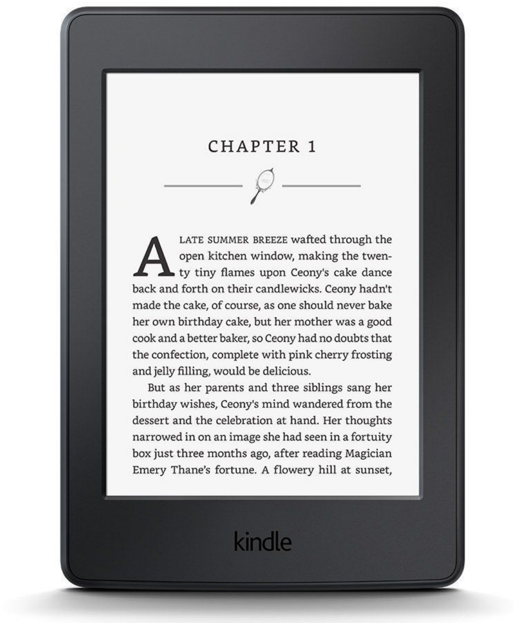
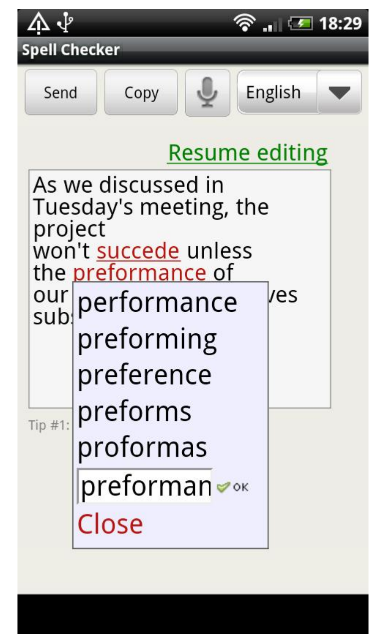
CSE202
Design and Analysis of Algorithms

Week 11 — String Algorithms 1 - Search

Strings Everywhere






```
class Node:

    def __init__(self,key,left=
        self.key = key
        self.left = left
        self.right = right
        self.size = 1

    def __str__(self):
        if self is None: return
        return str(self.left)+"


class BST:

    def __init__(self):
```

Definitions:
letter=character=symbol (usually 7,8, or 16 bits);
alphabet: set of letters (often denoted Σ and $R = |\Sigma|$);
string=word=text: *finite* sequence of letters;
length=size of a word: number of letters.

Substring Search

Notation:
$|T| = n$,
$|P| = m$.

Input: two strings (text T and pattern P)

Output: answer to "is P a substring of T?"

$\exists i, \forall j \in \{0, \ldots, m-1\}, T_{i+j} = P_j$.

Aim: *small number of character accesses*

Substring Search

Notation:
$|T| = n$,
$|P| = m$.

Input: two strings (text T and pattern P)

Output: answer to "is P a substring of T?"

$\exists i, \forall j \in \{0, \ldots, m-1\}, T_{i+j} = P_j$.

Aim: *small number of character accesses*

Algorithms of the Day:

	worst case	average case
Brute Force	$\leq nm$	$\leq 2(n-m+1)$
Knuth-Morris-Pratt	$\leq n+m$	$\geq n$
Boyer-Moore	$\leq 3n$	$\approx n/m$

Recall also Rabin-Karp from tutorial 7

Substring Search

Notation:
$|T| = n$,
$|P| = m$.

Input: two strings (text T and pattern P)

Output: answer to "is P a substring of T?"

$\exists i, \forall j \in \{0, \ldots, m-1\}, T_{i+j} = P_j$.

Aim: *small number of character accesses*

Algorithms of the Day:

	worst case	average case
Brute Force	$\leq nm$	$\leq 2(n - m + 1)$
Knuth-Morris-Pratt	$\leq n + m$	$\geq n$
Boyer-Moore	$\leq 3n$	$\approx n/m$

difficult, not done here

Recall also Rabin-Karp from tutorial 7

I. Brute Force

Algorithm

```
def bruteforce(text,pattern):
    for i in range(len(text)-len(pattern)):
        for j in range(len(pattern)):
            if text[i+j]!=pattern[j]: break
        else: return i
    return -1
```

Worst-case: $P = a^{m-1}b,\ T = a^{n-1}b$

$\rightarrow m(n - m + 1)$ comparisons

Expected Number of Comparisons for All Matches

Fixed pattern, uniform random text

$$\text{Expectation} \leq \frac{n - m + 1}{1 - 1/R}$$

Expected Number of Comparisons for All Matches

Fixed pattern, uniform random text

texts of length n having at least the k first letters of the pattern at a given location: R^{n-k}

Expectation $\leq \dfrac{n-m+1}{1-1/R}$

Expected Number of Comparisons for All Matches

Fixed pattern, uniform random text

texts of length n having at least the k first letters of the pattern at a given location: R^{n-k}

having exactly the k first letters: $R^{n-k} - R^{n-k-1}$

Expectation $\leq \dfrac{n-m+1}{1-1/R}$

Expected Number of Comparisons for All Matches

Fixed pattern, uniform random text

\# texts of length n having at least the k first letters of the pattern at a given location: R^{n-k}

\# having exactly the k first letters: $R^{n-k} - R^{n-k-1}$

\# comparisons at this location: $\sum_{k=0}^{m} (k+1)(R^{n-k} - R^{n-k-1})$

Expectation $\leq \dfrac{n-m+1}{1-1/R}$

Expected Number of Comparisons for All Matches

Fixed pattern, uniform random text

texts of length n having at least the k first letters of the pattern at a given location: R^{n-k}

having exactly the k first letters: $R^{n-k} - R^{n-k-1}$

comparisons at this location:
$$\sum_{k=0}^{m} (k+1)(R^{n-k} - R^{n-k-1}) \leq \frac{R^n}{1 - 1/R}$$

Expectation $\leq \dfrac{n - m + 1}{1 - 1/R}$

Expected Number of Comparisons for All Matches

$$\sum_{k=0}^{m} (k+1)(R^{n-k} - R^{n-k-1})$$

Expected Number of Comparisons for All Matches

$$\sum_{k=0}^{m} (k+1)(R^{n-k} - R^{n-k-1})$$

$$= R^n - R^{n-1} + {\color{red}2}(R^{n-1} - R^{n-2}) + {\color{red}3}(R^{n-2} - R^{n-2}) + \ldots + {\color{red}(m+1)}(R^{n-m} - R^{n-m-1})$$

$$R^{n-1} \qquad R^{n-2} \qquad R^{n-m}$$

Expected Number of Comparisons for All Matches

$$\sum_{k=0}^{m} (k+1)(R^{n-k} - R^{n-k-1})$$

$$= R^n - R^{n-1} + 2(R^{n-1} - R^{n-2}) + 3(R^{n-2} - R^{n-2}) + \ldots + (m+1)(R^{n-m} - R^{n-m-1})$$

R^{n-1} R^{n-2} R^{n-m}

$$= R^n + R^{n-1} + R^{n-2} + \ldots + R^{n-m} - (m+1)R^{n-m-1}$$

Expected Number of Comparisons for All Matches

$$\sum_{k=0}^{m} (k+1)(R^{n-k} - R^{n-k-1})$$

$$= R^n - R^{n-1} + 2(R^{n-1} - R^{n-2}) + 3(R^{n-2} - R^{n-2}) + \ldots + (m+1)(R^{n-m} - R^{n-m-1})$$

$$R^{n-1} \qquad R^{n-2} \qquad R^{n-m}$$

$$= R^n + R^{n-1} + R^{n-2} + \ldots + R^{n-m} - (m+1)R^{n-m-1}$$

$$\leq R^n + R^{n-1} + R^{n-2} + \ldots + R^{n-m}$$

Expected Number of Comparisons for All Matches

$$\sum_{k=0}^{m}(k+1)(R^{n-k}-R^{n-k-1})$$

$$= R^n - R^{n-1} + 2(R^{n-1} - R^{n-2}) + 3(R^{n-2} - R^{n-2}) + \ldots + (m+1)(R^{n-m} - R^{n-m-1})$$

$$R^{n-1} \qquad\qquad R^{n-2} \qquad\qquad R^{n-m}$$

$$= R^n + R^{n-1} + R^{n-2} + \ldots + R^{n-m} - (m+1)R^{n-m-1}$$

$$\leq R^n + R^{n-1} + R^{n-2} + \ldots + R^{n-m}$$

$$\leq \quad \frac{R^{n+1} - R^{n-m}}{R-1} \quad \leq \quad \frac{R^{n+1}}{R-1} \quad \leq \quad \frac{R^n}{1-1/R}$$

Expected Number of Comparisons for All Matches

Fixed pattern, uniform random text

texts of length n having at least the k first letters of the pattern at a given location: R^{n-k}

having exactly the k first letters: $R^{n-k} - R^{n-k-1}$

comparisons at this location: $\sum_{k=0}^{m} (k+1)(R^{n-k} - R^{n-k-1}) \leq \frac{R^n}{1 - 1/R}$

comparisons at all locations: $\leq \frac{(n-m+1)R^n}{1 - 1/R}$

some texts are counted several times

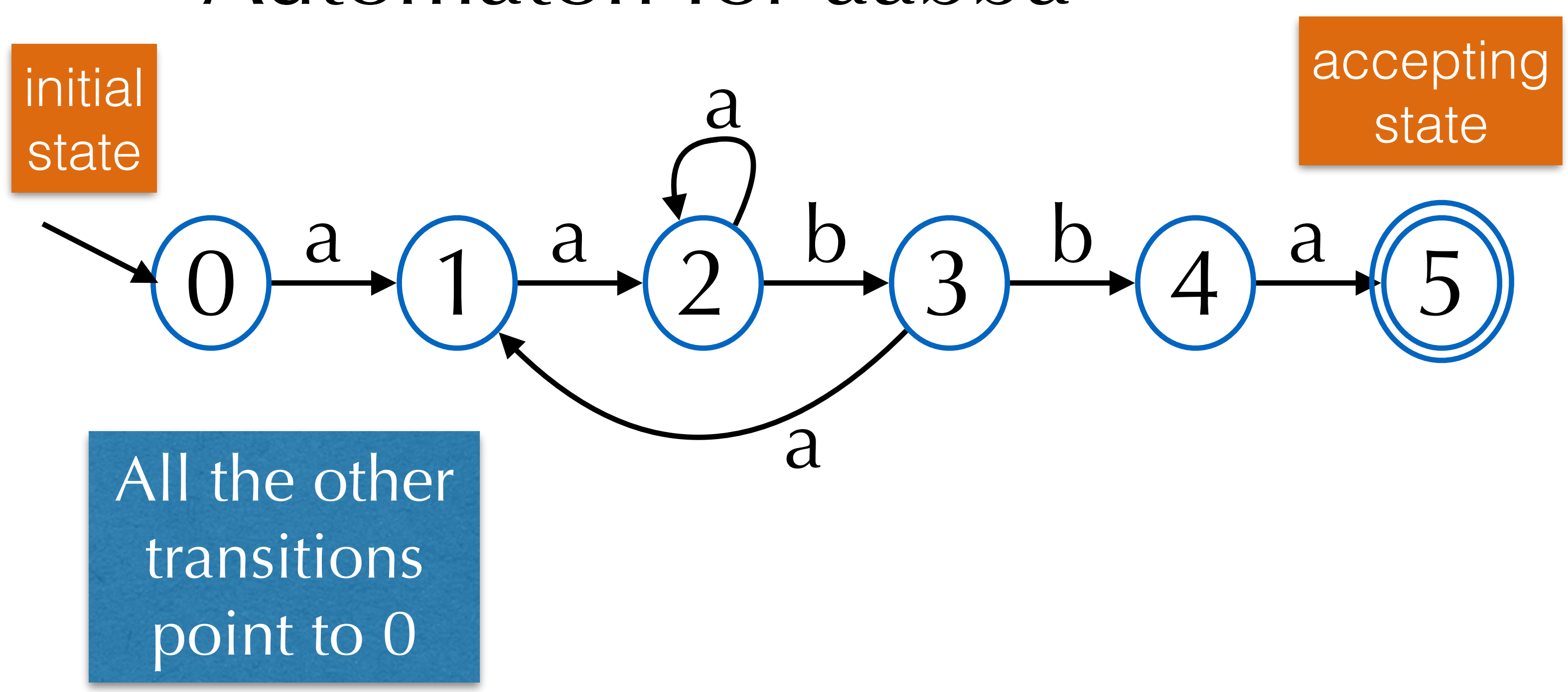
Expectation $\leq \frac{n-m+1}{1 - 1/R}$

II. Knuth-Morris-Pratt

Exploit the structure of the pattern

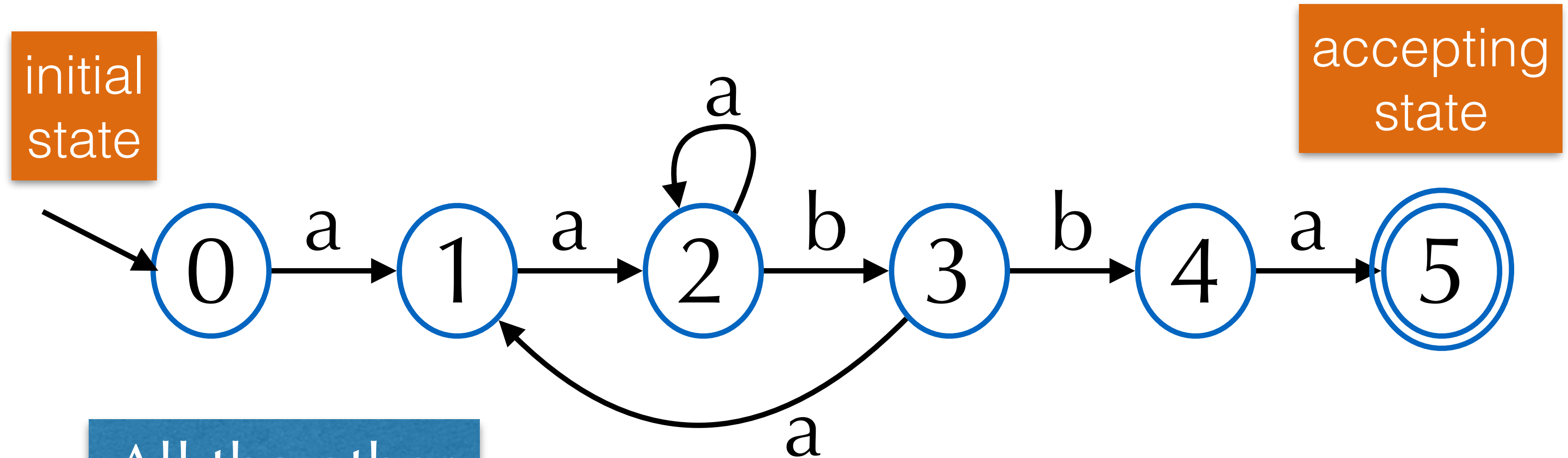
Pattern Compiled into Automaton

Automaton for aabba



Pattern Compiled into Automaton

Automaton for aabba

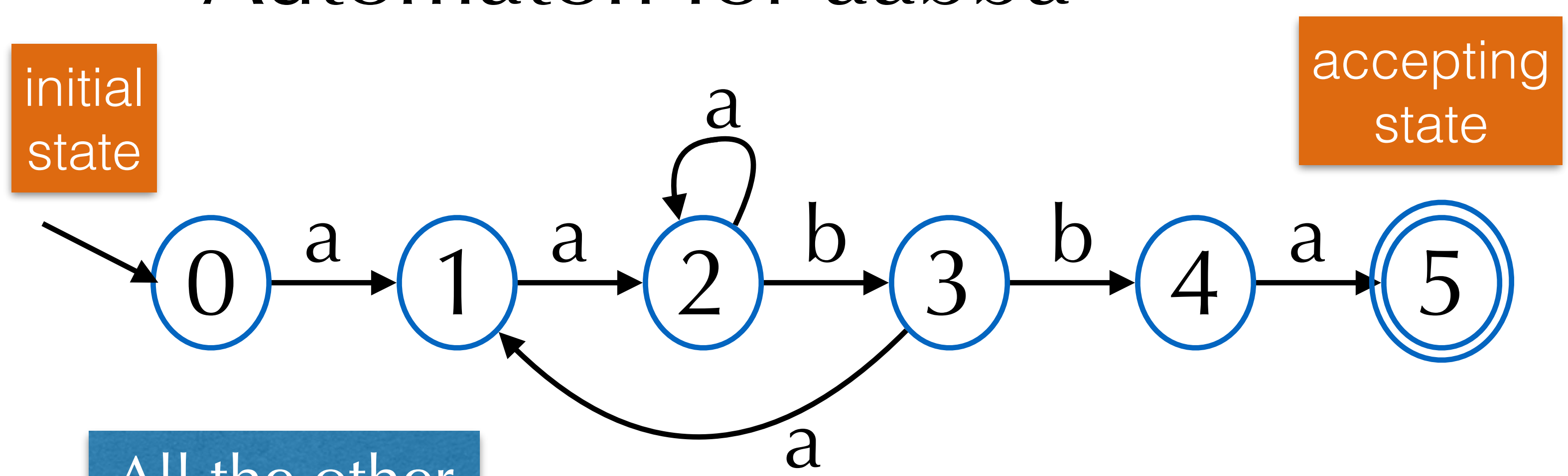


All the other transitions point to 0

Text ex.: b b a b a a a b a a b b a b b
0 0 0 1 0 1 2 2 3 1 2 3 4 5

Pattern Compiled into Automaton

Automaton for aabba



All the other transitions point to 0

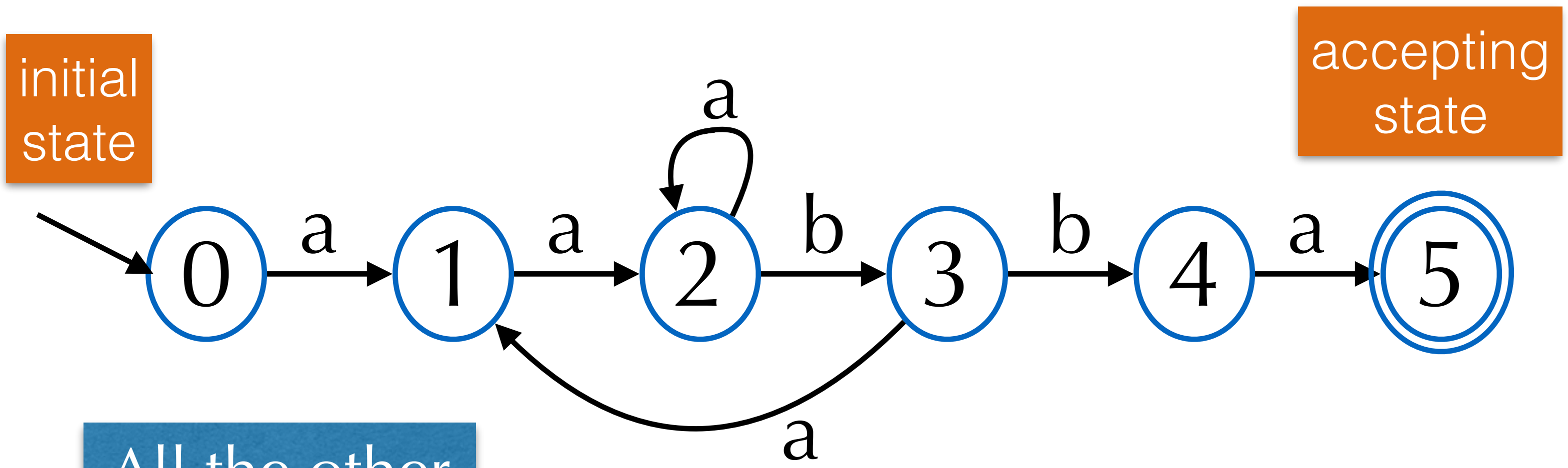
Text ex.: b b a b a a a b a a b b a b b
0 0 0 1 0 1 2 2 3 1 2 3 4 5

Def. Deterministic Finite Automaton

- a finite set Q of *states*;
- a *transition function* $\delta : Q \times \Sigma \to Q$;
- an *initial* state;
- one or several *accepting* states.

Pattern Compiled into Automaton

Automaton for aabba



All the other transitions point to 0

```python
def kmp(text,dfa):
    m=len(dfa)
    s=0
    for i in range(len(text)):
        s=dfa[s].get(text[i],0)
        if s==m: return i
    return -1
```

If the alphabet is small, dictionaries are replaced by arrays.

Text ex.: b b a b a a a b a a b b a b b

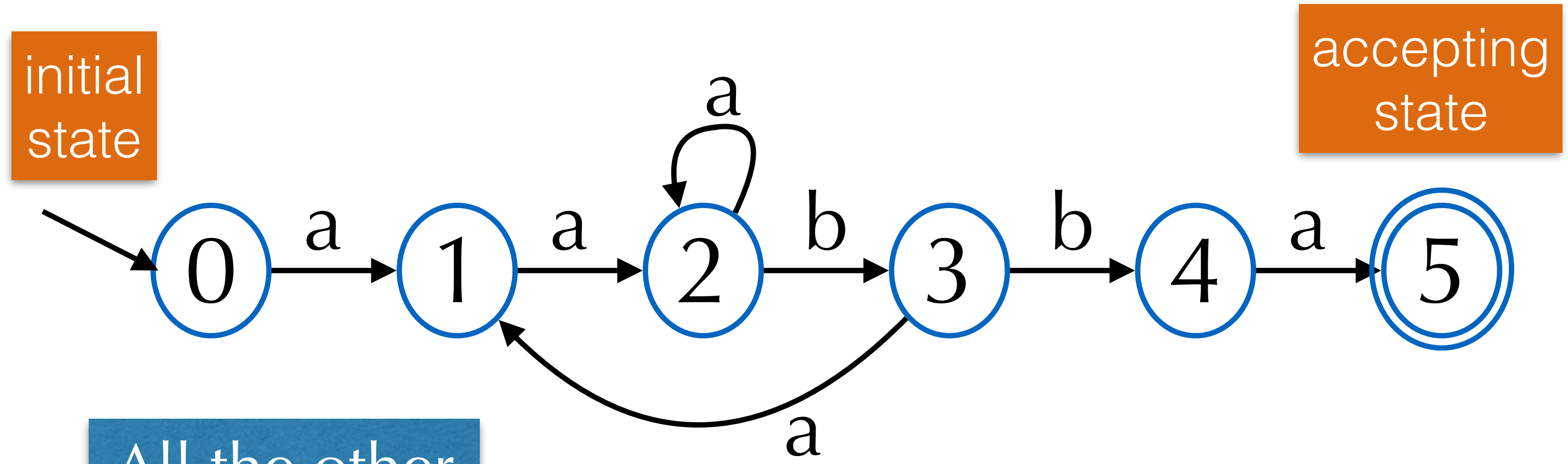
0 0 0 1 0 1 2 2 3 1 2 3 4 5

Def. Deterministic Finite Automaton

- a finite set Q of *states*;
- a *transition function* $\delta : Q \times \Sigma \to Q$;
- an *initial* state;
- one or several *accepting* states.

Pattern Compiled into Automaton

Automaton for aabba



All the other transitions point to 0

```
def kmp(text,dfa):
    m=len(dfa)
    s=0
    for i in range(len(text)):
        s=dfa[s].get(text[i],0)
        if s==m: return i
    return -1
```

If the alphabet is small, dictionaries are replaced by arrays.

Text ex.: b b a b a a a b a a b b a b b
0 0 0 1 0 1 2 2 3 1 2 3 4 5

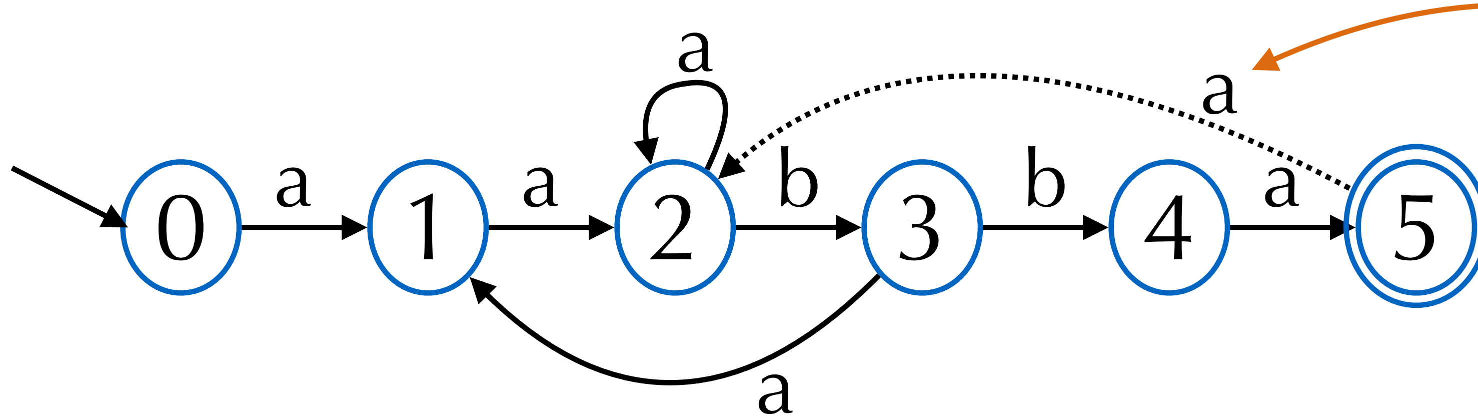
Def. Deterministic Finite Automaton

- a finite set Q of *states*;
- a *transition function* $\delta : Q \times \Sigma \to Q$;
- an *initial* state;
- one or several *accepting* states.

Each letter of the text is accessed once

KMP works on streams: never looks back

Computation of the Transitions



necessary if not looking only for the first match

When a match fails at index i in the pattern,
$i - 1$ characters of the text are known
$\longrightarrow$ imagine starting over from the 2nd one

```
def preprocess(pattern):
    m=len(pattern)
    dfa=[{} for i in range(m)]
    dfa[0][pattern[0]]=1
    state=0
    for i in range(1,m):
        for key in dfa[state]: dfa[i][key]=dfa[state][key]
        state=dfa[state].get(pattern[i],0)
        dfa[i][pattern[i]]=i+1
    return dfa
```

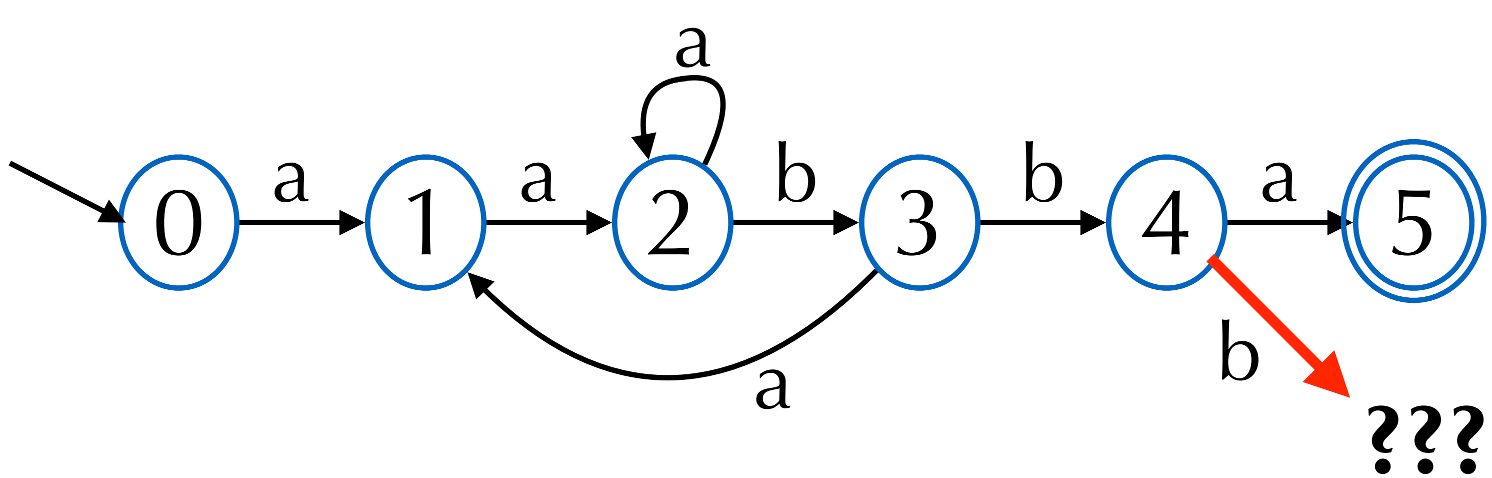
state reads pattern[1:m]

$O(m)$ operations

Exercise:
execute it step-by-step on this example

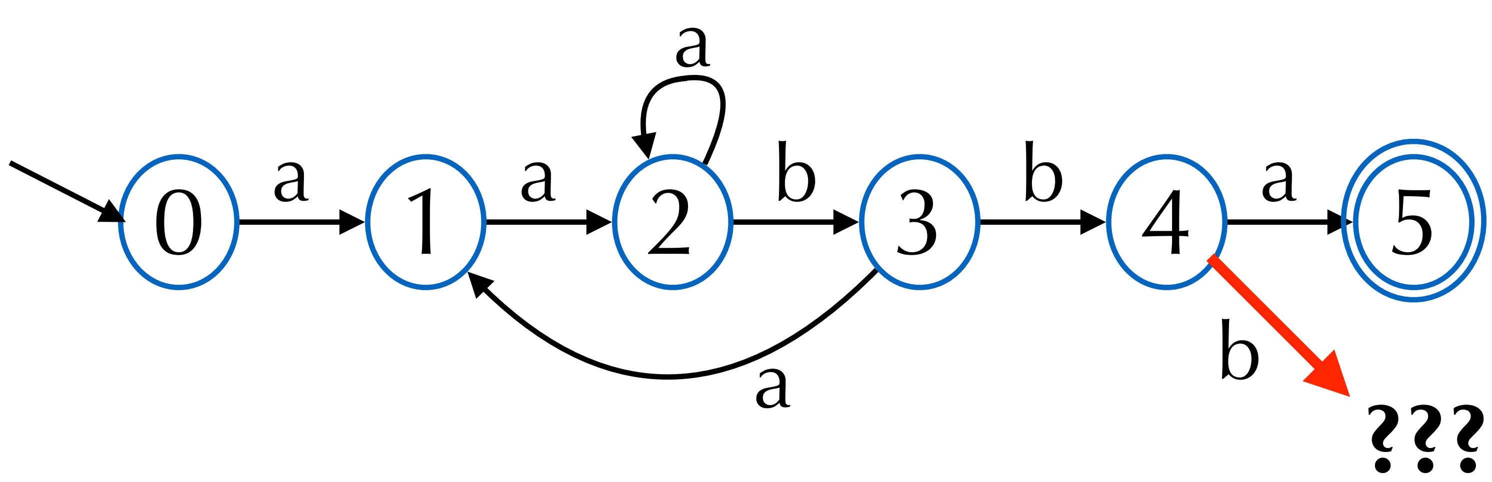
Computation of the Transitions

Reading a b from state 4 ?



Computation of the Transitions

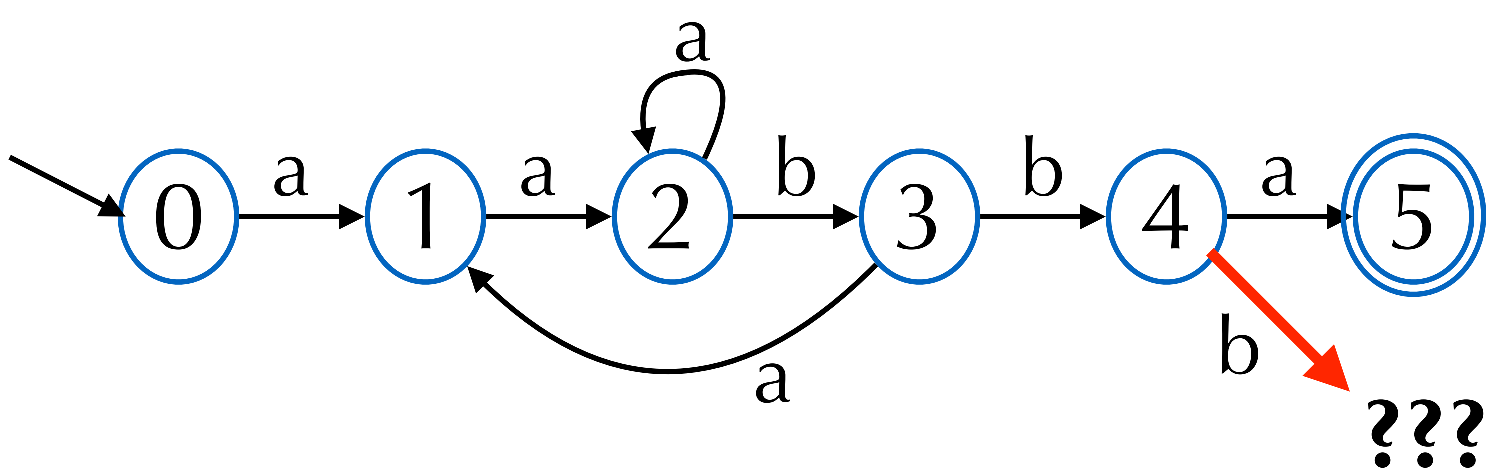
Reading a b from state 4 ?



The last 4 characters I have read: … aabb

Computation of the Transitions

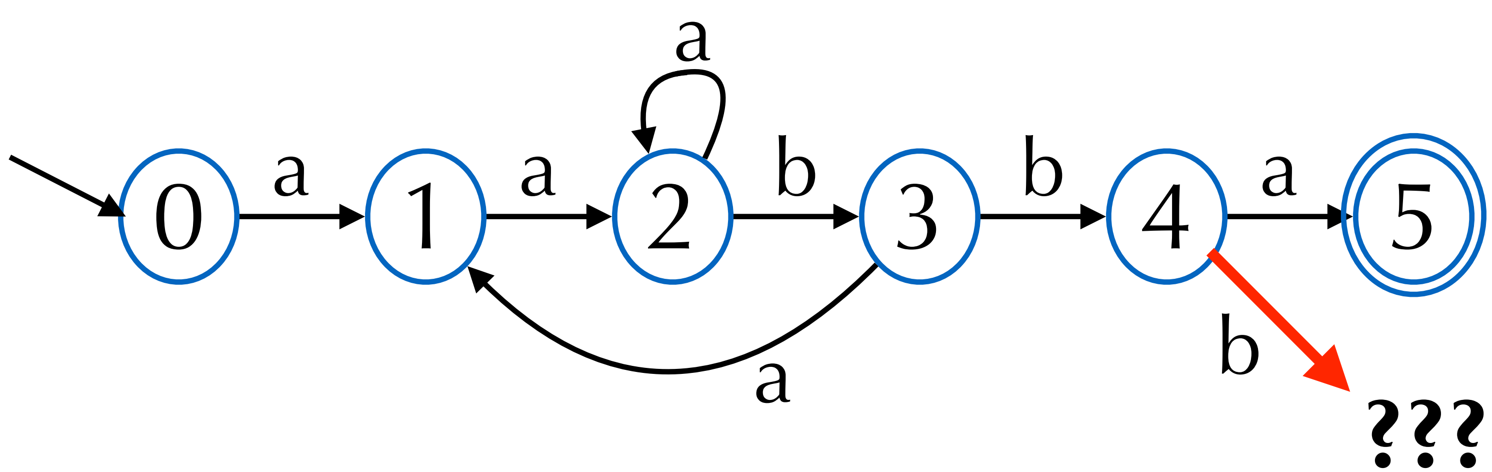
Reading a b from state 4 ?



The last 4 characters I have read: … aabb b

Computation of the Transitions

Reading a b from state 4 ?



The last 4 characters I have read: … aabb b

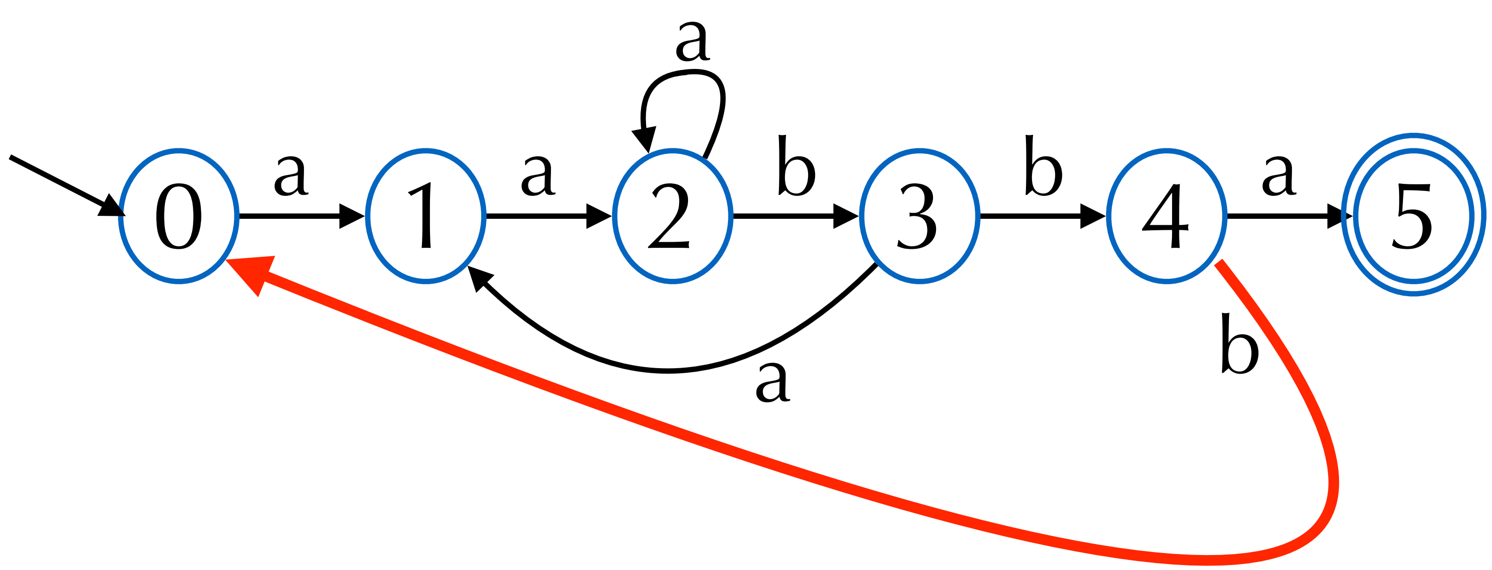
The pattern is not present.

What state if I started one position later ?

- reading abbb from state 0 -> state 0

Computation of the Transitions

Reading a b from state 4 ?



The last 4 characters I have read: … aabb b

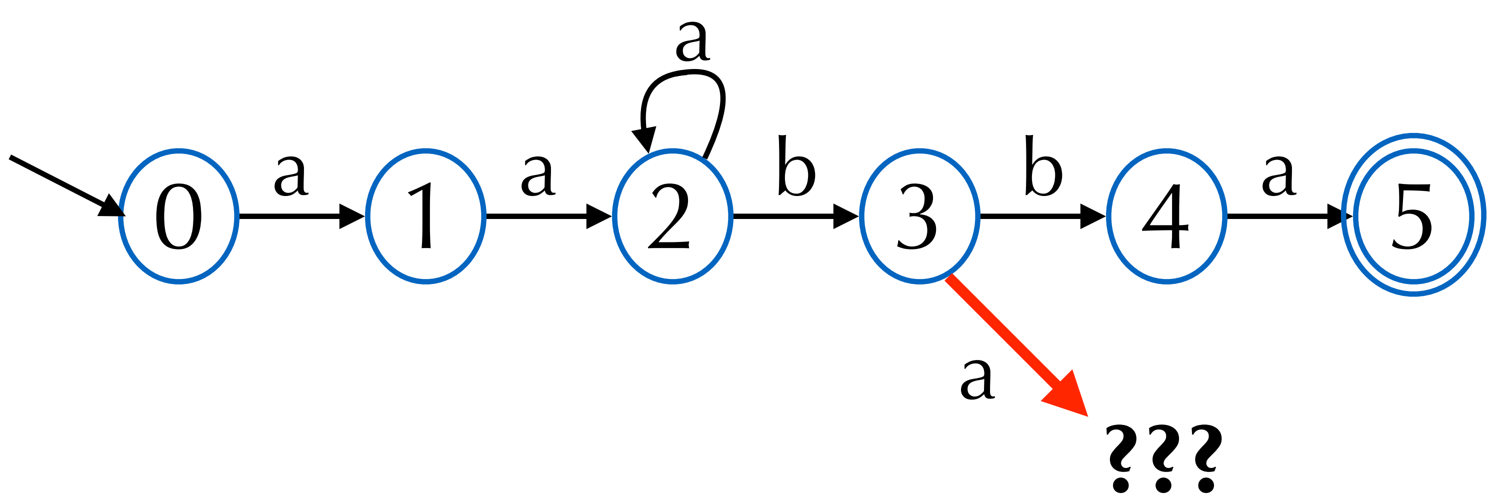
The pattern is not present.

What state if I started one position later ?

- reading abbb from state 0 -> state 0

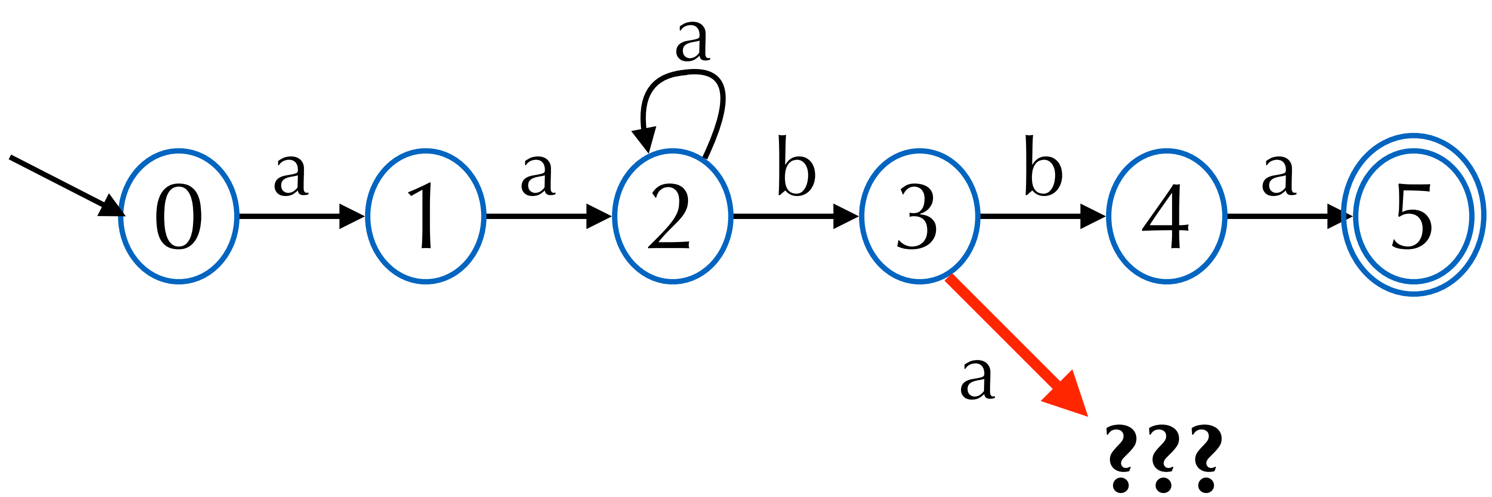
Computation of the Transitions

Reading an a from state 3 ?



Computation of the Transitions

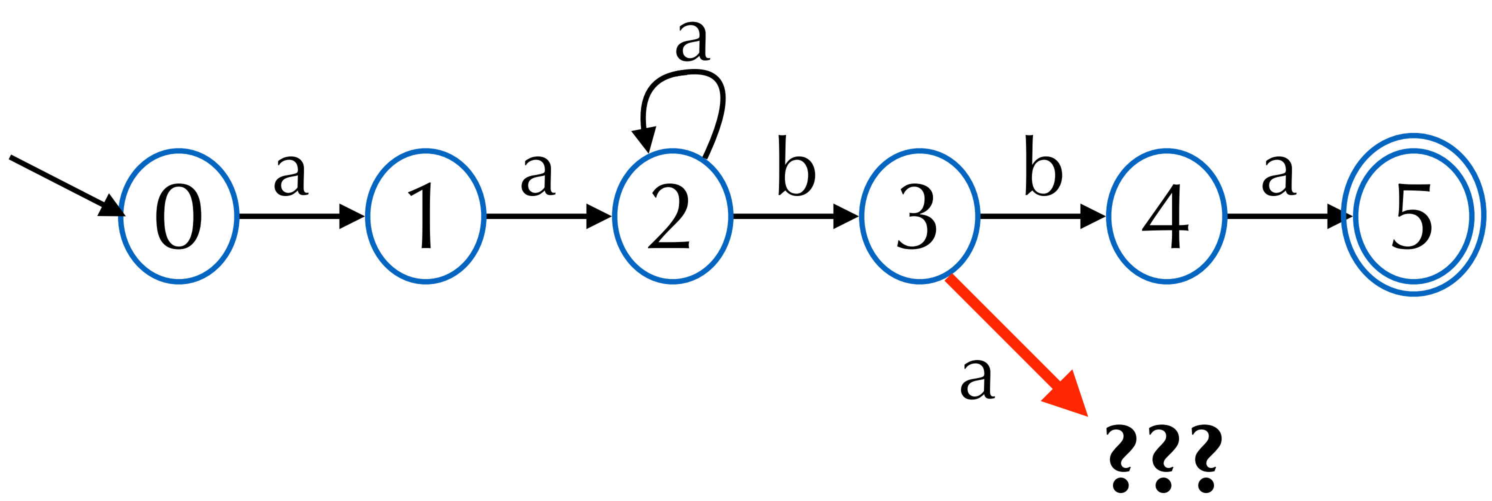
Reading an a from state 3 ?



The last 3 characters I have read: … aab

Computation of the Transitions

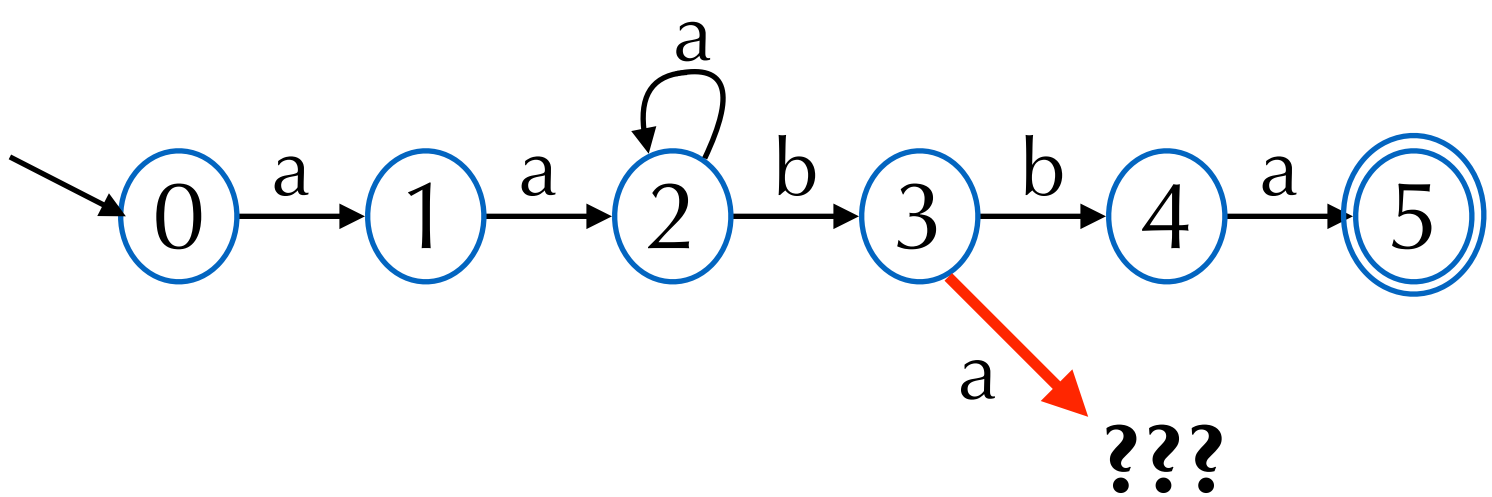
Reading an a from state 3 ?



The last 3 characters I have read: … aab a

Computation of the Transitions

Reading an a from state 3 ?



The last 3 characters I have read: … aab a

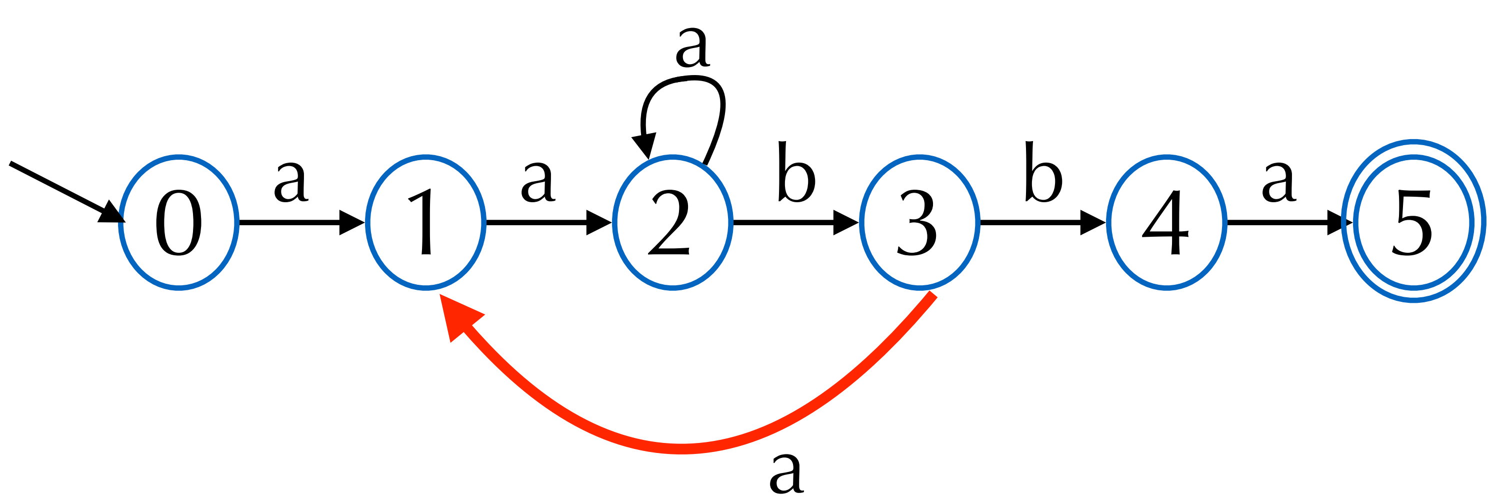
The pattern is not present.

What state if I started one position later ?

- reading aba from state 0 -> state 1

Computation of the Transitions

Reading an a from state 3 ?



The last 3 characters I have read: … aab a

The pattern is not present.

What state if I started one position later ?

- reading aba from state 0 -> state 1

III. Boyer-Moore

The Boyer–Moore algorithm is considered as the most efficient string matching algorithm in usual applications. Crochemore Hancart Lecroq (2003)

Idea: Read from the End

```
Text:       To be, or not to be,…
Pattern:    not to
                  not to
                       not to
```

last character shift

Boyer-Moore shift (defined later)

```
def bm(text,pattern,lcs,bms):
    n=len(text)
    m=len(pattern)
    i=0
    while i<=n-m:
        for j in range(m-1,-1,-1):
            if text[i+j]!=pattern[j]:
                i += max(1,j-lcs.get(text[i+j],-1),bms[j])
                break
        else: return i
    return -1
```

```
def lastoccurrence(pattern):
    m=len(pattern)
    lcs={}
    for i in range(m-1):
        lcs[pattern[i]]=i
    return lcs
```

Idea: Read from the End

```
Text:      To be, or not to be,…
Pattern:   not to
                 not to
                       not to
```

last character shift

Boyer-Moore shift (defined later)

```python
def bm(text,pattern,lcs,bms):
    n=len(text)
    m=len(pattern)
    i=0
    while i<=n-m:
        for j in range(m-1,-1,-1):
            if text[i+j]!=pattern[j]:
                i += max(1,j-lcs.get(text[i+j],-1),bms[j])
                break
        else: return i
    return -1
```

```python
def lastoccurrence(pattern):
    m=len(pattern)
    lcs={}
    for i in range(m-1):
        lcs[pattern[i]]=i
    return lcs
```

Worst-case for last character heuristic:

$$P = ba^{m-1},\ T = a^n$$

Idea: Read from the End

Text: `To be, or not to be,…`

Pattern:
```
not to
    not to
        not to
```

last character shift

Boyer-Moore shift (defined later)

```
def bm(text,pattern,lcs,bms):
    n=len(text)
    m=len(pattern)
    i=0
    while i<=n-m:
        for j in range(m-1,-1,-1):
            if text[i+j]!=pattern[j]:
                i += max(1,j-lcs.get(text[i+j],-1),bms[j])
                break
        else: return i
    return -1
```

```
def lastoccurrence(pattern):
    m=len(pattern)
    lcs={}
    for i in range(m-1):
        lcs[pattern[i]]=i
    return lcs
```

Worst-case for last character heuristic:

$$P = ba^{m-1},\ T = a^n$$

Worst-case becomes linear with Boyer-Moore shift

Average-Case Complexity of the Last Character Heuristic

Fixed pattern, uniform random text, $m \leq R$

$$\mathbb{E}(\#\text{comparisons}) \approx \frac{n}{m} \text{ for large } R/m$$

n/m is optimal

Also observed in practice

Document on moodle: step-by-step proof

Shift by Longest Suffixes

Text: `abaabbbaababaabaabaababaabaa`

Pattern: `abaababaabaa`

Find the smallest shift compatible with the letters read so far

Shift by Longest Suffixes

Text: abaabbbaababaabaabaababaabaa

Pattern: abaababaabaa

Find the smallest shift compatible with the letters read so far

Shift by Longest Suffixes

Text: abaabbbaababaabaabaababaabaa

Pattern: abaababaabaa

abaababaabaa

Find the smallest shift compatible with the letters read so far

Shift by Longest Suffixes

Text: abaabbbaababaabaabaababaabaa

Pattern: abaababaabaa

abaababaabaa

Find the smallest shift compatible with the letters read so far

Shift by Longest Suffixes

Text: abaabbbaababaabaabaababaabaa

Pattern: abaababaabaa

abaababaabaa

Find the smallest shift compatible with the letters read so far

Shift by Longest Suffixes

Text: abaabbbaababaabaabaababaabaa

Pattern: abaababaabaa

abaababaabaa

abaababaabaa

Find the smallest shift compatible with the letters read so far

Shift by Longest Suffixes

Text: abaabbbaababaabaabaababaabaa

Pattern: abaababaabaa

abaababaabaa

abaababaabaa

Find the smallest shift compatible with the letters read so far

Shift by Longest Suffixes

Text: abaabbbaababaabaabaababaabaa

Pattern: abaababaabaa

abaababaabaa

abaababaabaa

Find the smallest shift compatible with the letters read so far

Shift by Longest Suffixes

Text: abaabbbaababaabaabaababaabaa

Pattern: abaababaabaa

abaababaabaa

abaababaabaa

abaababaabaa

Find the smallest shift compatible with the letters read so far

Shift by Longest Suffixes

```
Text:        abaabbbaababaabaabaababaabaa
Pattern:     abaababaabaa
               abaababaabaa
                  abaababaabaa
                         abaababaabaa
```

Find the smallest shift compatible with the letters read so far

Shift by Longest Suffixes

Text: abaabbbaababaabaabaababaabaa

Pattern: abaababaabaa

abaab$\boxed{\text{abaa}}$baa

$\boxed{\text{abaa}}$babaabaa

abaab$\boxed{\text{abaa}}$baa

Find the smallest shift compatible with the letters read so far

Shift by Longest Suffixes

Text: abaabbbaababaabaabaababaabaa

Pattern: abaababaabaa

abaababaabaa

abaababaabaa

abaababaabaa

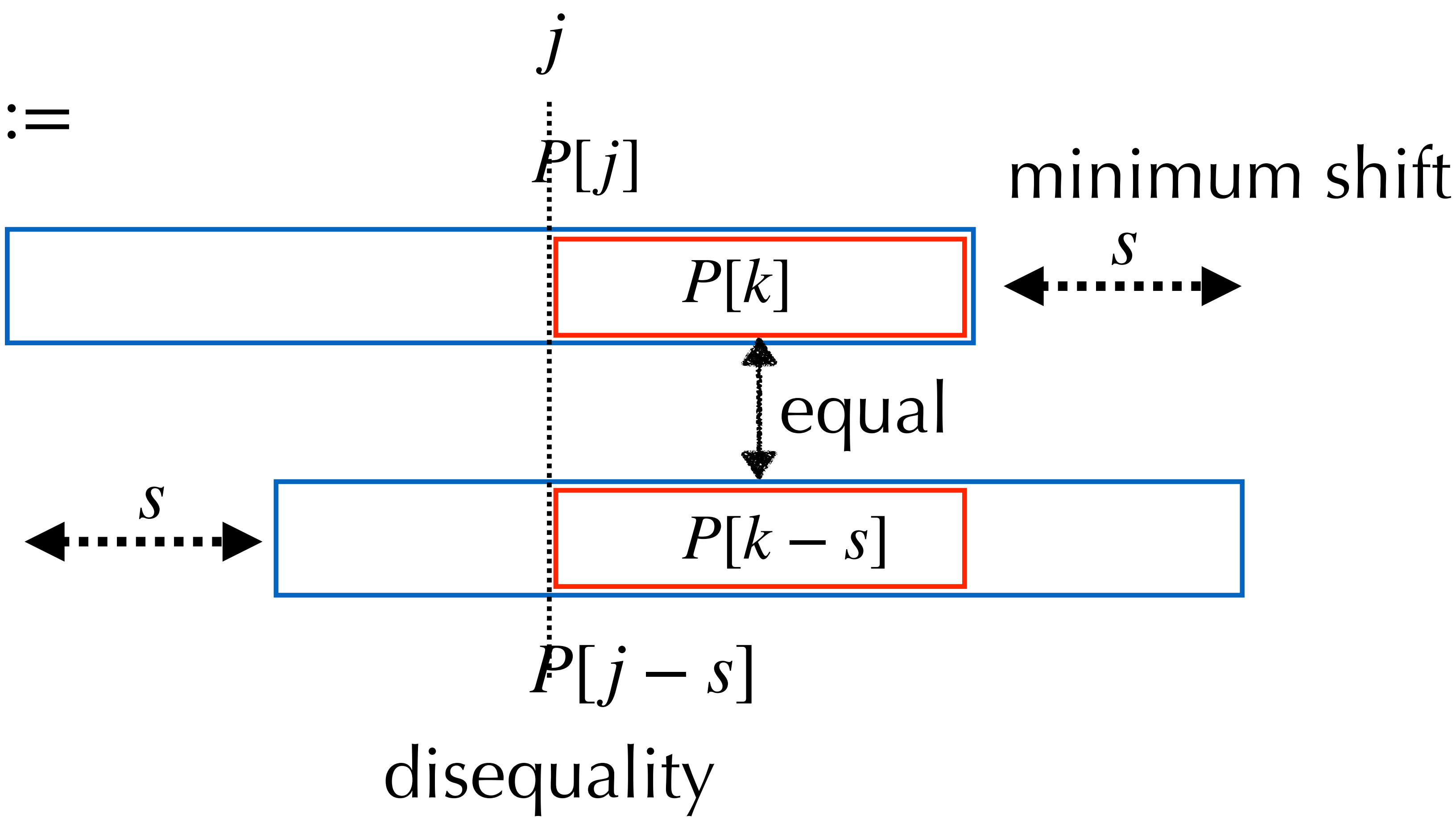
abaababaabaa

Find the smallest shift compatible with the letters read so far

Shift by Longest Suffixes

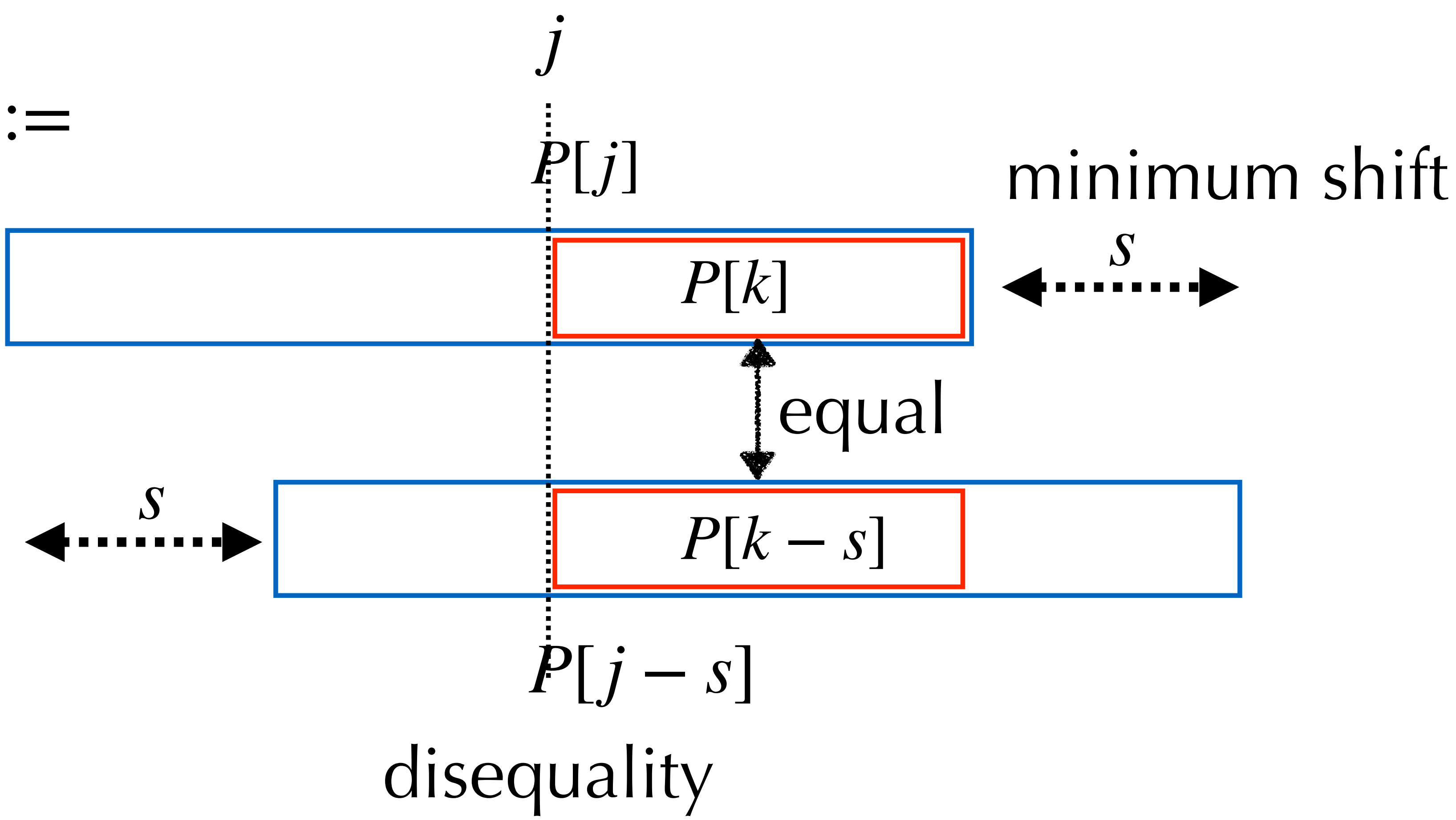
$$\mathrm{bms}[j] := \min \left\{ s > 0 \;\middle|\; \left(\forall k \in \{j+1, \ldots, m\}, s > k \text{ or } P[k-s] = P[k] \right) \right.$$
$$\left. \text{and } \left(s > j \text{ or } P[j-s] \neq P[j] \right) \right\}$$

$\mathrm{bms}[j] :=$



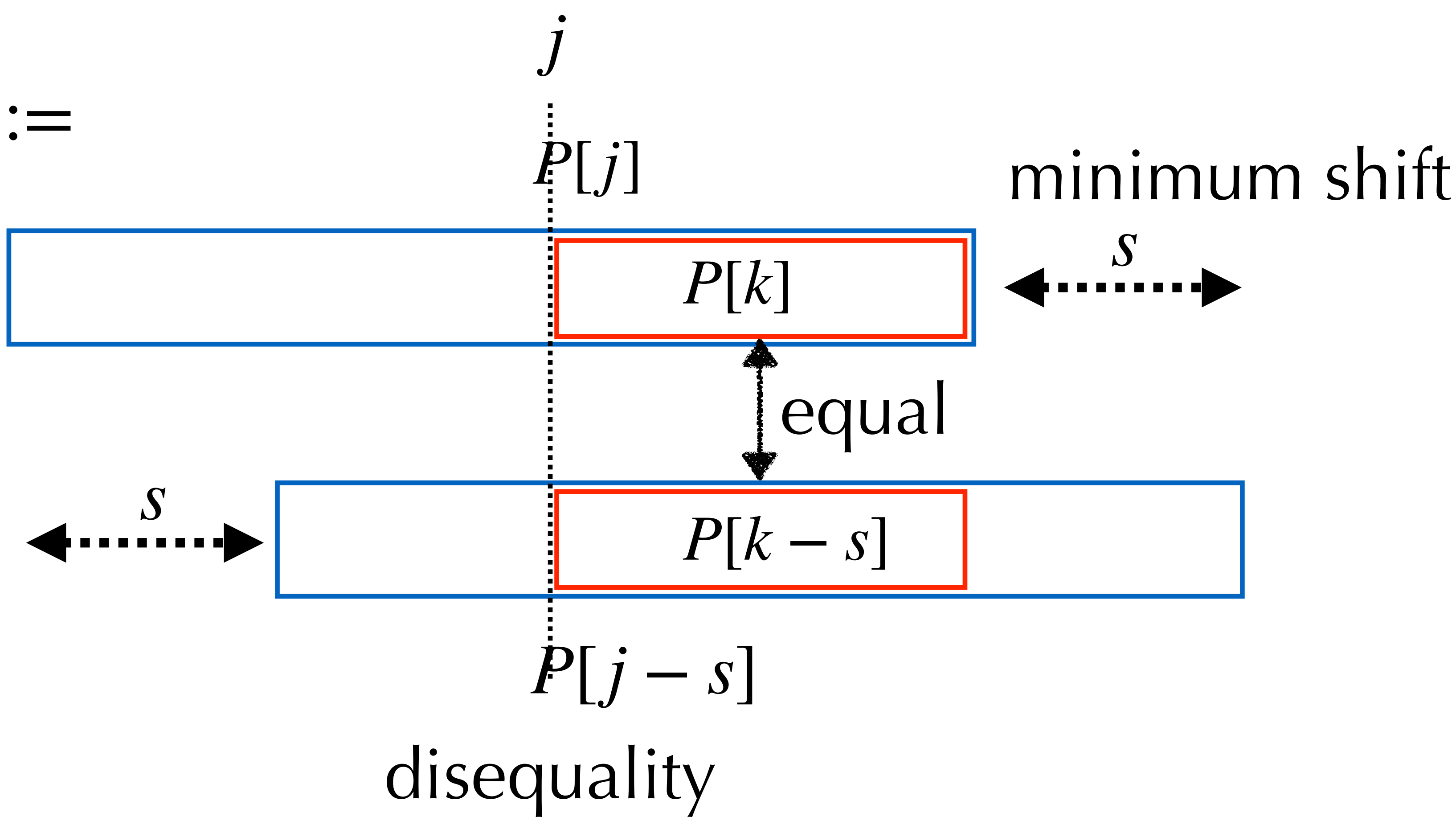
Shift by Longest Suffixes

bms[j] :=



Shift by Longest Suffixes

bms[j] :=



Pattern	a	b	a	a	b	a	b	a	a	b	a	a
Longest suffix	**1**	0	1	**4**	0	1	0	1	4	0	1	
BM shift	**8**	**8**	**8**	**8**	**8**	**8**	**8**	3	**11**	**11**	1	2

slide the pattern to the left over itself and measure overlap

a. prefixes that are also suffixes
b. internal overlap

2. shift leftmost abaa

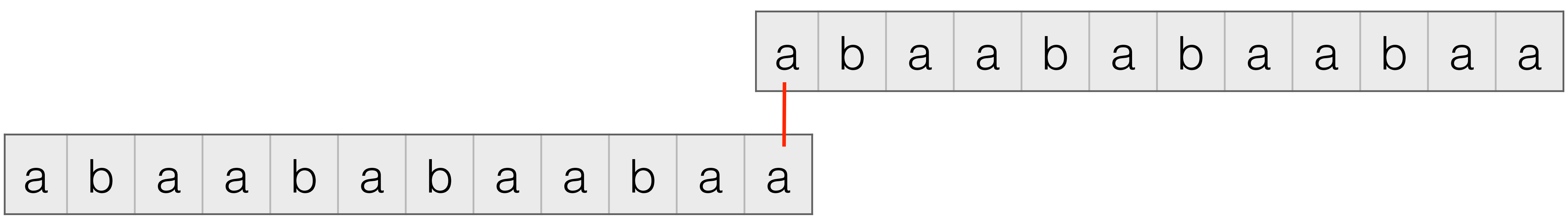
next abaa

1. leftmost a

Shift by Longest Suffixes

slide the pattern to the left over itself and measure overlap

Pattern	a	b	a	a	b	a	b	a	a	b	a	a
Longest suffix	1	0	1	4	0	1	0	1	4	0	1	



Shift by Longest Suffixes

slide the pattern to the left over itself and measure overlap

Pattern	a	b	a	a	b	a	b	a	a	b	a	a
Longest suffix	1	0	1	4	0	1	0	1	4	0	1	

a	b	a	a	b	a	b	a	a	b	a	a

a	b	a	a	b	a	b	a	a	b	a	a

Shift by Longest Suffixes

slide the pattern to the left over itself and measure overlap

Pattern	a	b	a	a	b	a	b	a	a	b	a	a
Longest suffix	**1**	0	1	**4**	0	1	0	1	4	0	1	

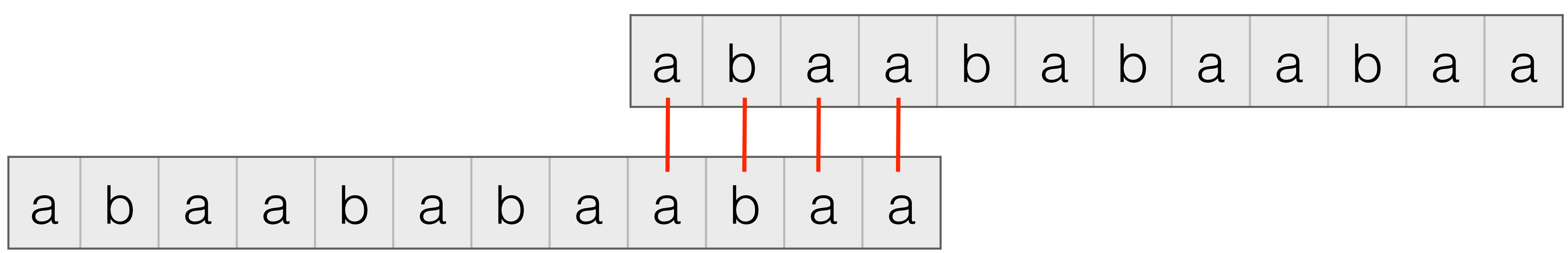
a	b	a	a	b	a	b	a	a	b	a	a

a	b	a	a	b	a	b	a	a	b	a	a

Shift by Longest Suffixes

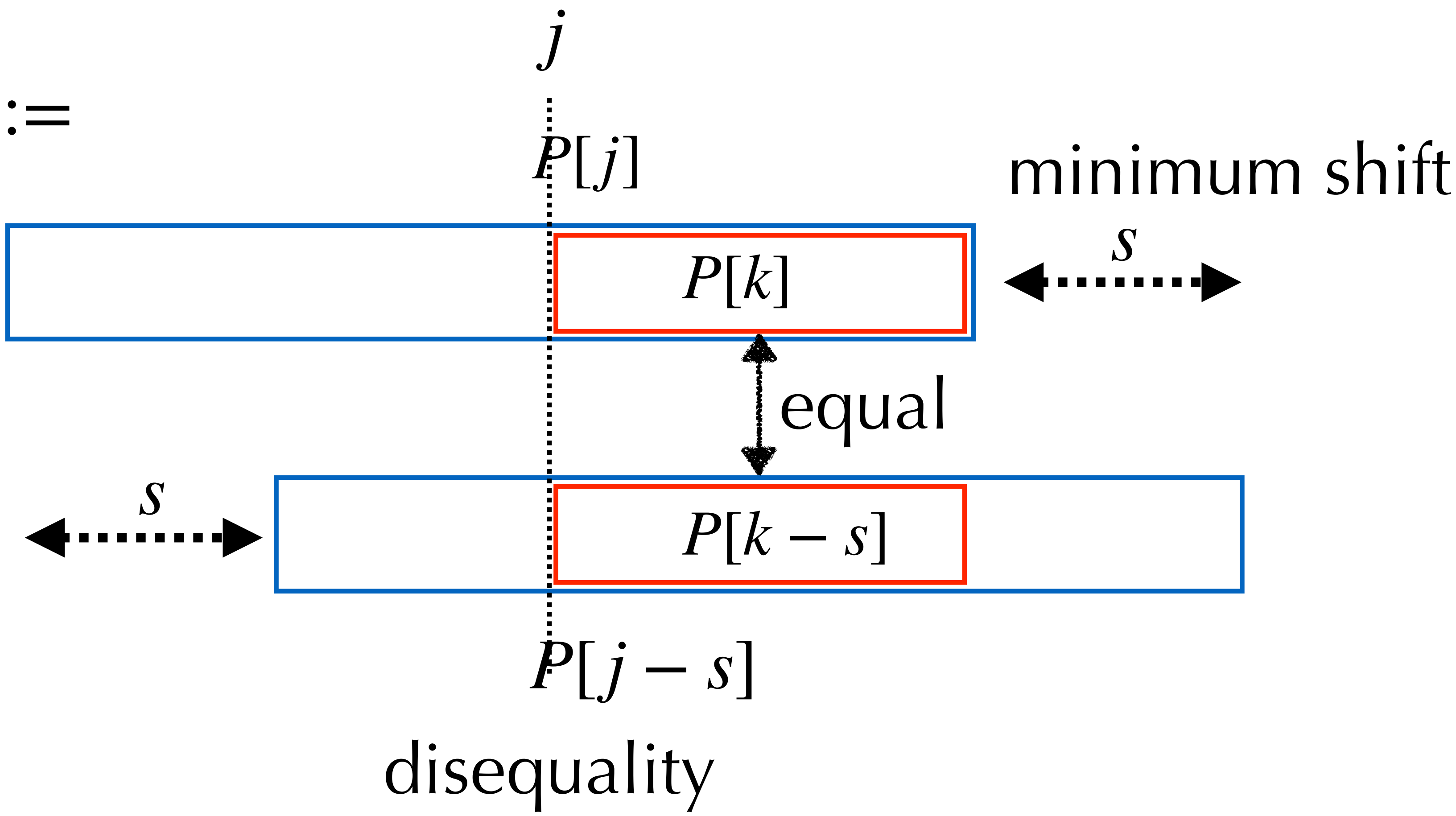
slide the pattern to
the left over itself
and measure overlap

Pattern	a	b	a	a	b	a	b	a	a	b	a	a
Longest suffix	1	0	1	4	0	1	0	1	4	0	1	

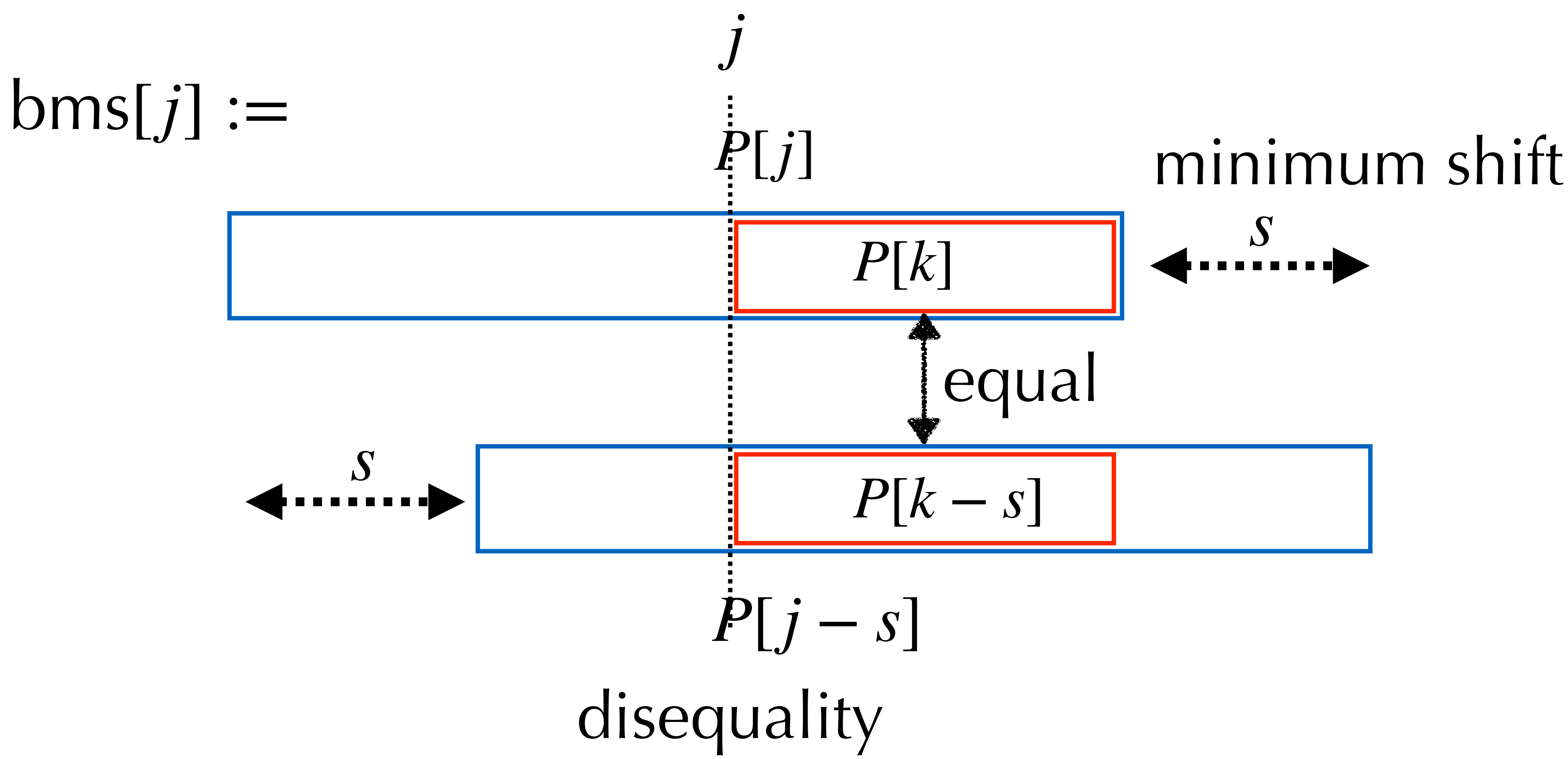


Shift by Longest Suffixes

bms[j] :=



Shift by Longest Suffixes



Pattern	a	b	a	a	b	a	b	a	a	b	a	a
Longest suffix	**1**	0	1	**4**	0	1	0	1	4	0	1	
BM shift	**8**	**8**	**8**	**8**	**8**	**8**	**8**	3	**11**	**11**	1	2

2. shift leftmost abaa

next abaa

1. leftmost a

slide the pattern to the left over itself and measure overlap

a. prefixes that are also suffixes
b. internal overlap

Corresponding Code

Linear complexity possible, but harder

```
def longestsuffix(pattern):
    m=len(pattern)
    ls=[0]*(m-1)
    for i in range(m-2,-1,-1):
        for j in range(i+1):
            if pattern[m-1-j]==pattern[i-j]:
                ls[i] += 1
            else: break
    return ls
```

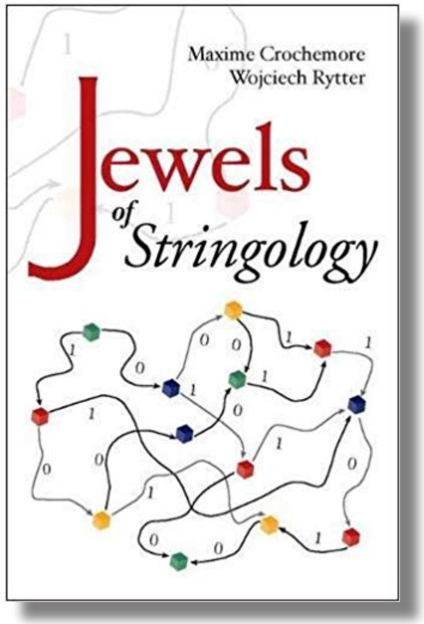
Linear complexity

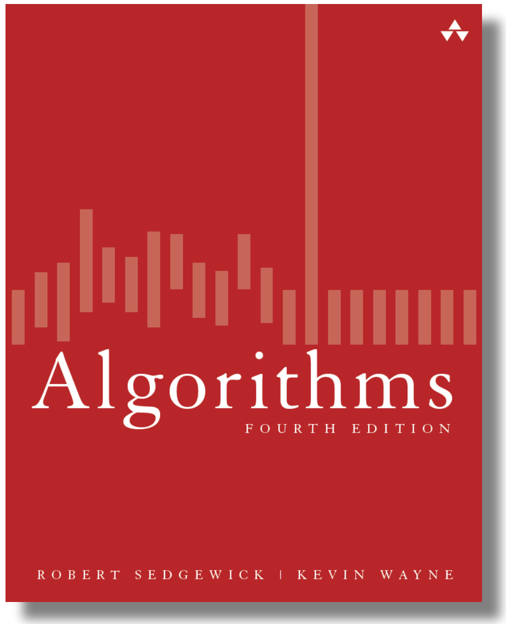
```
def bmshift(pattern):
    ls=longestsuffix(pattern)
    m=len(pattern)
    bms=[m]*m                # init: default shift m
    j=0
    for i in range(m-2,-1,-1):
        if ls[i]==i+1:       # a prefix is a suffix
            for j in range(j,m-i-1): bms[j]=m-1-i
    for i in range(m-1): # rightmost match
        bms[m-1-ls[i]]=m-1-i
    return bms
```

References for this lecture

The slides are designed to be self-contained.

They were prepared using the following books that I recommend if you want to learn more:





Next

NO Assignment

Next tutorial: searching for regular expressions

Next week: String Algorithms 2 — Compression

Feedback

Moodle

Questions: constantin.enea@polytechnique.edu