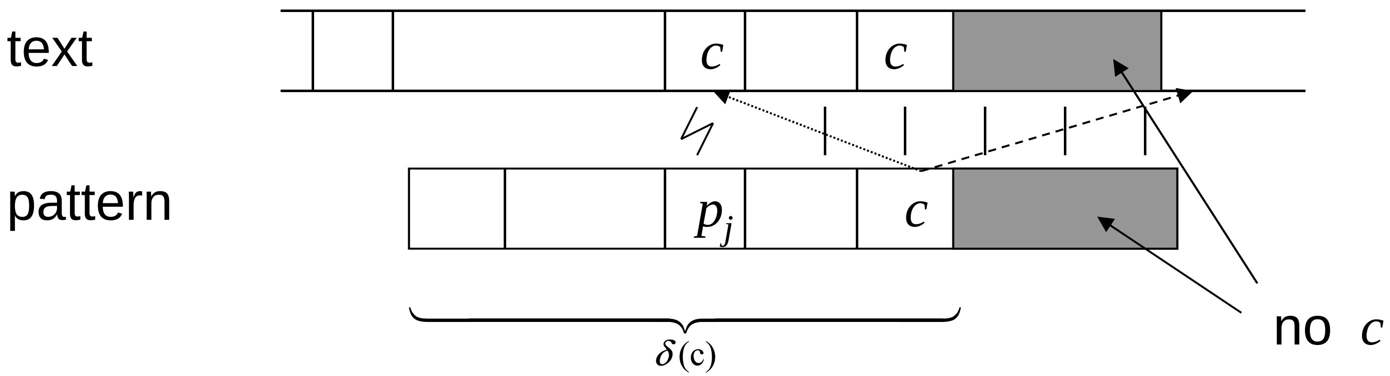
Case 2 *c* occurs in the pattern ($\delta[c] \neq 0$)

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: $\delta[c] > j$

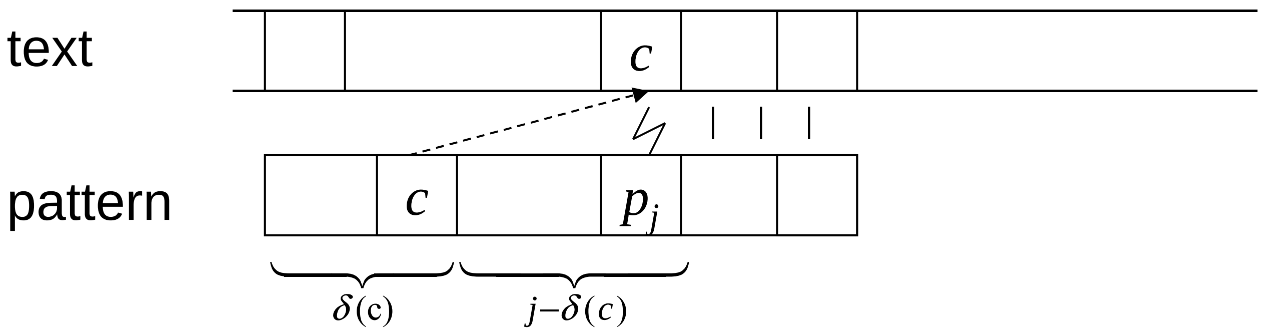


Shift the rightmost *c* in the pattern to a potential *c* in the text.

$$\Rightarrow \text{shift by } \Delta[i] = m - \delta[c] + 1$$

BM: last-occurrence function

Case 2 b: $\delta[c] < j$



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

$$\Rightarrow \text{shift by } \Delta[i] = j - \delta[c]$$

BM: Algorithm (version 1)

Algorithm *BM-search1*

Input: text T, 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;
```

BM: Algorithm (version 1)

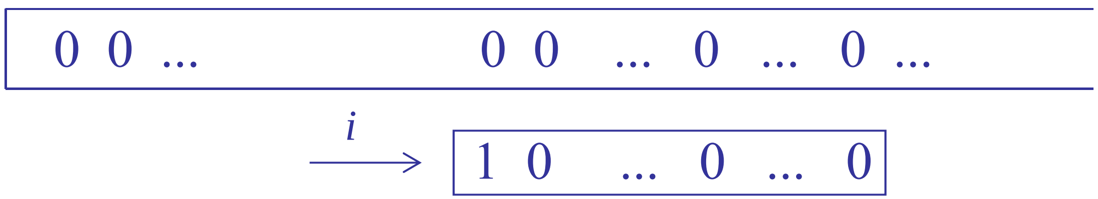
```
    if j = 0
     then output position i
            i := i + 1
     else if δ[ T[i + j] ] > j
        then i := i + m + 1 - δ[ T[i + j] ]
        else i := i + j - δ[ T[i + j] ]
end while;
```

BM: Algorithm (version 1)

Analysis:

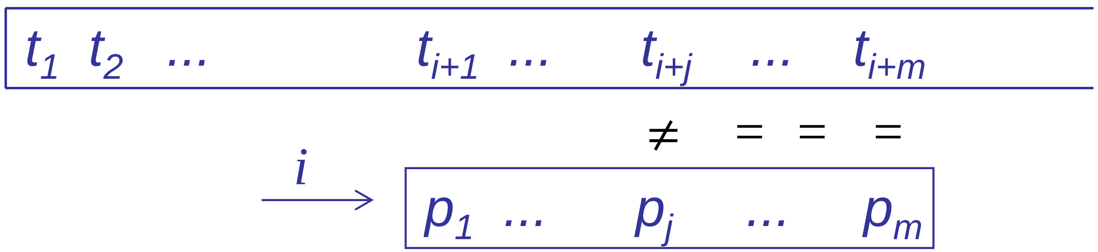
Desired running time: O(m + n/m)

Worst-case running time: Ω(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 $P_{j+1 \ldots m}$ from the right that is not preceded by character P_j (good suffix function)

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

Example of 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	further occurrence	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

Example of 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
```

BM: Algorithm (version 2)

Algorithm *BM-search2*

Input: text *T*, pattern *P*

Output: shift for all occurrences of *P* in *T*

n := length(T); m := length(P)
compute δ and γ
$i := 0$
while $i \leq n - m$ **do**
    $j := m$
    while $j > 0$ **and** $P[j] = T[i + j]$ **do**
        $j := j - 1$
    end while;

BM: Algorithm (version 2)

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