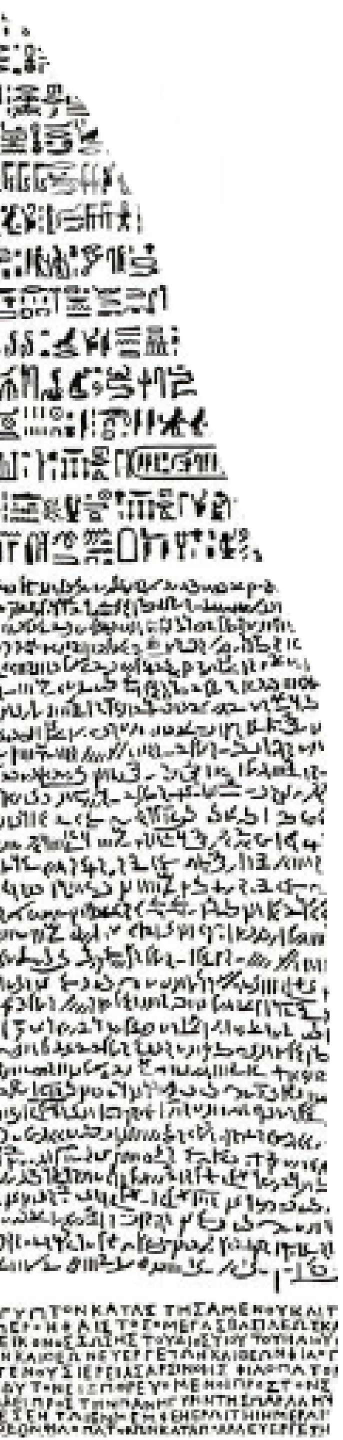


String Search

7th September 2022

Petter Kristiansen

Searching is increasingly important

- Vast ammounts of information is available
  - Google and other search engines search for given strings (or sets of strings) on all registered web-pages
  - The amount of stored digital information grows steadily (rapidly – 61 % compound rate)
    - 3 zettabytes (10^{21} = 1 000 000 000 000 000 000 000 bytes) in 2012
    - 4.4 zettabytes in 2013
    - 59 zettabytes in 2020 (44 ZB estimated in 2019)
    - 175 zettabytes in 2025 (estimated)

Searching is increasingly important

- Search for a given pattern in DNA strings (about 3 «giga-letters» (10^9) in human DNA – four letters in the alphabet: A, C, G, T)

- Searching for similar patterns is also relevant
  - The genetic sequences in organisms are changing over time because of mutations
  - We will look at searches for similar patterns that in connection with **Dynamic Programming**

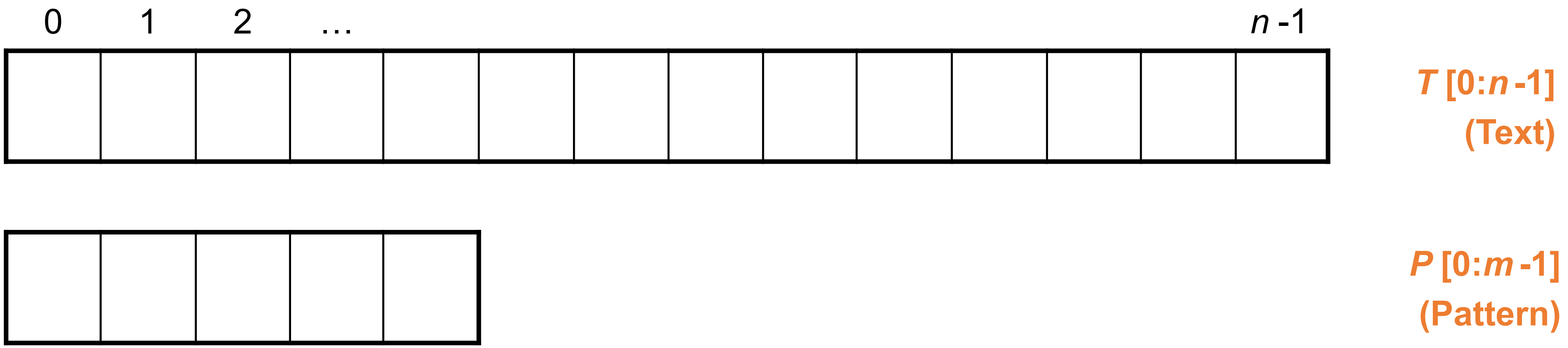
Definitions

- An **alphabet** is a finite set of «symbols» $A = \{a_1, a_2, \ldots, a_k\}$
- A **string** $S = S[0{:}\,n\text{-}1]$ or $S = \langle s_0 s_1 \ldots s_{n\text{-}1} \rangle$ of length n is a sequence of n symbols from A

String Search:

Given two strings T (= Text) and P (= Pattern), P is usually much shorter than T

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



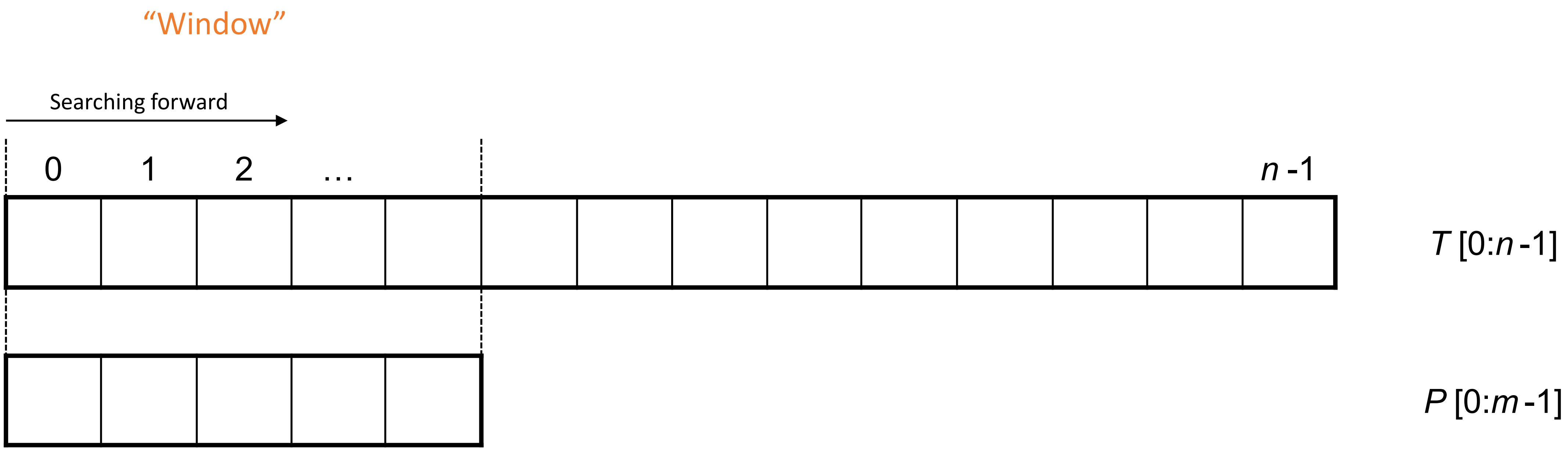
Variants of String Search

- Naive algorithm, no preprocessing of *T* or *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* $O(n^2)$

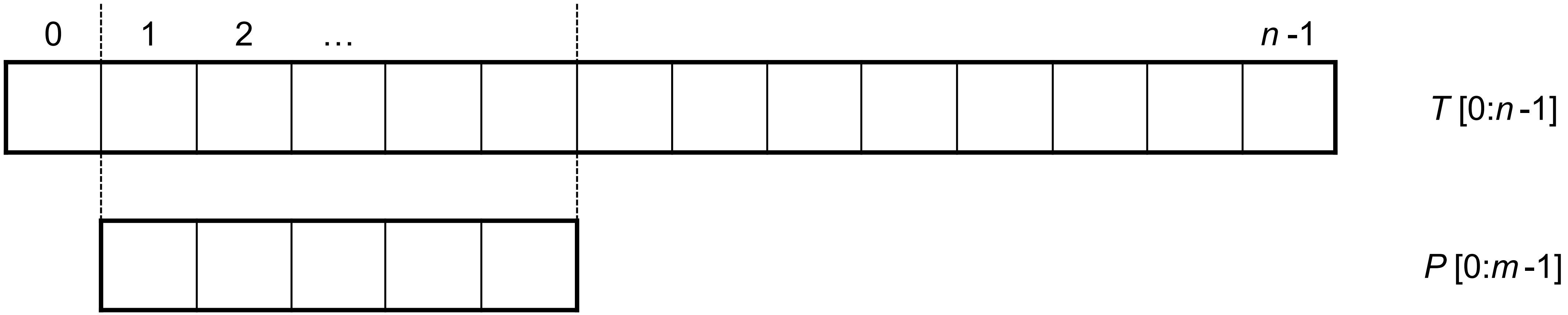
- Preprocessing of *P* (the pattern) for each new *P*
  - Prefix-search: The Knuth-Morris-Pratt algorithm
  - Suffix-search: The Boyer-Moore algorithm
  - Hash-based: The Karp-Rabin algorithm

- Preprocessing of the text *T*
  (Used when we search the same text a lot of times (with different patterns), done to an extreme degree in search engines)
  - Suffix trees: Data structure that relies on a structure called a Trie

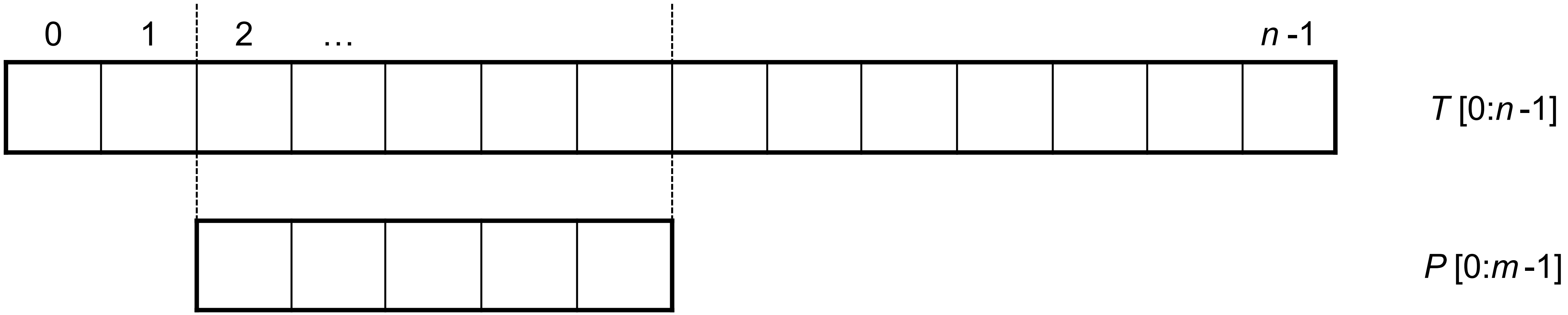
The naive algorithm (Prefix based)



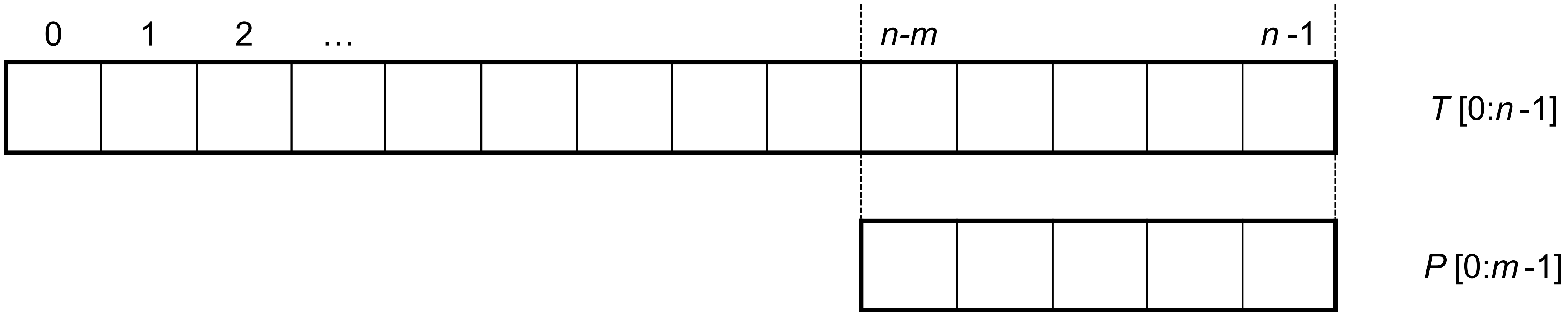
The naive algorithm



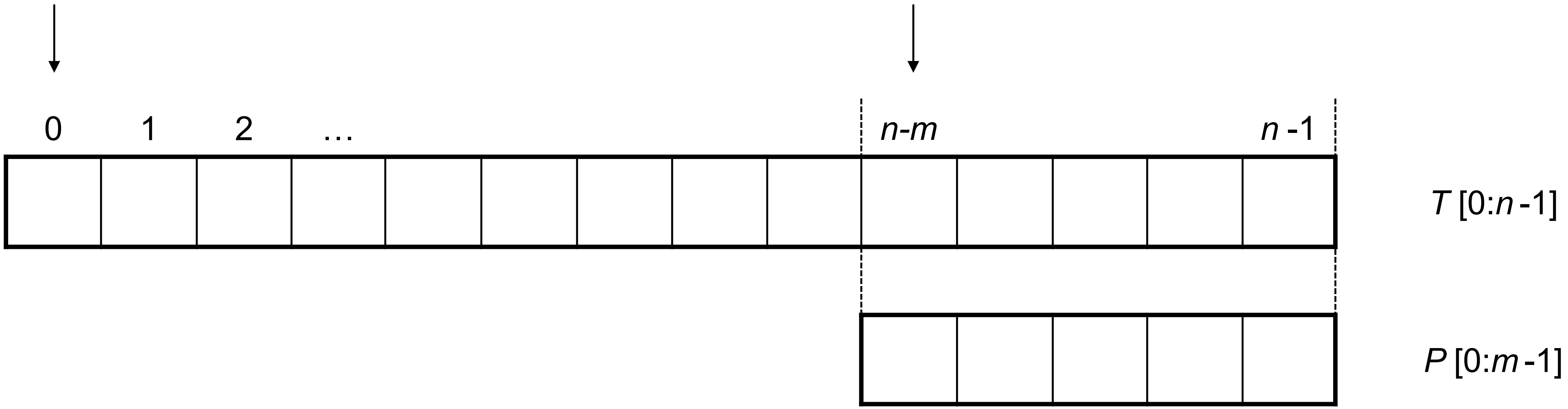
The naive algorithm



The naive algorithm

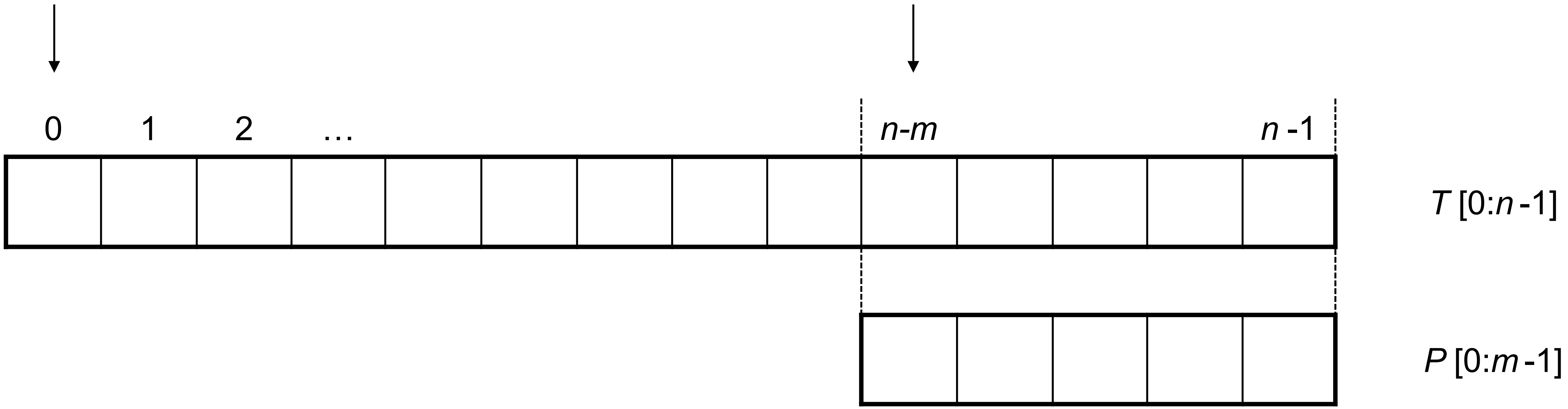


The naive algorithm




```
function NaiveStringMatcher (P [0:m -1], T [0:n -1])
   for s ← 0 to n - m do                  // loop through all window positions
         if T [s :s + m - 1] = P  then    // is window = P?
            return(s)                     // if so, return start-index  of window
         endif
   endfor
   return(-1)                             // no match found
end NaiveStringMatcher
```

The naive algorithm



function *NaiveStringMatcher* (*P* [0:*m* -1], *T* [0:*n* -1])

```
for s ← 0 to n - m do                // loop through all window positions
        if T [s :s + m - 1] = P  then    // is window = P?
            return(s)                    // if so, return start-index  of window
        endif
endfor
return(-1)                           // no match found
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 (Prefix based)

- There is room for improvement in the naive algorithm
  - The naive algorithm moves the window (pattern) only one character at a time
  - But we can move the window farther, based on what we know from earlier comparisons
    - USE WHAT WE KNOW, TO START FROM FIRST POSISTION WHERE A MATCH IS POSSIBLE
  - **How far can we move it?**

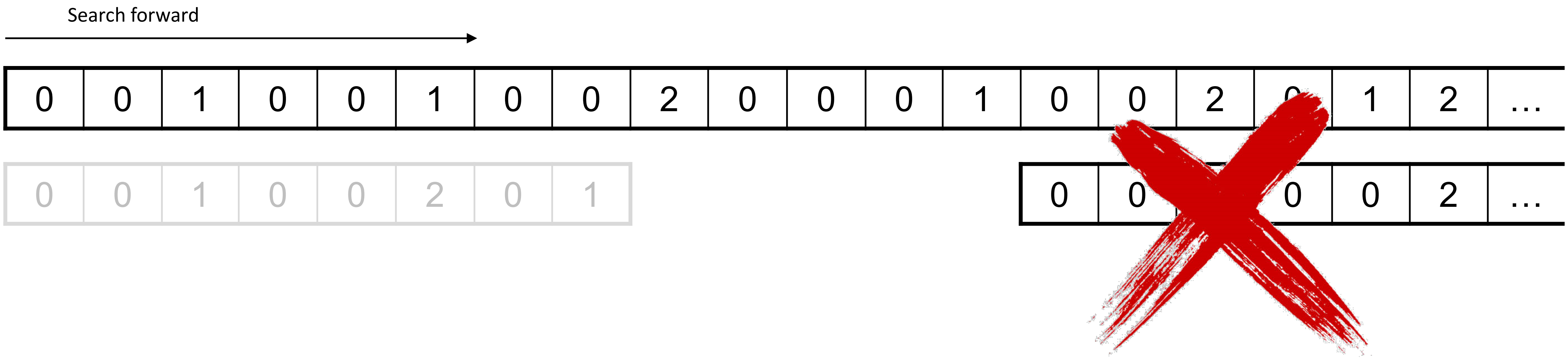
Search forward

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

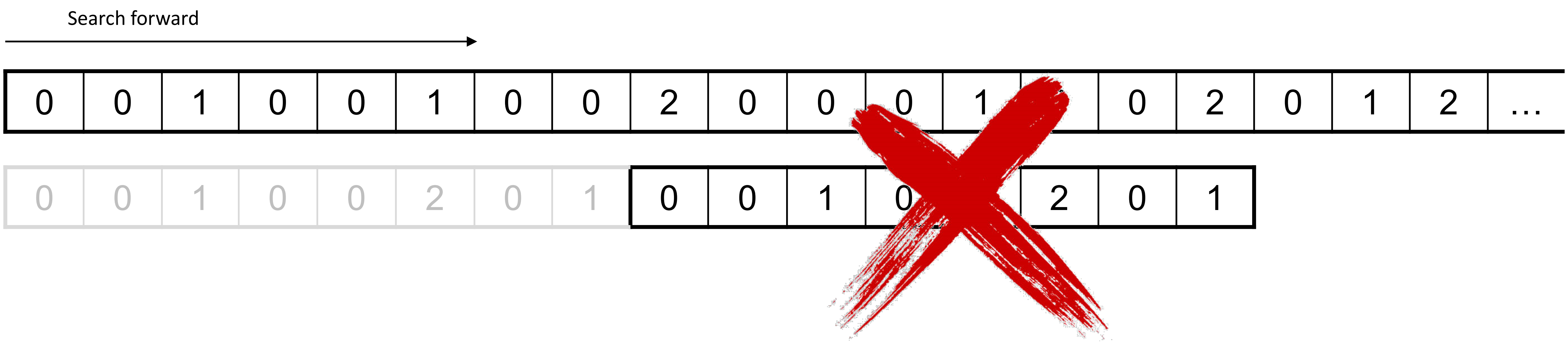
The Knuth-Morris-Pratt algorithm (Prefix based)

- There is room for improvement in the naive algorithm
  - The naive algorithm moves the window (pattern) only one character at a time
  - But we can move the window farther, based on what we know from earlier comparisons
    - USE WHAT WE KNOW, TO START FROM FIRST POSISTION WHERE A MATCH IS POSSIBLE
  - **How far can we move it?**



The Knuth-Morris-Pratt algorithm (Prefix based)

- There is room for improvement in the naive algorithm
  - The naive algorithm moves the window (pattern) only one character at a time
  - But we can move the window farther, based on what we know from earlier comparisons
    - USE WHAT WE KNOW, TO START FROM FIRST POSISTION WHERE A MATCH IS POSSIBLE
  - **How far can we move it?**



The Knuth-Morris-Pratt algorithm (Prefix based)

- There is room for improvement in the naive algorithm
  - The naive algorithm moves the window (pattern) only one character at a time
  - But we can move the window farther, based on what we know from earlier comparisons
    - USE WHAT WE KNOW, TO START FROM FIRST POSISTION WHERE A MATCH IS POSSIBLE
  - **How far can we move it?**

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 (Prefix based)

- There is room for improvement in the naive algorithm
  - The naive algorithm moves the window (pattern) only one character at a time
  - But we can move the window farther, based on what we know from earlier comparisons
    - USE WHAT WE KNOW, TO START FROM FIRST POSISTION WHERE A MATCH IS POSSIBLE
  - **How far can we move it?**

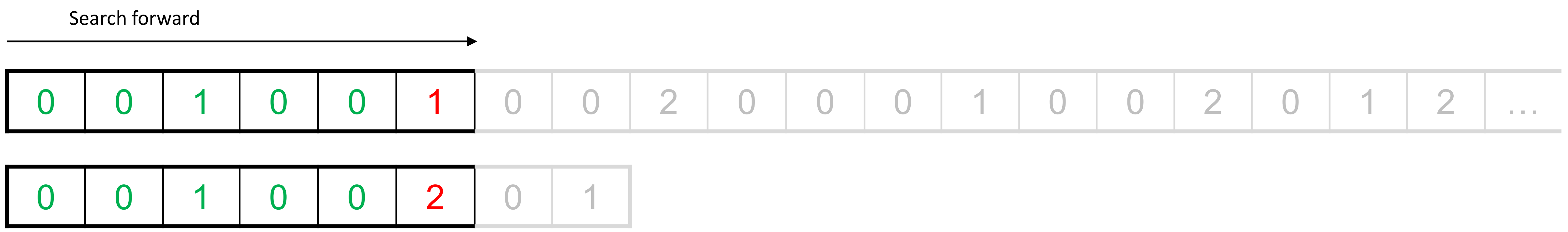
Search forward →

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

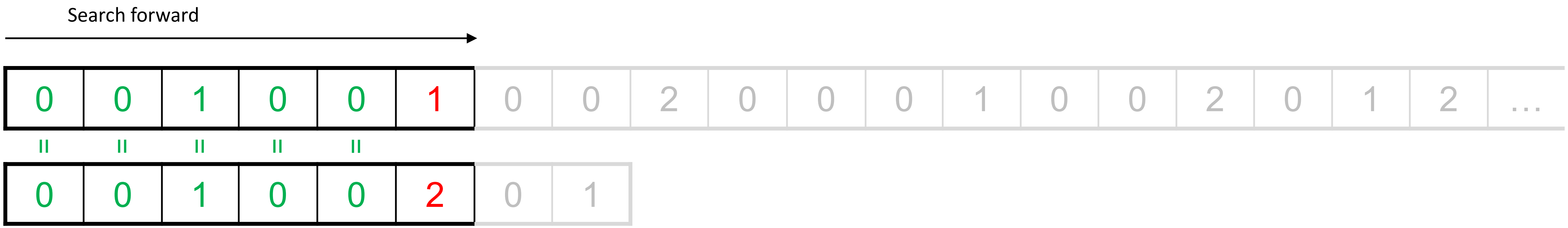
The Knuth-Morris-Pratt algorithm (Prefix based)

- There is room for improvement in the naive algorithm
  - The naive algorithm moves the window (pattern) only one character at a time
  - But we can move the window farther, based on what we know from earlier comparisons
    - USE WHAT WE KNOW, TO START FROM FIRST POSISTION WHERE A MATCH IS POSSIBLE
  - **How far can we move it?**



The Knuth-Morris-Pratt algorithm (Prefix based)

- There is room for improvement in the naive algorithm
  - The naive algorithm moves the window (pattern) only one character at a time
  - But we can move the window farther, based on what we know from earlier comparisons
    - USE WHAT WE KNOW, TO START FROM FIRST POSISTION WHERE A MATCH IS POSSIBLE
  - **How far can we move it?**



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

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											

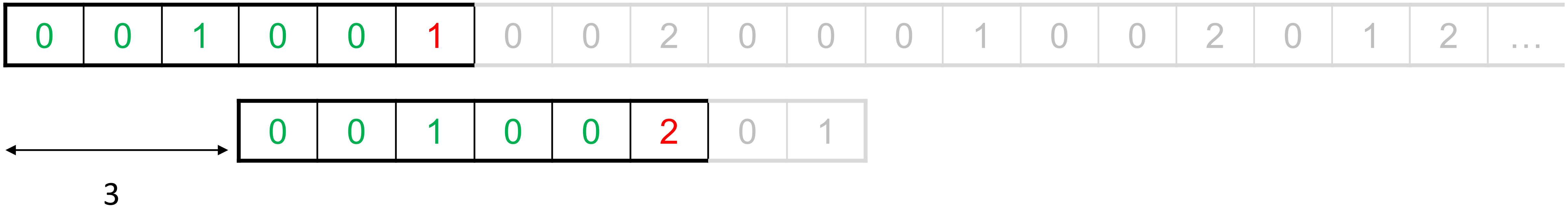
We move the pattern one step: Mismatch (in the second symbol)
(We have to move at least one step…)

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										

We move the pattern two steps: Mismatch (in the first symbol)

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 previously

The Knuth-Morris-Pratt algorithm

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

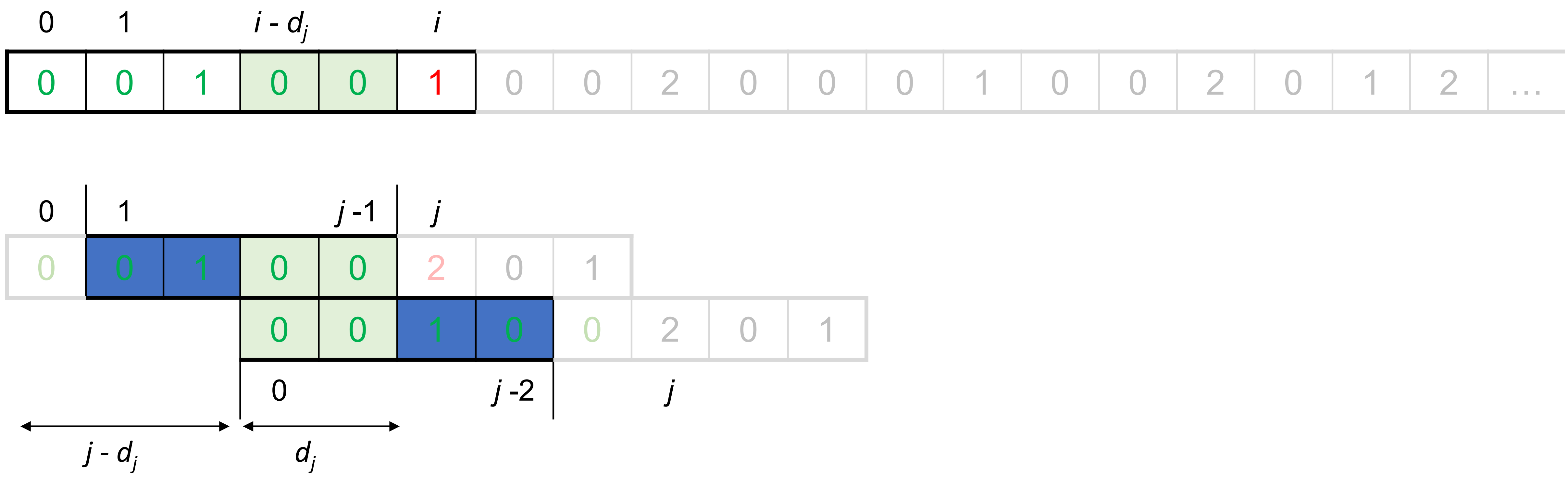
			0	0	1	0	0	2	0	1	

3

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

- We can skip a number of tests and move the pattern **more than one step** before we start comparing characters again. (3 in the above situation.)
- The key is that **we know what the characters of *T* and *P* are, up to the point where *P* and *T* got different.** **(*T* and *P* are equal up to this point.)**
- 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. **(We only need to look at P for this!)**
- It may well be that we never get an overlap 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, $j = 0,1,2, \ldots , m-1$

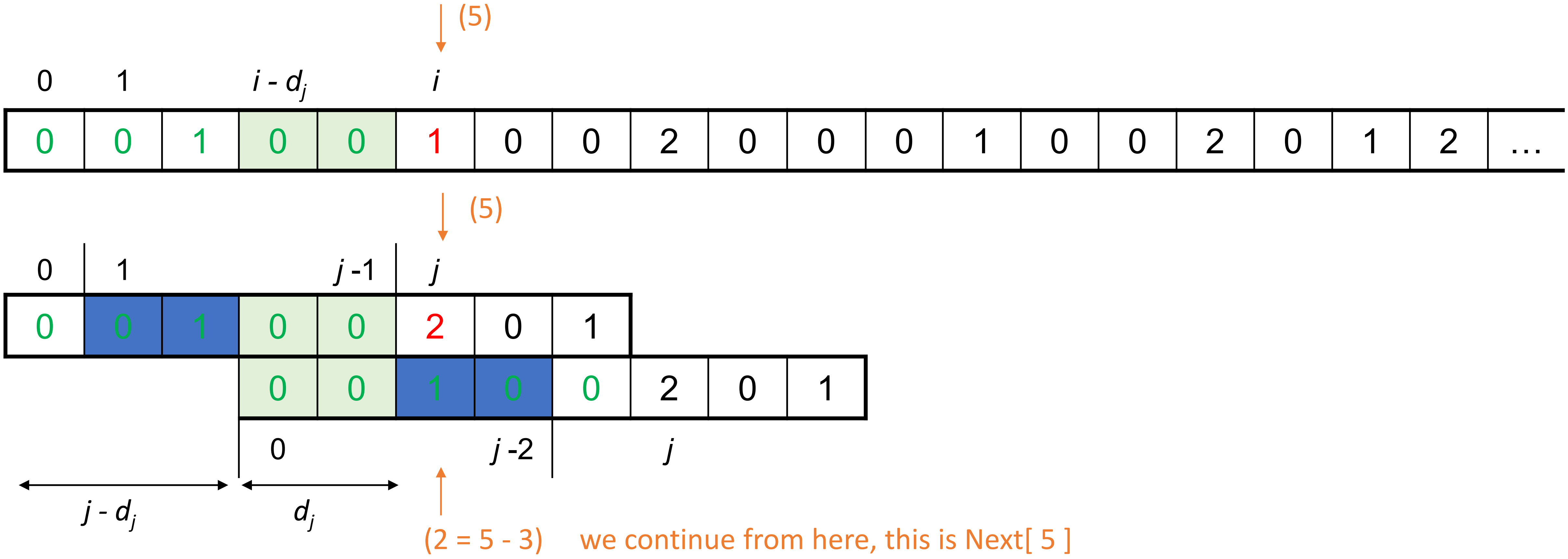
- But the array *Next* will not give this number directly. Instead, *Next* $[\,j\,]$ will contain the new (and smaller value) that j should have when we resume the search after a mismatch at j in P (see below)
  - That is: *Next* $[\,j\,] = j -$ <number of steps that P should be moved>
  - or: *Next* $[\,j\,]$ *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

- **The array *Next*[] can be computed from *P* alone!**

The Knuth-Morris-Pratt algorithm



(From index 5, starting again from 2 is the same as moving the pattern 3 steps)

```
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])          // preprocessing of the pattern P
    while i < n do                               // loop until we have a full match, or get to the end of T
        if P [ j ] = T [ i ] then                // if the symbols match, we can continue looking for a full match
            if j = m –1 then                     // check if match is full
                return(i – m + 1)                // if so, return start-index of the (full) match
            endif
            i ← i +1                             // if match is not full, check next symbol
            j ← j +1
        else                                     // if the symbols did not match, we must move the window
            j ← Next [ j ]                       // move window by decreasing j – implicit shift according to the preprosessing
            if j = 0 then                        // if j then becomes 0 (it can not be decreased any more)
                if T [ i ] ≠ P [0] then          // and symbols do not match
                    i ← i +1                     // move window by increasing i – explicit shift
                endif
            endif
        endif
    endwhile
    return(-1)                                   // no match found
end KMPStringMatcher
```

$O(n)$

Calculating the array Next[] from P

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 $O(m^2)$
- A more clever approach finds the array *Next* in time $O(m)$
- We will look at the procedure in 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	1	1	**2**	0	1

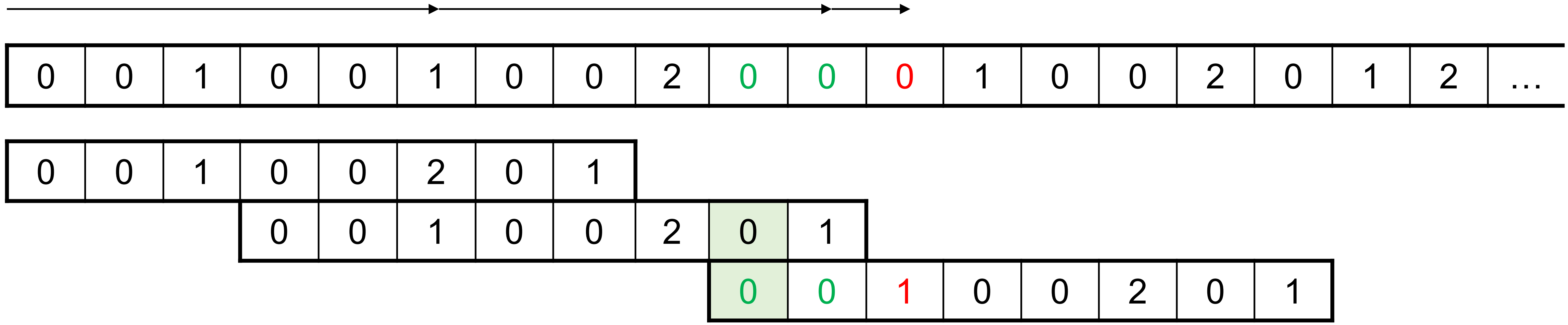
The Knuth-Morris-Pratt algorithm, example

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	1	1	2	0	**1**

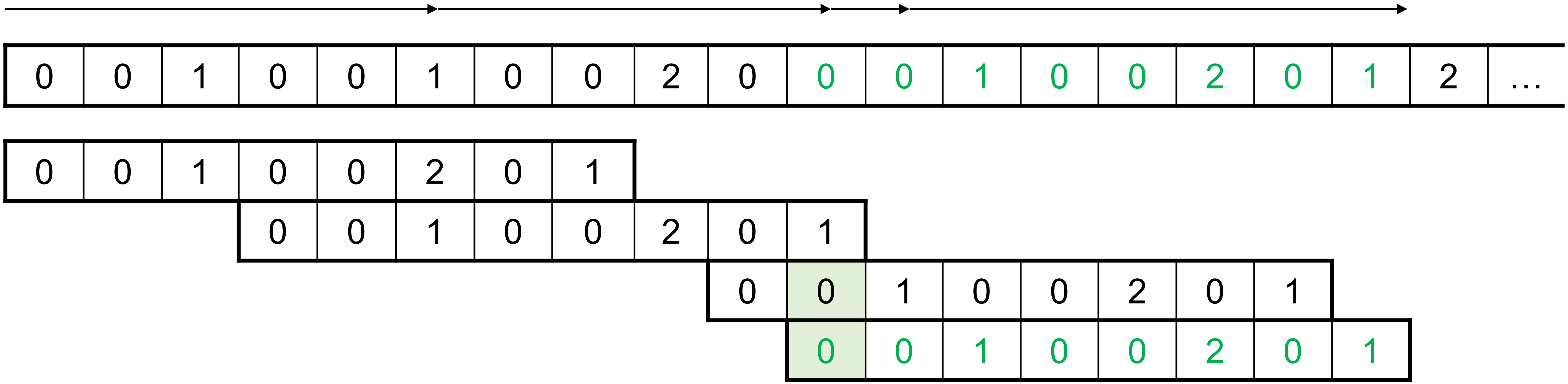
The Knuth-Morris-Pratt algorithm, example



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

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

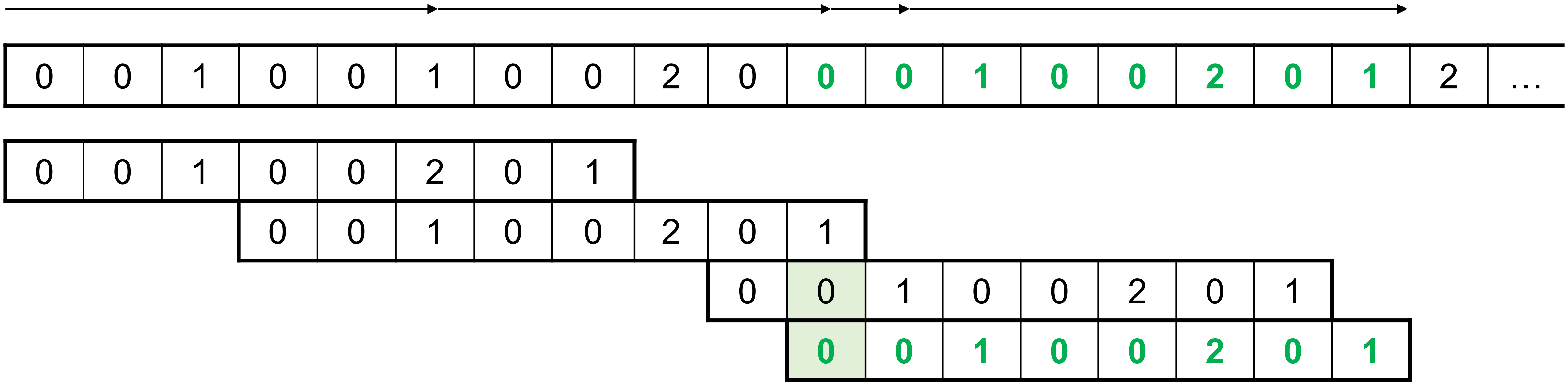
The Knuth-Morris-Pratt algorithm, example



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

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

The Knuth-Morris-Pratt algorithm, example



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

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

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

The Boyer-Moore algorithm (Suffix based)

- 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

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

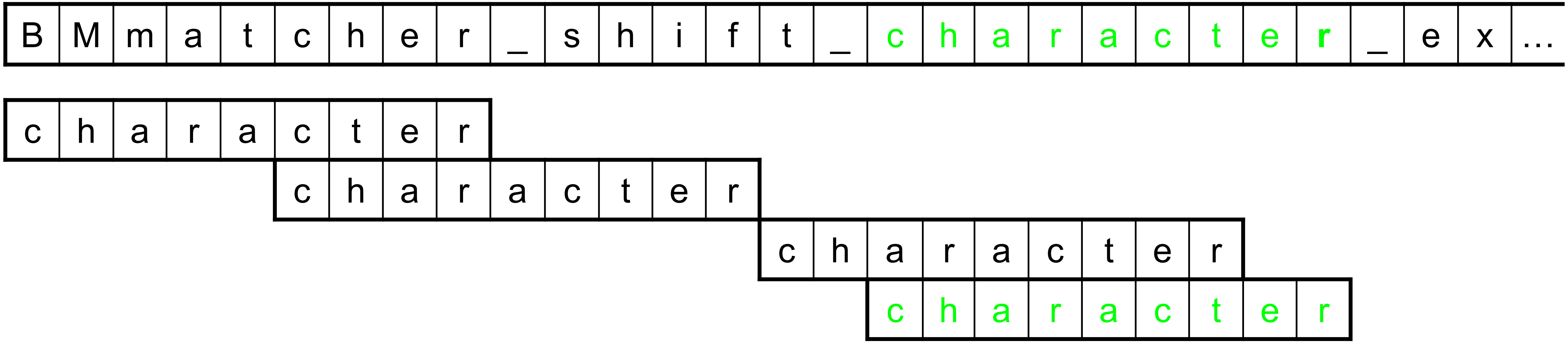
The Boyer-Moore algorithm (Horspool)

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															

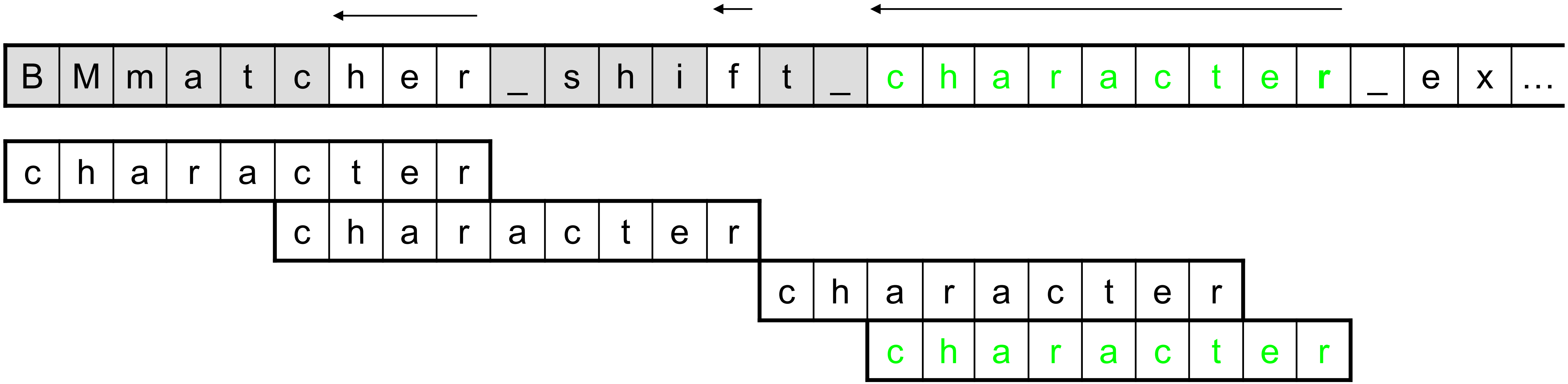
The Boyer-Moore algorithm (Horspool)

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)



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])            // preprocessing of the pattern P
   while i < n – m do                                     // loop through all window positions (from left)
         j ← m – 1
         while j ≥ 0 and T [ i + j ] = P [ j ] do         // compare window and pattern (from right)
               j ← j -1
         endwhile
         if j = 0 then                                    // if we have a full match,
               return( i )                                // return start-index of window
         endif
         i ← i + Shift[ T[ i + m -1] ]                    // if not, move pattern to the right, and align
         endwhile                                         // according to the last symbol of the window
   return(-1)
end HorspoolStringMatcher
```

Calculating the array Shift[] from P

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

...

end *CreateShift*

- We must preprocess *P* to find the array *Shift*
- The size 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

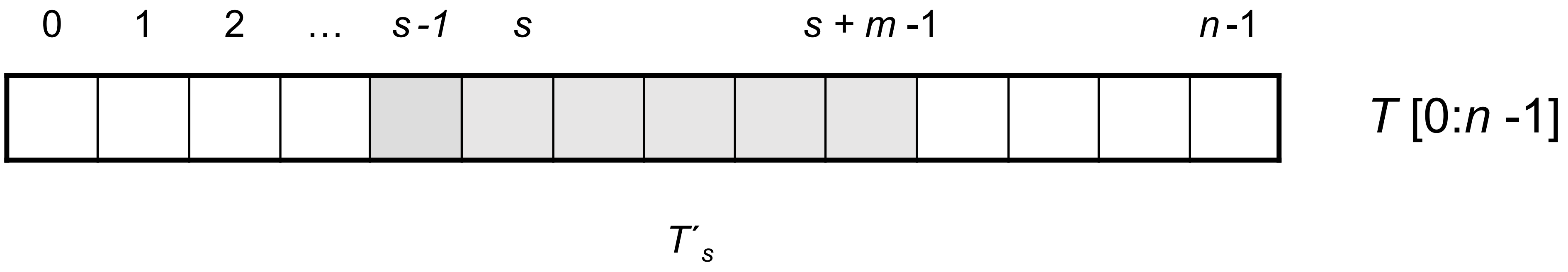
- Given a string $P[0: m-1]$. We can then calculate the corresponding number P' using $m - 1$ multiplications and $m - 1$ additions (Horners rule, computed from the innermost right expression and outwards):

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

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 compare 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

- We can compute T'_s in constant time when we know T'_{s-1} and k^{m-1}
- We can therefore compute
  - P' and
  - T'_s, $s = 0, 1, \ldots, n - m$ ($n - m + 1$ numbers)

  in time $O(n)$
- We can threfore "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 – back to the naive algorithm again)
- 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. 64 bits)
- A prime number is chosen as this will distribute the values well

The Karp-Rabin algorithm

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

  $$\left.\begin{aligned} T'^{(q)}_s &= T'_s \bmod q, \\ P'^{(q)} &= P' \bmod q, \textit{(only once)} \end{aligned}\right\}$$

  x mod *y* is the remainder when deviding *x* with *y*, this is always in the interval $\{0, 1, \ldots, y-1\}$.

  and compare

- 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 ← k^(m -1) mod q                                        // initialize
    P´(q) ← 0
    T´(q)_s ← 0

    for i ← 1 to m do
            P´(q) ← (k * P´(q) + P [ i ]) mod q                // calculate value for P
            T´(q)_0 ← (k * T´(q)_0 + T [ i ]) mod q            // and first position of window
    endfor

    for s ← 0 to n - m do                                     // loop through all positions for the window
            if s > 0 then                                     // calculate value for the (new) window
                    T´(q)_s ← (k * ( T´(q)_(s -1) - T [ s ] * c) + T [ s + m ]) mod q   // (based on previous window)
            endif
            if T´(q)_s = P´(q) then                           // if we have a match mod q,
                    if T_s = P then                           // then we must check the actual strings
                            return(s)                         // and return the start-index
                    endif
            endif
    endfor
    return(-1)                                                // no match found
    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 $P'^{(q)}$ (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)

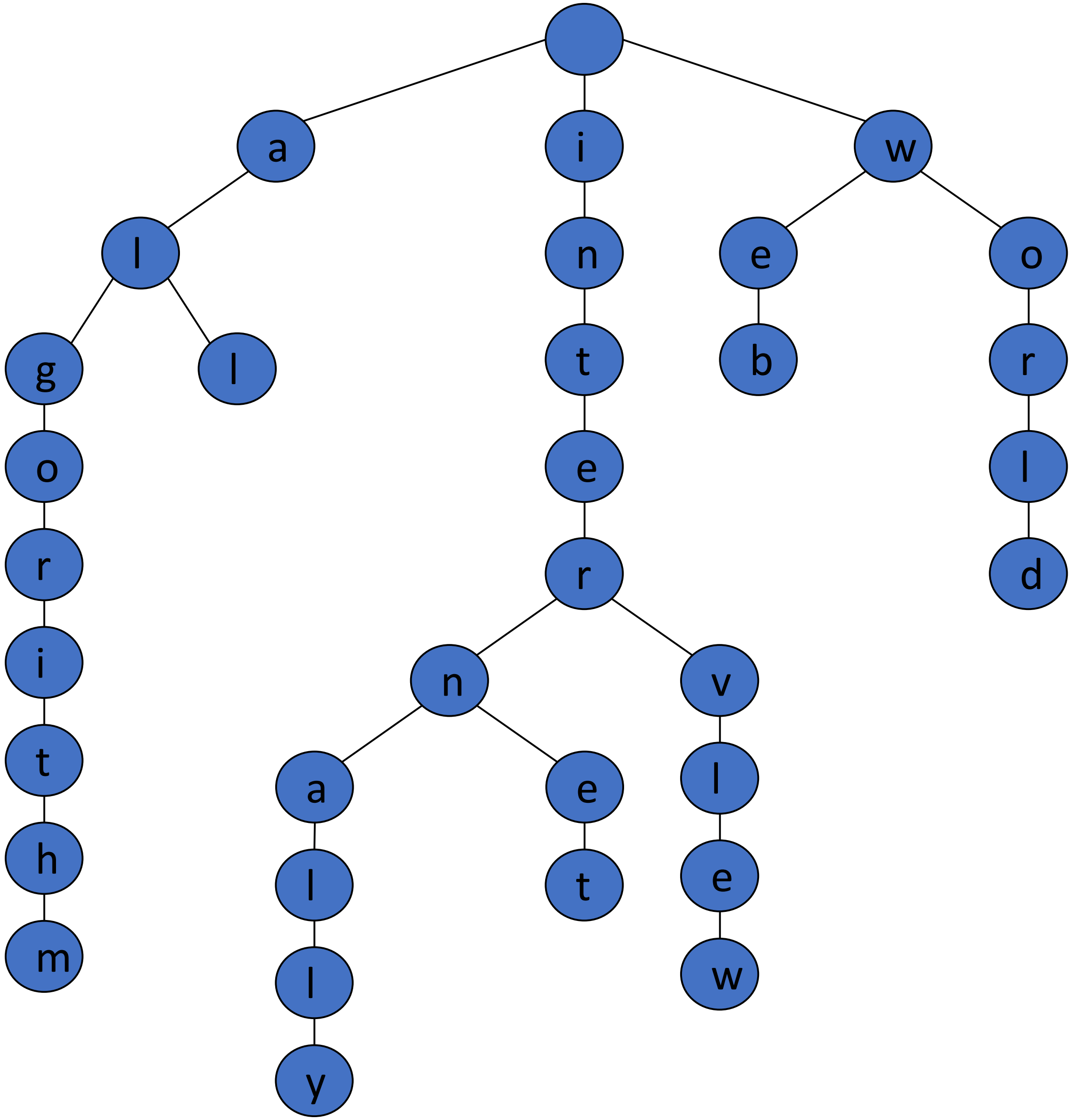
- It is then usually smart to *preprocess T*, so that later searches in *T* for different patterns *P* will be fast
  - Search engines (like Google or Bing) do this in a very clever way, so that searches in huge number of web-pages can be done extremely fast

- 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

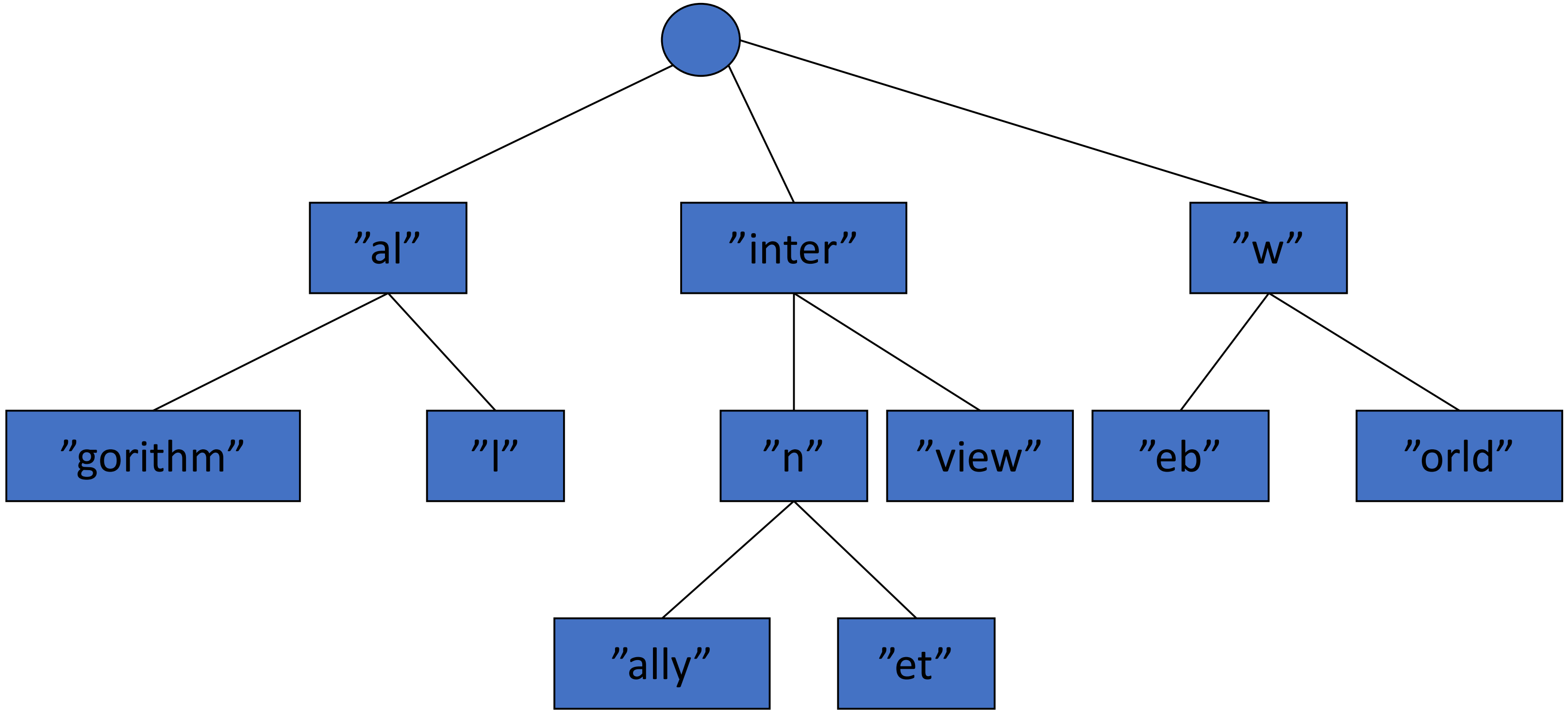
- T may also gradually change over time. We then have to update the index for each such change
  - The index of a search engine is updated when the crawler finds a new web page

Tries (word play on **Tree/Retrieval**)

There is a small error in the textbook here

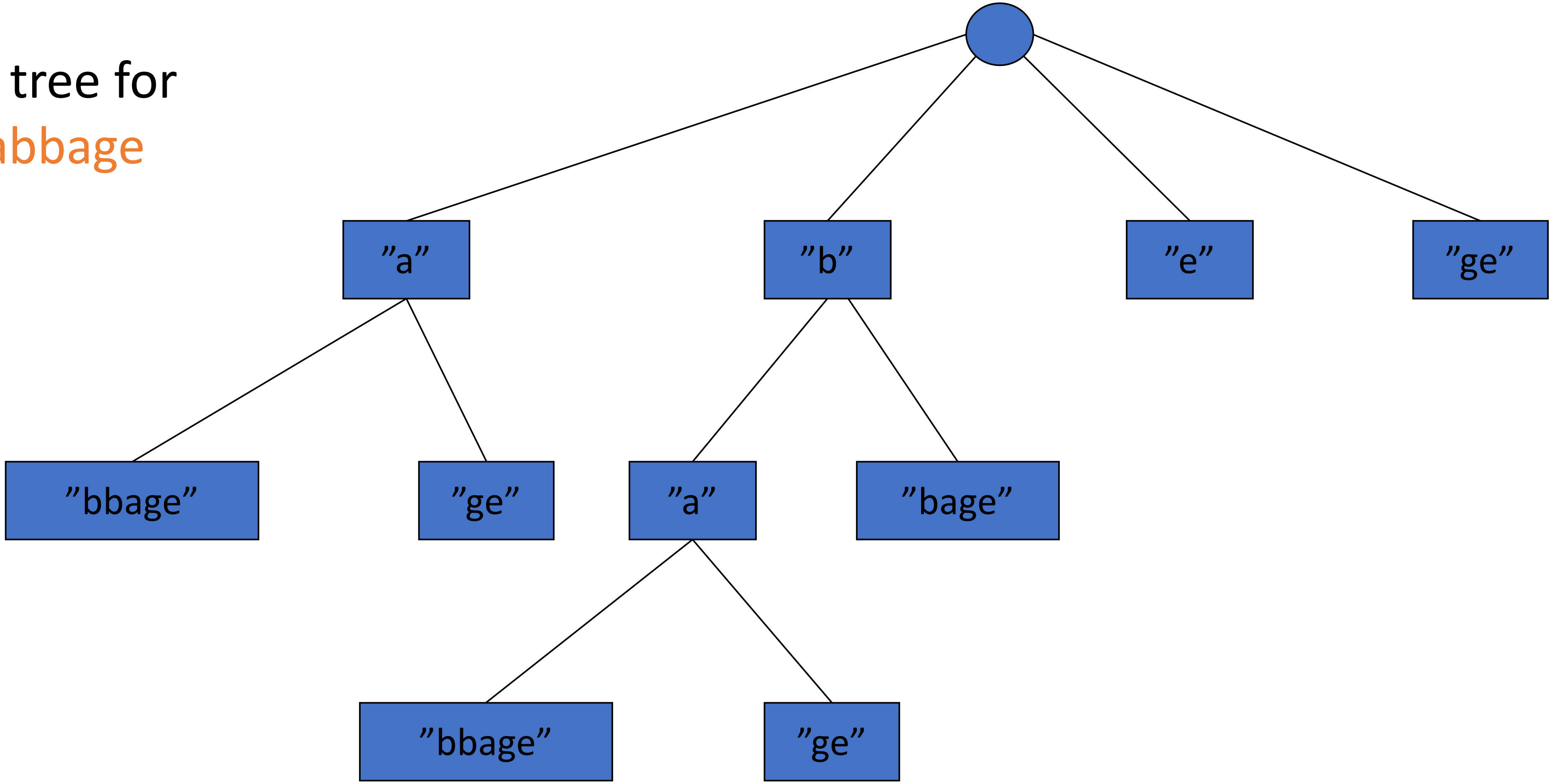


Compressed trie

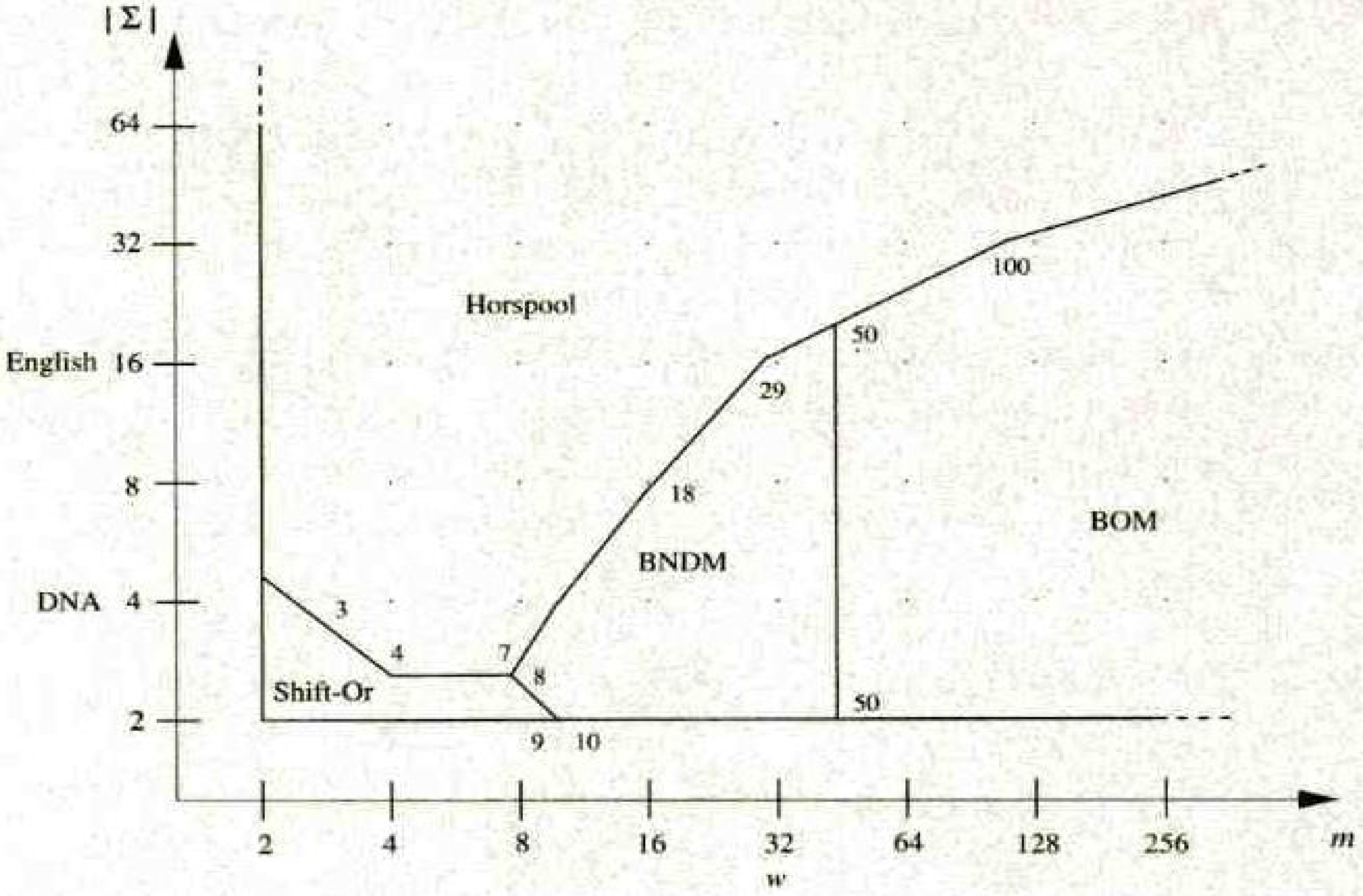


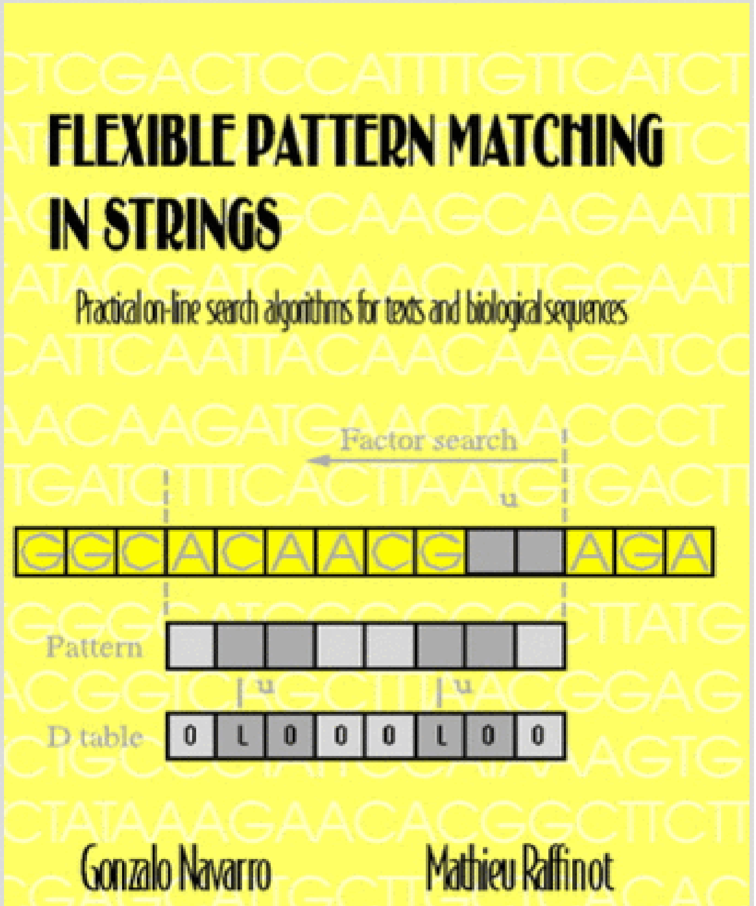
Suffix trees (compressed)

Suffix tree for
T = babbage



- Looking for P in this trie will decide whether P occurs as a substring of T, all substrings have a path strting in the root


|Σ|
64
32
English 16
8
DNA 4
2
Horspool
BNDM
BOM
Shift-Or
3
4
7
8
9
10
18
29
50
50
100
2
4
8
16
32
64
128
256
m
w


FLEXIBLE PATTERN MATCHING IN STRINGS
Practical on-line search algorithms for texts and biological sequences
Factor search
u
Pattern
D table
Gonzalo Navarro
Mathieu Raffinot