Applied Databases

Lecture 13
KMP, Boyer-Moore, and Horspool Algorithms

Sebastian Maneth

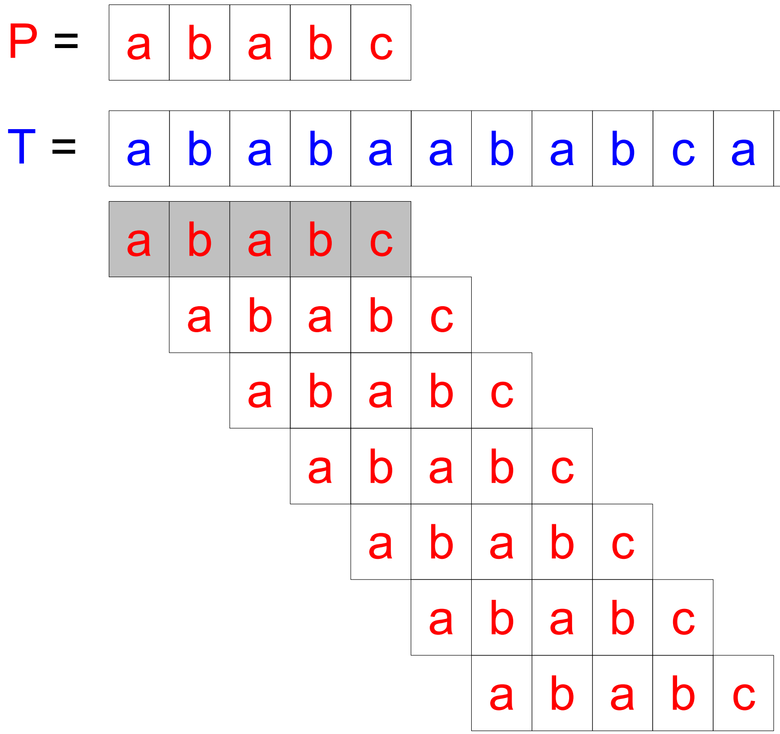
University of Edinburgh - March 6th, 2017

Outline

1. Morris-Pratt Algorithm
2. KMP
3. Boyer-Moore
4. Horspool

Recap: Naive Method

Given Pattern P (length m) Text T (length n) find all occurrences of P in T.

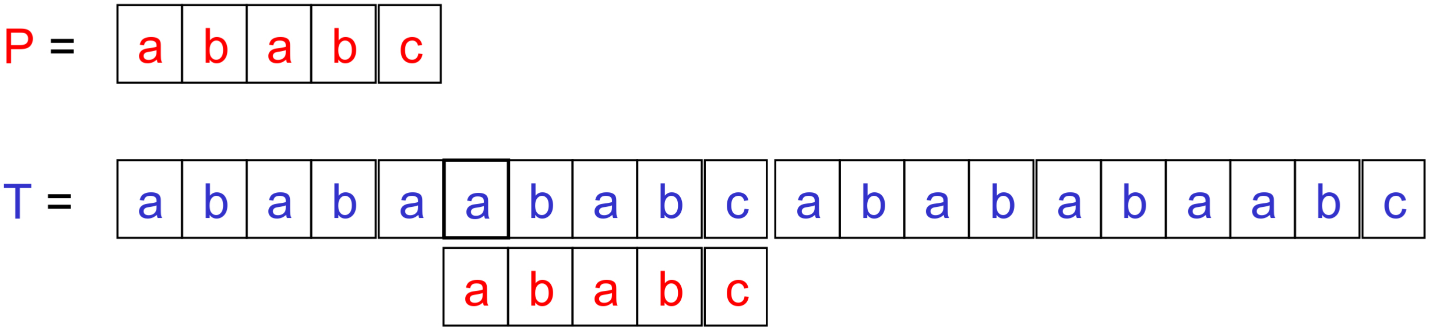


(1) **Naive method**

→ shift always by **one**
→ worst-case complexity
$m(n - m + 1)$
i.e., in $O(mn)$

Recap: Naive Method

Given Pattern P, Text T, find all occurrences of P in T.



P = a b a b c

T = a b a b a a b a b c a b a b a b a a b c

a b a b c

Best-Case O(n)

Average-Case → show that if P and T are *randomly chosen* from an alphabet with d letters, then #character-comparisons of Naive Algorithm is:

$$(n - m + 1)\frac{1 - d^{-m}}{1 - d^{-1}} <= 2(n - m + 1)$$

Questions

Best-Case Complexity?

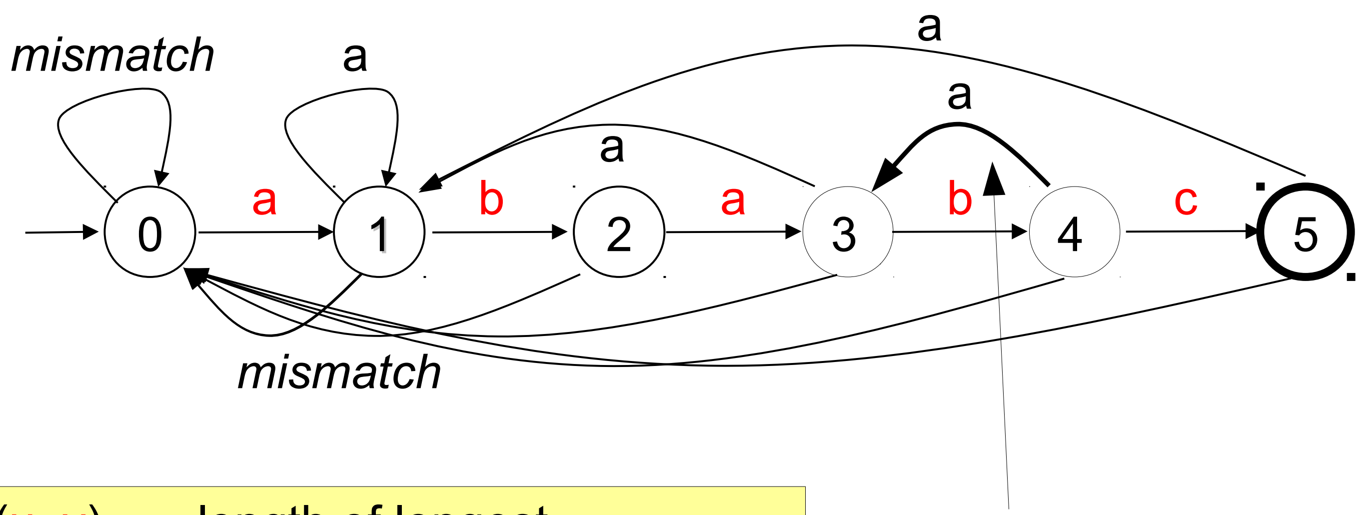
Average-Case Complexity?
(on random strings)

Recap: Automaton Method

Given Pattern P (length m) Text T (length n) find all occurrences of P in T.

P =	a	b	a	b	c

T =	a	b	a	b	a	a	b	a	b	c	a	b	a	b	a	b	a	a	b	c



lospre(u, v) = length of longest **proper suffix** of u that is **prefix** of v

to state
lospre(ababa, P) = |aba| = 3

Recap: Automaton Method

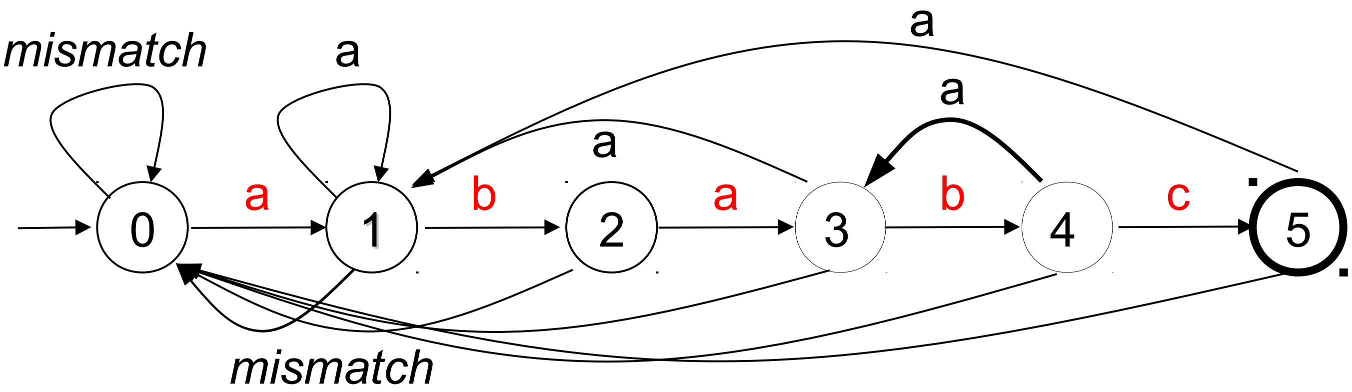
Given Pattern P (length m) Text T (length n) find all occurrences of P in T.

P =

a	b	a	b	c

T =

a	b	a	b	a	a	b	a	b	c	a	b	a	b	a	b	a	a	b	c



lospre(u, v) = length of longest
proper suffix of u that is **prefix** of v

Automaton method

→ O(m|**S**|) size & time to build
→ O(n) matching time

|**S**| = size of alphabet of P
→ for automata *wo mismatch-transitions*, |**S**| = size of alphabet of T [large!!]

1. Morris-Pratt Algorithm

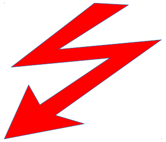
Brief History:

→ 1970: James H. Morris built a text editor for the CDC 6400 computer

→ with Vaughan Pratt, developed "A linear pattern matching algorithm"
[Report 40, University of California, Berkely, 1970]

→ Matching time: O(n + m)

→ rigorous analysis (Knuth) revealed: delay at a character can be O(m)



→ Knuth added one check to Morris&Pratt's conditions, and was then able to prove *logarithmic delay (tight bound)*

→ #character comparisons is <= 2n–1

1. Morris-Pratt Algorithm

	1	2	3	4	5
P =	a	b	a	a	b
	0	0	1	1	2

MP[k] = **lospre**(P[1..k], P)

MP[3] = length of longest **proper suffix** of aba, that is **prefix** of abaab
= |a| = 1

MP[5] = length of longest **proper suffix** of abaab, that is **prefix** of abaab
= |ab| = 2

Note: a *suffix* of a string w is **proper**, if it is *strictly shorter* than w.

1. Morris-Pratt Algorithm

	1	2	3	4	5
P =	a	b	a	a	b
	0	0	1	1	2

MP[k] = **lospre**(P[1..k], P)

MP[3] = |a| = 1
MP[5] = |ab| = 2

	0																			
T =	a	b	b	a	b	a	a	a	b	a	a	b	a	b	a	b	a	a	b	c
	a	b	a	a	b															

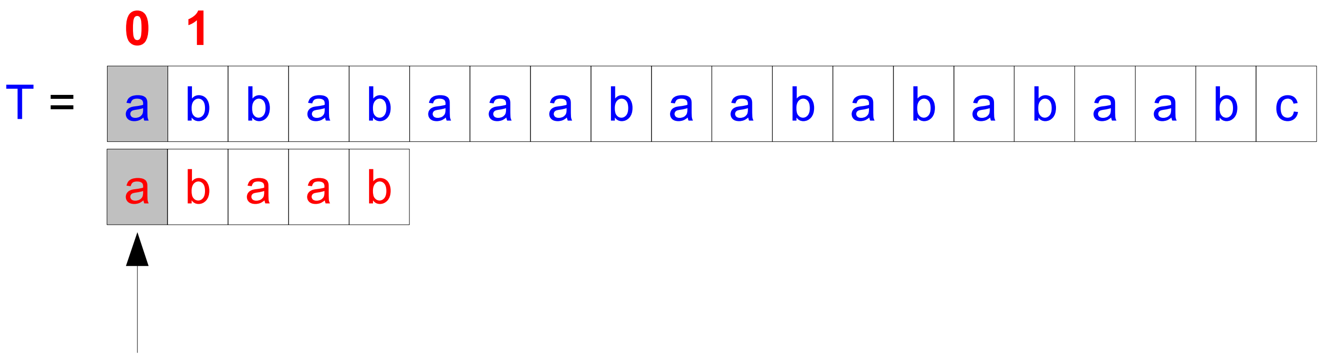
Start matching with
“**current lospre** = **0**”

1. Morris-Pratt Algorithm

	1	2	3	4	5
P =	a	b	a	a	b
	0	0	1	1	2

MP[k] = **lospre**(P[1..k], P)

MP[3] = |a| = 1
MP[5] = |ab| = 2



	0	**1**																		
T =	a	b	b	a	b	a	a	a	b	a	a	b	a	b	a	b	a	a	b	c
	a	b	a	a	b															

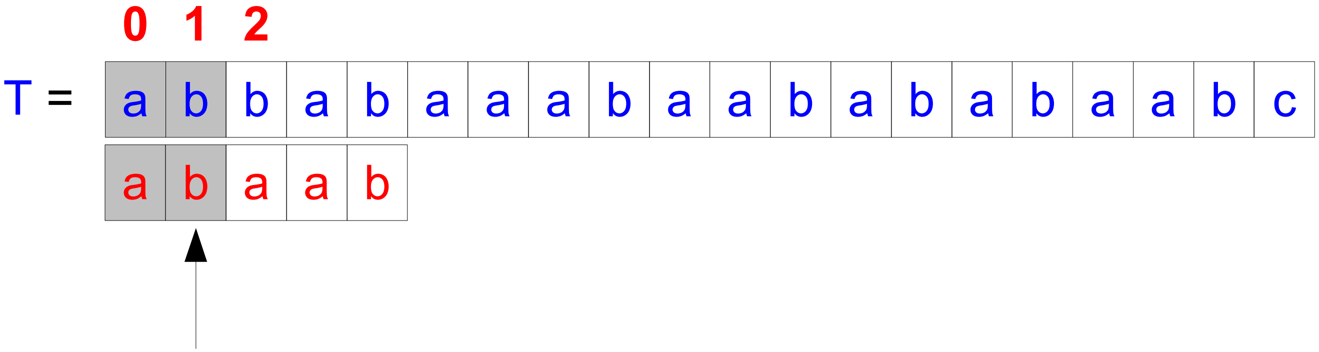
Match!
→ increase **lospre** by **1**

1. Morris-Pratt Algorithm

	1	2	3	4	5
P =	a	b	a	a	b
	0	0	1	1	2

MP[k] = **lospre**(P[1..k], P)

MP[3] = |a| = 1
MP[5] = |ab| = 2



Match!
→ increase **lospre** by **1**

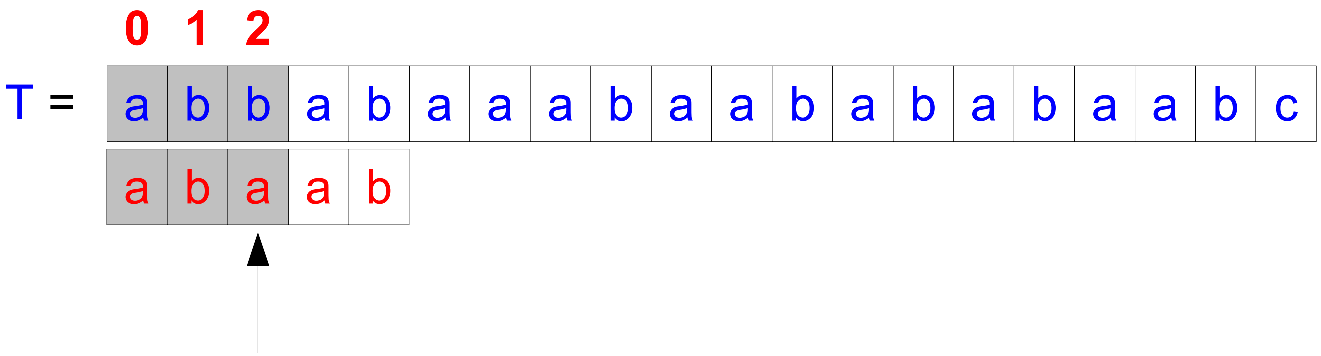
1. Morris-Pratt Algorithm

P =

1	**2**	3	4	5
a	b	a	a	b
0	**0**	1	1	2

MP[k] = **lospre**(P[1..k], P)

MP[3] = |a| = 1
MP[5] = |ab| = 2



T =

0	**1**	**2**																	
a	b	b	a	b	a	a	a	b	a	a	b	a	b	a	b	a	a	b	c
a	b	a	a	b															

Mismatch!
→ set current **lospre** to MP[**2**] = **0**

→ continue matching at *same position* (do **not** advance)

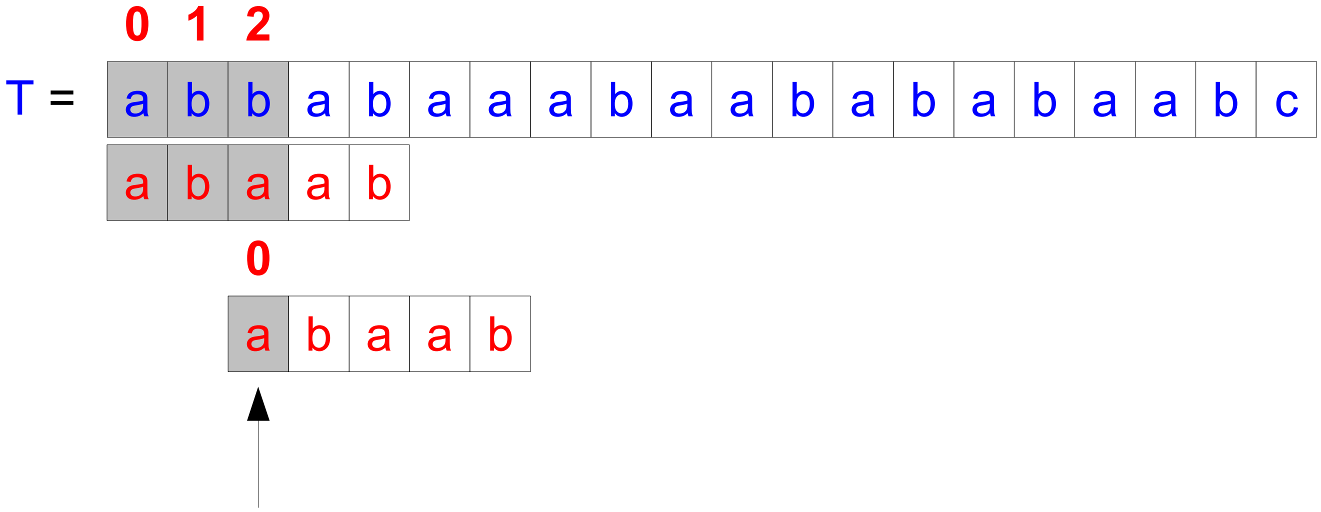
1. Morris-Pratt Algorithm

P =

1	2	3	4	5
a	b	a	a	b
0	0	1	1	2

MP[k] = **lospre**(P[1..k], P)

MP[3] = |a| = 1
MP[5] = |ab| = 2



	0	**1**	**2**																	
T =	a	b	b	a	b	a	a	a	b	a	a	b	a	b	a	b	a	a	b	c
	a	b	a	a	b															
			0																	
			a	b	a	a	b													

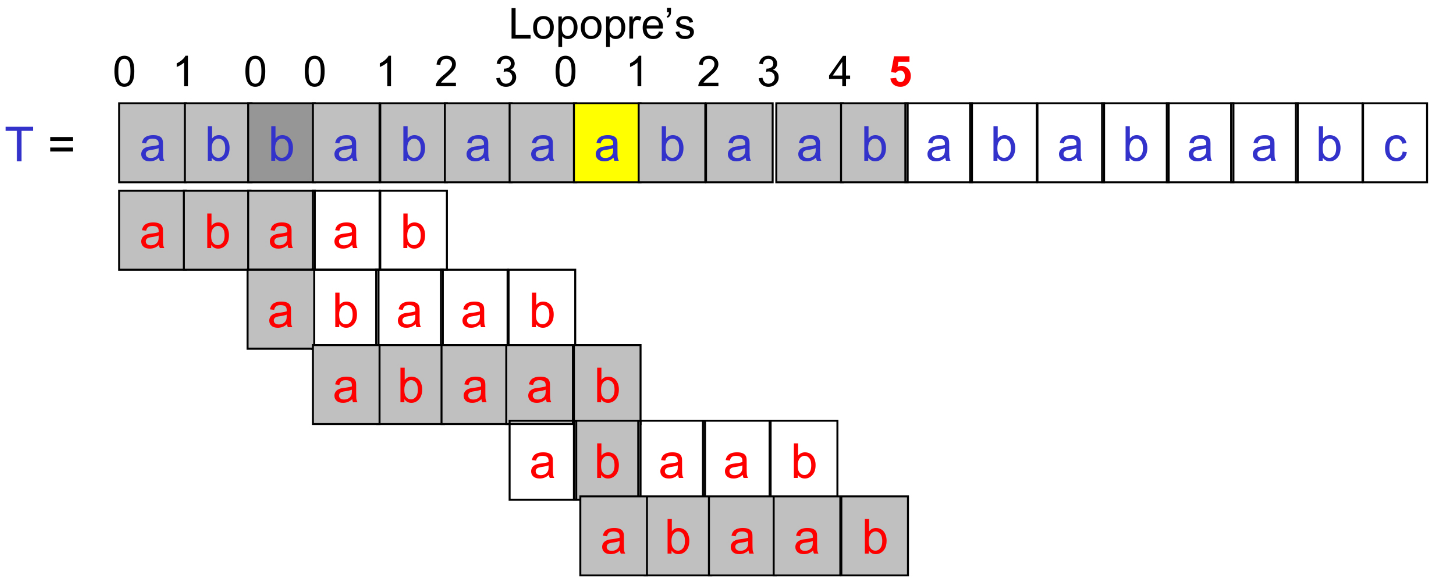
Mismatch!
Since **lospre=0**, advance one letter to the right
(and leave **lospre=0**)

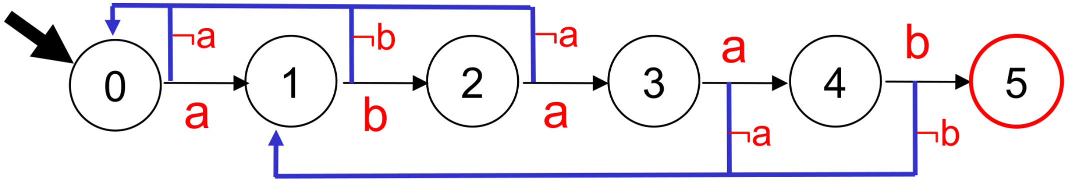
1. Morris-Pratt (1970)

Given Pattern P, Text T, find all occurrences of P in T.

	1	2	3	4	5
P =	a	b	a	a	b
	0	0	1	1	2

will never be used





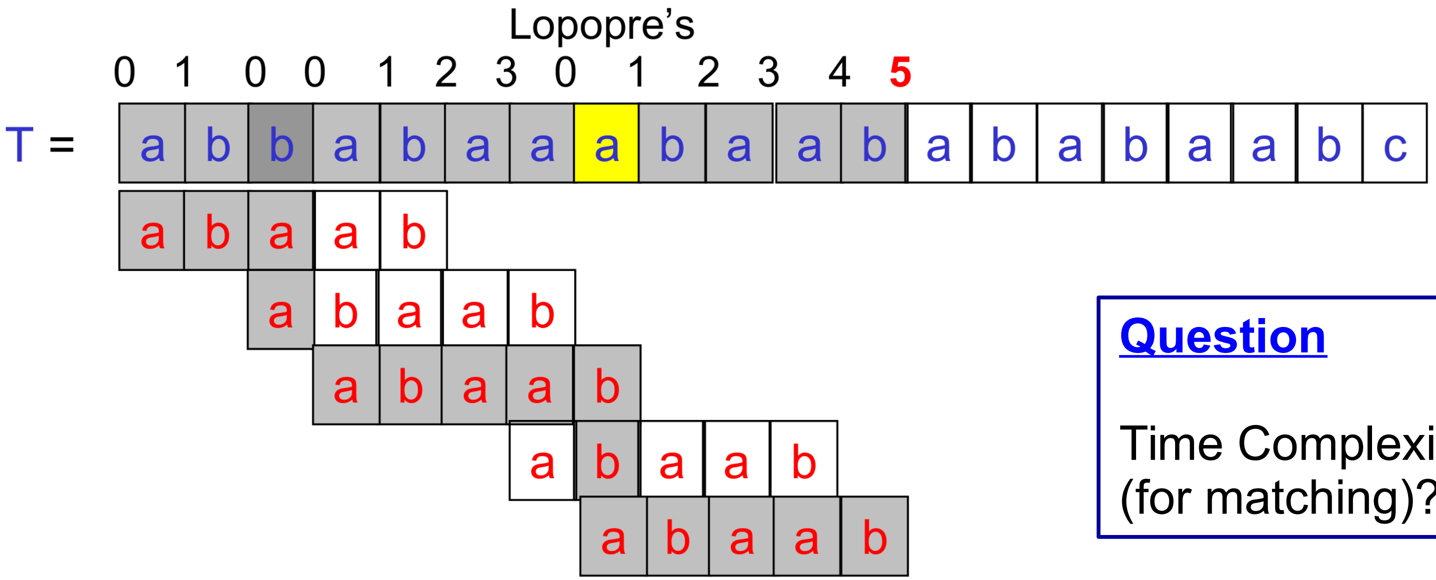
Blue arrow =
read current symbol **again**

1. Morris-Pratt (1970)

Given Pattern P, Text T, find all occurrences of P in T.

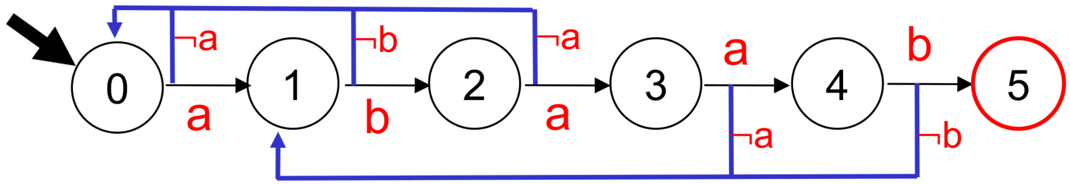
	1	2	3	4	5
P =	a	b	a	a	b
	0	0	1	1	🟩 2

will never be used



Question

Time Complexity (for matching)?



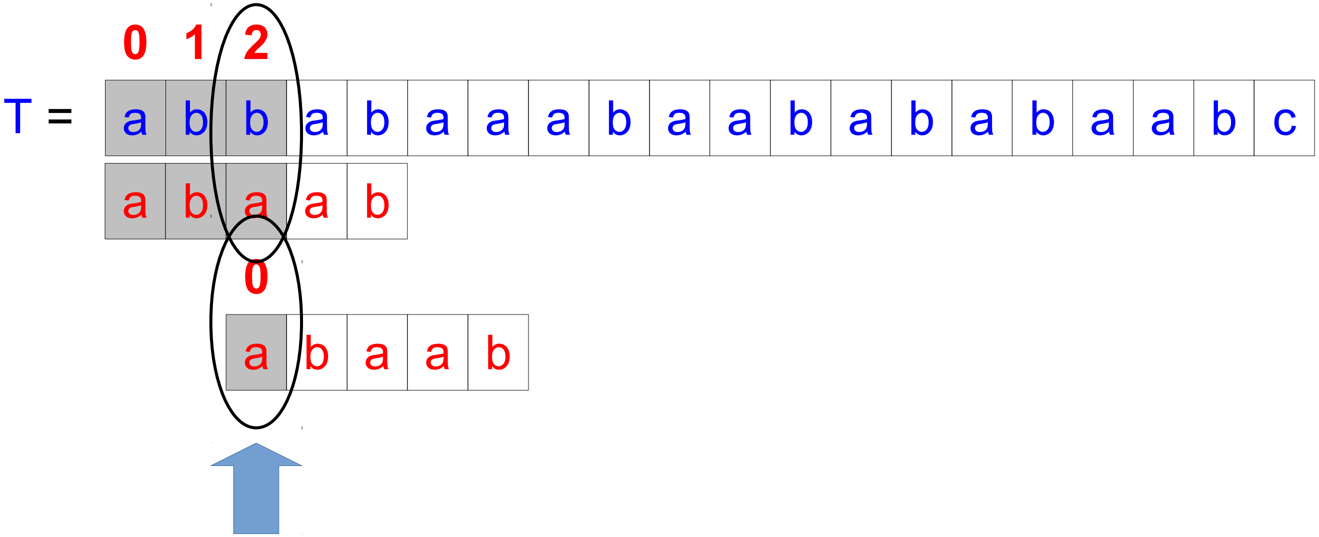
Blue arrow =
read current symbol **again**

Delay

	1	2	3	4	5
P =	a	b	a	a	b
	0	0	1	1	2

MP[k] = **lospre**(P[1..k], P)

MP[3] = |a| = 1
MP[5] = |ab| = 2



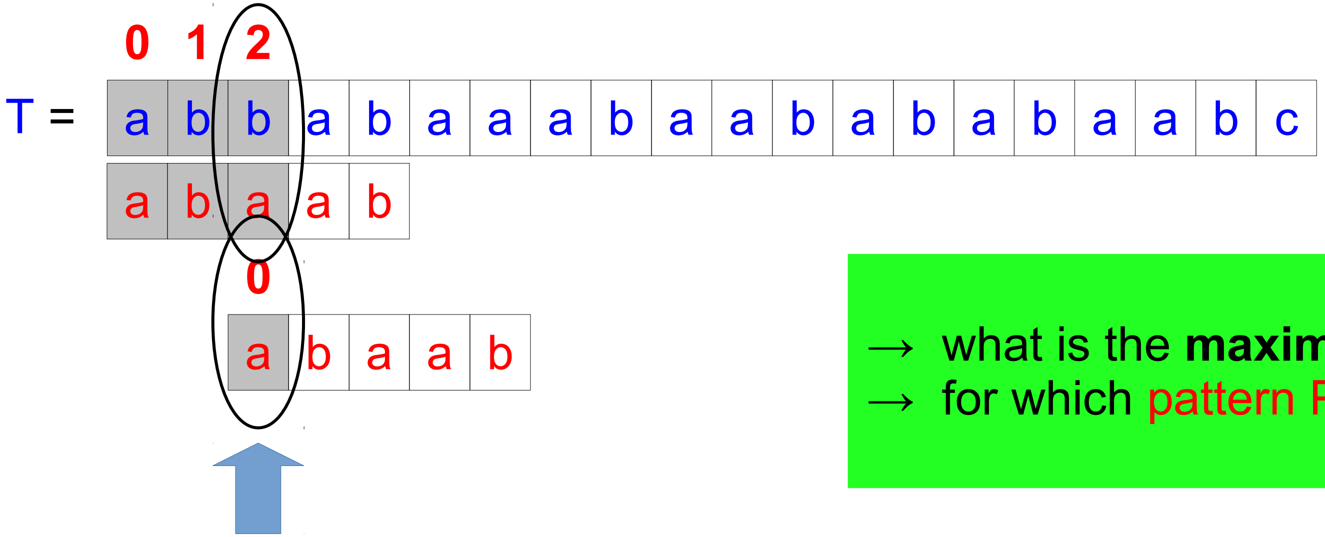
delay = 2   (number of checks at this position)

Delay

	1	2	3	4	5
P =	a	b	a	a	b
	0	0	1	1	2

MP[k] = **lospre**(P[1..k], P)

MP[3] = |a| = 1
MP[5] = |ab| = 2



- → what is the **maximum delay**?
- → for which pattern P?

delay = 2 (number of checks at this position)

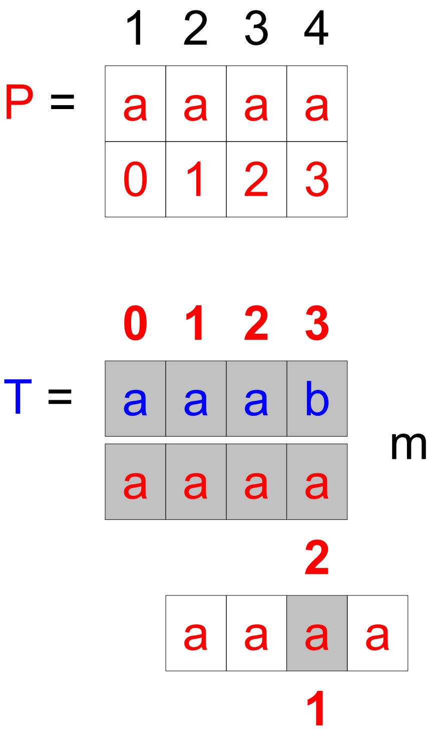
Maximum Delay

	1	2	3	4
P =	a	a	a	a
	0	1	2	3

	0	**1**	**2**	**3**
T =	a	a	a	b
	a	a	a	a
				2

mismatch: lospre = MP[3] = 2

Maximum Delay



P =

1	2	3	4
a	a	a	a
0	1	2	3

T =

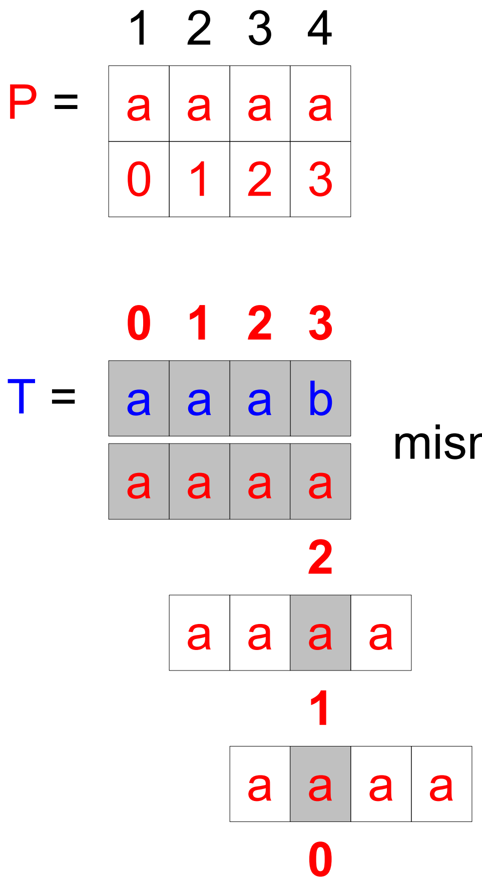
0	**1**	**2**	**3**
a	a	a	b
a	a	a	a
			2

mismatch: lospre = MP[3] = 2

	a	a	a	a
		1		

mismatch: lospre = MP[2] = 1

Maximum Delay

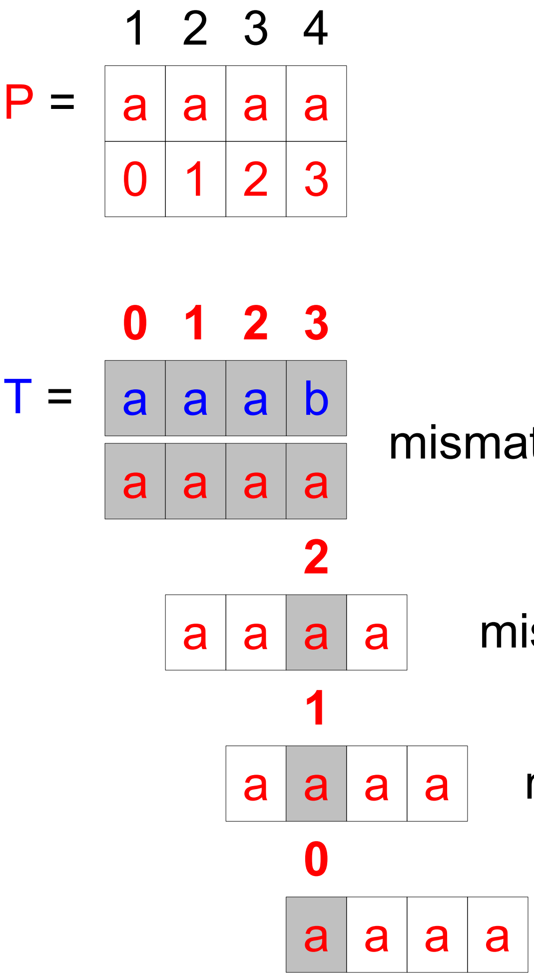


mismatch: lospre = MP[3] = 2

mismatch: lospre = MP[2] = 1

mismatch: lospre = MP[1] = 0

Maximum Delay



	1	2	3	4
P =	a	a	a	a
	0	1	2	3

	0	1	2	3
T =	a	a	a	b
	a	a	a	a

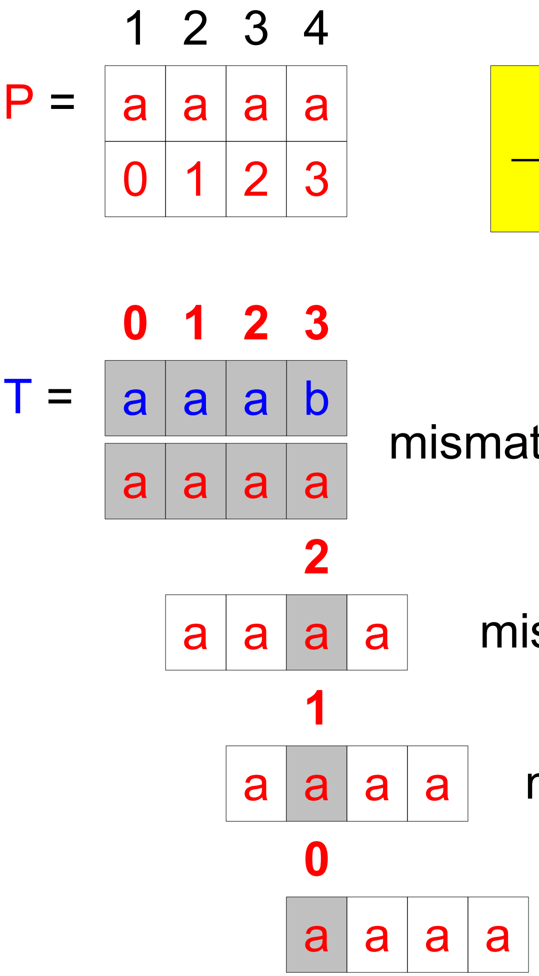
mismatch: lospre = MP[3] = 2

mismatch: lospre = MP[2] = 1

mismatch: lospre = MP[1] = 0

mismatch at lospre = 0 → **advance**

Delay in MP Algorithm



	1	2	3	4
P =	a	a	a	a
	0	1	2	3

→ how to decrease the delay??

	0	**1**	**2**	**3**
T =	a	a	a	b
	a	a	a	a

mismatch: lospre = MP[3] = 2

2

a a a a

mismatch: lospre = MP[2] = 1

1

a a a a

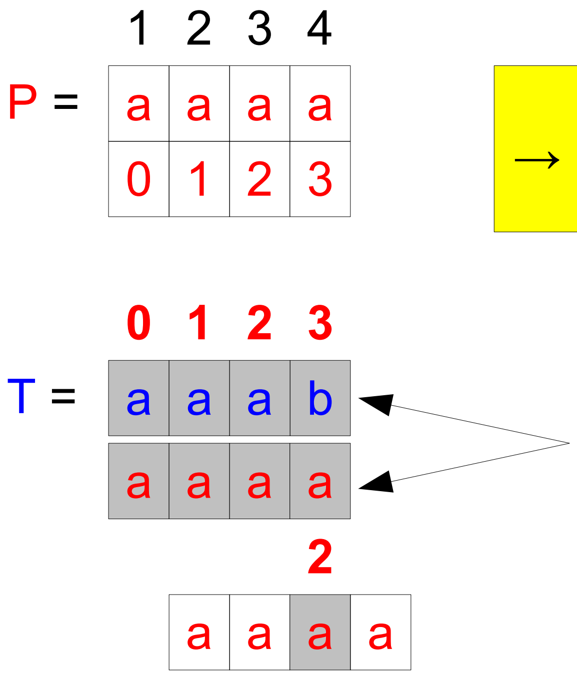
mismatch: lospre = MP[1] = 0

0

a a a a

mismatch at lospre = 0 → **advance**

2. **Knuth**-Morris-Pratt (KMP)



	1	2	3	4
P =	a	a	a	a
	0	1	2	3

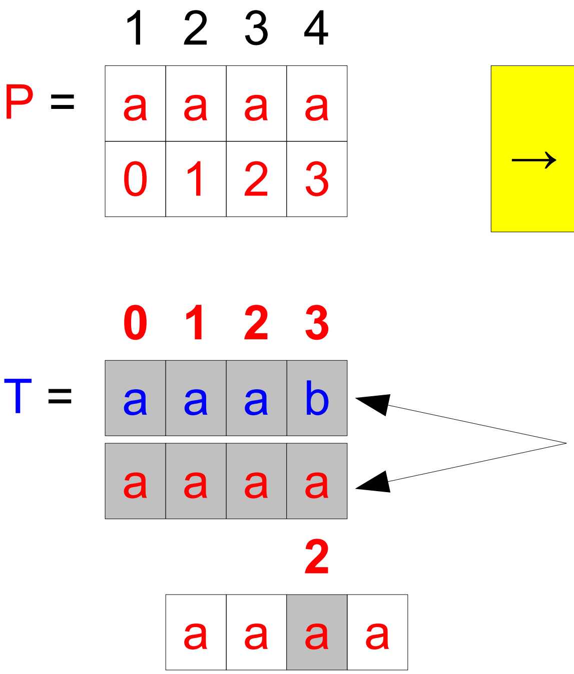
→ how to decrease the delay??

	0	**1**	**2**	**3**	
T =	a	a	a	b	
	a	a	a	a	
				2	
		a	a	a	a

from this check, we know
T[3] != "a"

Thus: if next check-letter in P is "a",
then mismatch

2. KMP



	1	2	3	4
P =	a	a	a	a
	0	1	2	3

→ how to decrease the delay??

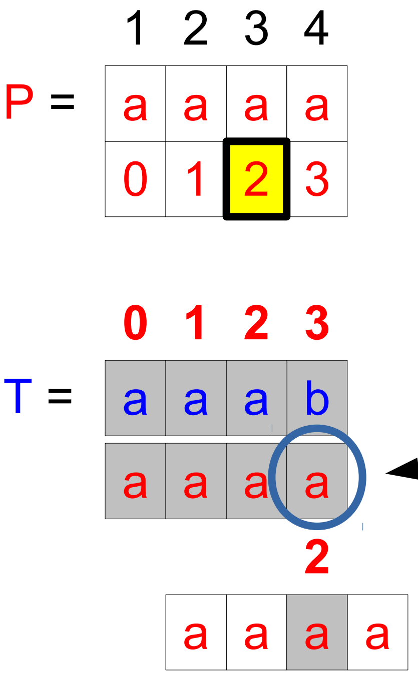
from this check, we know
T[3] != “a”

Thus: if next check-letter in P is “a”,
then mismatch

→ does next check-letter in P
equal fail-letter in P?

If so, then KMP-entry should be smaller!!
(= bigger shift = smaller delay)

2. KMP



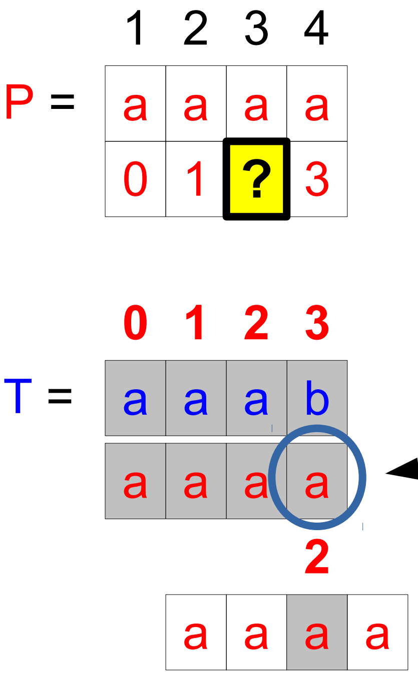
Mismatch at **lospre** = 3

fail-letter = 4
next check-letter = MP[3] + 1 = 2 + 1 = 3

P[4] = P[3]

Thus: KMP[3] should **not** equal “2”!

2. KMP



→ value for KMP[3]?

Mismatch at **lospre** = 3

fail-letter = 4
next check-letter = MP[3] + 1 = 2 + 1 = 3

P[4] = P[3]

Thus: KMP[3] should **not** equal “2”!

→ find longest **lospre**, such that next-letter **!=** fail-letter

→ if does not exist, then mark “**–1**” = ADVANCE (and match with P[1])
= no further check (delay)

2. KMP

P =

1	2	3	4	5
a	b	a	a	b
0	0	1	1	2

Previous table (Morris-Pratt)

-1	1	2	3	4	5
	a	b	a	a	b
	0	-1	1	0	2

Knuth-Morris-Pratt table

KMP[j] largest k>=0 such that strong_cond(j,k) holds, or -1 if such k does not exist

strong_cond(j,k): P[1..k] is a proper suffix of P[1..j] **and**
P[k+1] != P[j+1]

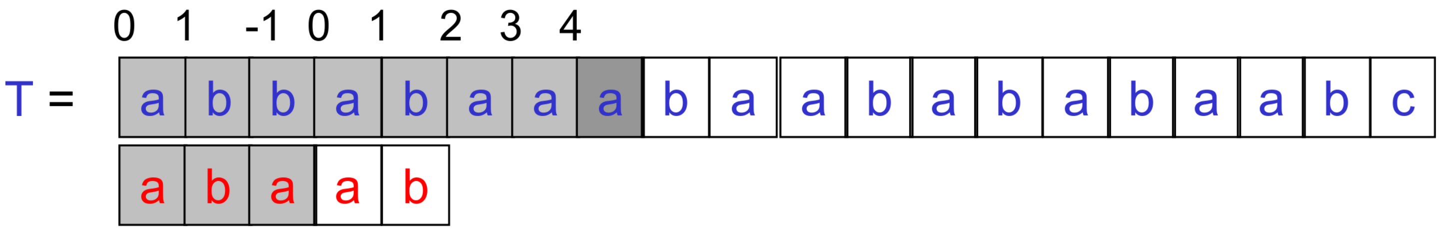
2. KMP

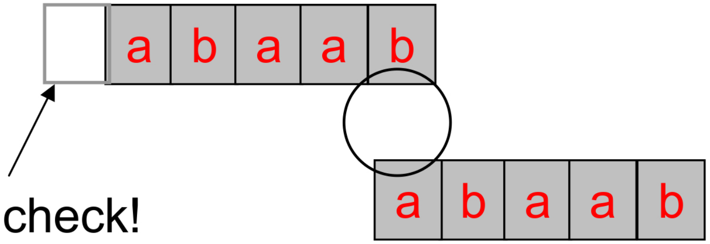
-1	1	2	3	4	5
	a	b	a	a	b
	0	**-1**	1	0	2

Knuth-Morris-Pratt table

a	b	a	a	b
0	0	1	1	2

MP table





no 2nd check!

KMP[j] largest k>=0 such that strong_cond(j,k) holds:
strong_cond(j,k): P[1..k] is a proper suffix of pat[1..j] **and** P[k+1] != P[j+1]

2. KMP

Matching complexity of KMP: O(m + n)

→ what is the maximum delay for KMP?

→ in O(log m) [Knuth]

→ can actually occur (e.g., for Fibonacci strings)

2. KMP

Lemma
For KMP, delay(m) = O(log m), and the bound is **tight**.

KMP1977 (p. 333)
Delay per scanned character is at most $1+\log_{\phi} m$,
where $\phi=(1+\mathrm{sqrt}\{5\})/2 = 1.618\ldots$ is the golden ratio.

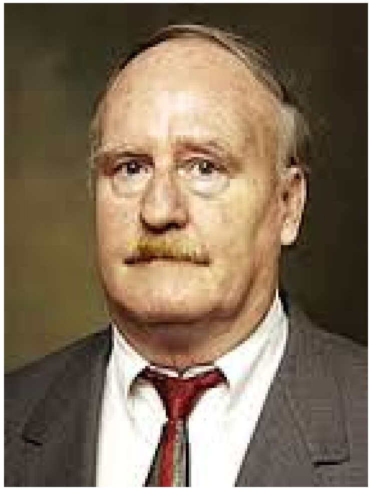
1) Define Fibonacci strings: f(1)=b, f(2)=a, f(n)=f(n – 1)f(n – 2)
b, a, ab, aba, abaab, abaababa, …

j =	1	2	**3**	4	5	**6**	7	8	9	10	**11**	12	13	14	15	16	17	18	**19**	20	21
	a	b	a	a	b	a	b	a	a	b	a	a	b	a	b	a	a	b	a	b	a
KMP[j] =	0	0	**1**	0	0	**3**	0	1	0	0	**6**	0	0	3	0	1	0	0	**11**	0	3

$\mathrm{KMP}[F_k - 2] = F_{k-1} - 2$
Mismatch at position 19 = $F_8 - 2$, check at: **19, 11, 6, 3, 1, 0**.
$F_k = \mathrm{round}(\phi^k / \mathrm{sqrt}\{5\})$ → delay(m) = $\log_{\phi}(m) - 2$.

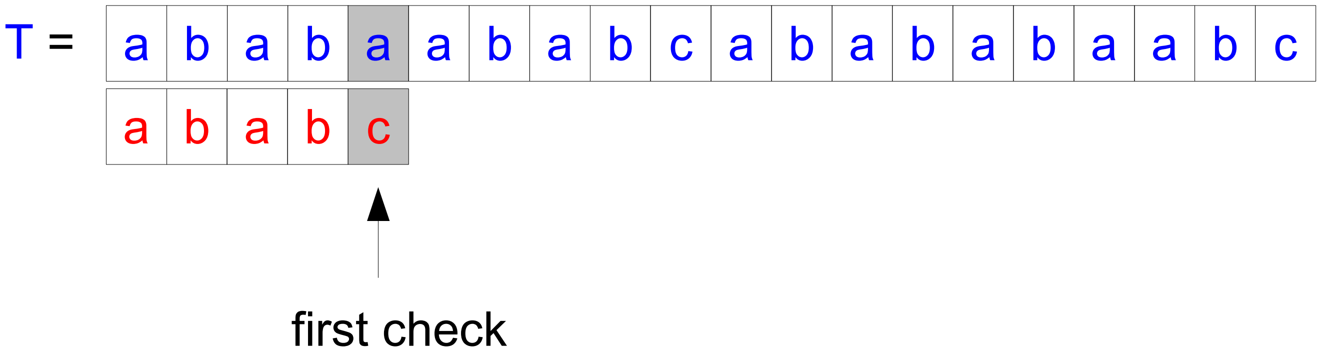
3. Boyer-Moore

Robert S. Boyer and J. Strother Moore in 1977



3. Boyer-Moore

Idea 1 Match **RIGHT-TO-LEFT** in window



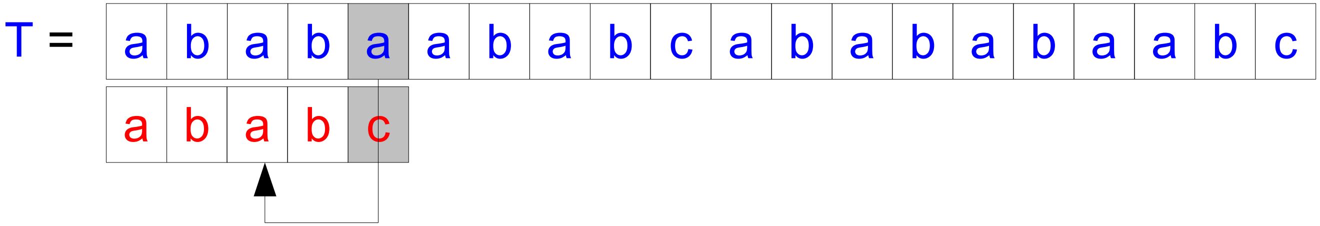
T = a b a b a a b a b c a b a b a b a a b c

a b a b c

first check

3. Boyer-Moore

Idea 1 Match **RIGHT-TO-LEFT** in window



T = a b a b a a b a b c a b a b a b a a b c

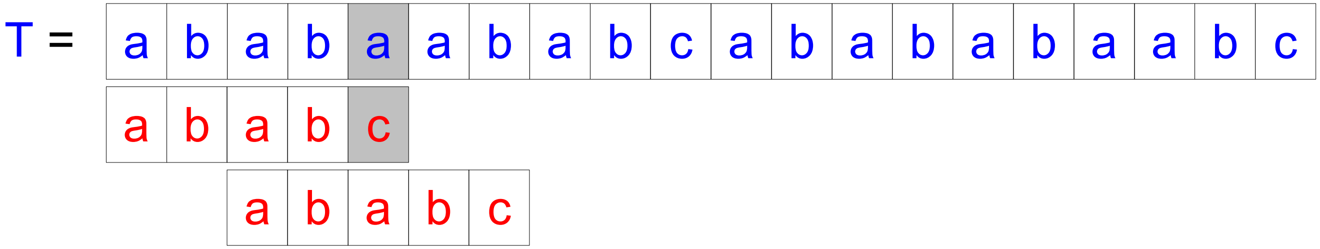
a b a b c

R(a) = 2

For each letter z, let
R(z) = distance from right-most occurrence of z in P, to the end of P
(and |P| if there is no occurrence)

3. Boyer-Moore

Idea 1 Match **RIGHT-TO-LEFT** in window

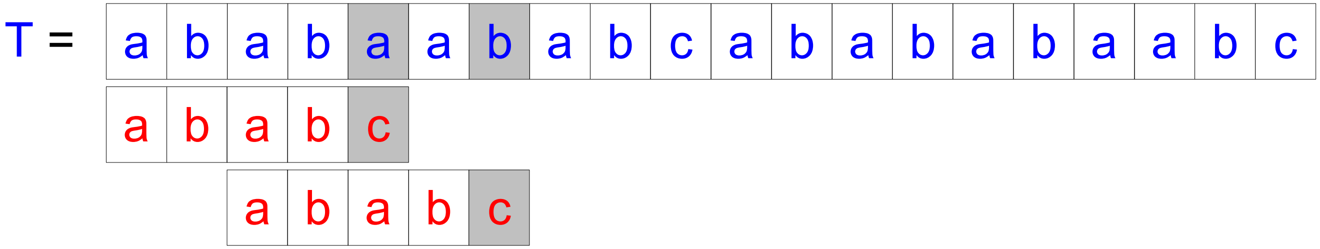


For each letter z, let
R(z) = distance from right-most occurrence of z in P, to the end of P
(and |P| if there is no occurrence)

→ shift R(a) to the right at mismatch with “a”
(for all smaller shifts, we get a mismatch)

3. Boyer-Moore

Idea 1 Match **RIGHT-TO-LEFT** in window


T = a b a b a a b a b c a b a b a b a a b c

a b a b c

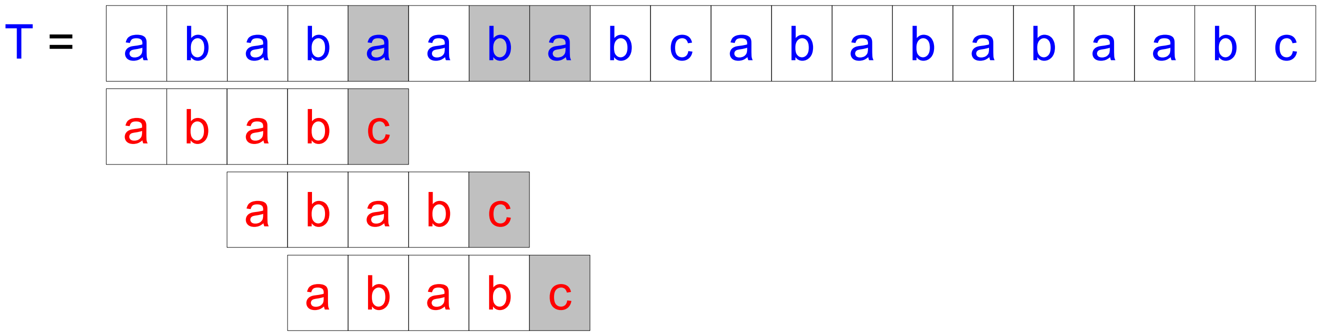
a b a b c

For each letter z, let
R(z) = distance from right-most occurrence of z in P, to the end of P
(and |P| if there is no occurrence)
R(a) = 2, R(b) = 1

→ may shift R(a) to the right at mismatch with "a"
(for all smaller shifts, we get a mismatch)

3. Boyer-Moore

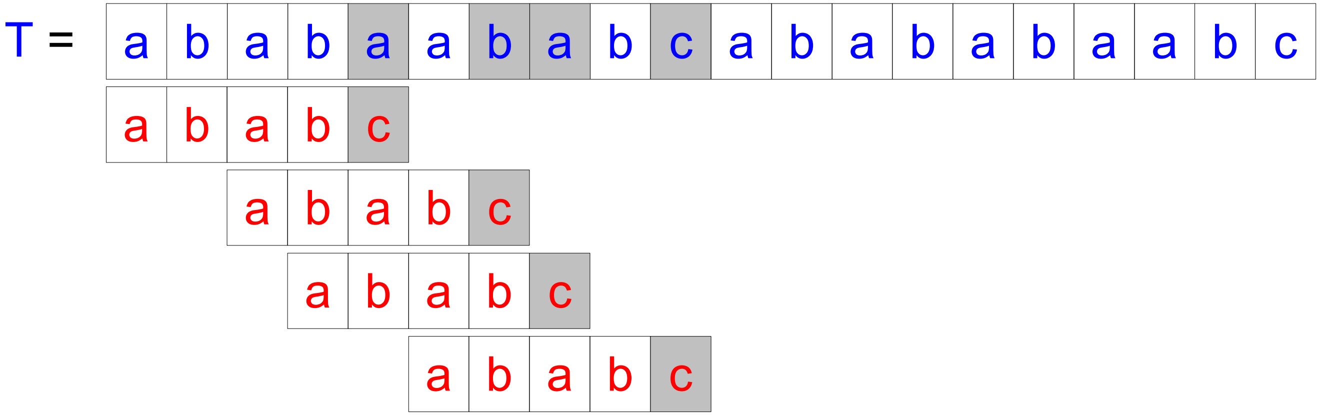
Idea 1 Match **RIGHT-TO-LEFT** in window



T =	a	b	a	b	a	a	b	a	b	c	a	b	a	b	a	b	a	a	b	c
	a	b	a	b	c															
			a	b	a	b	c													
				a	b	a	b	c												

3. Boyer-Moore

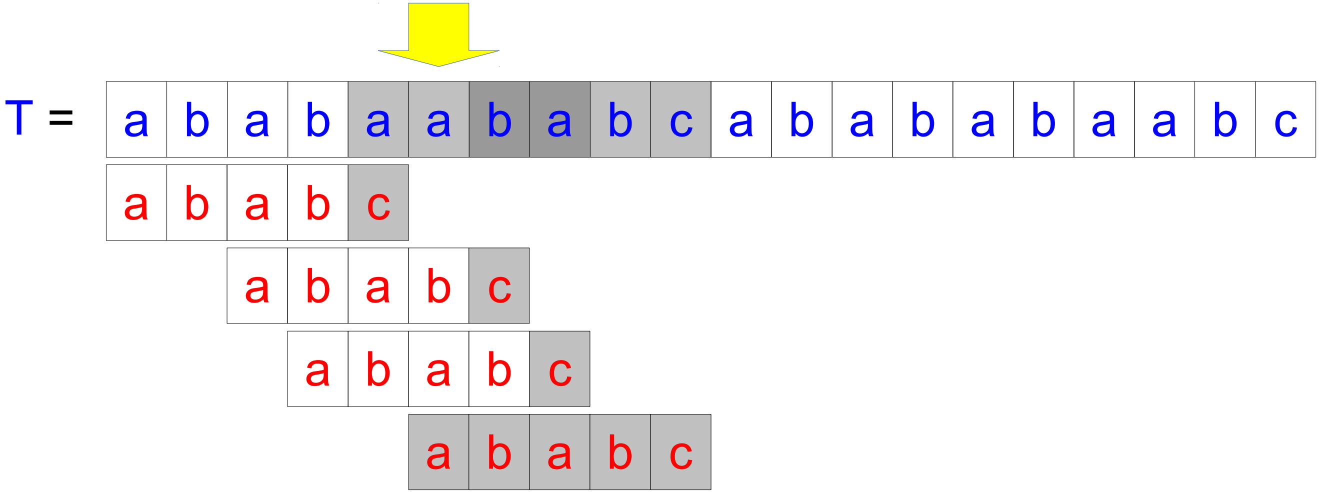
Idea 1 Match **RIGHT-TO-LEFT** in window



T =	a	b	a	b	a	a	b	a	b	c	a	b	a	b	a	b	a	a	b	c
	a	b	a	b	c															
			a	b	a	b	c													
				a	b	a	b	c												
						a	b	a	b	c										

3. Boyer-Moore

Idea 1 Match **RIGHT-TO-LEFT** in window



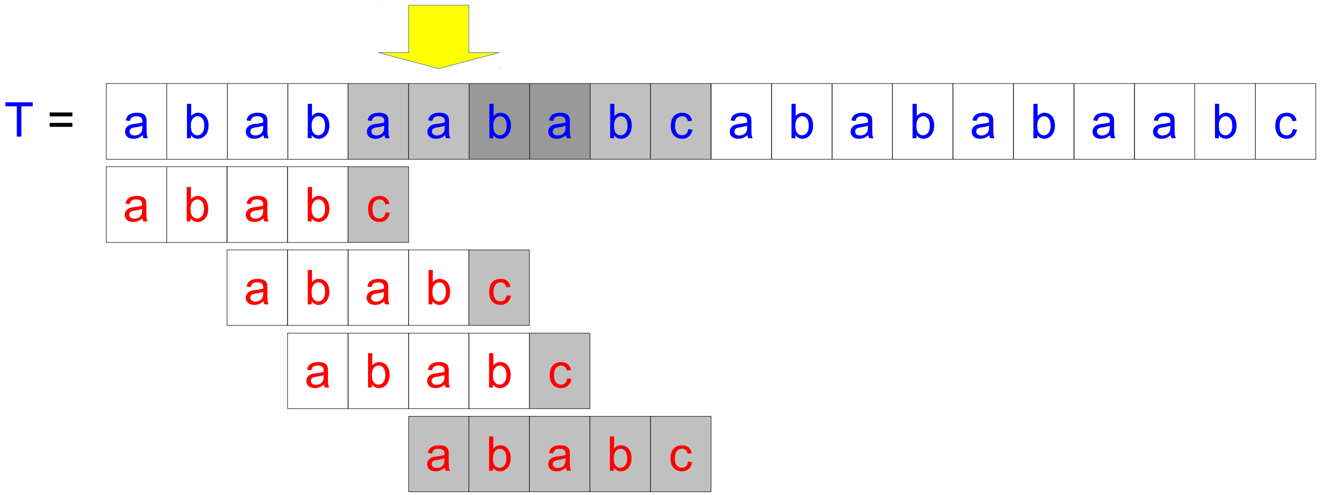
T =	a	b	a	b	a	a	b	a	b	c	a	b	a	b	a	b	a	a	b	c
	a	b	a	b	c															
			a	b	a	b	c													
				a	b	a	b	c												
						a	b	a	b	c										

→ after **only 8 comparisons**, detects the first match!

→ compare this with the previous methods!

3. Boyer-Moore

Idea 1 Match **RIGHT-TO-LEFT** in window



T = a b a b a a b a b c a b a b a b a a b c

a b a b c

a b a b c

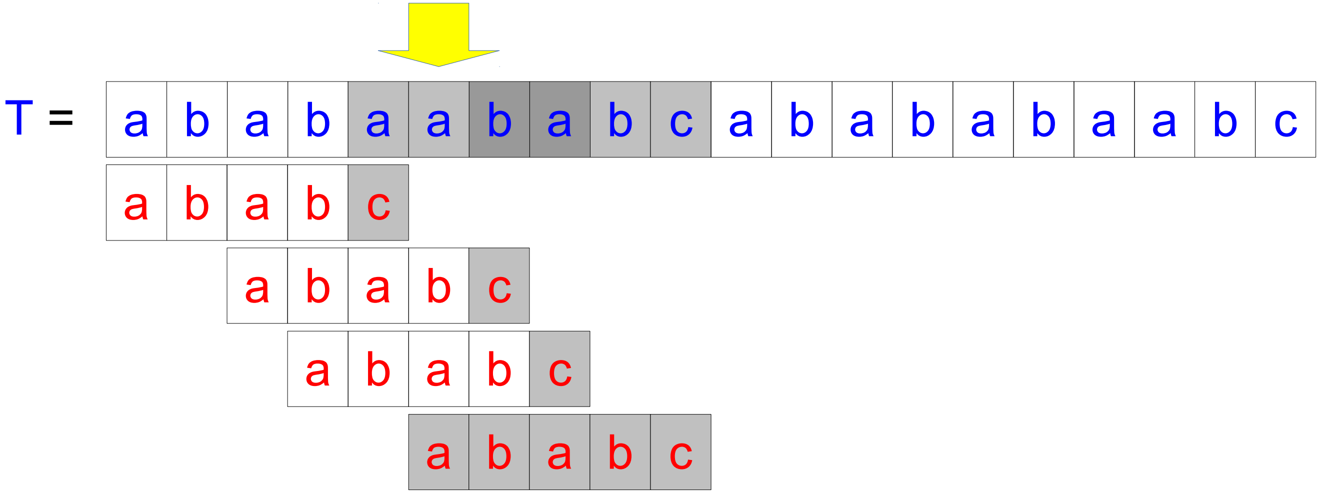
a b a b c

a b a b c

- → after **only 8 comparisons**, detects the first match!
- → compare this with the previous methods!
- → letters T[1], T[2], T[3], and T[4] are never checked!
- → allows for *sub-linear matching time* wrt n = |T|

3. Boyer-Moore

Idea 1 Match **RIGHT-TO-LEFT** in window



T =	a	b	a	b	a	a	b	a	b	c	a	b	a	b	a	b	a	a	b	c
	a	b	a	b	c															
			a	b	a	b	c													
				a	b	a	b	c												
						a	b	a	b	c										

→ after **only 8 comparisons**, detects the first match!

→ compare this with the previous methods!

→ letters T[1], T[2], T[3], and T[4] are never checked!

→ allows for *sub-linear matching time* wrt n = |T|

→ for natural language, almost always sublinear time!

3. Boyer-Moore

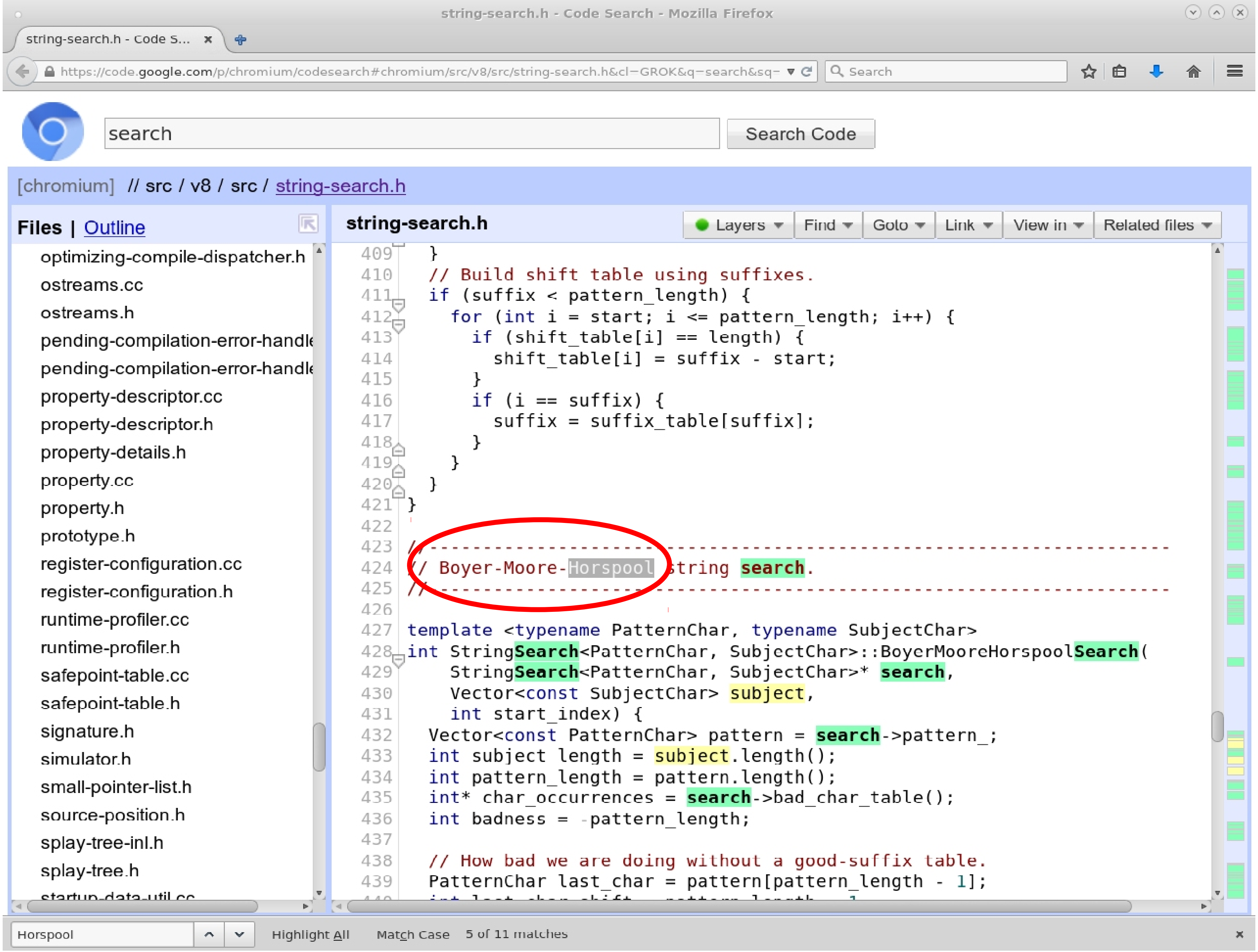
Idea 1 → match **RIGHT-TO-LEFT**
→ shift R(z) to the right at mismatch

"Horspool algorithm"

→ How does Google Chrome search text (Ctrl^ F) on a page so quickly?

It uses a search algorithm inspired by Boyer-Moore and
Boyer-Moore-Horspool

→ V8 string search package (chromium project)



search

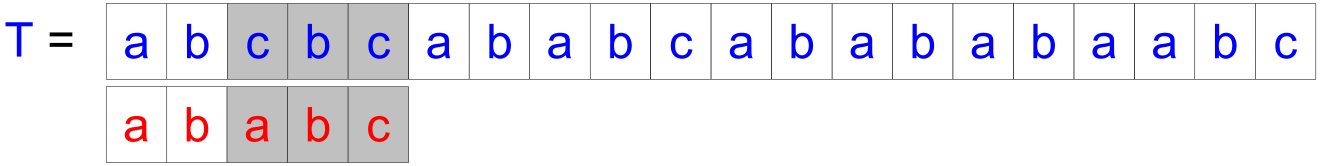
```
  }
  // Build shift table using suffixes.
  if (suffix < pattern_length) {
    for (int i = start; i <= pattern_length; i++) {
      if (shift_table[i] == length) {
        shift_table[i] = suffix - start;
      }
      if (i == suffix) {
        suffix = suffix_table[suffix];
      }
    }
  }
}

//-----------------------------------
// Boyer-Moore-Horspool string search.
//-----------------------------------

template <typename PatternChar, typename SubjectChar>
int StringSearch<PatternChar, SubjectChar>::BoyerMooreHorspoolSearch(
    StringSearch<PatternChar, SubjectChar>* search,
    Vector<const SubjectChar> subject,
    int start_index) {
  Vector<const PatternChar> pattern = search->pattern_;
  int subject_length = subject.length();
  int pattern_length = pattern.length();
  int* char_occurrences = search->bad_char_table();
  int badness = -pattern_length;

  // How bad we are doing without a good-suffix table.
  PatternChar last_char = pattern[pattern_length - 1];
```

Horspool

5 of 11 matches

3. Boyer-Moore

Idea 2



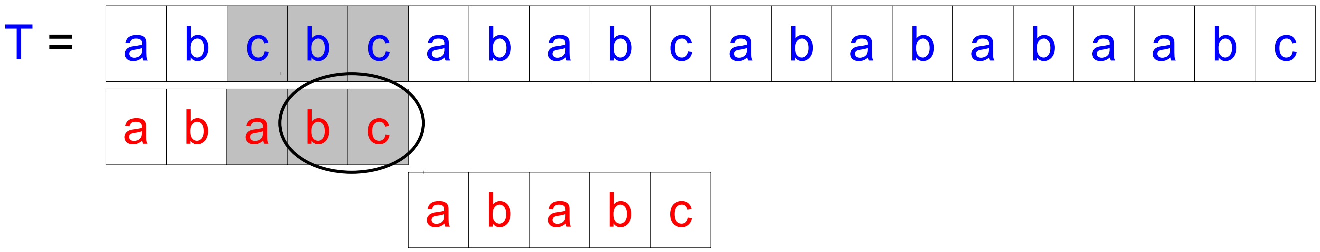
T = a b c b c a b a b c a b a b a b a a b c

a b a b c

→ how far can we shift?

3. Boyer-Moore

Idea 2



T = a b c b c a b a b c a b a b a b a a b c

a b a b c

a b a b c

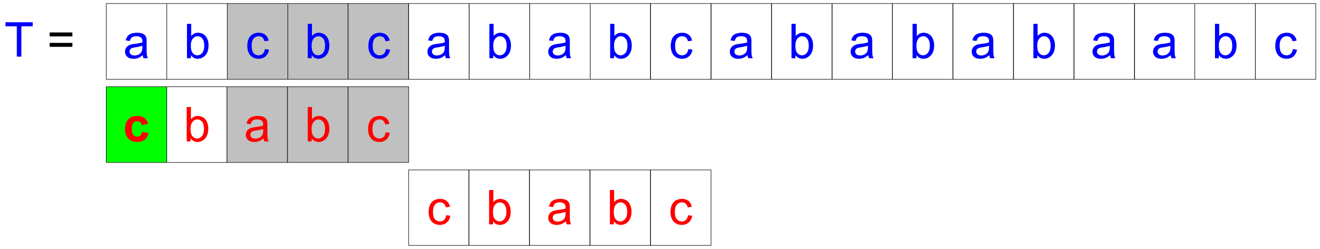
→ how far can we shift?

→ all of |P|.
because “bc” does not occur to the left in P

→ for every suffix u of P, let D(u) be the distance to the next occurrence of u to the left (if none exists, then D(u)=|P|)

3. Boyer-Moore

Idea 2



T = a b c b c a b a b c a b a b a b a a b c

c b a b c

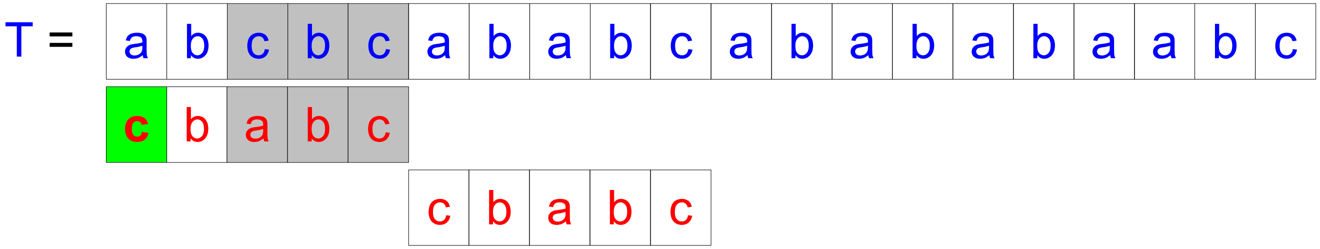
c b a b c

→ D(bc) = 5

→ now **not OK**, to shift by 5! Why??

3. Boyer-Moore

Idea 2



T = a b c b c a b a b c a b a b a b a a b c

c b a b c

c b a b c

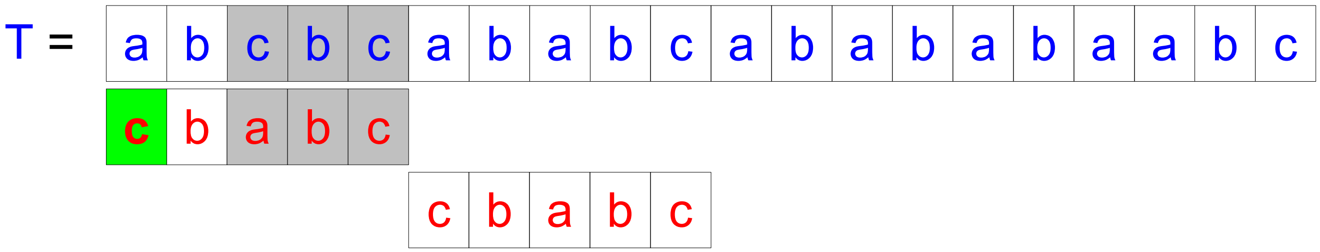
→ D(bc) = 5

→ now **not OK**, to shift by 5! Why??

→ a suffix of u is a prefix of P!

3. Boyer-Moore

Idea 2



T = a b c b c a b a b c a b a b a b a a b c

c b a b c

c b a b c

→ D(bc) = 5

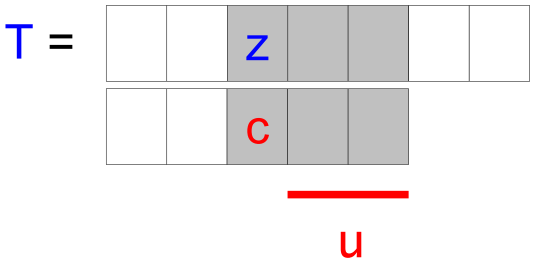
→ now **not OK**, to shift by 5! Why??

→ a suffix of u is a prefix of P!

→ for every suffix u of P, let
L(u) = **lospre**(u, P)

3. Boyer-Moore

Idea 2

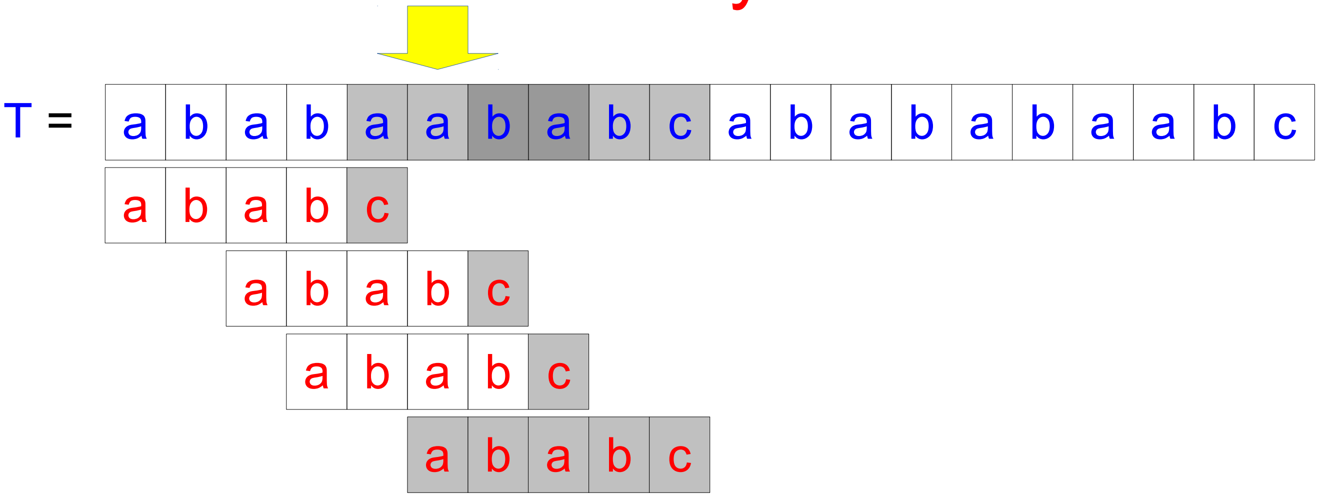


```
if mismatch after cu on symbol z, then
  k = max( R(z), D(u) )
  k = min( k, |P|-L(u) )
else
    report occurrence of P
    k := |P|-L(u)
shift by k
```

maximum shift

restrict by **lospre**

3. Boyer-Moore



T = a b a b a a b a b c a b a b a b a a b c

a b a b c

a b a b c

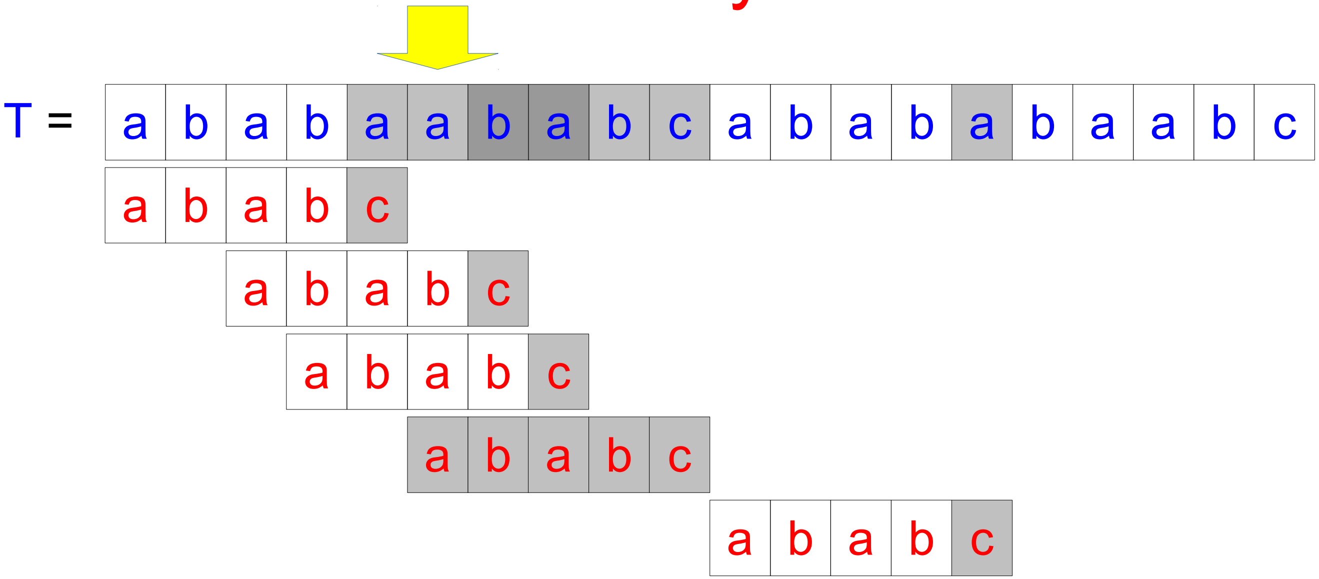
a b a b c

a b a b c

```
if mismatch after cu on symbol z, then
  k = max( R(z), D(u) )
  k = min( k, |P|-L(u) )
else
    report occurrence of P
    k := |P|-L(P)
shift by k
```

L(P) = 0, hence we shift by |P| = 5

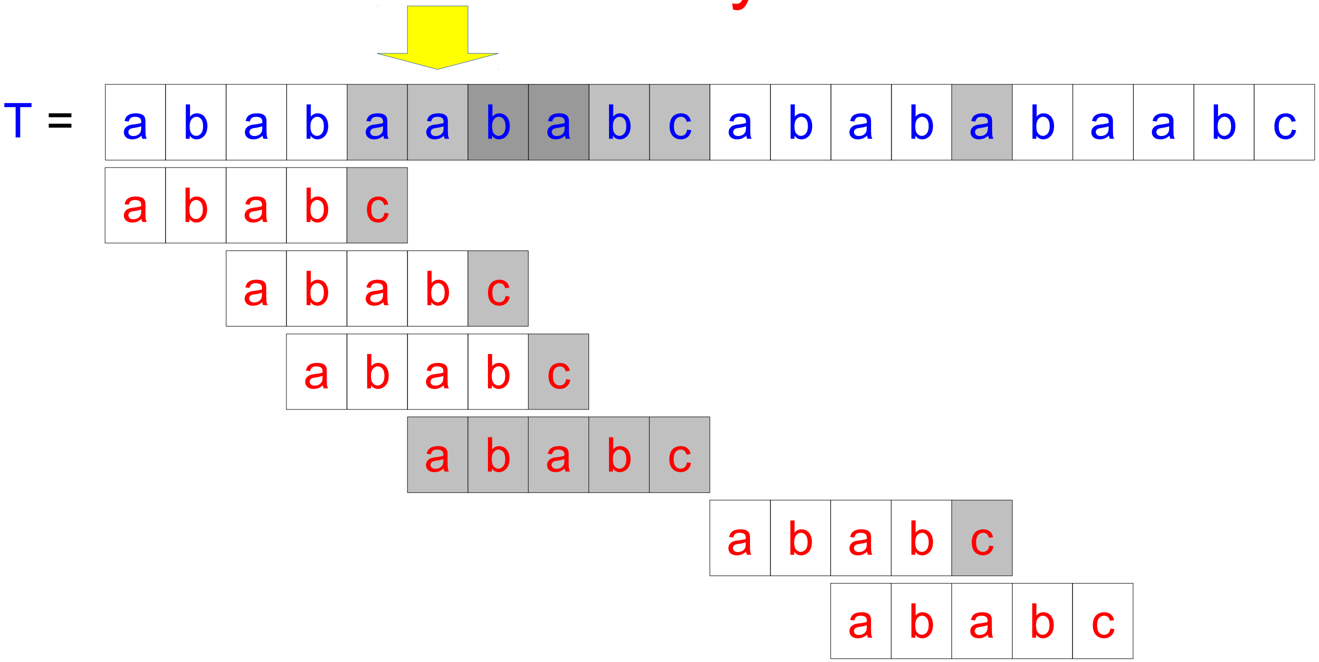
3. Boyer-Moore



```
if mismatch after cu on symbol z, then
  k = max( R(z), D(u) )
  k = min( k, |P|-L(u) )
else
    report occurrence of P
    k := |P|-L(P)
shift by k
```

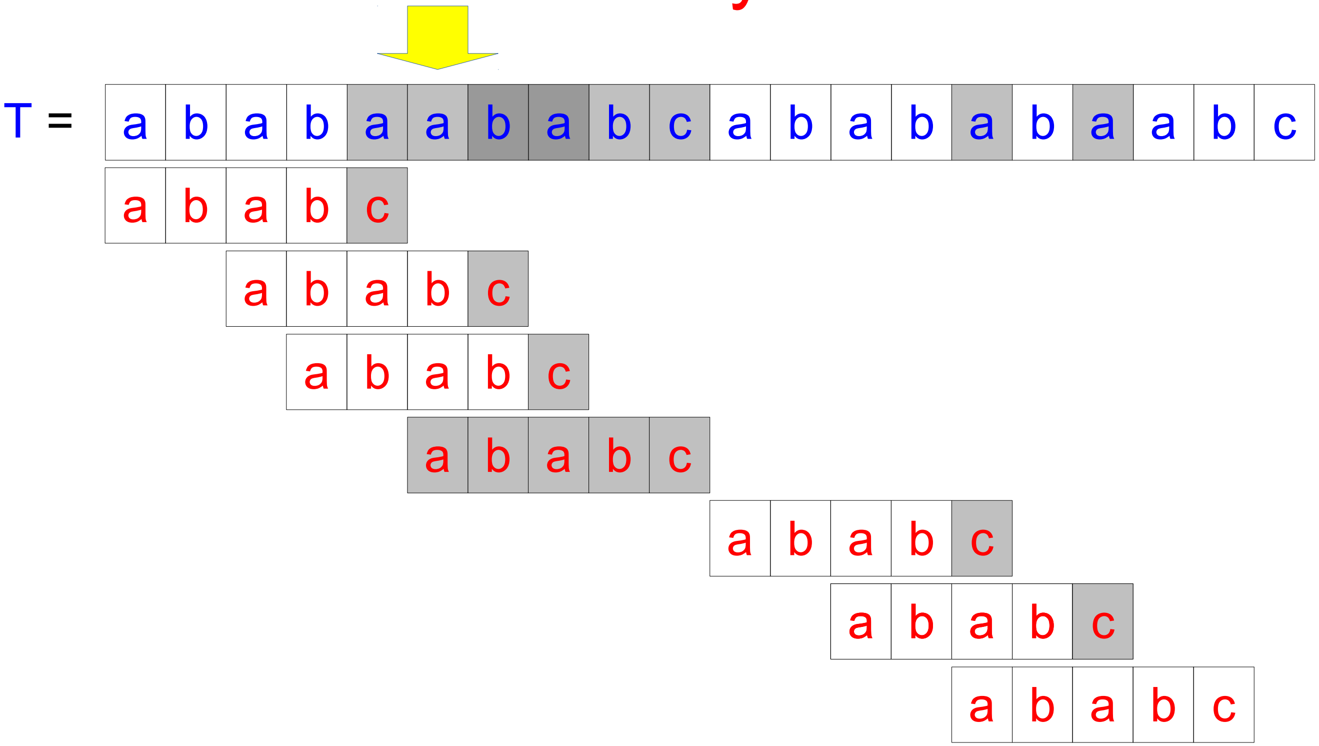
L(P) = 0, hence we shift by |P| = 5

3. Boyer-Moore



```
if mismatch after cu on symbol z, then
  k = max( R(z), D(u) )
  k = min( k, |P|-L(u) )
else
    report occurrence of P
    k := |P|-L(P)
shift by k
```

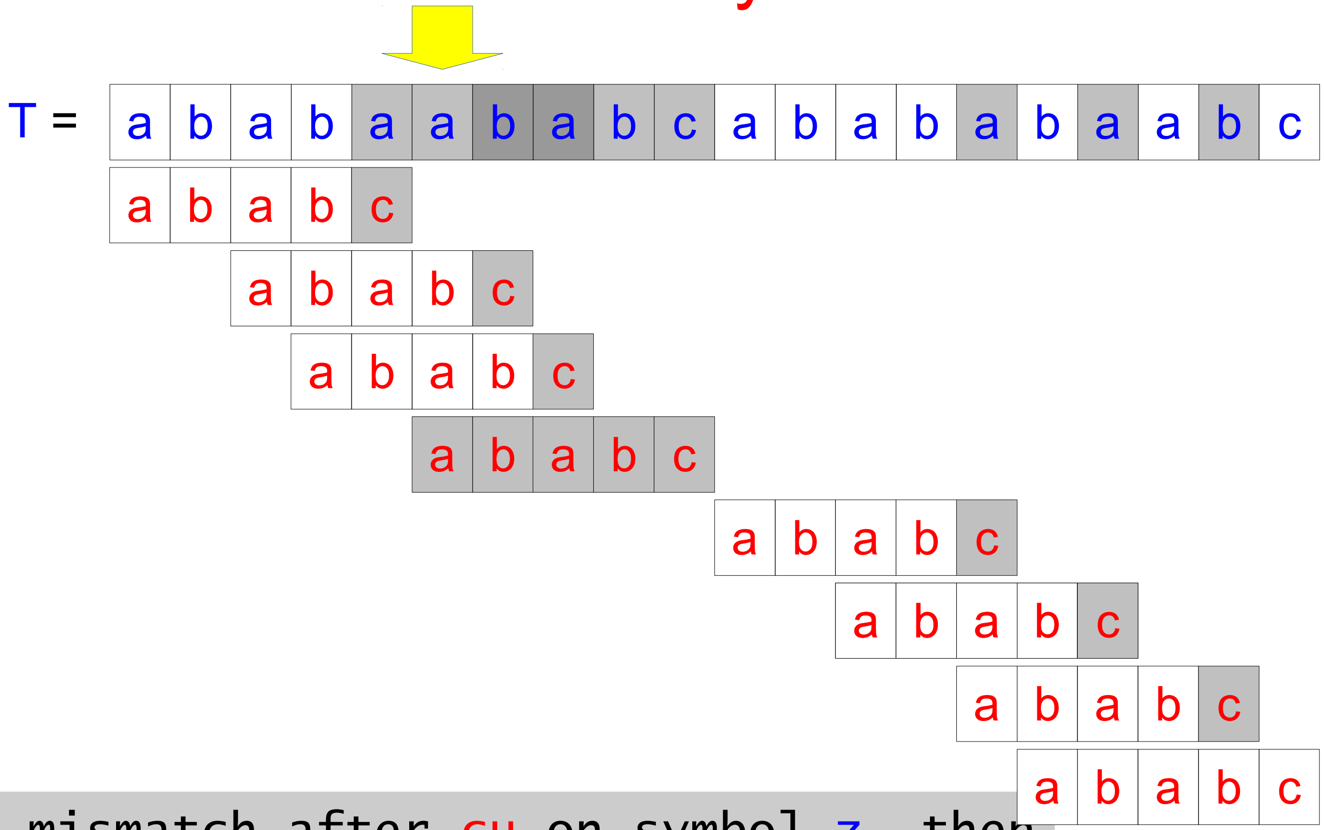
3. Boyer-Moore



T =	a	b	a	b	a	a	b	a	b	c	a	b	a	b	a	b	a	a	b	c
	a	b	a	b	c															
			a	b	a	b	c													
				a	b	a	b	c												
						a	b	a	b	c										
											a	b	a	b	c					
													a	b	a	b	c			
															a	b	a	b	c	

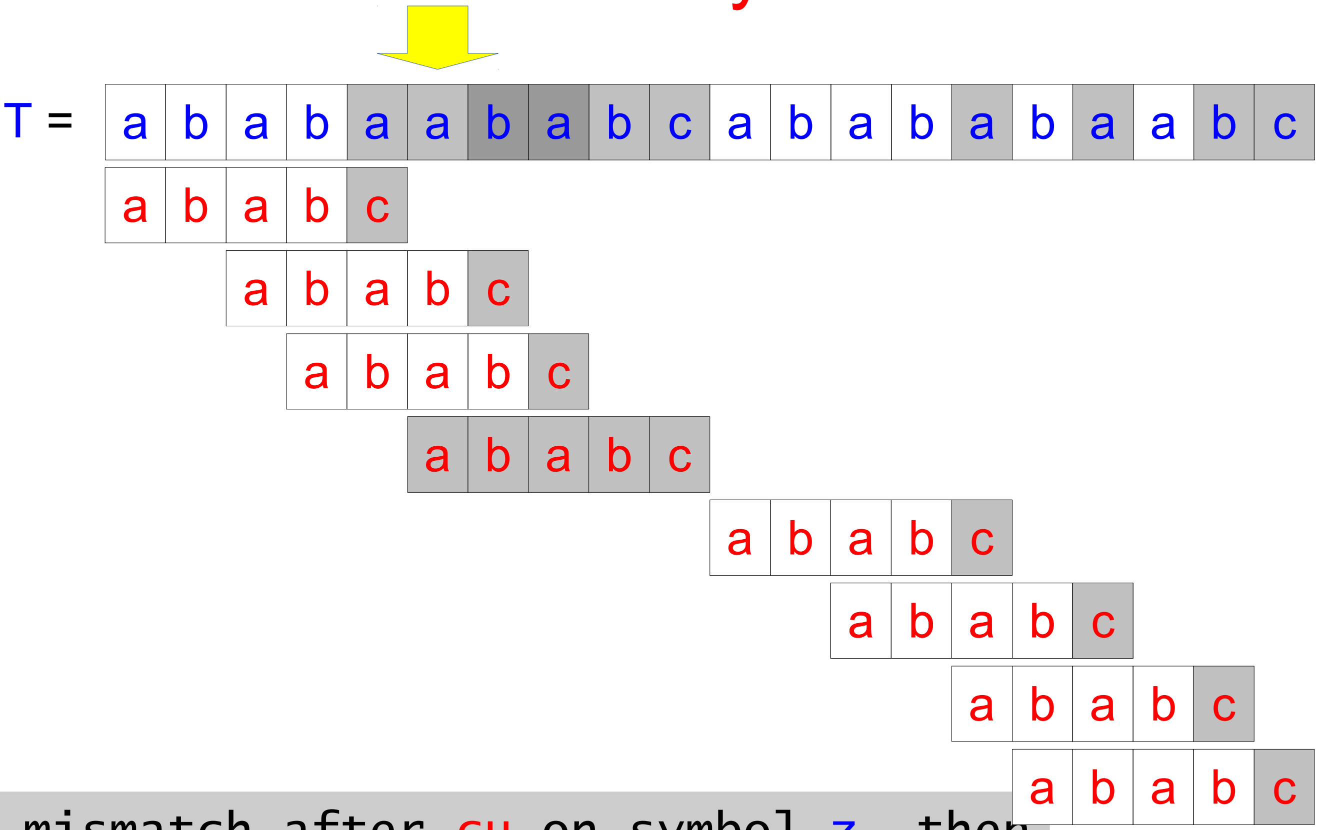
```
if mismatch after cu on symbol z, then
  k = max( R(z), D(u) )
  k = min( k, |P|-L(u) )
else
    report occurrence of P
    k := |P|-L(P)
shift by k
```

3. Boyer-Moore



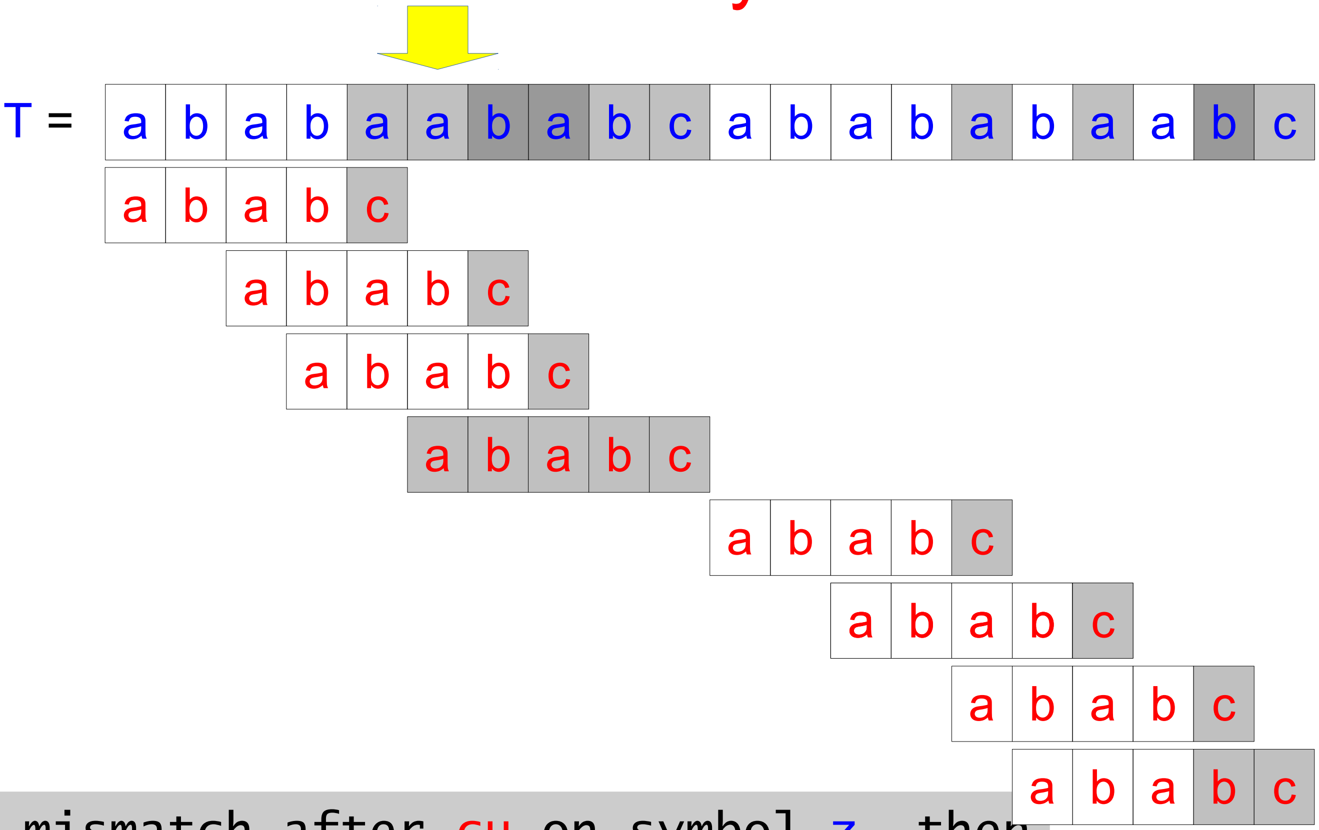
```
if mismatch after cu on symbol z, then
  k = max( R(z), D(u) )
  k = min( k, |P|-L(u) )
else
    report occurrence of P
    k := |P|-L(P)
shift by k
```

3. Boyer-Moore



```
if mismatch after cu on symbol z, then
  k = max( R(z), D(u) )
  k = min( k, |P|-L(u) )
else
    report occurrence of P
    k := |P|-L(P)
shift by k
```

3. Boyer-Moore



T = a b a b a a b a b c a b a b a b a a b c

a b a b c

a b a b c

a b a b c

a b a b c

a b a b c

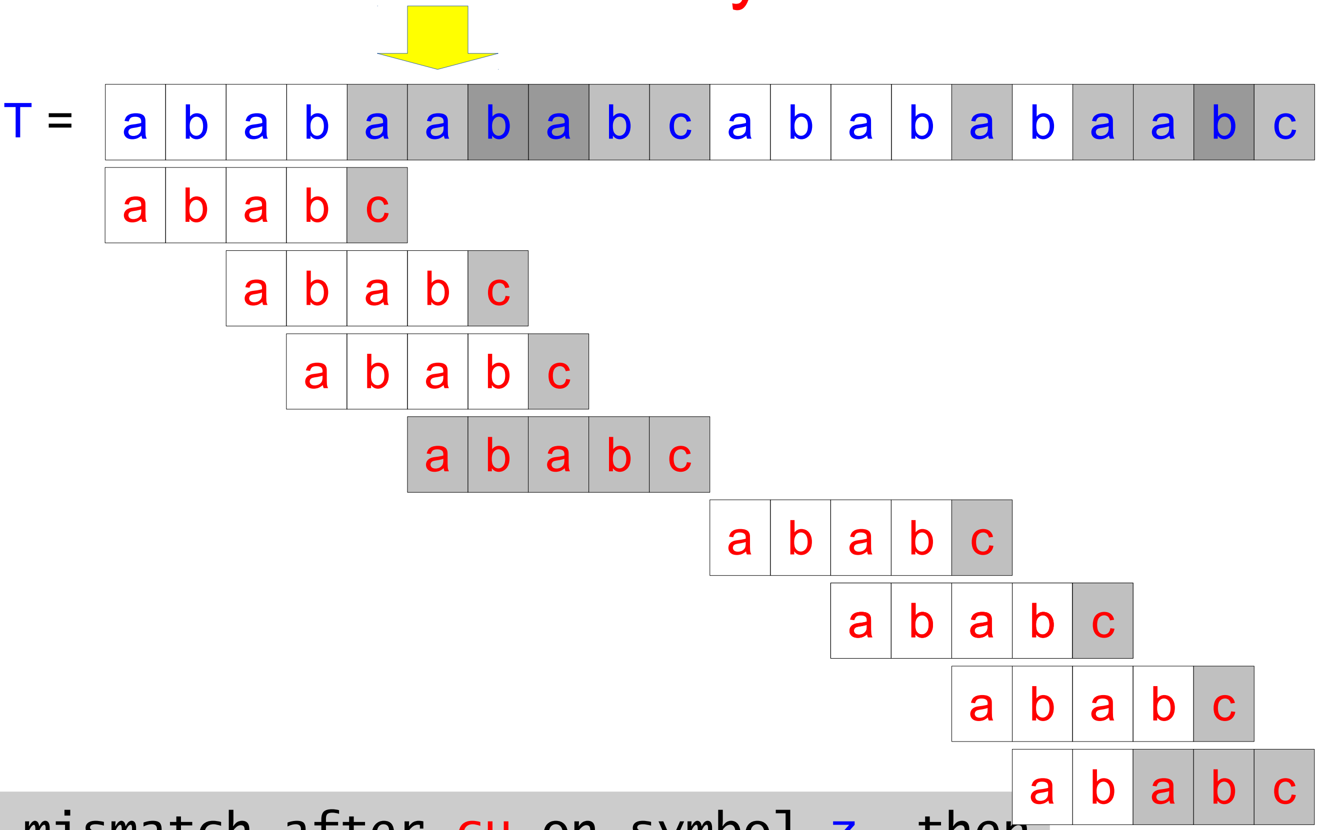
a b a b c

a b a b c

a b a b c

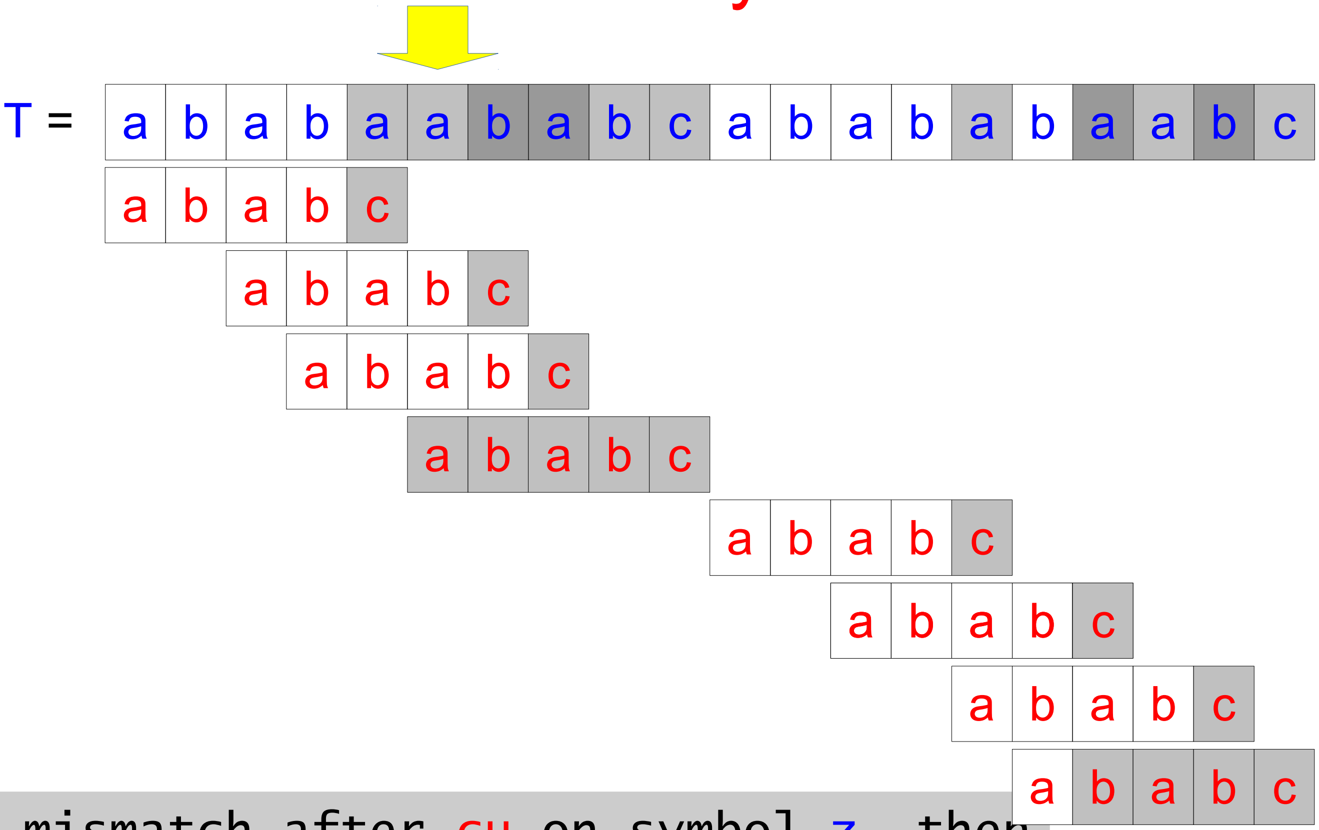
```
if mismatch after cu on symbol z, then
  k = max( R(z), D(u) )
  k = min( k, |P|-L(u) )
else
    report occurrence of P
    k := |P|-L(P)
shift by k
```

3. Boyer-Moore



```
if mismatch after cu on symbol z, then
  k = max( R(z), D(u) )
  k = min( k, |P|-L(u) )
else
    report occurrence of P
    k := |P|-L(P)
shift by k
```

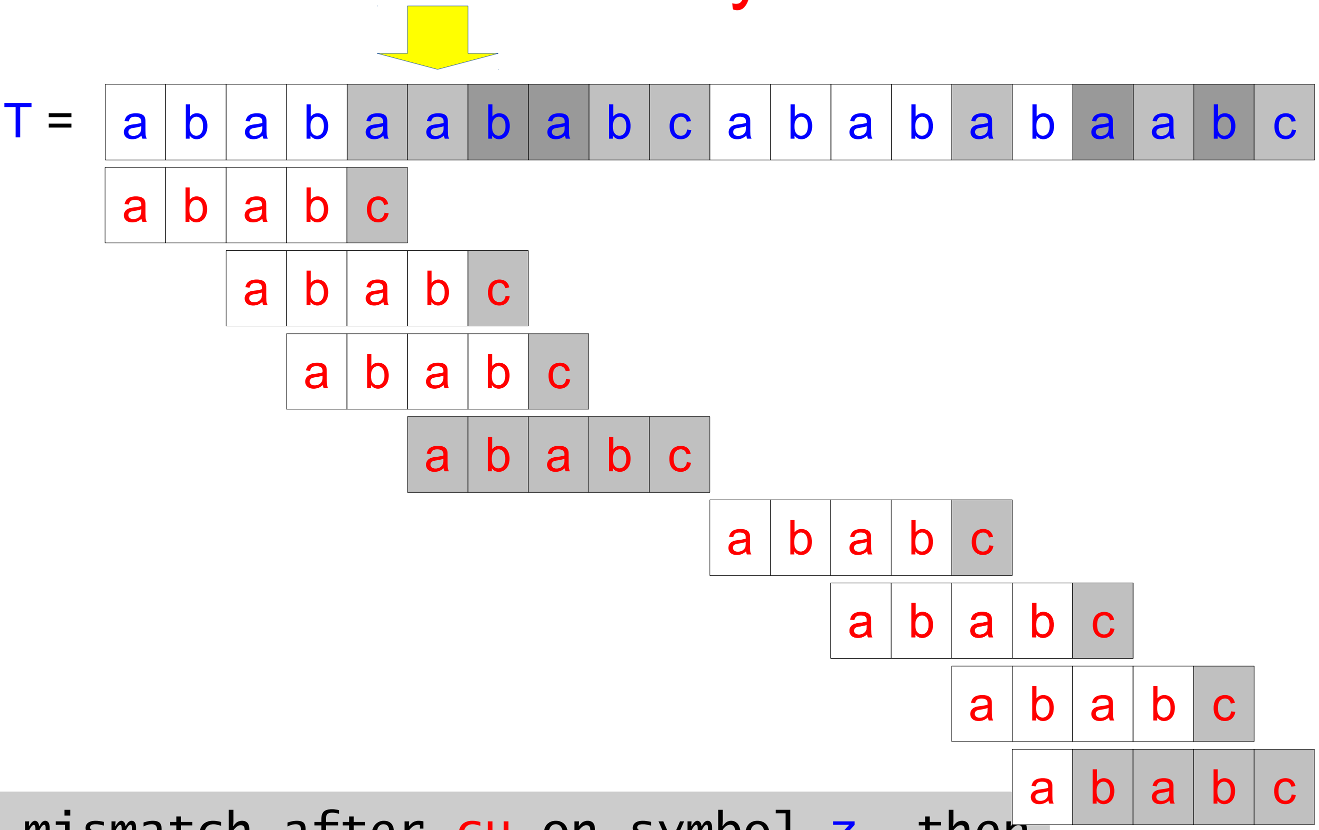
3. Boyer-Moore



```
T =  a b a b a a b a b c a b a b a b a a b c
     a b a b c
         a b a b c
           a b a b c
               a b a b c
                         a b a b c
                             a b a b c
                                 a b a b c
                                   a b a b c
```

```
if mismatch after cu on symbol z, then
  k = max( R(z), D(u) )
  k = min( k, |P|-L(u) )
else
    report occurrence of P
    k := |P|-L(P)
shift by k
```

3. Boyer-Moore



```
if mismatch after cu on symbol z, then
  k = max( R(z), D(u) )
  k = min( k, |P|-L(u) )
else
    report occurrence of P
    k := |P|-L(P)
shift by k
```

→ finished!

→ would shift k=2, but
end of text reached

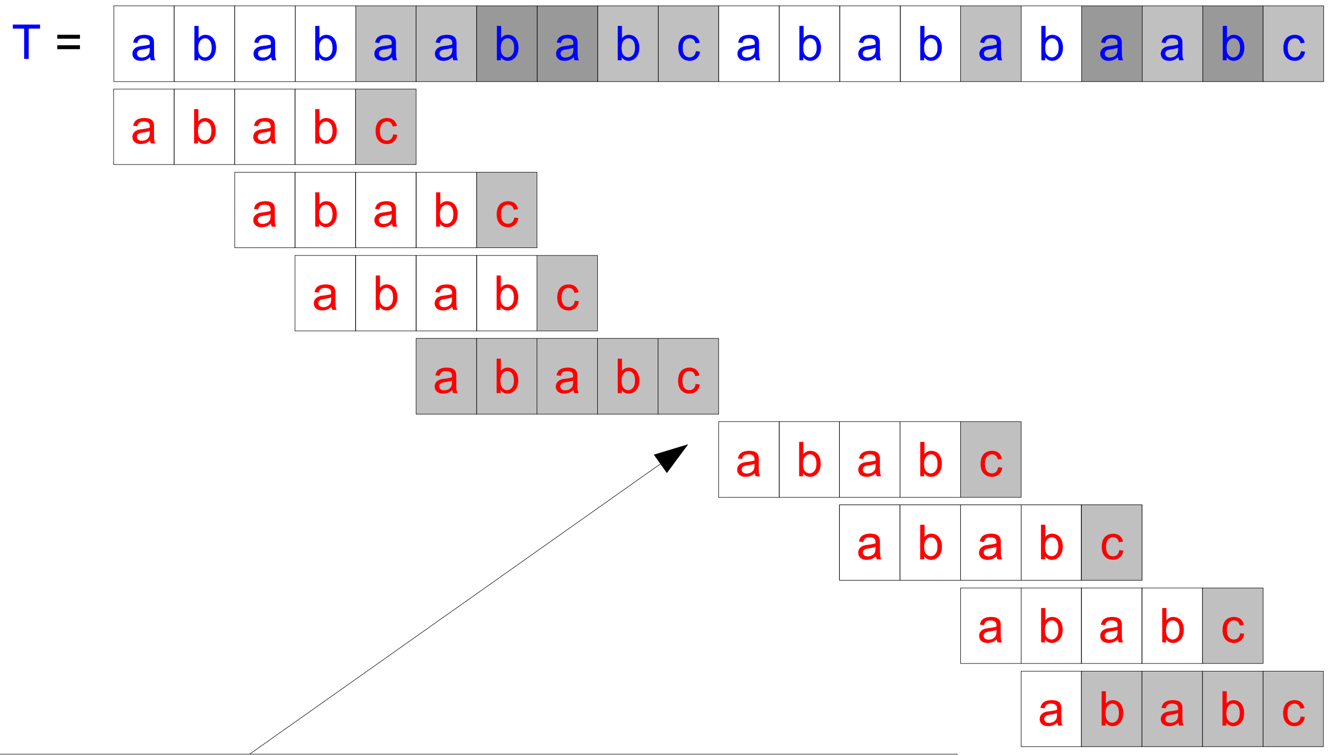
4. Horspool

Only **Idea 1**

→ match from right-to-left
→ at mismatch (with z): shift to R(z)

R(z) = distance from right-most occurrence of z in P[1..m-1], to the end of P
(|P| if there is no occurrence)

4. Horspool

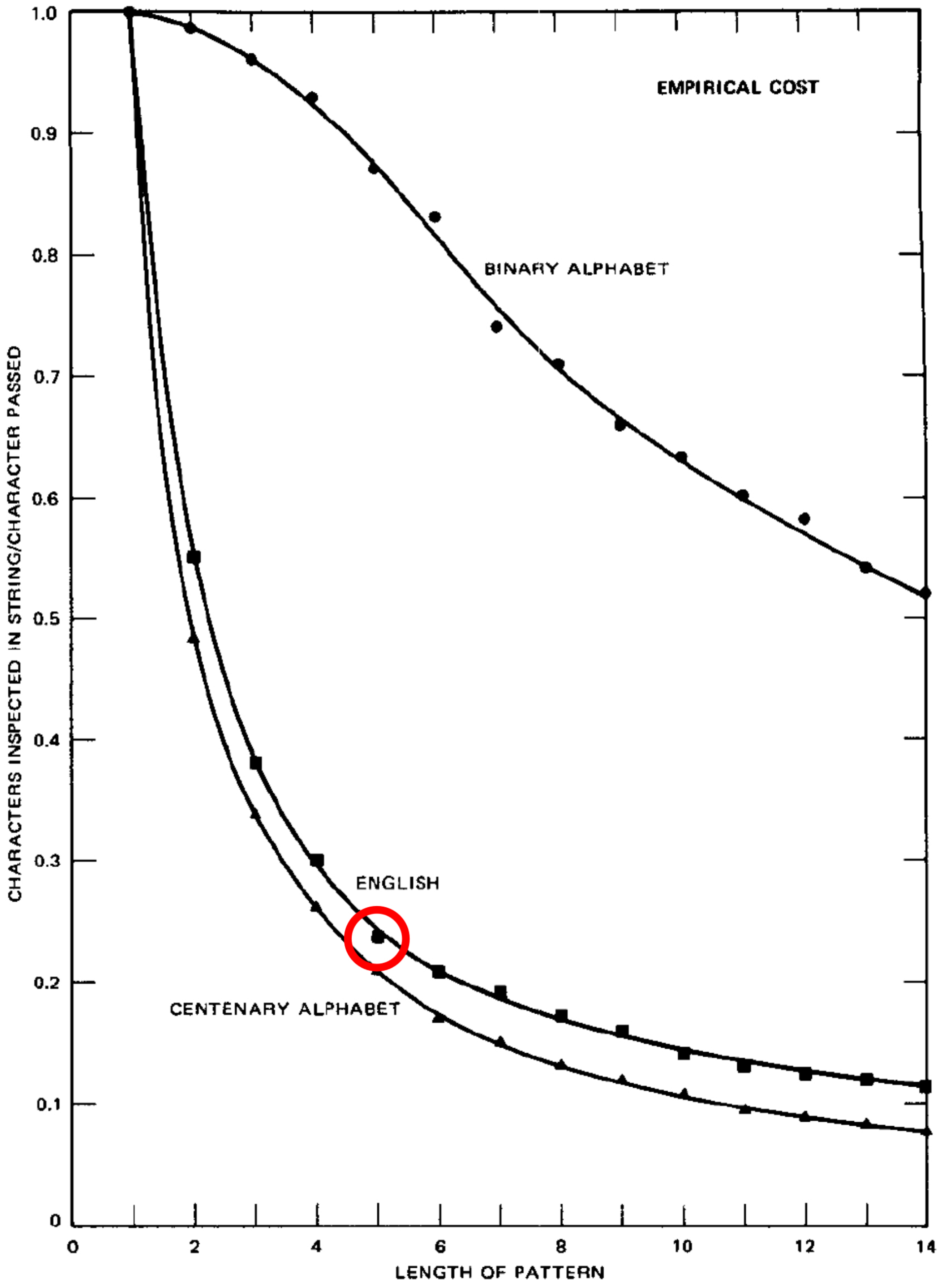


For each letter z, let
R(z) = distance from right-most occurrence of z in **P[1..,m–1]**, to the end of P (and |P| if there is no occurrence)

R(c) = 5

← correct definition of R(z)

BM – Average Case



To find first occurrence i
of an arbitrary 5-letter
word in an English text
Inspects on average

(0.25 * i)

text symbols.

In Practise

Experimental Map

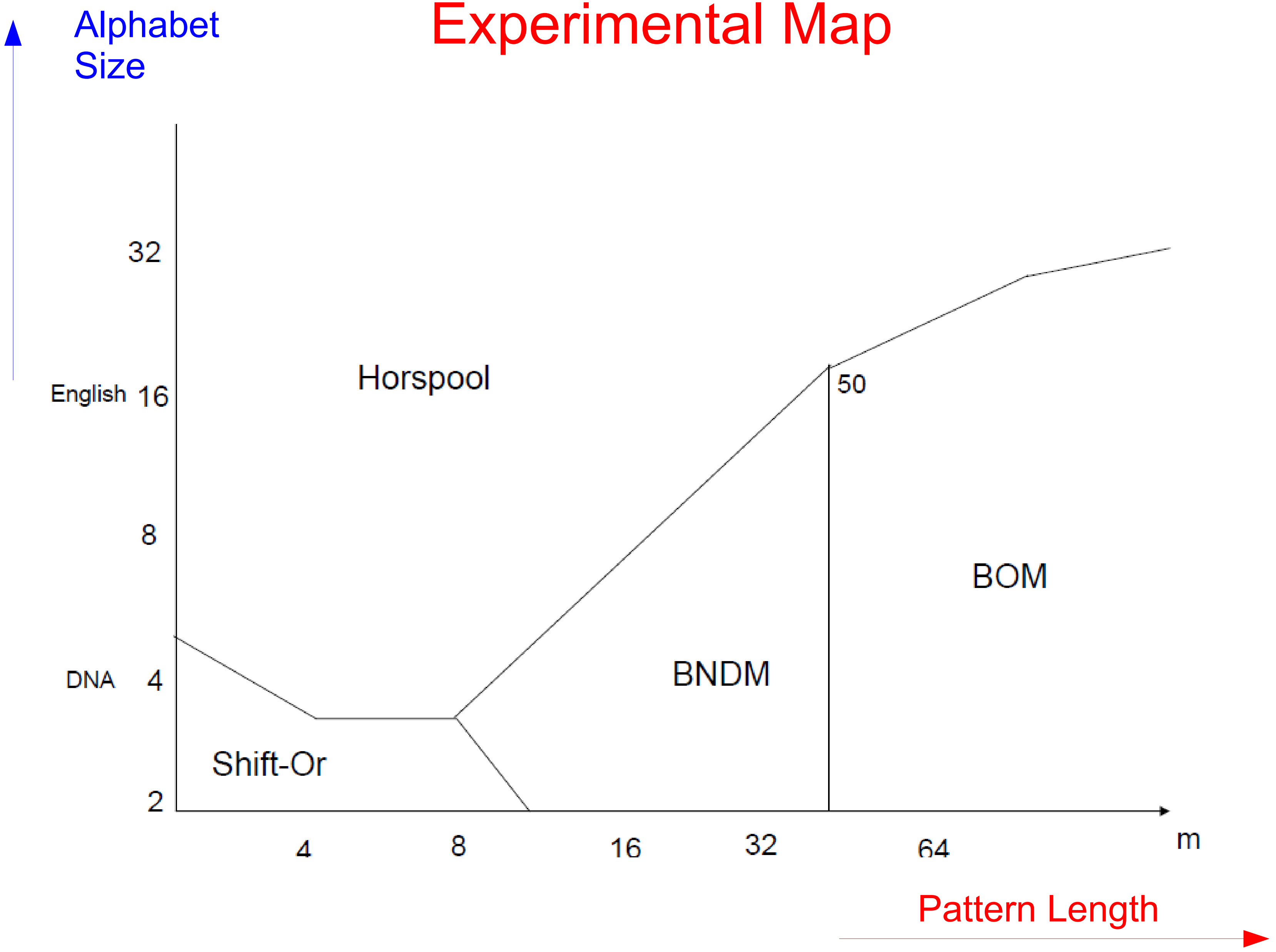
- → Random text (10MB) and random patterns
- → Tested all algorithms (KMP, BM, BDM, etc)
- → 32-bit machine (UltraSPARC)
- → Only 4 algorithms have a zone on the map.

Surprising:

- → Results on DNA are same as random for $|\Sigma|=4$
- → Results on English text are same as random for $|\Sigma|=16$ (!)

alphabet sizes

Experimental Map



END
Lecture 13