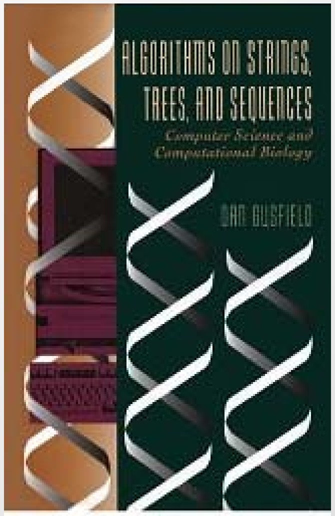
Exact sub-string matching in deterministic linear time O(n+m)

Prof. Manolis Kellis

IMPORTANT NOTE: sub-string matching does not include gaps. Sub-sequence matching, which includes gaps is O(n*m)



For more information see Dan Guslfied book

The exact matching problem

- Inputs:
  - a string ***P***, called the pattern
  - a longer string ***T***, called the text
- Output:
  - Find all occurrences, if any, of pattern ***P*** in text ***T***
- Example

P=

a	b	a

T=

b	a	a	b	a	c	a	b	a	b	a	d
1	2	3	4	5	6	7	8	9	10	11	12

Basic string definitions

S=

b	a	a	b	a	c	a	b	a	b	a	d
1	2	3	4	5	6	7	8	9	10	11	12

- A *string* S
  - Ordered list of characters
  - Written contiguously from left to write
- A *substring* S[i..j]
  - all contiguous characters from i to j
  - Example: S[3..7] = abaxa
- A *prefix* is a substring starting at 1
- A *suffix* is a substring ending at |S|
- |S| denotes the number of characters in string S

The naïve string-matching algorithm

- NAÏVE STRING MATCHING
  1 – n ← length[T]
  2 – m ← length[P]
  3 – **for** shift ← 0 **to** n
  4 • **do if** P[1..m] == T[shift+1 .. shift+m]
  5 – **then** print “Pattern occurs with shift” shift

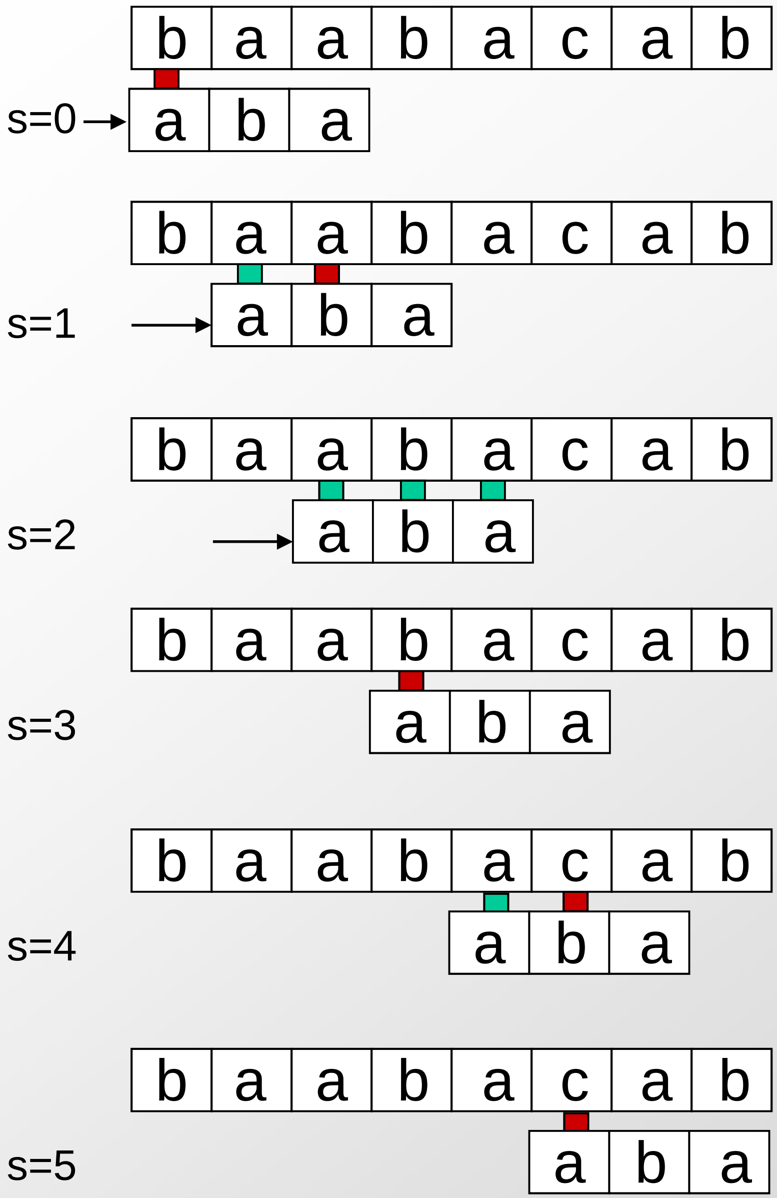
Running time:
O(n)
→ O(m)

- Where the test operation in line 4:
  - Tests each position in turn
    - If match, continue testing
    - else: stop
- Running time ~ number of comparisons

  number of shifts (with one comparison each)

  \+ number of successful character comparisons

Comparisons made with naïve algorithm

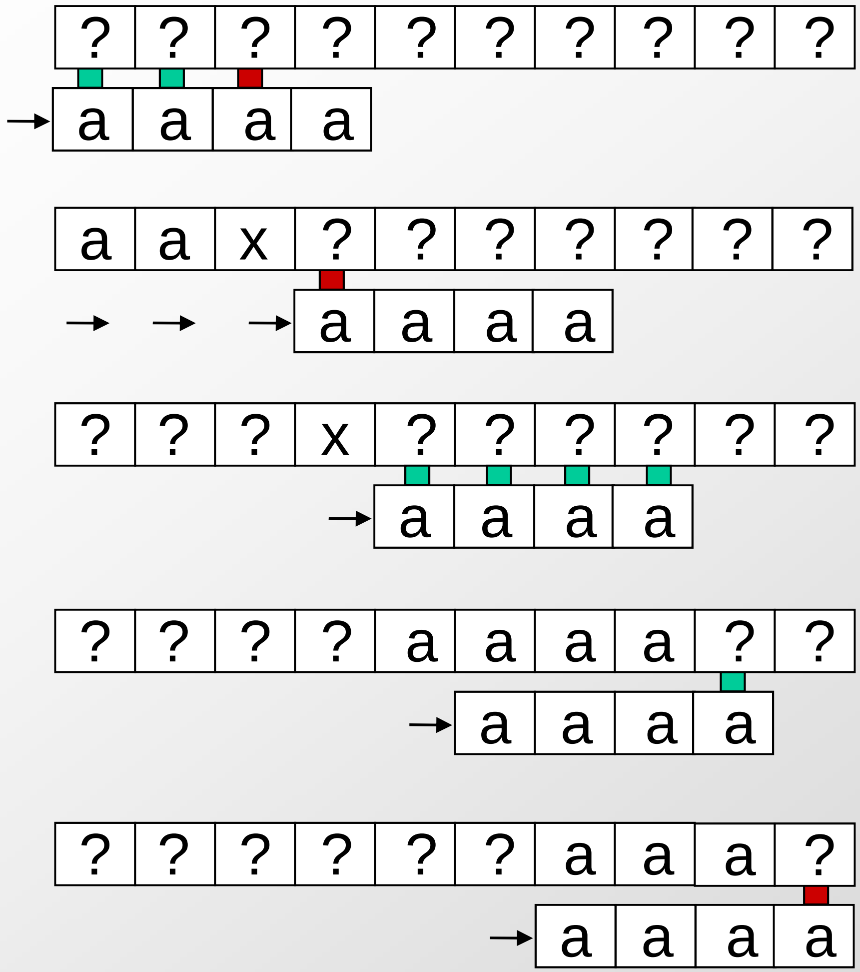


- Worst case running time:
  - Test every position
  - P=aaaa, T=aaaaaaaaaaa
- Best case running time:
  - Test only first position
  - P=bbbb, T=aaaaaaaaaaa

Can we do better?

Key insight: make bigger shifts!

- If all characters in the pattern are the **same**:



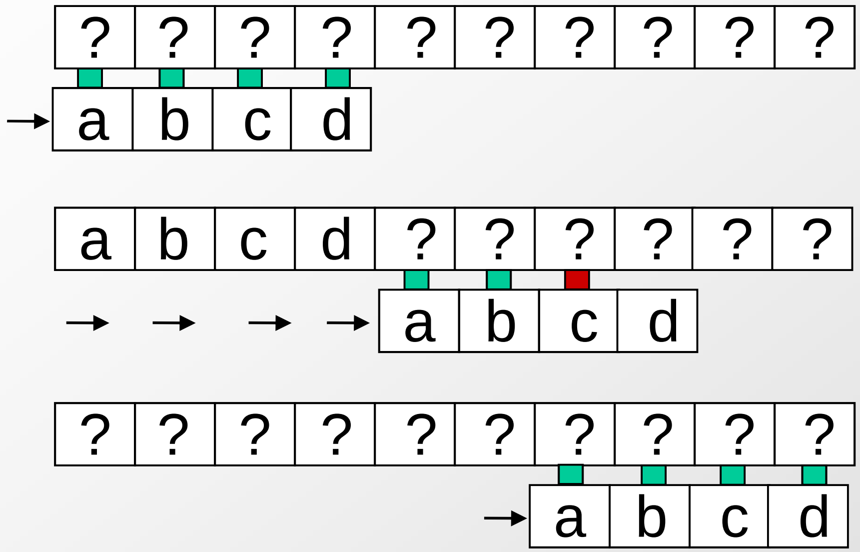
Information gathered at every comparison

Knowledge of the internal structure of P

Number of comparisons: O(n)

Key insight: make bigger shifts!

- If all characters in the pattern are **different**:



Number of comparisons:

•At most n matching comparisons

•At most n non-matching comparisons

➔ Number of comparisons: O(n)

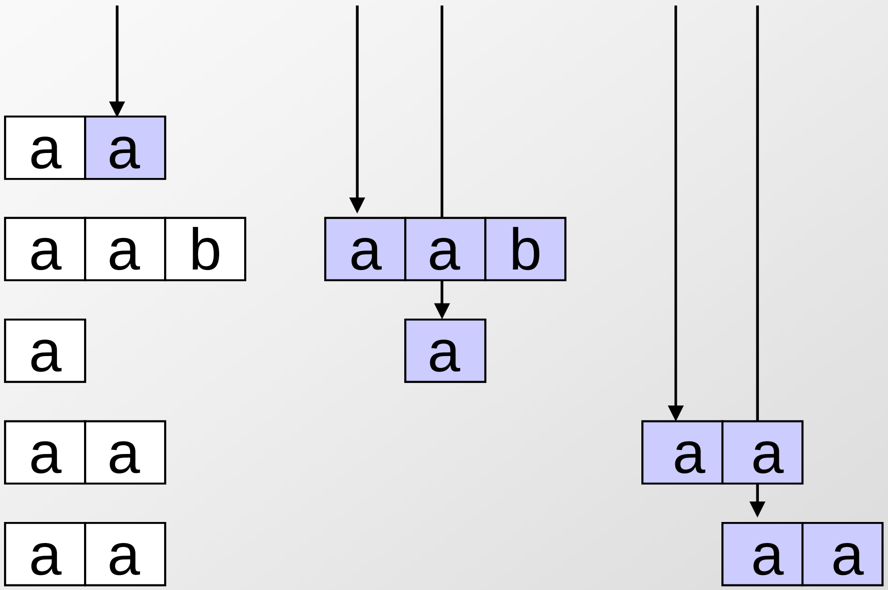
Key insight: make bigger shifts!

- Special case:
  - If all characters in the pattern are **the same**: O(n)
  - If all characters in the pattern are **different**: O(n)
- General case:
  - Learn internal redundancy structure of the pattern
  - Pattern pre-processing step
- Methods:
  - Fundamental pre-processing
  - Knuth-Morris-Pratt
  - Finite State Machine

Fundamental pre-processing

- Learning the redundancy structure of a string S

	1	2	3	4	5	6	7	8	9	10	11
S =	a	a	b	c	a	a	b	x	a	a	a
Z =	0	**1**	0	0	**3**	**1**	0	0	**2**	**2**	1



- Zi = length of longest prefix in common for S[i..] and S

(Length of the longest prefix of S[i..] that's also a prefix of S)

Fundamental pre-processing

- Learning the redundancy structure of a string S

	1	2	3	4	5	6	7	8	9	10	11
S =	a	a	b	c	a	a	b	x	a	a	a
Z =	0	**1**	0	0	**3**	**1**	0	0	**2**	**2**	1
Z-box =	a	a	b	c	a	a	b	x	a	a	a
r =	a	a	b	c	a	a	b	x	a	a	a
l =	a	a	b	c	a	a	b	x	a	a	a

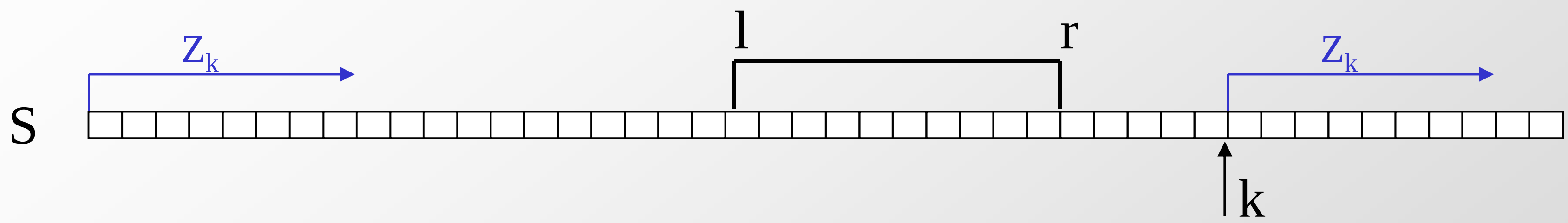
Z_1	Z_2	Z_3	…	Z_{k-1}	$\mathbf{Z_k}$					
a	a	b	c	a	a	b	x	a	a	a

left k right

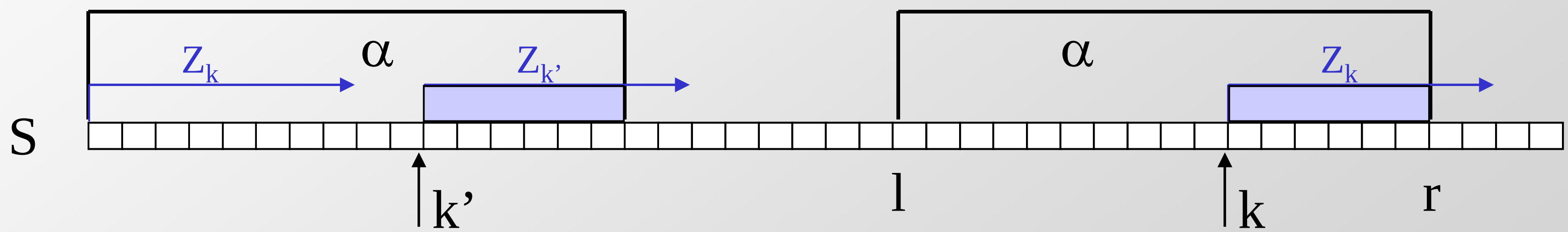
Can we compute Z, r, l in linear time O(|S|)?

Computing Z_k given Z_1 .. Z_{k-1}

- Case 1: k is outside a Z-box: simply compute Z_k



- Case 2: k is inside a Z-box: Look up $Z_{k'}$

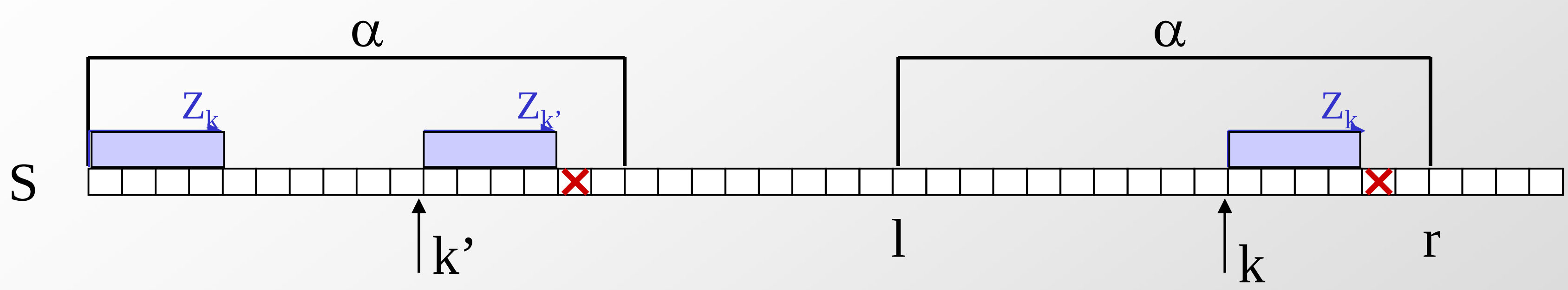


→ Case 2a: Zk' < r-k

→ Case 2b: Zk' >= r-k

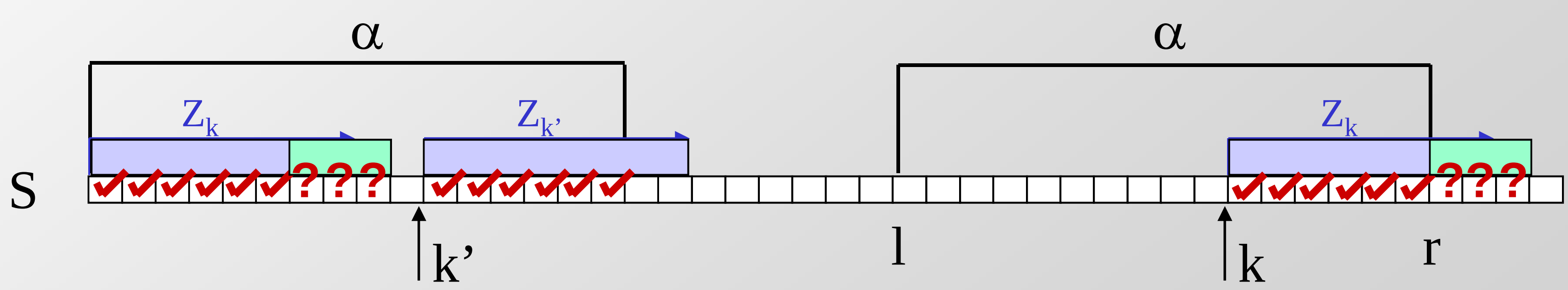
Computing Z_k given Z_1 .. Z_{k-1}

Case 2a: $Z_{k'} < r-k$



Set $Z_k = Z_{k'}$

Case 2b: $Z_{k'} >= r-k$



Explicitly compare starting at r+1

Putting it all together

- FUNDAMENTAL-PREPROCESSING(S):

Z_2,l,r = explicitly compare S[1..] with S[2..]

for k in 2..n:

if k > r: Z_k,l,r = explicitly compare S[1..] with S[k..]

if k <= r:

if $Z_{k'}$<(r-k): $Z_k = Z_{k'}$

else:

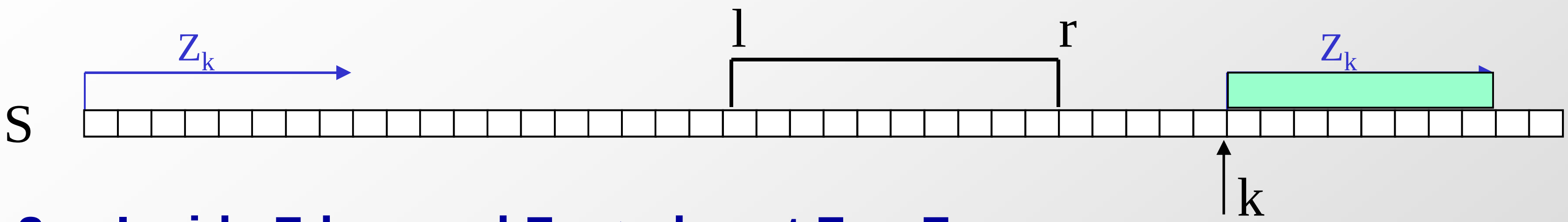
Z_k = explicitly compare S[r+1..] with S[(r-k)+1..]

l = k

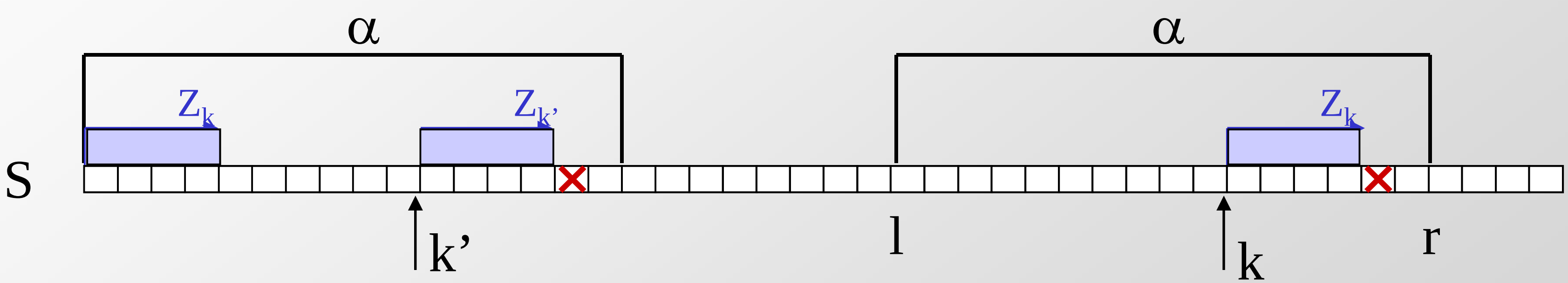
r = l+Z_k

Correctness of Z computation

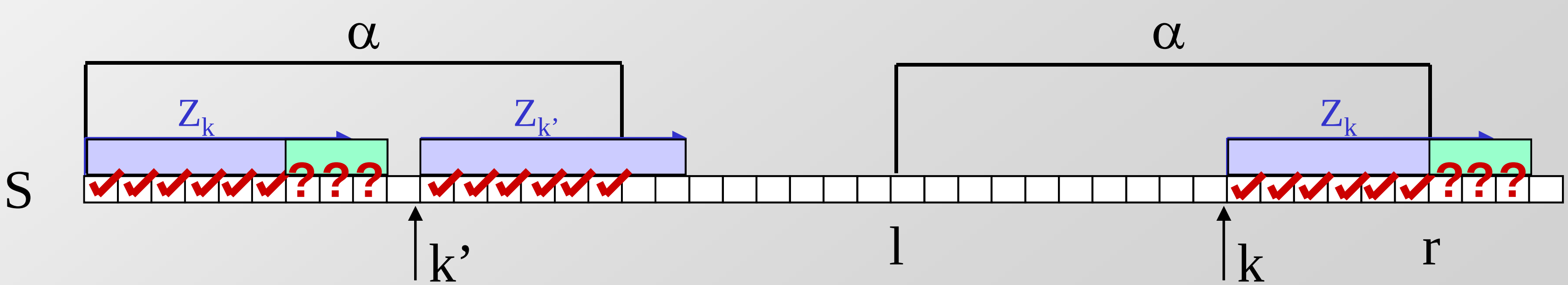
Case 1: k is outside a Z-box: explicitly compute Z_k



Case 2a: Inside Z-box and $Z_{k'}$ < r-k: set Z_k = $Z_{k'}$

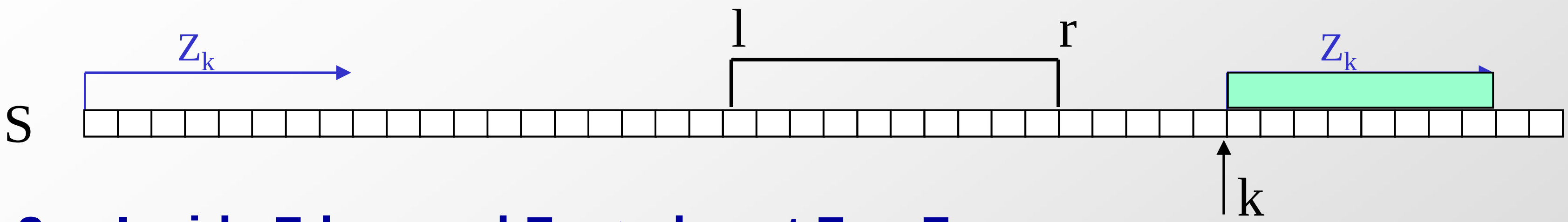


Case 2b: Inside Z-box and $Z_{k'}$ >= r-k: explicitly compute starting at r+1

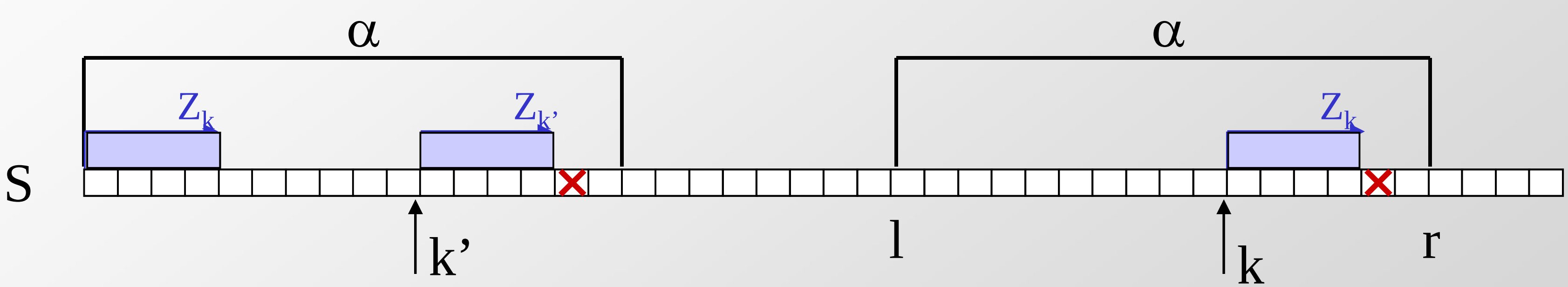


Running time of Z computation

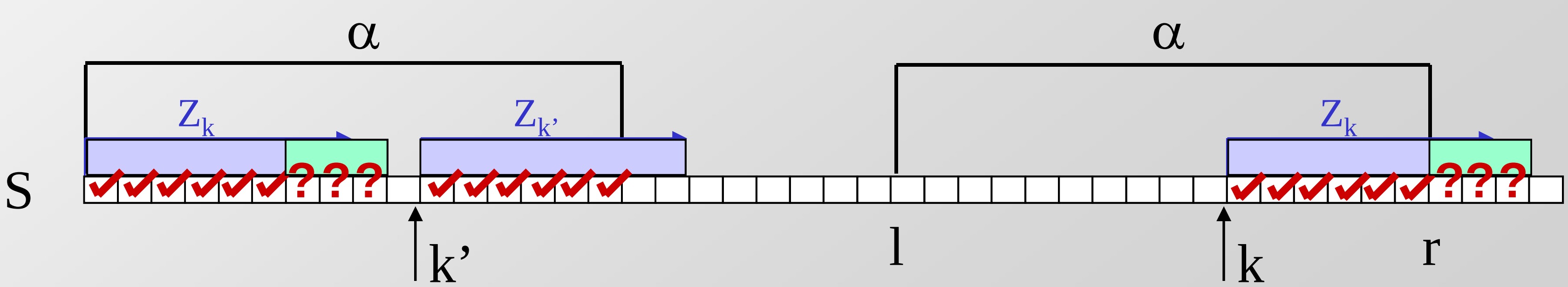
Case 1: k is outside a Z-box: explicitly compute Z_k



Case 2a: Inside Z-box and $Z_{k'} < r\text{-}k$: set $Z_k = Z_{k'}$



Case 2b: Inside Z-box and $Z_{k'} >= r\text{-}k$: explicitly compute starting at r+1



What’s so fundamental about Z?

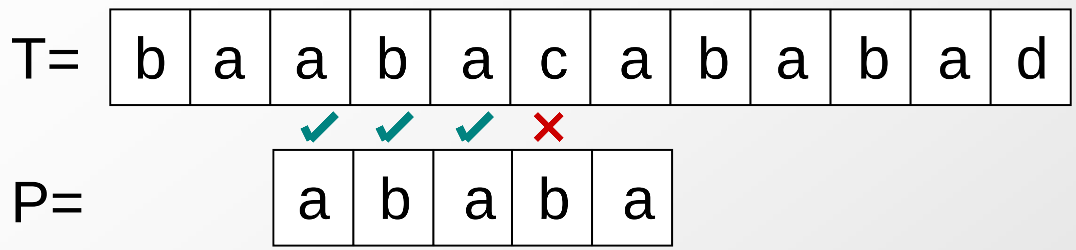
- Learning the redundancy structure of a string S

S =	a	a	b	c	a	a	b	x	a	a	a
Z =	0	**1**	0	0	**3**	**1**	0	0	**2**	**2**	0

a	a	b		a	a	b

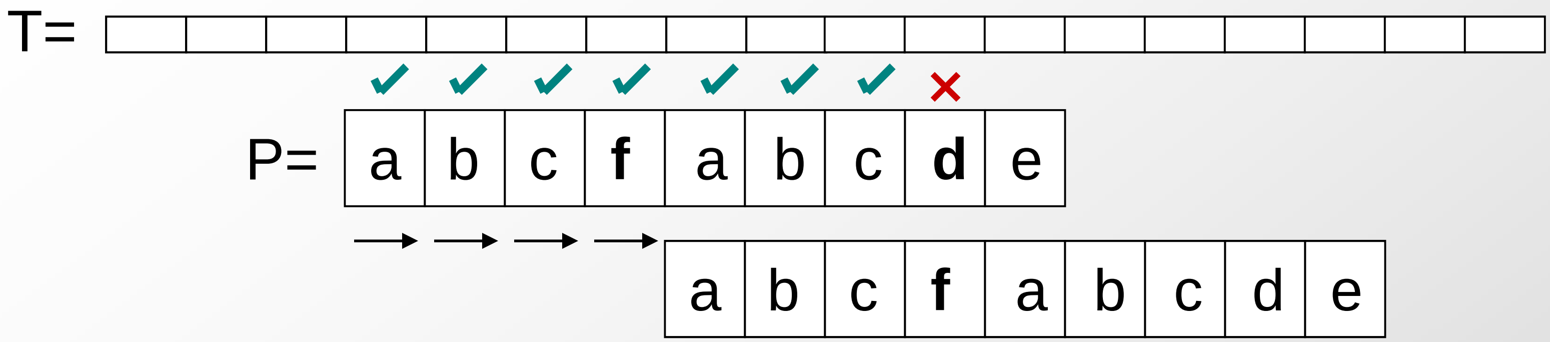
- Z_i = fundamental property of internal redundancy structure
- Most pre-processings can be expressed in terms of Z
  - Length of the longest **prefix** starting/ending at position i
  - Length of the longest **suffix** starting/ending at position i

Back to string matching

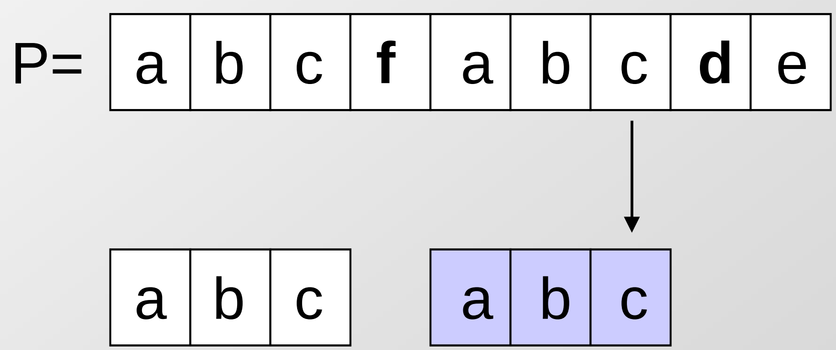


- Given the fundamental pre-processing of pattern P
  - Compare pattern P to text T
  - Shift P by larger intervals based on values of Z
- Three algorithms based on these ideas
  - Knuth-Morris-Pratt algorithm
  - Boyer-Moore algorithm
  - Z algorithm

Knuth-Morris-Pratt algorithm

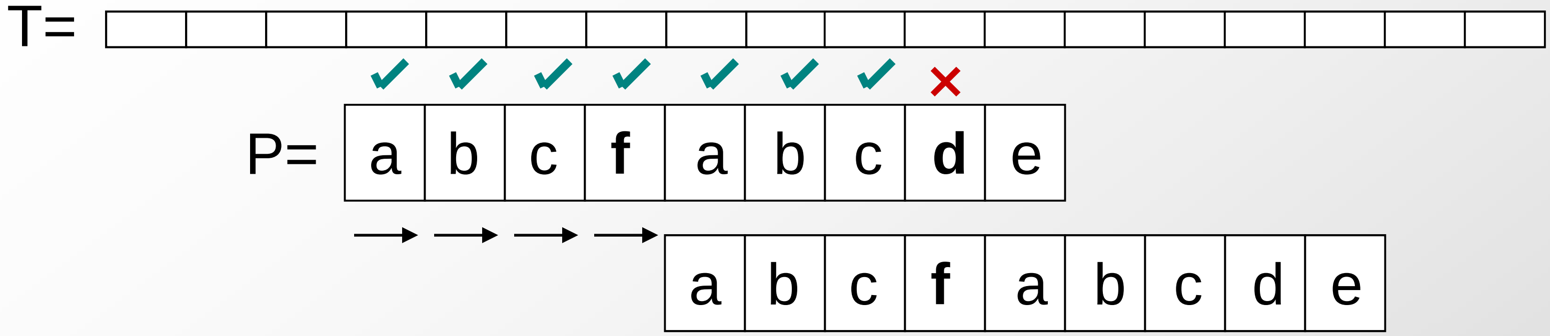


- Pre-processing:
  - $Sp_i(P)$ = length of longest proper suffix of P[1..i] that matches a prefix of P



– No other than the right-hand-side of the Z-boxes

Knuth-Morris-Pratt running time



- Number of comparisons bounded by characters in T
  - Every comparison starts at text position where last comparison ended
  - Every shift results in at most one extra comparison
  - At most |T| shifts → Running time bounded by 2*|T|

Boyer-Moore algorithm

T=	b	a	a	b	x	c	a	b	a	b	a	d
P=	a	b	a	b	x							

- Three fundamental ideas:
  1. Right-to-left comparison
  2. Alphabet-based shift rule
  3. Preprocessing-based shift rule
- Results in:
  - Very good algorithm in practice
  - Rule 2 results in large shifts and sub-linear time
    - for larger alphabets, ex: English text
  - Rule 3 ensures worst-case linear behavior
    - even in small alphabets, ex: DNA sequences

The Z algorithm

P+T=	a	b	a	b	a	**$**	b	a	a	b	a	c	a	b	a	b	a	d

- The Z algorithm
  - Concatenate P + ‘$’ + T
  - Compute fundamental pre-processing O(m+n)
  - Report all starting positions i for which Z_i=|P|