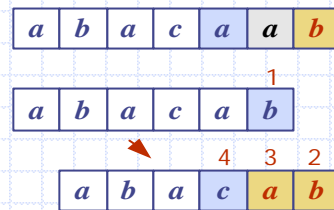


Pattern Matching



Strings

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

"Happy" = {72, 97, 112, 112, 121}

"Happy" = {72, 97, 112, 112, 121}

"01101"

"ACGGTCAT"

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 Pattern Matching

- An algorithmic design pattern for matching a pattern string P , of length m to a substring of a text string T , of length n .
- By “match”, we mean that, for some substring $T[i, i+m-1]$,
 $T[i+j] = P[j]$ for all values of j such that $0 \leq j < m$.
- The brute-force pattern matching algorithm compares the pattern P with the text T for consecutive values of i such that $0 \leq i < n-m$, until either
 - a match is found, or
 - all placements of the pattern have been tried and no match has been found.

Brute-Force Pattern Matching



- ✦ Brute-force pattern matching runs in time $O(nm)$
- ✦ Example of worst case:
 - $T = aaa \dots ah$
 - $P = aaah$
 - may occur in images and DNA sequences
 - unlikely in English text

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

```

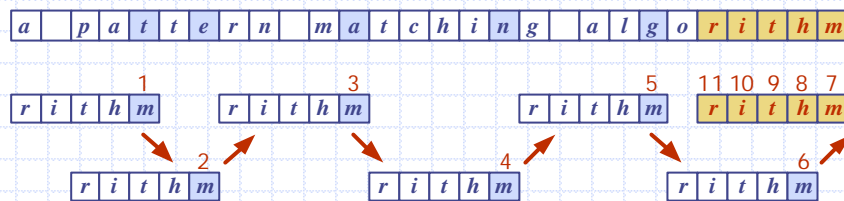
for  $i \leftarrow 0$  to  $n - m$ 
  { test shift  $i$  of the pattern }
   $j \leftarrow 0$ 
  while  $j < m \wedge T[i + j] = P[j]$ 
     $j \leftarrow j + 1$ 
  if  $j = m$ 
    return  $i$  { match at  $i$  }
  else
    break while loop { mismatch }
return  $-1$  { no match anywhere }
  
```

Other Approaches

- ✦ Naïve view is that it is always necessary to examine every character in T in order to locate a pattern P as a substring.
- ✦ This is not always the case...
- ✦ We can sometimes avoid comparisons between P and a sizable fraction of T if we assume that the alphabet is a fixed and finite size.
- ✦ We call this approach the **Boyer-Moore Algorithm**.

Boyer-Moore Heuristics

- The Boyer-Moore's pattern matching algorithm is based on two heuristics
 - Looking-glass heuristic:** Compare P with a subsequence of T moving backwards
 - Character-jump heuristic:** 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



© 2004 Goodrich, Tamassia

Pattern Matching

7

Last-Occurrence Function

- Boyer-Moore's algorithm preprocesses the pattern P and the alphabet S to build the last-occurrence function L mapping S 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:

- $S = \{a, b, c, d\}$
- $P = abacab$

| | | | | |
|--------|-----|-----|-----|-----|
| 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 S

© 2004 Goodrich, Tamassia

Pattern Matching

8

The Boyer-Moore Algorithm

Algorithm *BoyerMooreMatch*(T, P, S)

$L \leftarrow \text{lastOccurrenceFunction}(P, S)$

$i \leftarrow m - 1$

$j \leftarrow m - 1$

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 }

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

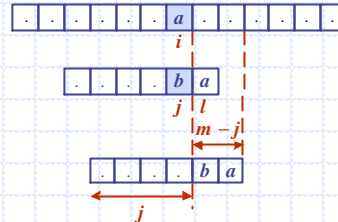
$i \leftarrow i + m - \min(j, 1 + l)$

$j \leftarrow m - 1$

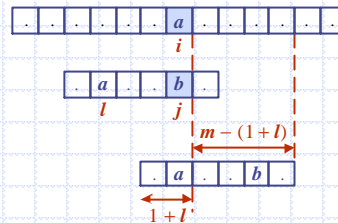
until $i > n - 1$

return -1 { no match }

Case 1: $j \leq 1 + l$



Case 2: $1 + l \leq j$

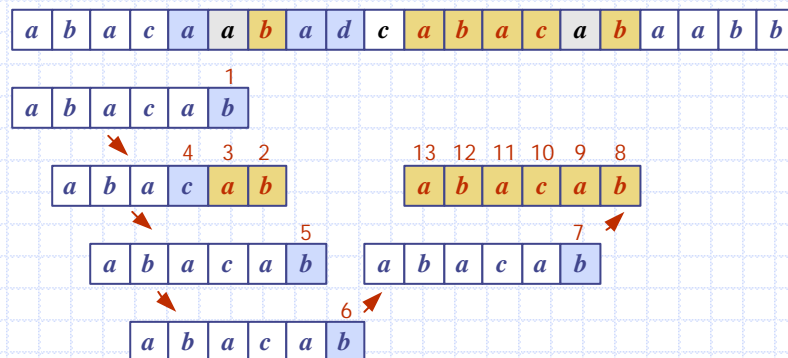


© 2004 Goodrich, Tamassia

Pattern Matching

9

Example



© 2004 Goodrich, Tamassia

Pattern Matching

10

Analysis

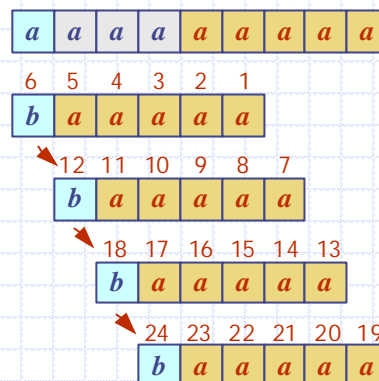
- Boyer-Moore's algorithm runs in time $O(nm + s)$

- Example of worst case:

- $T = aaa \dots a$
 - $P = baaa$

- 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

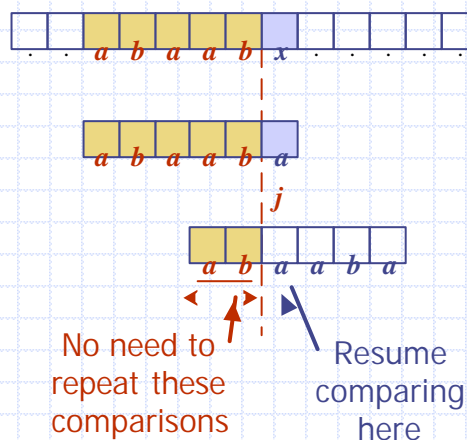


The KMP Algorithm (§ 11.2.3)

- 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 $P[1..j]$

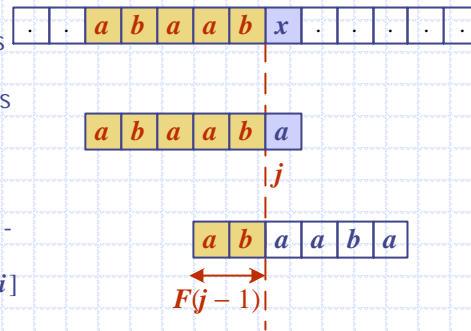


KMP Failure Function

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

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

- 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 $j \leftarrow F(j-1)$

© 2004 Goodrich, Tamassia

Pattern Matching

13

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

© 2004 Goodrich, Tamassia

Pattern Matching

14

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 }
     $i \leftarrow i + 1$ 

```

Example Problem

- ◆ We want to match the pattern "abacab" to the text "abacaabaccabacabaabb"
 - We start by computing the fail function.
 - Then we run the search using the fail function.

Example – Computing the Fail Function

| | | | | | | |
|-----------------------|----------|----------|----------|----------|----------|----------|
| <i>j</i> | 0 | 1 | 2 | 3 | 4 | 5 |
| <i>P</i> [<i>j</i>] | <i>a</i> | <i>b</i> | <i>a</i> | <i>c</i> | <i>a</i> | <i>b</i> |
| <i>F</i> (<i>j</i>) | 0 | 0 | 1 | 0 | 1 | 2 |

| | | | | | | |
|-----------------------|---|---|---|---|---|---|
| <i>i</i> | 1 | 2 | 3 | 3 | 4 | 5 |
| <i>j</i> | 0 | 0 | 1 | 0 | 0 | 1 |
| <i>F</i> (<i>i</i>) | 0 | 1 | | 0 | 1 | 2 |

$P[1...i] = P[1...1] = \text{"b"}$
 $P[1...i] = P[1...2] = \text{"ba"}$
 $P[1...i] = P[1...3] = \text{"bac"}$
 $P[1...i] = P[1...4] = \text{"baca"}$
 $P[1...i] = P[1...5] = \text{"bacab"}$

Example – Doing the search

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

a b a c a b

$j = 0$, so $\text{retrieve}(0) = 0$

a b a c a b

a b a c a b

a b a c a b

a b a c a b

| | | | | | | |
|-----------------------|----------|----------|----------|----------|----------|----------|
| <i>j</i> | 0 | 1 | 2 | 3 | 4 | 5 |
| <i>P</i> [<i>j</i>] | <i>a</i> | <i>b</i> | <i>a</i> | <i>c</i> | <i>a</i> | <i>b</i> |
| <i>F</i> (<i>j</i>) | 0 | 0 | 1 | 0 | 1 | 2 |

A brief comparison...

- The pattern "abacab" was matched to the text "abacaabaccabacabaabb" using the 3 pattern matching techniques discussed in this course:
 - Brute Force took 29 steps
 - Boyer-Moore took 19 steps
 - Knuth-Morris-Pratt also took 19 steps
- After making a small change to the text "abacaabacdabacabaabb" the results were:
 - Brute Force still took 29 steps
 - Boyer-Moore took 16 steps
 - Knuth-Morris-Pratt still took 19 steps
- The Boyer-Moore technique improved mainly because the "d" is not in the pattern.