INF 4130 / 9135

30/8-2016

- **Mandatory assignments («Oblig-1», «-2», and «-3»):**

  All three must be approved

  Deadlines around: 25. sept, 25. oct, and 15. nov

- **Other courses on similar themes:**

INF-MAT 3100	Linear optimization
INF-MAT 4110	Mathematical optimization
MAT-INF 3600	Mathematical logic
INF 1080*	Logical methods for computer science
INF 5840	Computability theory

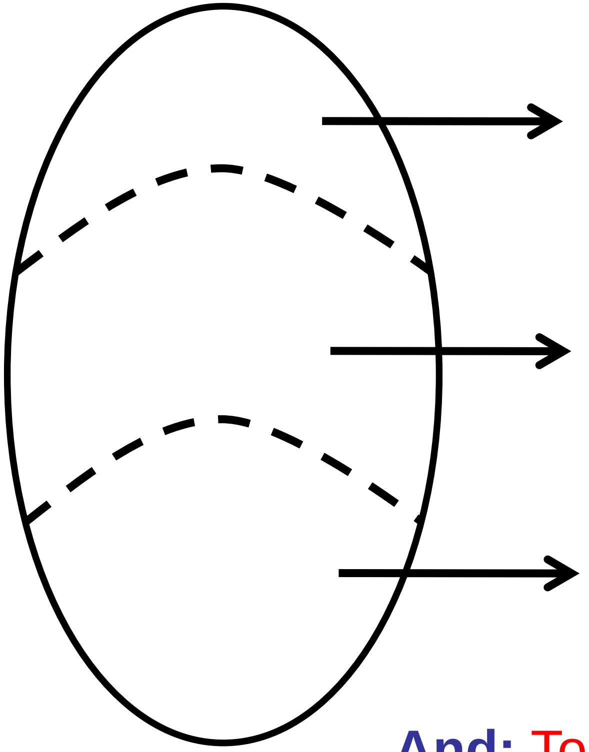
Algorithms, efficiency, and complexity

Problem classes:
The elements (the points inside the ellipse to the right) are «problems», and three problem classes are indicated.

These classes are defined from what type of algorithms can (or cannot) solve problems in it.

E.g. the class P consists of all problems that (in the worst case) can be solved by algorithms running in «polynomial time»,

Note: Each probem will in turn consist of a number of «instances». To be interesting, a problem must have an unlimited number of instances.

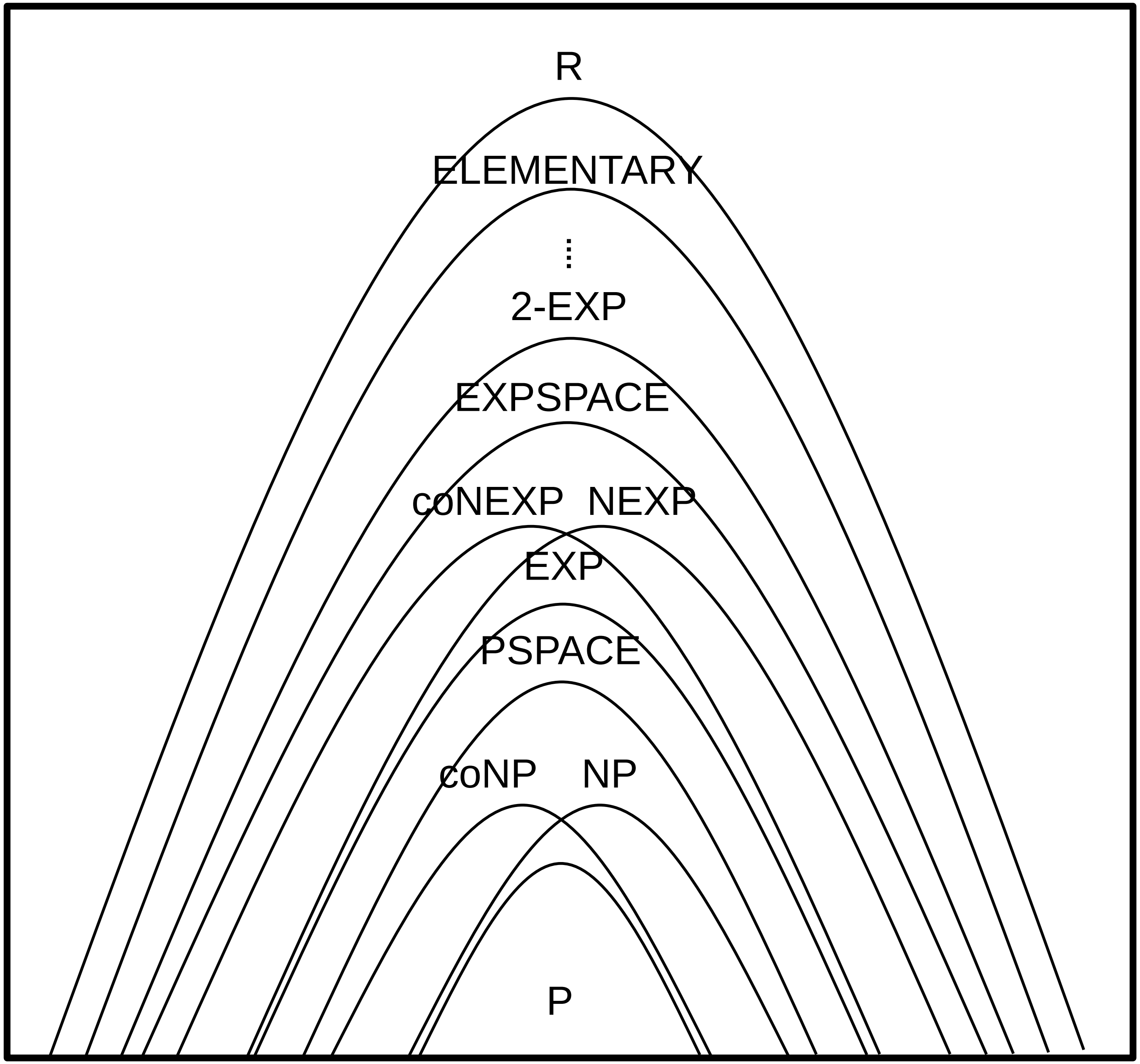


Unsolvable

Intractable
(NP-complete)

Simple (P)

And: Today we will work within P


R
ELEMENTARY
⋮
2-EXP
EXPSPACE
coNEXP
NEXP
EXP
PSPACE
coNP
NP
P

Search for a given (short) string in a long string

Such search problems have become more important lately

- The amount of stored digital information grows steadily (about 3 zettabytes [billion terabytes] 2012).
- Search for a given pattern in DNA strings (about 3 giga-letters in humans).
- Google and similar programs search for given strings (or sets of strings) on all registered web-pages.

Searching for *similar* patterns is also relevant for DNA-strings

- The genetic sequences in organisms are changing over time because of mutations.
- Searches for similar patterns are treated in Ch. 20.5. We will look at that in connection with dynamic programming (Ch. 9, next week).

.

String search

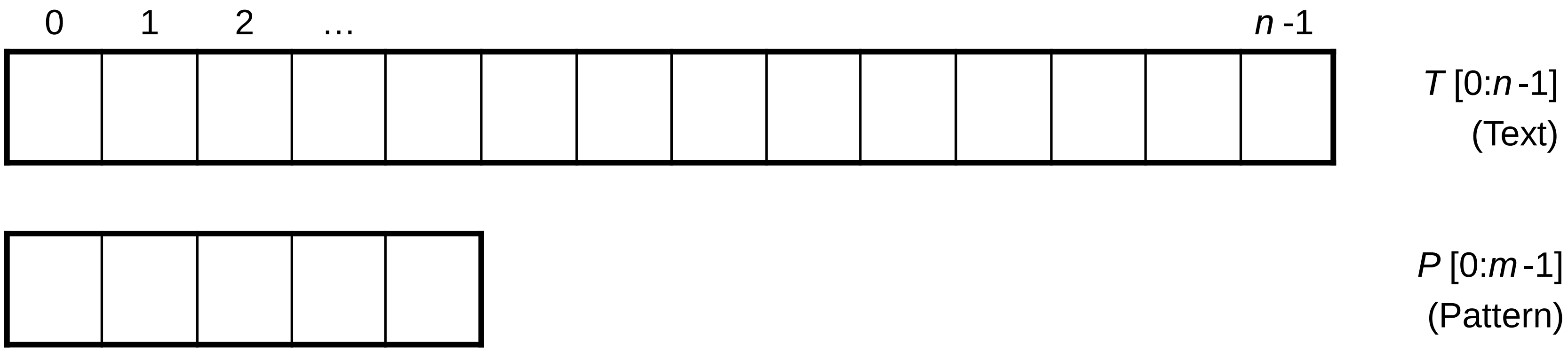
An **alphabet** is a finite set of «symbols» $A = \{a_1, a_2, \ldots, a_k\}$.

A **string** $S = S\,[0{:}\,n\text{-}1]$ of length n is a sequence of symbols from A.

We can consider the string either as an array $S[0{:}n\text{-}1]$ or as a string of symbols $S = < s_0\, s_1 \ldots\, s_{n\text{-}1} >$

The search problem: Given two strings T (= Text) and P (= Pattern), where P is shorter than T (usually much shorter).

Decide whether P occurs as a (continuous) substring in T, and if so, find where it occurs.



0 1 2 … n -1

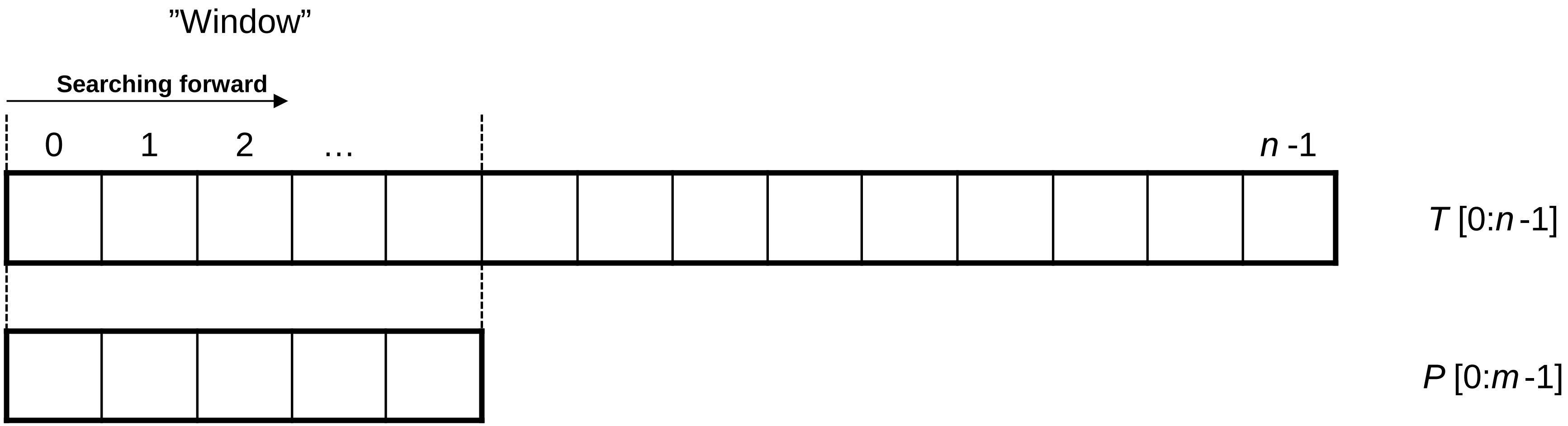
$T\,[0{:}n\text{-}1]$
(Text)

$P\,[0{:}m\text{-}1]$
(Pattern)

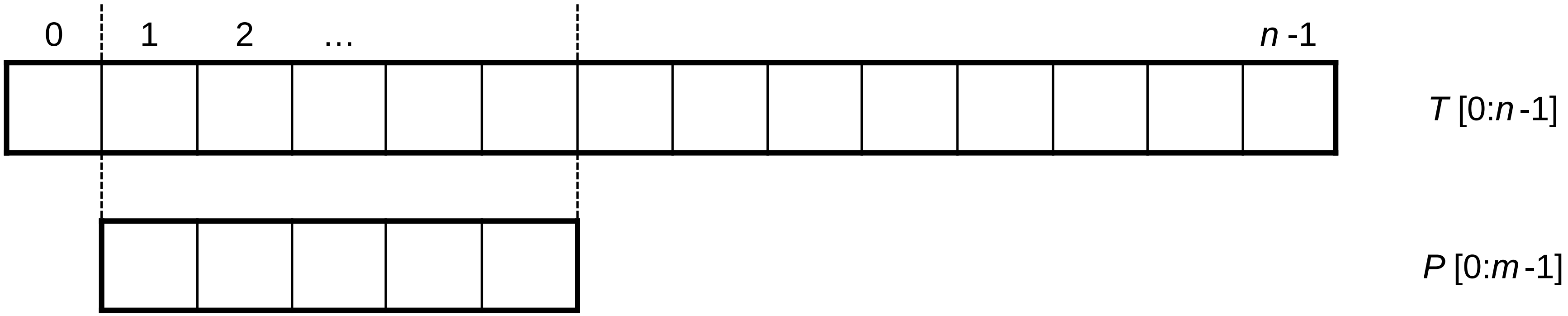
Variants of string search

- Naive algorithm, no preprocessing of neither *T* nor *P*
  - Assume that the length of *T* and *P* are *n* and *m* respectively
  - The naive algorithm is already a polynomial-time algorithm, with worst case execution time $O(n*m)$*, which is also* $O(n^2)$.
- Preprocessing of *P* (the pattern) for each new *P*
  - Prefix-search: The Knuth-Morris-Pratt algorithm
  - Suffix-search: The Boyer-Moore algorithm
  - Hash-based: The Karp-Rabin algorithm
- When searching in the same text a lot of times (with different patterns):
  - Preprocess the text T (which is done to an extreme degree in search engines)
  - We shall look at Suffix trees that relie on a structure called a Trie.

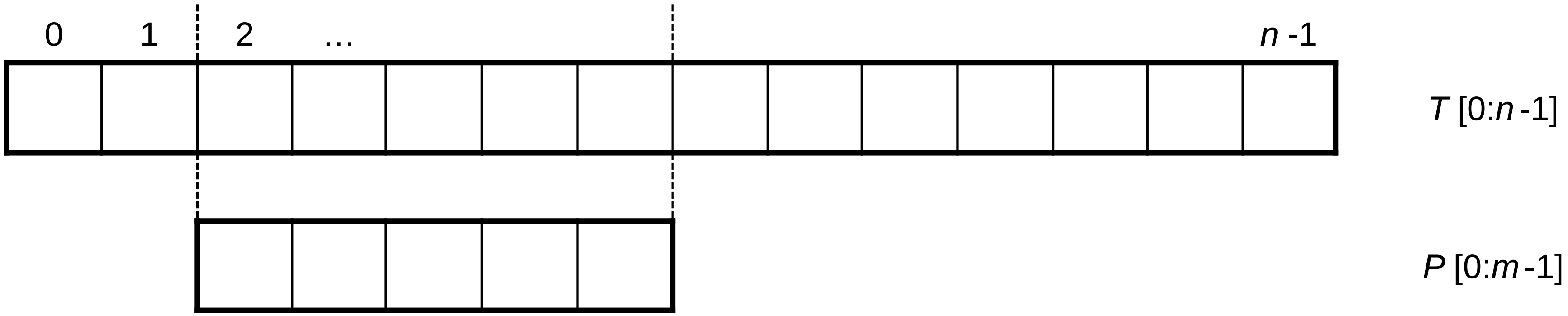
The naive algorithm



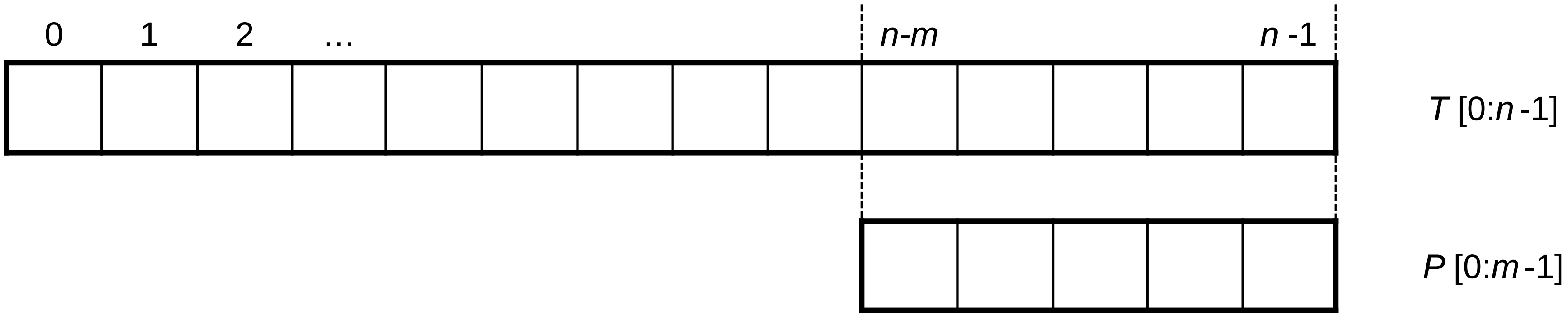
The naive algorithm



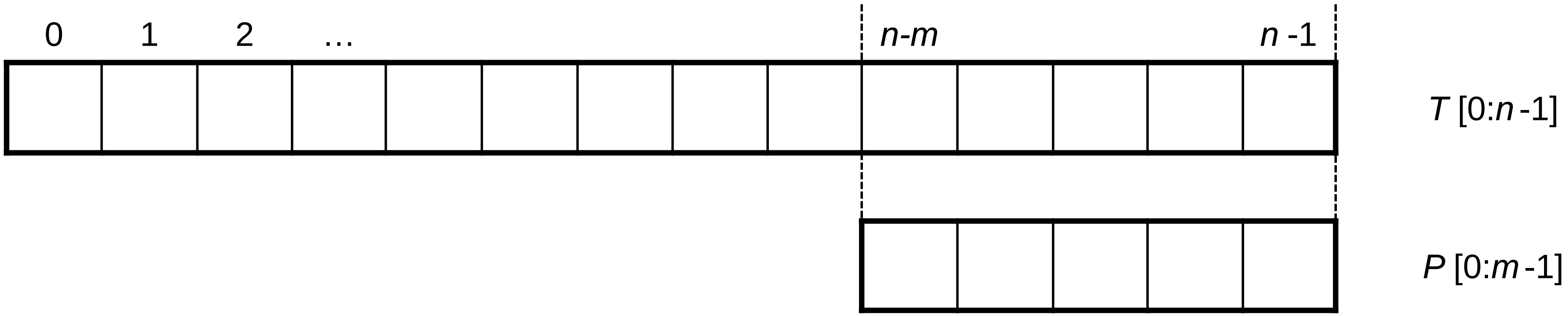
The naive algorithm



The naive algorithm

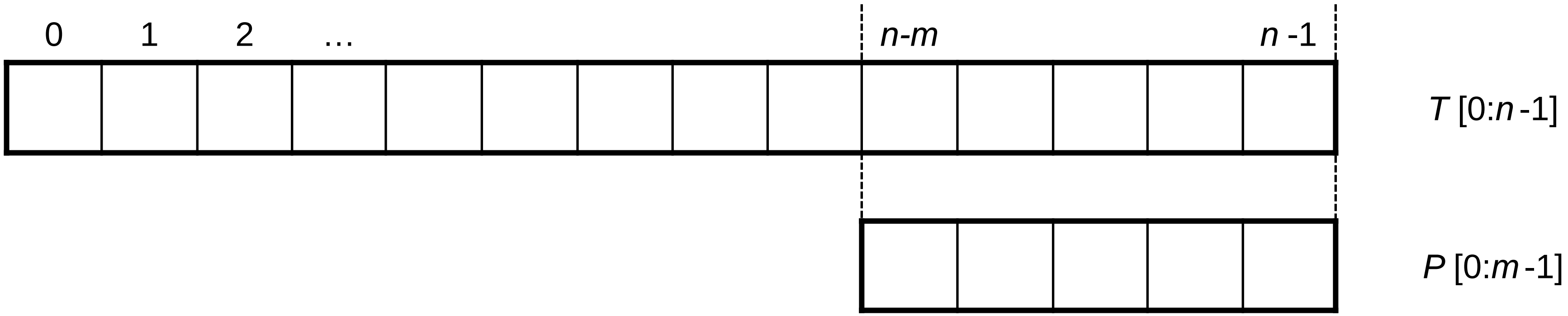


The naive algorithm



```
function NaiveStringMatcher (P [0:m -1], T [0:n -1])
    for s ← 0 to n - m do
        if T [s :s + m - 1] = P  then
            return(s)
        endif
    endfor
    return(-1)
end NaiveStringMatcher
```

The naive algorithm



```
function NaiveStringMatcher (P [0:m -1], T [0:n -1])
    for s ← 0 to n - m do
        if T [s :s + m - 1] = P then
            return(s)
        endif
    endfor
    return(-1)
end NaiveStringMatcher
```

The **for**-loop is executed $n - m + 1$ times.

Each string test has up to m symbol comparisons

$O(nm)$ execution time (worst case)

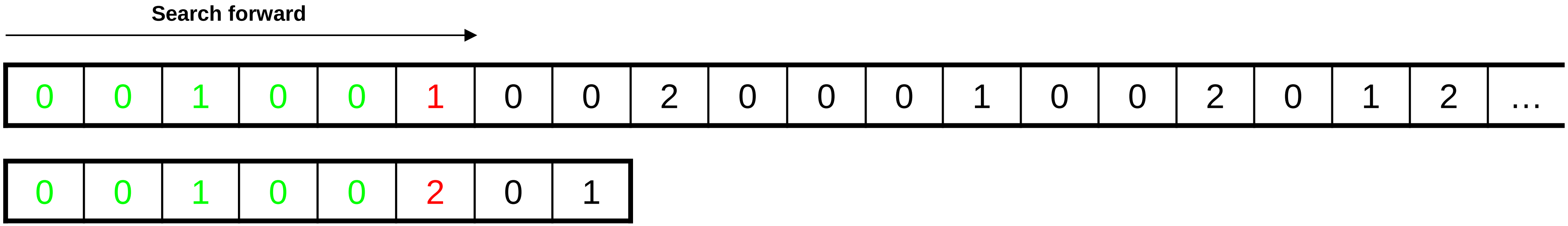
The Knuth-Morris-Pratt algorithm

- There is room for improvement in the naive algorithm
  - The naive algorithm moves the window (pattern) only one character at a time.
  - But maybe we can move it farther, based on what we know from earlier comparisons.
  - We look at the following example:

0	0	1	0	0	1	0	0	2	0	0	0	1	0	0	2	0	1	2	…

0	0	1	0	0	2	0	1

The Knuth-Morris-Pratt algorithm

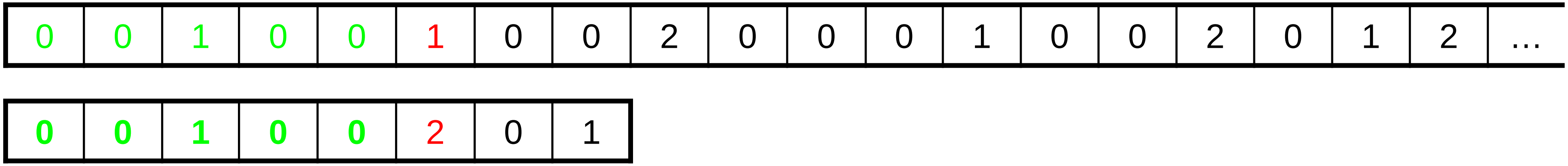


Search forward

0	0	1	0	0	1	0	0	2	0	0	0	1	0	0	2	0	1	2	…

0	0	1	0	0	2	0	1

The Knuth-Morris-Pratt algorithm

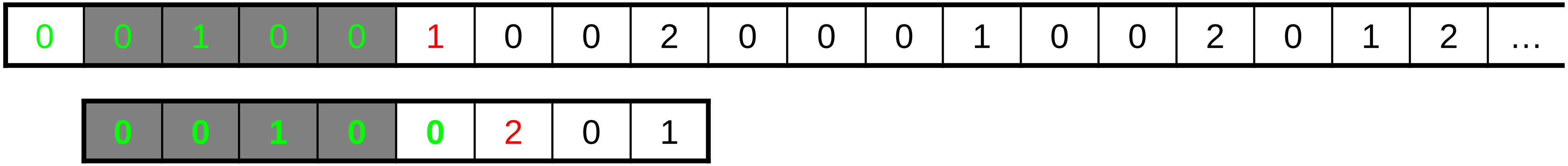


0	0	1	0	0	1	0	0	2	0	0	0	1	0	0	2	0	1	2	…

0	**0**	**1**	**0**	**0**	2	0	1

The Knuth-Morris-Pratt algorithm

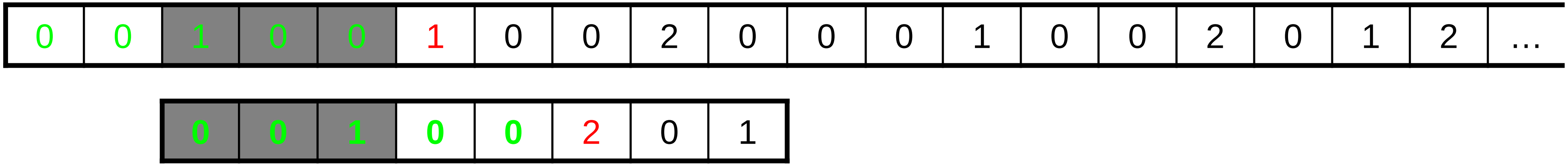
We move the pattern one step: Mismatch



0	0	1	0	0	1	0	0	2	0	0	0	1	0	0	2	0	1	2	…
	0	0	1	0	0	2	0	1											

The Knuth-Morris-Pratt algorithm

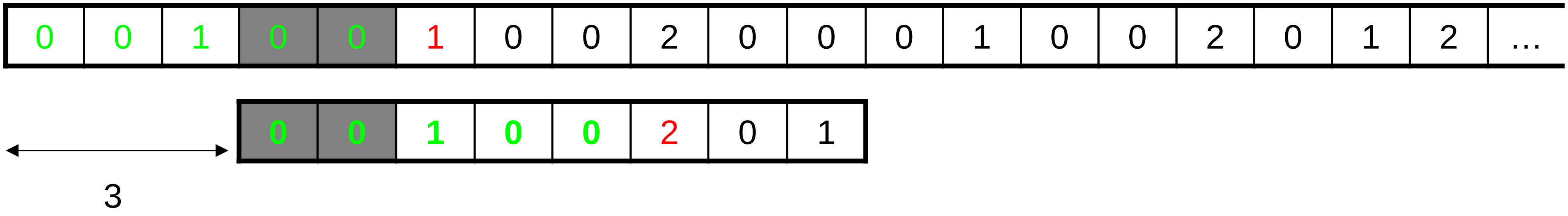
We move the pattern two steps: Mismatch



0	0	1	0	0	1	0	0	2	0	0	0	1	0	0	2	0	1	2	…
		0	**0**	**1**	**0**	**0**	2	0	1										

The Knuth-Morris-Pratt algorithm

We move the pattern three steps: Now, there is at least a match in the part of *T* where we had a match in the previous test



0	0	1	0	0	1	0	0	2	0	0	0	1	0	0	2	0	1	2	…
			0	**0**	**1**	**0**	**0**	2	0	1									

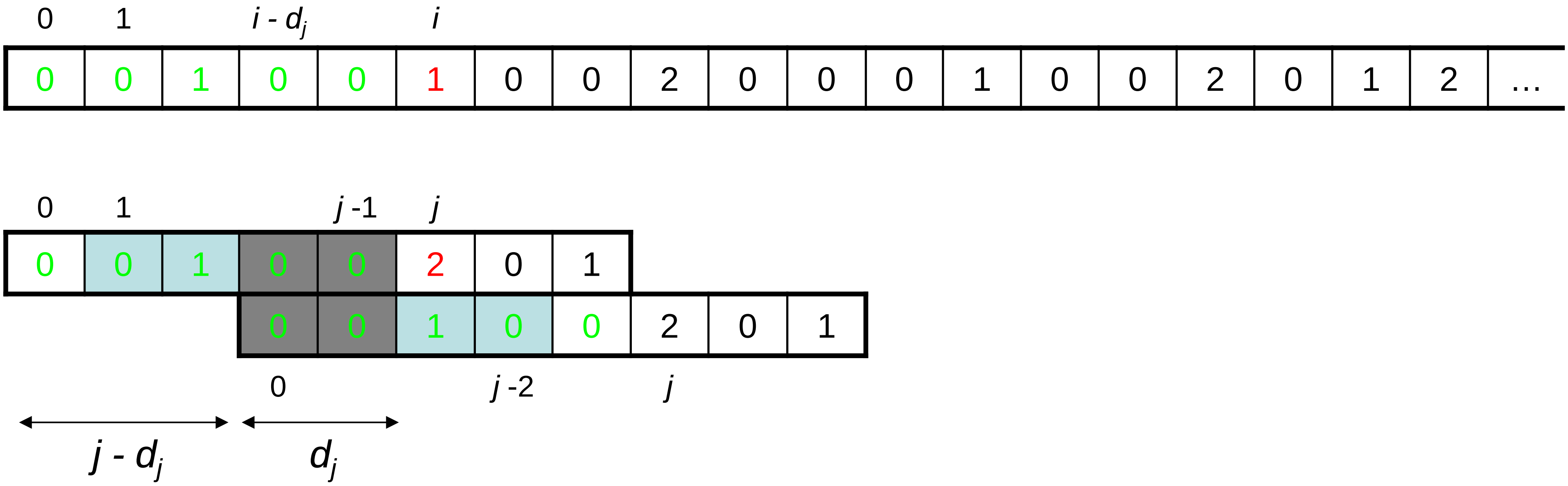
Thus, we can skip a number of tests and move the pattern more than one character forward (three in the above situation) before we start comparing characters again.

The key is that we know what the characters of *T* and *P* are up to the point where *P* and *T* got different in the previous comparison. (*T* and *P* are equal up to this point.)

Thus for each possible index *j* in *P,* we assume that the first difference between *P* and *T* occurs at *j*, and from that compute how far we can move *P* before the next string-comparison.

It may well be that we never get a match like the one above, and we can then move *P* all the way to the point in *T* where we found an inequality. This is the best case for the efficiency of the algorithm.

The Knuth-Morris-Pratt algorithm



d_j is the longest suffix of *P* [1 : j -1] that is also prefix of *P* [0 : j - 2]

We know that if we move *P* less than j - d_j steps, there can be no (full) match.

And we know that, after this move, *P* [0 : d_j -1] will match the corresponding part of *T*.

Thus we can start the comparison at d_j in *P* and compare *P* [d_j : m-1] with the symbols from index *i* in *T.*

Idea behind the Knuth-Morris-Pratt algorithm

- We will produce a table *Next* [0*: m*-1] that shows how far we can move *P* when we get a (first) mismatch at index *j* in *P, j* = 0,1,2, … , *m*-1
- But the array *Next* will not give this number directly. Instead, *Next* [*j*] will contain the new (and smaller value) *that j* should have when we resume the search after a mismatch at *j* in *P* (see below)
    - That is: *Next* [*j*] = *j* – <number of steps that *P* should be moved>,
    - or: *Next* [*j*] *is the value that is named* d_j on the previous slide
- After *P* is moved, we know that the first d_j symbols of *P* are equal to the corresponding symbols in T (that's how we chose d_j).
- So, the search can continue from index *i* in *T* and *Next* [*j*] in *P*.
- And, importantly, The array *Next* can be computed from *P* alone

```
function KMPStringMatcher (P [0:m -1], T [0:n -1])
    i ← 0   // indeks i T
    j ← 0   // indeks i P
    CreateNext(P [0:m -1], Next [n -1])
    while i < n do
        if P [ j ] = T [ i ] then
            if j = m –1 then                // check full match
                return(i – m + 1)
            endif
            i ← i +1
            j ← j +1
        else
            j ← Next [ j ]
            if j = 0 then
                if T [ i ] ≠ P [0] then
                    i ← i +1
                endif
            endif
        endif
    endwhile
    return(-1)
end KMPStringMatcher
```

$O(n)$

Calculating the array *Next* from *P*

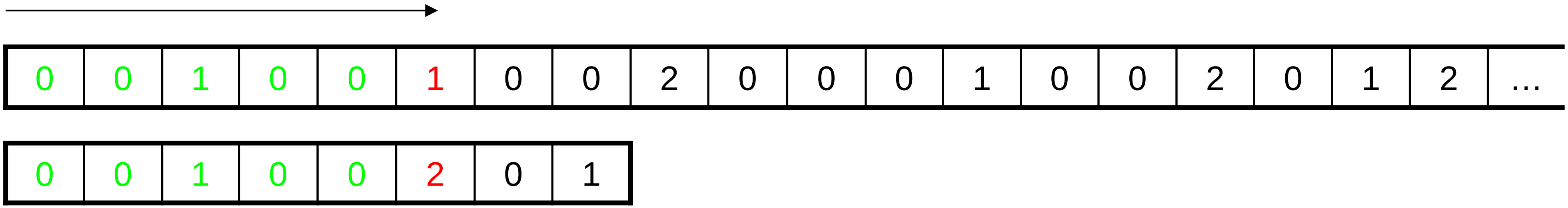
Main form:

```
function CreateNext (P [0:m -1], Next [0:m -1])
    …
end CreateNext
```

- This can be written straight-ahead with simple searches, and will then use time $O(m^2)$.
- However, one can use some of the tricks we used above, and can then find the array *Next* in time $O(m)$.
- The textbook discusses the complex one, but we do not include that one in the curriculum for INF4130.
- We will discuss the simple one as an exercise next week.

The Knuth-Morris-Pratt algorithm

Example



0	0	1	0	0	1	0	0	2	0	0	0	1	0	0	2	0	1	2	…

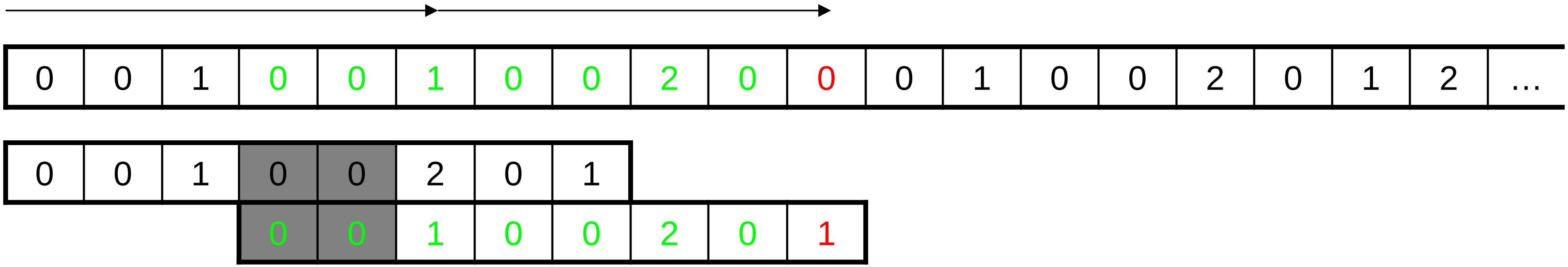
0	0	1	0	0	2	0	1

The array *Next* for the string *P* above:

j =	0	1	2	3	4	5	6	7
Next[j] =	0	0	1	0	1	2	0	1

The Knuth-Morris-Pratt algorithm

Example



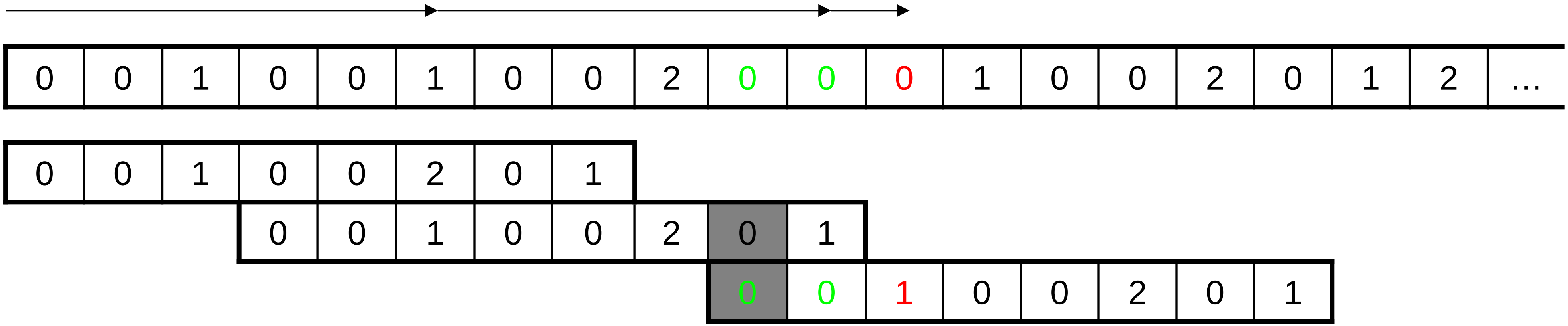
0	0	1	0	0	1	0	0	2	0	0	0	1	0	0	2	0	1	2	…
0	0	1	0	0	2	0	1												
			0	0	1	0	0	2	0	1									

The array *Next* for the string *P* above:

j =	0	1	2	3	4	5	6	7
Next[*j*] =	0	0	1	0	1	2	0	1

The Knuth-Morris-Pratt algorithm

Example



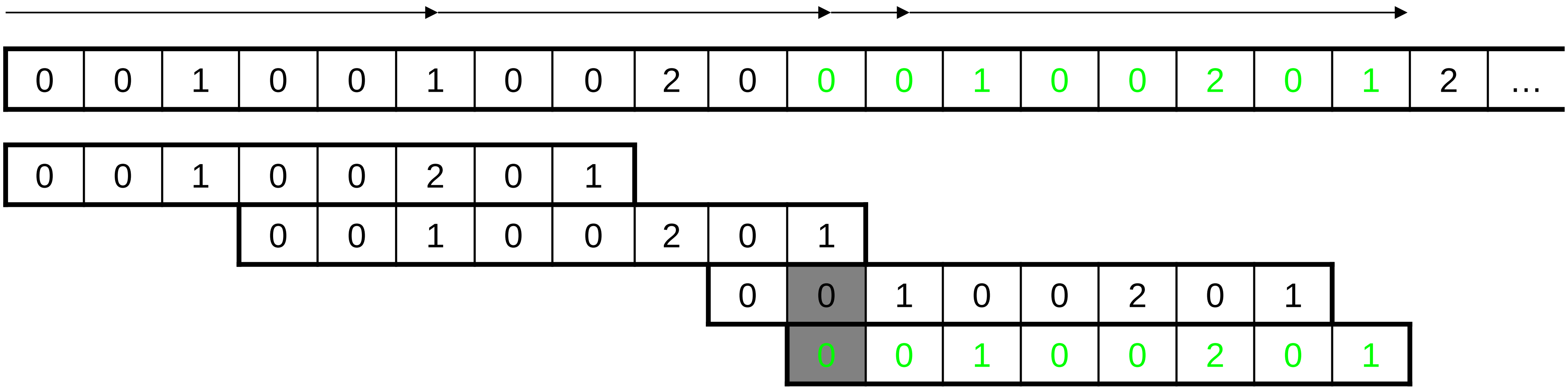
0	0	1	0	0	1	0	0	2	0	0	0	1	0	0	2	0	1	2	…
0	0	1	0	0	2	0	1												
			0	0	1	0	0	2	0	1									
									0	0	1	0	0	2	0	1			

The array *Next* for the string *P* above:

j =	0	1	2	3	4	5	6	7
Next[*j*] =	0	0	1	0	1	2	0	1

The Knuth-Morris-Pratt algorithm

Example



0 0 1 0 0 1 0 0 2 0 0 0 1 0 0 2 0 1 2 …

0 0 1 0 0 2 0 1

0 0 1 0 0 2 0 1

0 0 1 0 0 2 0 1

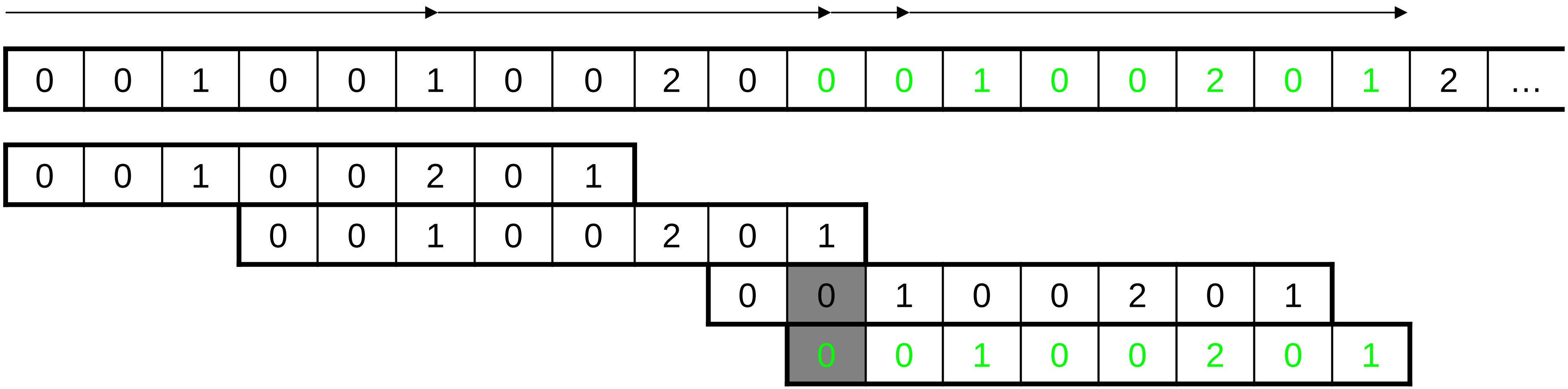
0 0 1 0 0 2 0 1

The array *Next* for the string *P* above:

j =	0	1	2	3	4	5	6	7
Next[*j*] =	0	0	1	0	1	2	0	1

The Knuth-Morris-Pratt algorithm

Example



0	0	1	0	0	1	0	0	2	0	0	0	1	0	0	2	0	1	2	…
0	0	1	0	0	2	0	1												
			0	0	1	0	0	2	0	1									
									0	0	1	0	0	2	0	1			
										0	0	1	0	0	2	0	1		

The array *Next* for the string *P* above:

j =	0	1	2	3	4	5	6	7
Next[*j*] =	0	0	1	0	1	2	0	1

This is a linear algorithm: worst case runtime $O(n)$.

The Boyer-Moore algorithm (Horspool)

- The naive algorithm, and Knuth-Morris-Pratt is prefix-based (from left to right through *P*)
- The Boyer-Moore algorithm (and variants of it) is suffix-based (from right to left in *P*)

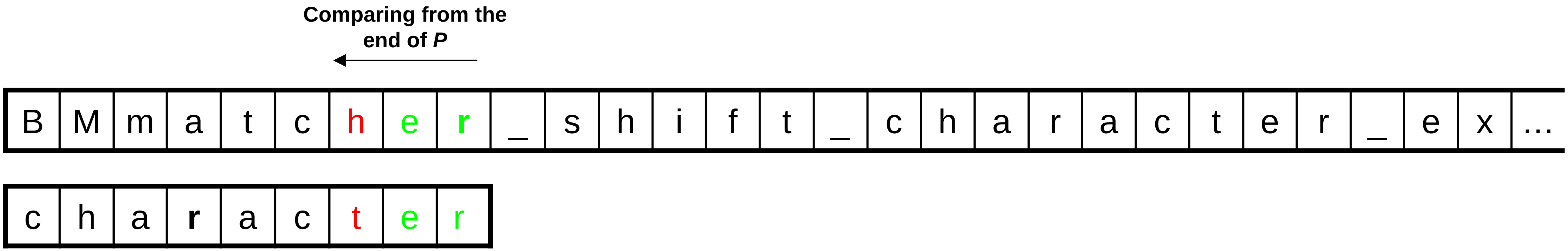
- Horspool proposed a simplification of Boyer-Moore, and we will look at the resulting algorithm here.

- We look at the following example:

B	M	m	a	t	c	h	e	r	_	s	h	i	f	t	_	c	h	a	r	a	c	t	e	r	_	e	x	…

c	h	a	r	a	c	t	e	r

The Boyer-Moore algorithm (Horspool)

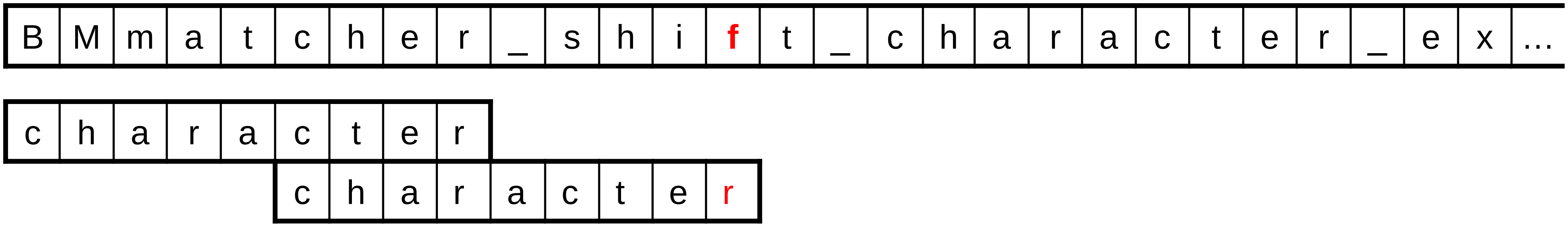


Comparing from the end of *P*

B	M	m	a	t	c	h	e	**r**	_	s	h	i	f	t	_	c	h	a	r	a	c	t	e	r	_	e	x	…

c	h	a	**r**	a	c	t	e	r

The Boyer-Moore algorithm (Horspool)

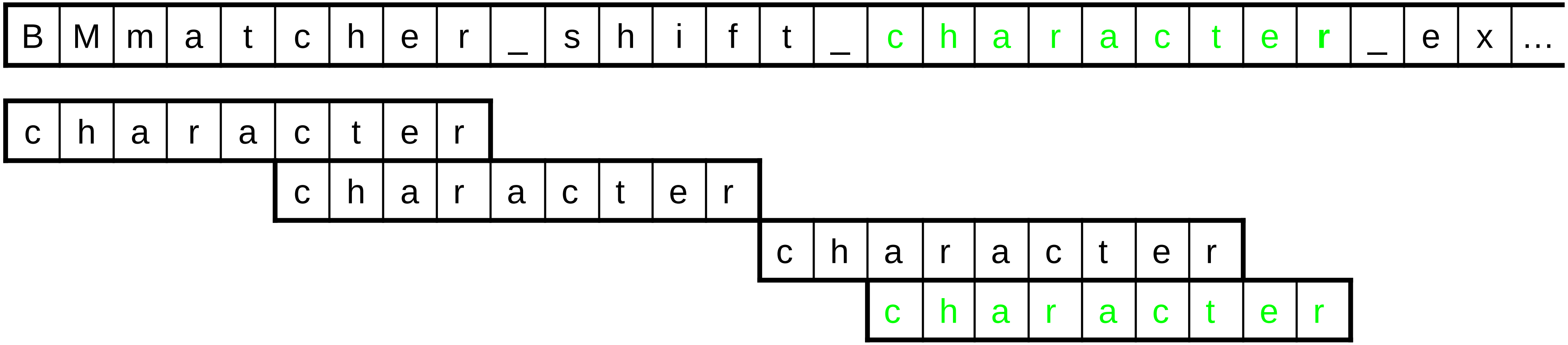


B	M	m	a	t	c	h	e	r	_	s	h	i	**f**	t	_	c	h	a	r	a	c	t	e	r	_	e	x	...
c	h	a	r	a	c	t	e	r																				
					c	h	a	r	a	c	t	e	r															

The Boyer-Moore algorithm (Horspool)

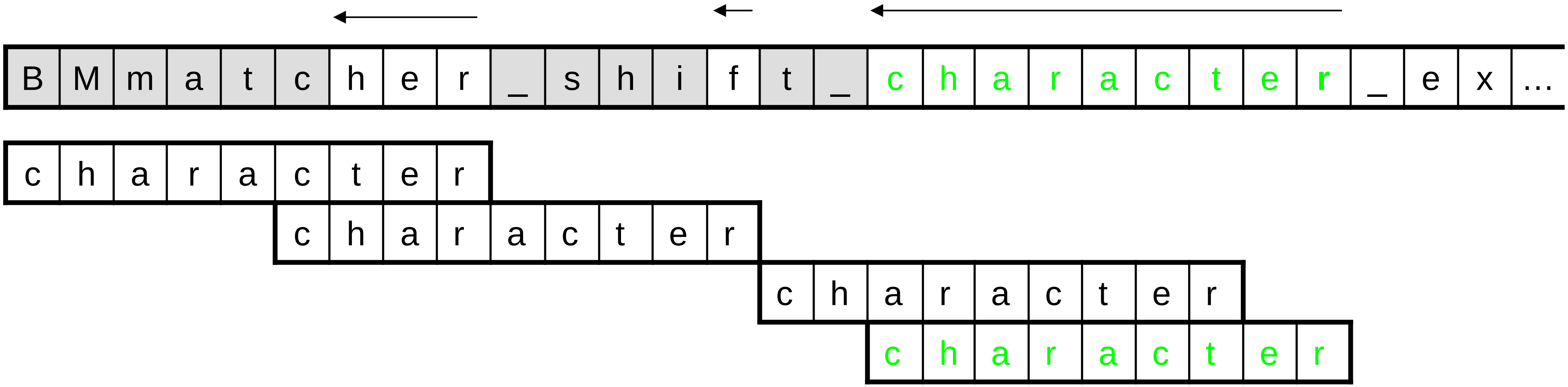
B	M	m	a	t	c	h	e	r	_	s	h	i	f	t	_	c	h	a	r	a	c	**t**	e	r	_	e	x	...
c	h	a	r	a	c	t	e	r																				
					c	h	a	r	a	c	t	e	r															
														c	h	a	r	a	c	**t**	e	r						

The Boyer-Moore algorithm (Horspool)



B	M	m	a	t	c	h	e	r	_	s	h	i	f	t	_	c	h	a	r	a	c	t	e	**r**	_	e	x	...
c	h	a	r	a	c	t	e	r																				
					c	h	a	r	a	c	t	e	r															
														c	h	a	r	a	c	t	e	r						
																c	h	a	r	a	c	t	e	r				

The Boyer-Moore algorithm (Horspool)



Worst case execution time $O(mn)$, same as for the naive algorithm!

However: Sub-linear ($\leq n$), as the average execution time is $O(n\ (\log_{|A|} m) / m)$.

```
function HorspoolStringMatcher (P [0:m -1], T [0:n -1])
    i ← 0
    CreateShift(P [0:m -1], Shift [0:|A| - 1])
    while i < n – m do
        j ← m – 1
        while j ≥ 0 and T [ i + j ] = P [ j ] do
            j ← j -1
        endwhile
        if j = 0 then
            return( i )
        endif
        i ← i + Shift[ T[ i + m -1] ]
    endwhile
    return(-1)
end HorspoolStringMatcher
```

Calculating the array *Shift* from *P*

Main form:

```
function CreateShift (P [0:m -1], Shift [0:|A| - 1])
        …
end CreateShift
```

- We must preprocess *P* to find the array *Shift*.
- The length of Shift[ ] is the number of symbols in the alphabet.
- We search from the end of *P* (minus the last symbol), and calculate the distance from the end for every first occurence of a symbol.
- For the symbols not occuring in *P*, we know:

  Shift [*t*] = <the length of *P*> (*m*)

  This will give a ”full shift”.

The Karp-Rabin algorithm (hash based)

- We assume that the alphabet for our strings is $A = \{0, 1, 2, \ldots, k - 1\}$.
- Each symbol in A can be seen as a digit in a number system with base k
- Thus each string in A^* can be seen as number in this system (and we assume that the most significant digit comes first, as usual)

Example:

k = 10, and A = {0,1, 2, …, 9} we get the traditional decimal number system

The string ”6832355” can then be seen as the number 6 832 355.

- Given a string $P[0: m - 1]$. We can then the corresponding number P' using *m* multiplications and *m* additions (Horners rule, computed from the innermost right expression and outwards):

$$P' = P[m - 1] + k\,(P[m - 2] + \ldots + k\,(P[1] + k\,(P[0])\ldots))$$

Example (written as it computed from left to right):

1234 = ((1*10 + 2)*10 + 3)*10 + 4

The Karp-Rabin algorithm

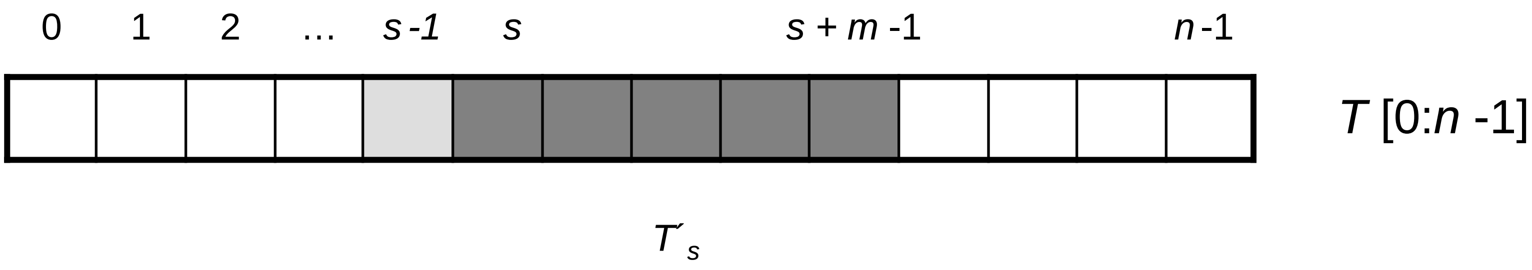
- Given a string $T\,[0{:}\,n\,-1]$, and an integer s (start-index), and a pattern of length m. We then refer to the substring $T\,[s{:}\ s+m\,-1]$ as T_s, and its value is referred to as T'_s

- The algorithm:
  - We first compute the value P' for the pattern P
  - Based on Horners rule we compute T'_0, T'_1 , T'_2 , … and successively compares these numbers to P´

- This is very much like the naive algorithm.

- However: Given $T'_{s\,-1}$ and $k^{\,m-1}$, we can compute T'_s in constant time:

!



The Karp-Rabin algorithm

This constant time computation can be done as follows (where T'_{s-1} is defined as on the previous slide, and k^{m-1} is pre-computed):

$$T'_s = k * (T'_{s-1} - k^{m-1} * T[s]) + T[s+m] \qquad s = 1, \ldots, n - m$$

Example:

$k = 10$, $A = \{0,1, 2, \ldots, 9\}$ (the usual decimal number system) and $m = 7$.

$T'_{s-1} = 7937245$

$T'_s = 9372458$

$$T'_s = 10 * (7937245 - (1000000 * 7)) + 8 = 9372458$$

The Karp-Rabin algorithm, time considerations

- We can compute T'_s in constant time when we know T'_{s-1} and k^{m-1}.
- We can therefore compute P' and the $n - m + 1$ numbers:

  $$T'_s,\ s = 0, 1, \ldots, n - m$$

  in time $O(n)$.
- Thus, we can ”theoretically” implement the search algorithm in time $O(n)$.
- However the numbers T'_s and P' will be so large that storing and comparing them will take too long time (in fact $O(m)$ time).
- The Karp-Rabin trick is to instead use modular arithmetic:
  - We do all computations modulo a value q.
- The value q should be chosen as a prime, so that kq just fits in a register (of e.g. 32/64 bits).
- A prime number is chosen as this will distribute the values well.

The Karp-Rabin algorithm, time considerations

- We compute $T'^{(q)}_s$ and $P'^{(q)}$, where

  $T'^{(q)}_s = T'_s \bmod q$,

  $P'^{(q)} = P' \bmod q$, *(only once)*

  and compare.

  «x mod y» is the remainder when deviding x with *y*, and this is always in the interval {0, 1, …, *y* -1}.

- We can get $T'^{(q)}_s = P'^{(q)}$ even if $T'_s \neq P'$. This is called a spurious match.

- So, if we have $T'^{(q)}_s = P'^{(q)}$, we have to fully check whether $T_s = P$.

- With large enough *q,* the probability for getting spurious matches is low (see next slides)

```
function KarpRabinStringMatcher ($P[0:m-1]$, $T[0:n-1]$, $k$, $q$)
    $c \leftarrow k^{m-1} \bmod q$
    $P'^{(q)} \leftarrow 0$
    $T'^{(q)}_s \leftarrow 0$

    for $i \leftarrow 1$ to $m$ do
        $P'^{(q)} \leftarrow (k * P'^{(q)} + P[i]) \bmod q$
        $T'^{(q)}_0 \leftarrow (k * T'^{(q)}_0 + T[i]) \bmod q$
    endfor

    for $s \leftarrow 0$ to $n - m$ do
        if $s > 0$ then
            $T'^{(q)}_s \leftarrow (k * (T'^{(q)}_{s-1} - T[s] * c) + T[s + m]) \bmod q$
        endif
        if $T'^{(q)}_s = P'^{(q)}$ then
            if $T_s = P$ then
                return($s$)
            endif
        endif
    endfor
    return(-1)
    end KarpRabinStringMatcher
```

The Karp-Rabin algorithm, time considerations

- The worst case running time occurs when the pattern *P* is found at the end of the string *T.*

- If we assume that the strings are distributed uniformally, the probability that $T'^{(q)}_s$ is equal to P' (which is in the interval {0, 1, …, *q*-1}) is 1/*q*

- Thus $T'^{(q)}_s$, for *s* = 0, 1, …, *n*-*m*-1 will for each *s* lead to a spurious match with probability 1/*q.*

- With the real match at the end of *T*, we will on average get (*n* - *m*) / *q* spurious matches during the search

- Each of these will lead to *m* symbol comparisons. In addition, we have to check whether $T'^{(q)}_{n-m}$ equals *P* when we finally find the correct match at the end.

- Thus the number of comparisons of single symbols and computations of new values $T'^{(q)}_s$ will be:

$$\left(\frac{n-m}{q}+1\right)m+(n-m+1)$$

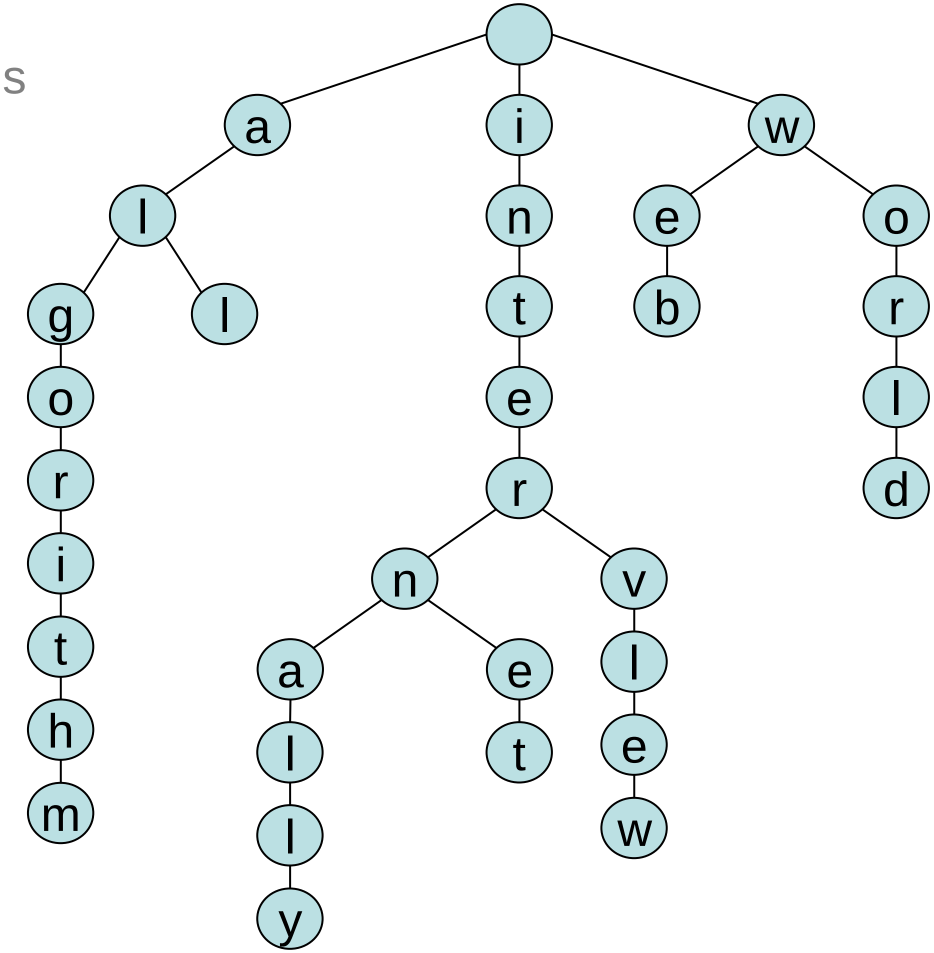
- We can choose values so that *q* >> *m.* Thus the runing time will be *O*(*n*).

Multiple searches in a fixed string *T* (structure)

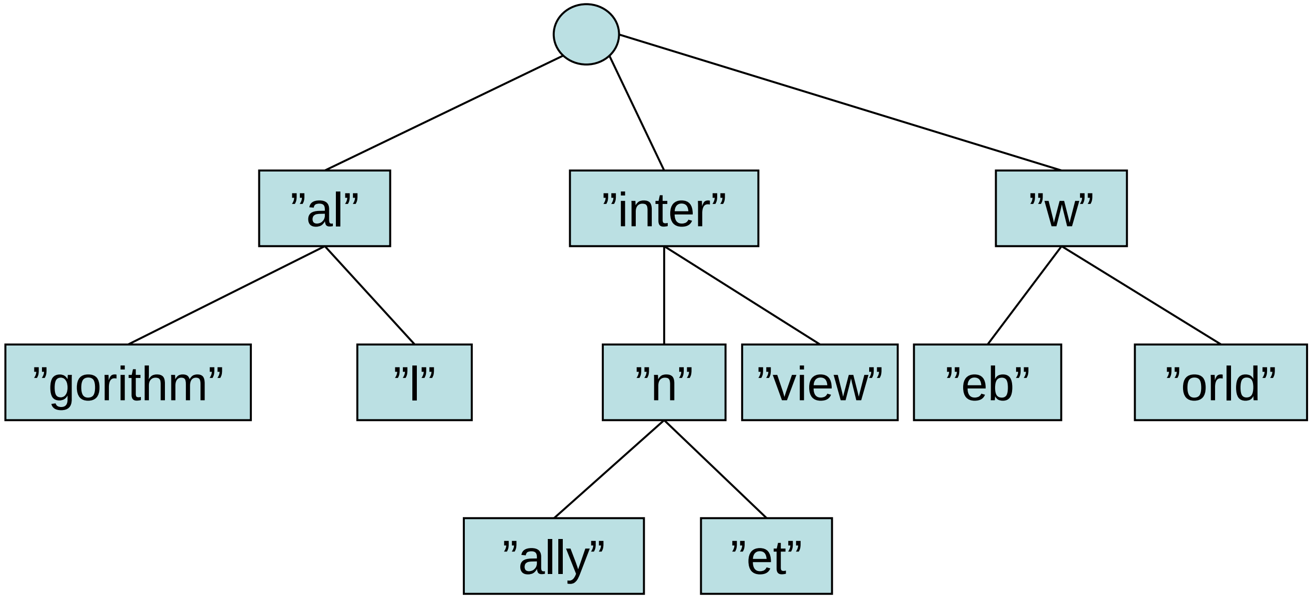
- It is then usually smart to *preprocess* *T*, so that later searches in *T* for different patterns *P* will be fast.
  - Search engines (like Google or Bing) do this in a very clever way, so that searches in huge number of web-pages can be done extremely fast.
- We often refer to this as *indexing* the text (or data set), and this can be done in a number of ways. We will look at the following technique:
  - Suffix trees, which relies on ”Tries” trees.
  - So we first look at Tries.
- T may also gradually change over time. We then have to update the index for each such change.
  - The index of a search engine is updated when the crawler finds a new web page.

First: Trie trees

In the textbook there is an error here

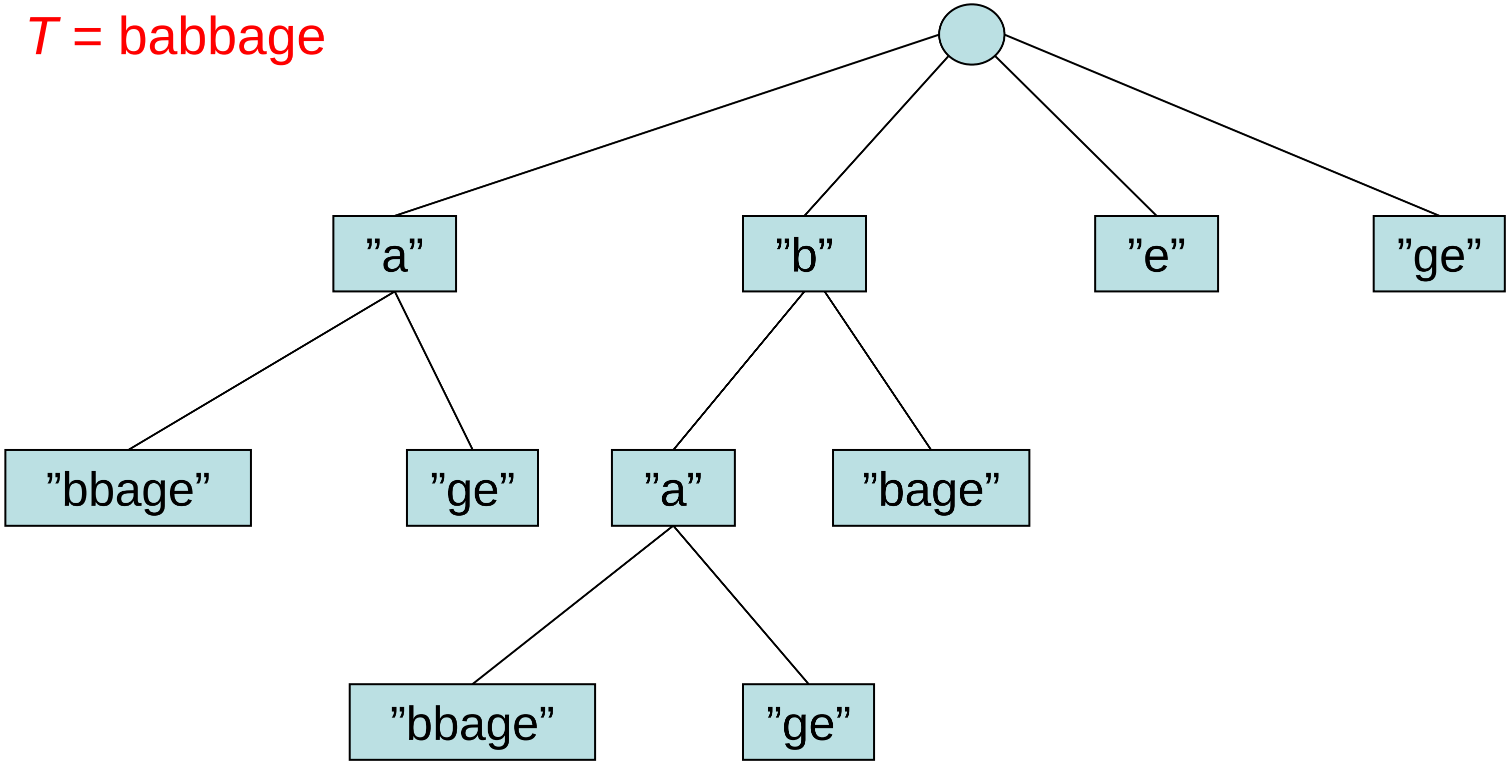


Compressed trie tree



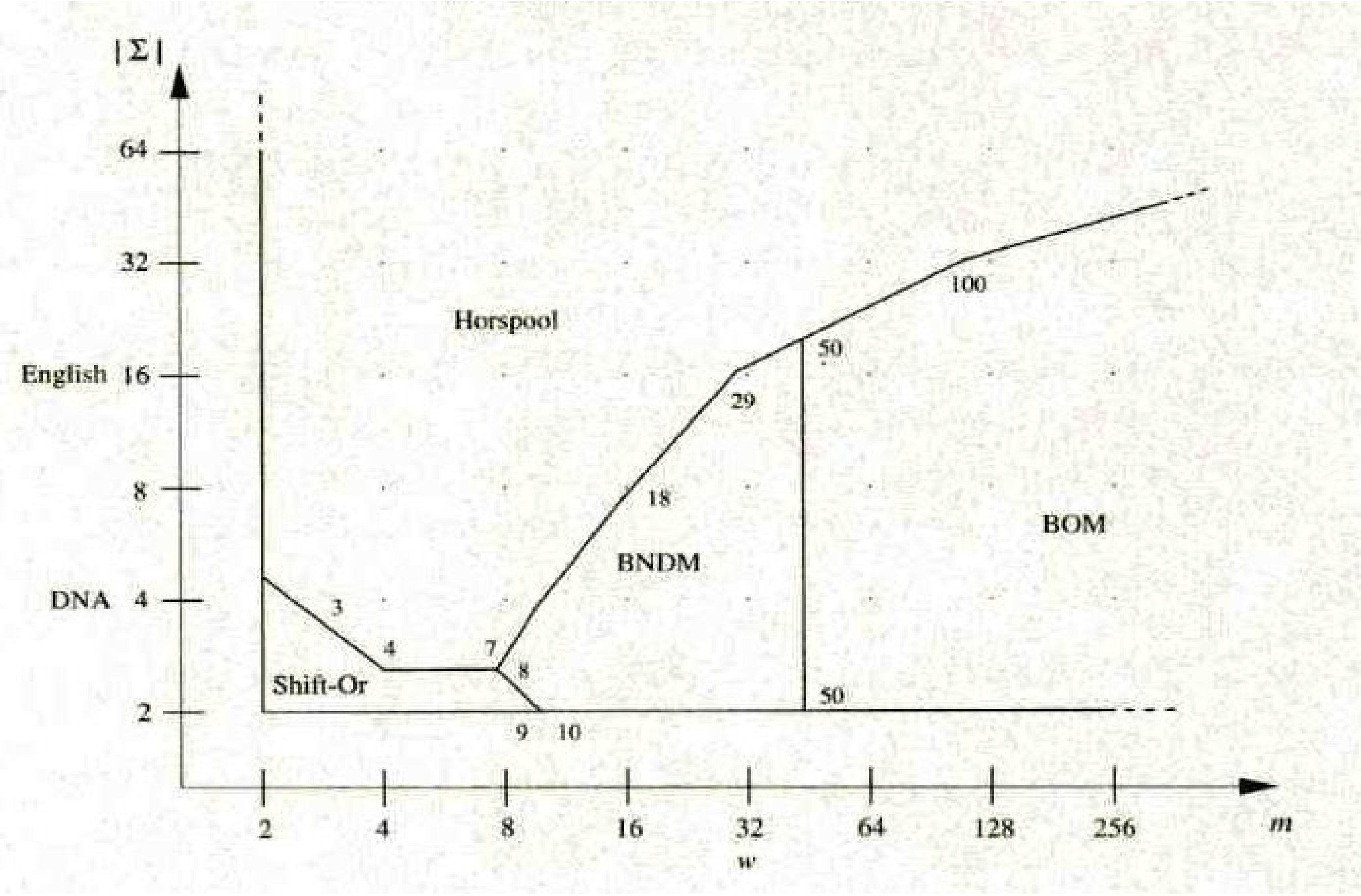
And finally: Suffix trees (compressed)

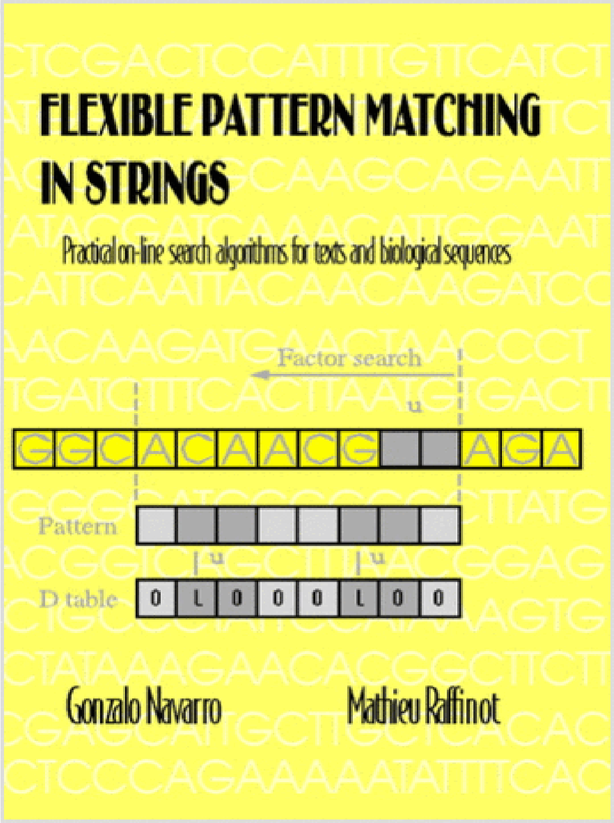
Suffix tree for
T = babbage



- Looking for *P* in this Trie will decide whether *P* occurs in *T*
- *There are a lot of optimizations that can be done for such trees*

Div.




FLEXIBLE PATTERN MATCHING IN STRINGS

Practical on-line search algorithms for texts and biological sequences

Gonzalo Navarro Mathieu Raffinot