Pattern Matching

a	b	a	c	a	a	b

1

a	b	a	c	a	b

4 3 2

a	b	a	c	a	b

Pattern Matching (Some slides
from **Dimitrios Katsaros)**
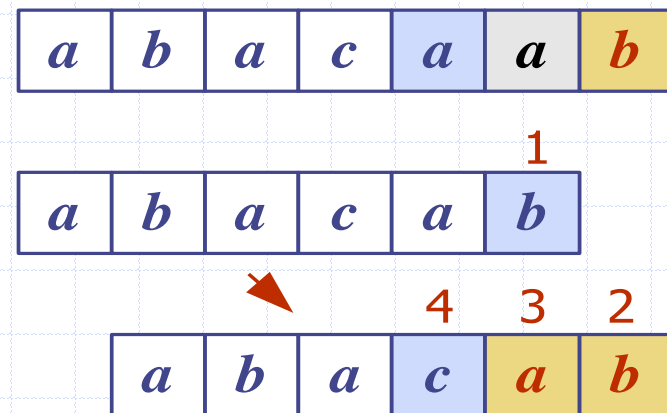
Outline

◆ Strings

◆ Pattern matching algorithms

- Brute-force algorithm
- Boyer-Moore algorithm
- Knuth-Morris-Pratt algorithm

Strings

- A string is a sequence of characters
- Examples of strings:
    - Java program
    - HTML document
    - DNA sequence
    - Digitized image
- An alphabet Σ is the set of possible characters for a family of strings
- Example of alphabets:
    - ASCII
   - Unicode
    - {0, 1}
    - {A, C, G, T}

Strings

- Let P be a string of size m
  - A substring $P[i .. j]$ of P is the subsequence of P consisting of the characters with ranks between i and j
  - A prefix of P is a substring of the type $P[0 .. i]$
  - A suffix of P is a substring of the type $P[i .. m - 1]$

- Given strings T (text) and P (pattern), the pattern matching problem consists of finding a substring of T equal to P

- Applications:
  - Text editors
  - Search engines
  - Biological research
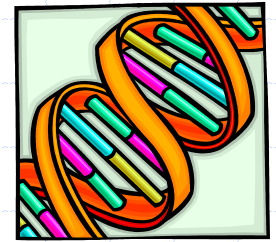
Brute-Force Algorithm

◈ The brute-force pattern matching algorithm compares the pattern P with the text T for each possible shift of P relative to T, until either
  - a match is found, or
  - all placements of the pattern have been tried
◈ Brute-force pattern matching runs in time $O(nm)$
◈ Example of worst case:
  - $T = aaa \ldots ah$
  - $P = aaah$
  - may occur in images and DNA sequences
  - unlikely in English text
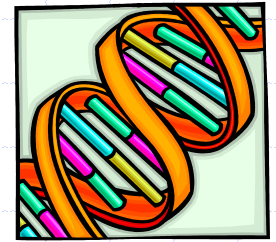
Brute-Force Algorithm

Algorithm *BruteForceMatch*(*T, P*)

    Input text *T* of size *n* and pattern
        P of size *m*

    Output starting index of a
        substring of *T* equal to *P* or -1
        if no such substring exists

    for $i \leftarrow 0$ **to** $n - m$

        { test shift *i* of the pattern }

        $j \leftarrow 0$

        while $j < m \wedge T[i + j] = P[j]$

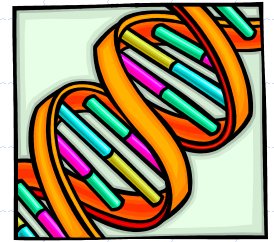 $j \leftarrow j + 1$

        if $j = m$

            return *i* {match at *i*}

        else

            break while loop {mismatch}

    return **-1** {no match anywhere}

Brute-Force Algorithm

http://whocouldthat.be/visualizing-string-matching/

```
function brute_force(text[], pattern[])
{
  // let n be the size of the text and m the size of the
  // pattern

  for(i = 0; i < n; i++) {
    for(j = 0; j < m && i + j < n; j++)
      if(text[i + j] != pattern[j]) break;
      // mismatch found, break the inner loop
    if(j == m) // match found
  }
}
```

The KMP Algorithm - Motivation

- Knuth-Morris-Pratt's algorithm compares the pattern to the text in **left-to-right**, but shifts the pattern more intelligently than the brute-force algorithm.
- When a mismatch occurs, what is the **most** we can shift the pattern so as to avoid redundant comparisons?
- Answer: the largest prefix of $P[0..j]$ that is a suffix of $P[1..j]$

.	.	a	b	a	a	b	x	.	.	.	.	.

a	b	a	a	b	a

j

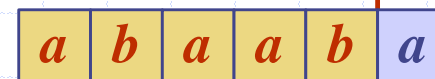| a	b	a	a	b	a

No need to repeat these comparisons

Resume comparing here

KMP Failure Function

◆ Knuth-Morris-Pratt's algorithm preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself

◆ The **failure function** $F(j)$ is defined as the size of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$

◆ Knuth-Morris-Pratt's algorithm modifies the brute-force algorithm so that if a mismatch occurs at $P[j] \neq T[i]$ we set $j \leftarrow F(j-1)$

j	0	1	2	3	4	5
$P[j]$	a	b	a	a	b	a
$F(j)$	0	0	1	1	2	3

| . | . | a | b | a | a | b | x | . | . | . | . | . |

| a | b | a | a | b | a |

j

| a | b | a | a | b | a |

$F(j-1)$

KMP Failure Function

- Knuth-Morris-Pratt's algorithm preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself

- The **failure function** $F(j)$ is defined as the size of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$

- Knuth-Morris-Pratt's algorithm modifies the brute-force algorithm so that if a mismatch occurs at $P[j] \neq T[i]$ we set $j \leftarrow F(j-1)$
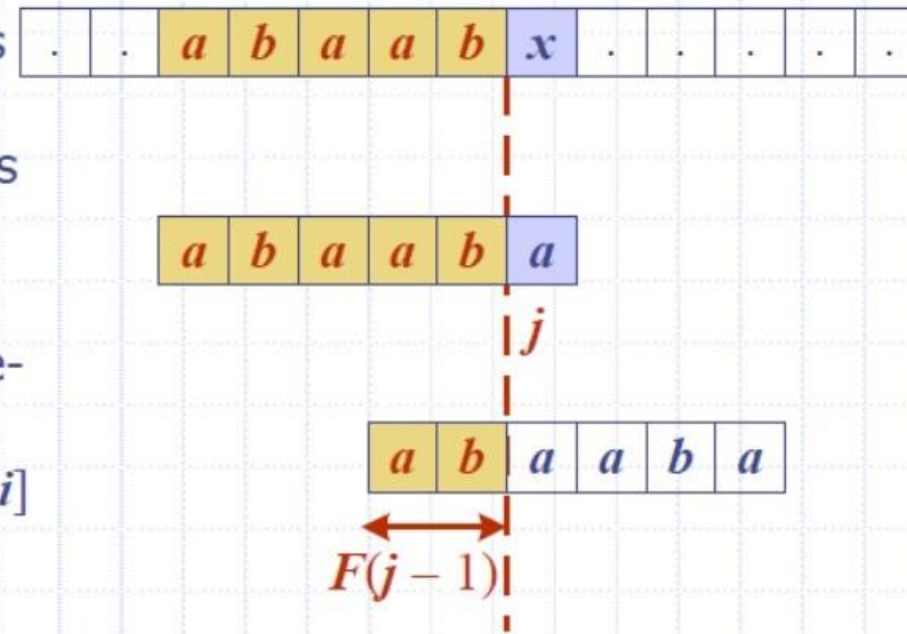
b is not appearing in prefex so set 0

a is appearing in prefix so set it to 1

ab is appearing in prefix so set it 2

aba is largest prefix which is also a suffix

j	0	1	2	3	4	5
$P[j]$	a	b	a	a	b	a
$F(j)$	0	0	1	1	2	3

.	.	a	b	a	a	b	x	.	.	.	.	.

a	b	a	a	b	a

j

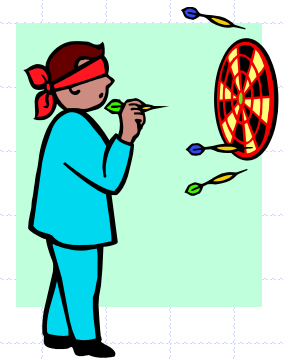| a	b	a	a	b	a

$F(j-1)$

The KMP Algorithm

- The failure function can be represented by an array and can be computed in $O(m)$ time

- At each iteration of the while-loop, either
  - i increases by one, or
  - the shift amount $i - j$ increases by at least one (observe that $F(j-1) < j$)

- Hence, there are no more than $2n$ iterations of the while-loop

- Thus, KMP's algorithm runs in optimal time $O(m + n)$

Algorithm *KMPMatch*(*T, P*)
$\quad F \leftarrow failureFunction(P)$
$\quad i \leftarrow 0$
$\quad j \leftarrow 0$
$\quad$ **while** $i < n$
$\quad\quad$ **if** $T[i] = P[j]$
$\quad\quad\quad$ **if** $j = m - 1$
$\quad\quad\quad\quad$ **return** $i - j$ { match }
$\quad\quad\quad$ **else**
$\quad\quad\quad\quad i \leftarrow i + 1$
$\quad\quad\quad\quad j \leftarrow j + 1$
$\quad\quad$ **else**
$\quad\quad\quad$ **if** $j > 0$
$\quad\quad\quad\quad j \leftarrow F[j - 1]$
$\quad\quad\quad$ **else**
$\quad\quad\quad\quad i \leftarrow i + 1$
$\quad$ **return** -1 { no match }

Computing the Failure Function

- The failure function can be represented by an array and can be computed in $O(m)$ time

- The construction is similar to the KMP algorithm itself

- At each iteration of the while-loop, either
  - i increases by one, or
  - the shift amount $i - j$ increases by at least one (observe that $F(j - 1) < j$)

- Hence, there are no more than $2m$ iterations of the while-loop

Algorithm *failureFunction*(*P*)
  $F[0] \leftarrow 0$
  $i \leftarrow 1$
  $j \leftarrow 0$
  while $i < m$
    if $P[i] = P[j]$
      {we have matched $j + 1$ chars}
      $F[i] \leftarrow j + 1$
      $i \leftarrow i + 1$
      $j \leftarrow j + 1$
    else if $j > 0$ **then**
      {use failure function to shift P}
      $j \leftarrow F[j - 1]$
    else
      $F[i] \leftarrow 0$ { no match }
      $i \leftarrow i + 1$

Search string = a b a c a b

Iteration 1 : i = 1, j = 0 ; P[1] == P[0] ; No so else F [1] = 0

J	0	1	2	3	4	5
P[j]	a	b	a	c	a	b
F[j]	0	0				

Iteration 2 : i = 2, j = 0 ; is P[2] == P[0] ; yes so else F [2] = j + 1 = 1

J	0	1	2	3	4	5
P[j]	a	b	a	c	a	b
F[j]	0	0	1			

Iteration 3 : i = 3, j = 1 ; is P[3] == P[1] ; no and
J > 0; so J = F[0]; is p [3] == P [0]; no so F [3] = 0

J	0	1	2	3	4	5
P[j]	a	b	a	c	a	b
F[j]	0	0	1	0		

Algorithm *failureFunction*(P)

$F[0] \leftarrow 0$
$i \leftarrow 1$
$j \leftarrow 0$
while $i < m$
    if $P[i] = P[j]$
        {we have matched $j + 1$ chars}
        $F[i] \leftarrow j + 1$
        $i \leftarrow i + 1$
        $j \leftarrow j + 1$
    else if $j > 0$ **then**
        {use failure function to shift P}
        $j \leftarrow F[j - 1]$
    else
        $F[i] \leftarrow 0$ { no match }
        $i \leftarrow i + 1$

Example

a	*b*	*a*	*c*	*a*	**a**	*b*	*a*	*c*	*c*	*a*	*b*	*a*	*c*	*a*	*b*	*a*	*a*	*b*	*b*

1 2 3 4 5 6

a	*b*	*a*	*c*	*a*	*b*

7

a	*b*	*a*	*c*	*a*	*b*

8 9 10 11 12

a	*b*	*a*	*c*	*a*	*b*

13

a	*b*	*a*	*c*	*a*	*b*

14 15 16 17 18 19

a	*b*	*a*	*c*	*a*	*b*

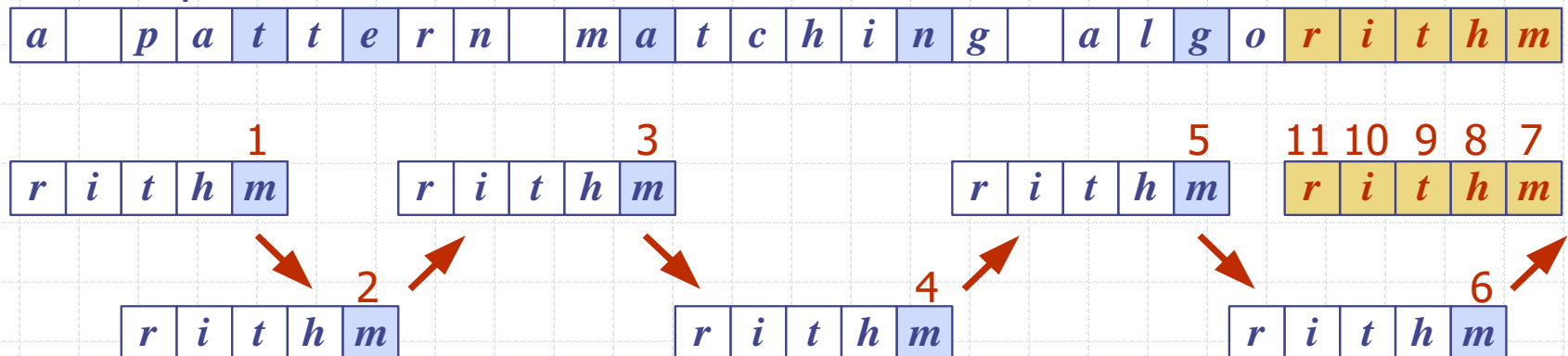| *j*	0	1	2	3	4	5
P[*j*]	*a*	*b*	*a*	*c*	*a*	*b*
F(*j*)	0	0	1	0	1	2

Boyer-Moore Heuristics

◆ The Boyer-Moore's pattern matching algorithm is based on two heuristics

Looking-glass heuristic (right-to-left matching): Compare P with a subsequence of T moving backwards

Character-jump heuristic (bad character shift rule): When a mismatch occurs at $T[i] = c$

- If P contains c, shift P to align the last occurrence of c in P with $T[i]$
- Else, shift P to align $P[0]$ with $T[i+1]$

◆ Example

| a | | p | a | t | t | e | r | n | | m | a | t | c | h | i | n | g | | a | l | g | o | r | i | t | h | m |

1   3   5   11 10 9 8 7

| r | i | t | h | m |

| r | i | t | h | m |

2   4   6

Last-Occurrence Function

◆ Boyer-Moore's algorithm preprocesses the pattern P and the alphabet Σ to build the last-occurrence function L mapping Σ to integers, where $L(c)$ is defined as
  ■ the largest index i such that $P[i] = c$ or
  ■ -1 if no such index exists

◆ Example:
  ■ $\Sigma = \{a, b, c, d\}$
  ■ $P = abacab$

c	a	b	c	d
$L(c)$	4	5	3	-1

◆ The last-occurrence function can be represented by an array indexed by the numeric codes of the characters

◆ The last-occurrence function can be computed in time $O(m + s)$, where m is the size of P and s is the size of Σ

The Boyer-Moore Algorithm

Algorithm *BoyerMooreMatch*(*T, P, Σ*)

   $L \leftarrow lastOccurenceFunction(P, \Sigma)$

   $i \leftarrow m - 1$

   $j \leftarrow m - 1$

   repeat

      if $T[i] = P[j]$

         if $j = 0$

            return i { match at i }

         else

            $i \leftarrow i - 1$

            $j \leftarrow j - 1$

      else

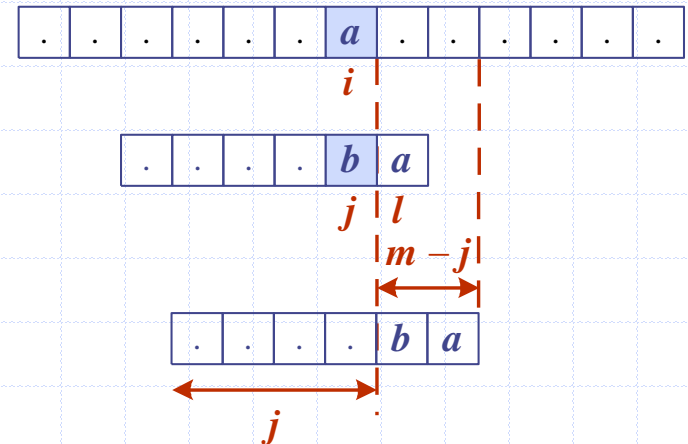   { character-jump }

         $l \leftarrow L[T[i]]$

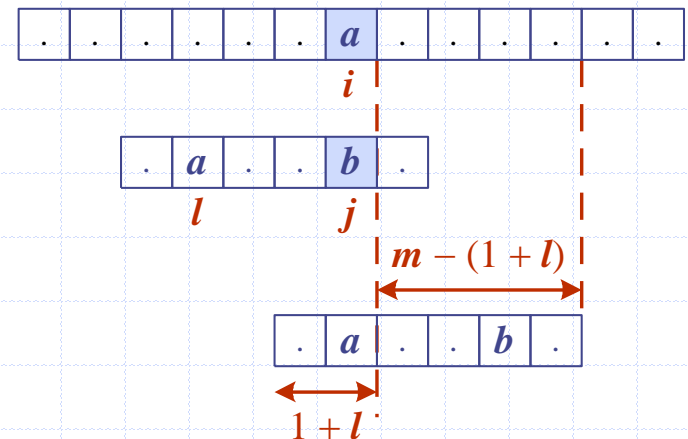 $i \leftarrow i + m - \min(j, 1 + l)$

         $j \leftarrow m - 1$

   until $i > n - 1$
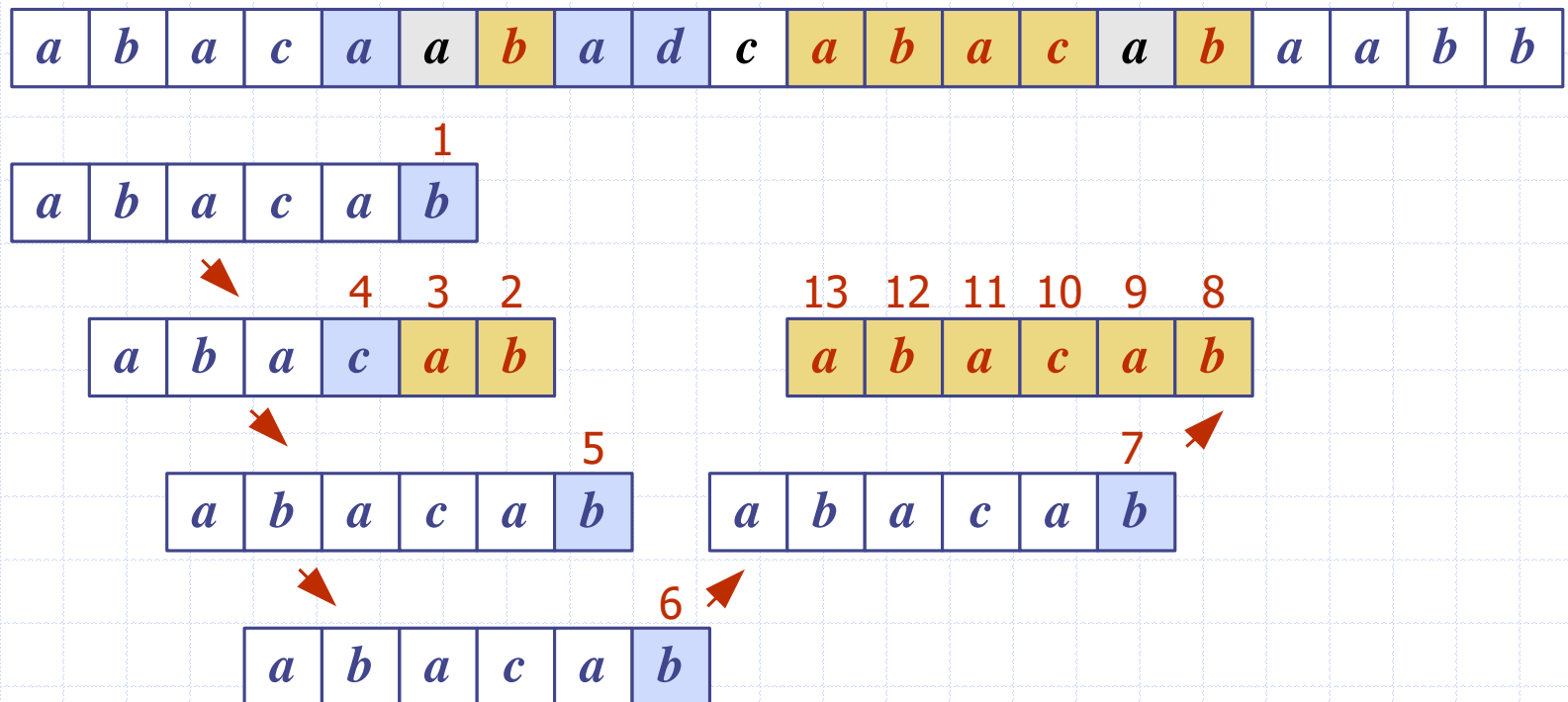
   return -1 { no match }

Case 1: $j \leq 1 + l$

Case 2: $1 + l \leq j$

Example

Analysis

- Boyer-Moore's algorithm runs in time $O(nm + s)$
- Example of worst case:
  - $T = aaa \ldots a$
  - $P = baaa$
- The worst case may occur in images and DNA sequences but is unlikely in English text
- Boyer-Moore's algorithm is significantly faster than the brute-force algorithm on English text

a	a	a	a	a	a	a	a	a

6	5	4	3	2	1

b	a	a	a	a	a

12	11	10	9	8	7

b	a	a	a	a	a

18	17	16	15	14	13

b	a	a	a	a	a

24	23	22	21	20	19

b	a	a	a	a	a