SCS2201

Data Structures & Algorithms III

String Matching

Dr. HND Thilini

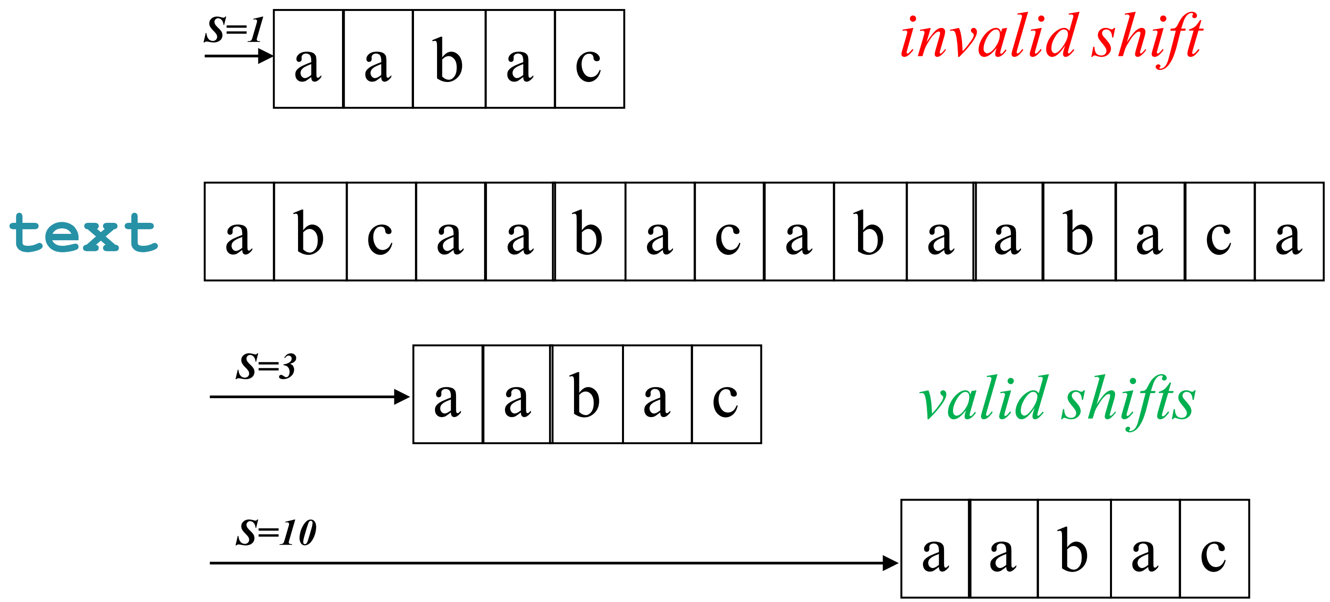
String Matching

- Finding occurrences of a *pattern* within a *text*
- Where a **text** and **pattern** are *strings of characters* with |**text**| ≥ |**pattern**|

- And 'finding' corresponds to discovering *valid shifts* of **`pattern`** along **`text`** that match all characters of **`pattern`** with those of **`text`** beginning at that position

String Matching

- Example



String Matching

- **Applications**
  - Contents Search
  - Editing
  - Text Processing
  - Signal Processing
  - Virus Scanning
  - Bioinformatics (DNA-strings)

String Notions

- An alphabet $\sum$ is a finite set of symbols (or characters).
- A string over $\sum$ is a finite sequence $a_1a_2...a_n$ where a_i € $\sum$.

The set of strings over $\sum$ is denoted by $\sum^*$

The empty string is denoted by ε

String Notions

- Let $x = a_1a_2...a_n$ and $y = b_1b_2...b_m$ Then :
- $|x|$ denotes the length n
- $x[i.....j]$ denotes the (sub-)string $a_ia_{i+1}...a_j$
- xy denotes the concatenation $a_1a_2....a_nb_1b_2...b_m$

Prefix

A string ω is a prefix of a string x, denoted $\omega \sqsubset x$,

if $x = \omega y$ for some string $y \in \Sigma^*$ and

$|\omega| <= |x|$

e.g. $ab \sqsubset abcca$

String Notions

Suffix

A string ω is a suffix of a string x, denoted $\omega \sqsupset x$,

if $x = y\omega$ for some string $y \in \Sigma^*$ and

$|\omega| <= |x|$

e.g. cca $\sqsupset$ abcca

The empty string ε is both a suffix and a prefix of every string.

String Matching

- Algorithms
  - Naïve String Matching
  - Knuth-Morris-Pratt (KMP) Algorithm
  - Rabin-Karp Algorithm
  - Boyer Moore Algorithm

Naive String Matching

- Naive (brute-force) algorithm:

  Simply test all the possible placements of P relative to T

- Checks the condition for each of the possible `n-m+1` positions

- Consists of two nested loops.

Naive String Matching

BruteForceMatch(T,P)

input : strings T with n characters and P(pattern)with m characters

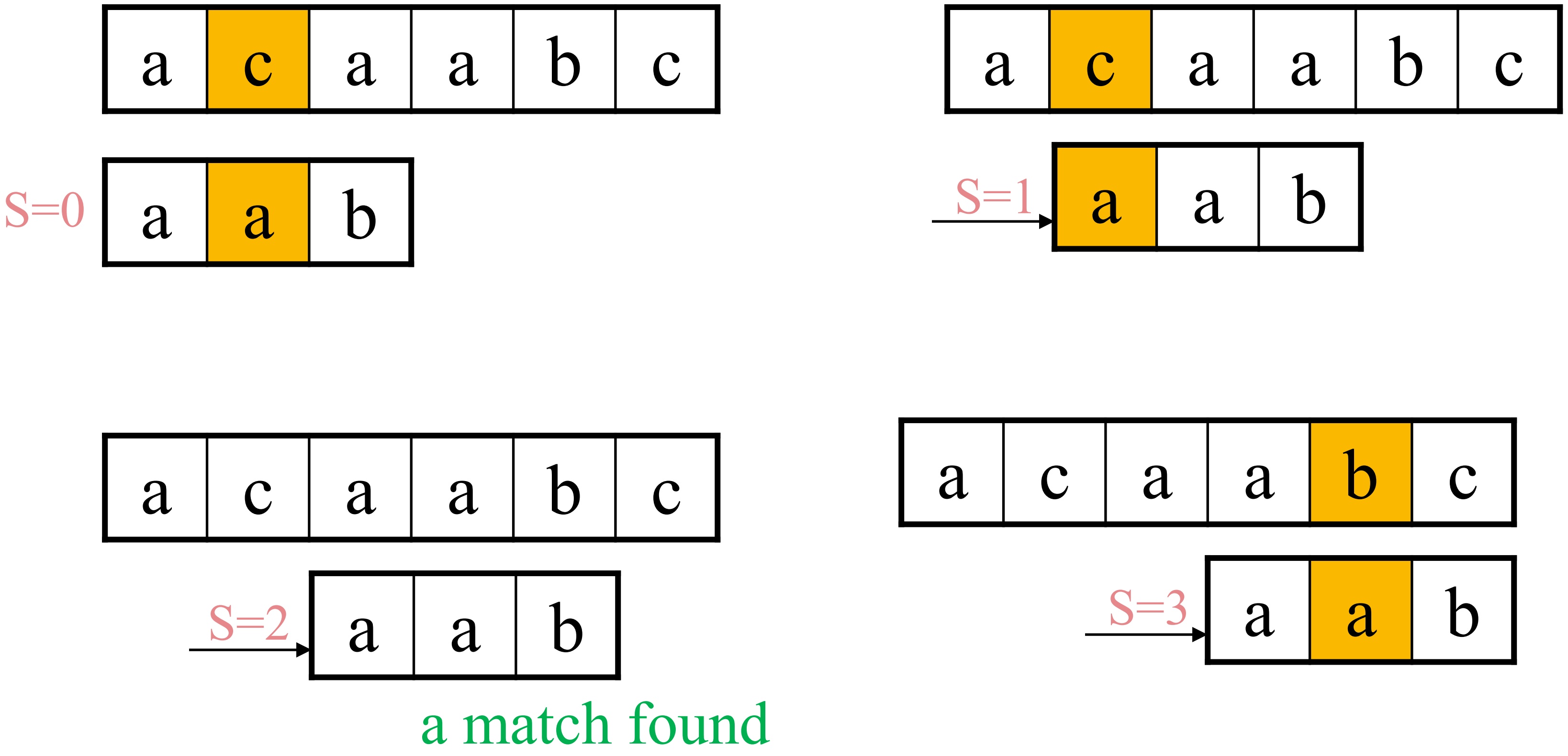
output : starting index of the first substring of T matching P, or an indication that P is not a substring of T

```
n = length(T);   m = length(P);
for i = 0 to n-m do
  j ← 0
while (j < m and T[i+j] = p[j]) do
      j ← j+1
      if j = m then return i
return "there is no substring of T matching P "
```

Naive String Matching

```
NAIVE_STRING_MATCHER (T, P)
n ← length [T]
m ← length [P]
for s ← 0 to n - m do
    if P[1 . . m] = T[s +1 . . s + m]
        then print "pattern occurs with shift s"
```

Naive String Matching



Naive String Matching

Text :

ABBCABBCBCCABCDABCFGCD

Pattern:

ABCD

1. Find the starting index of the first substring of Text matching Pattern
2. How many comparisons were made?

Naive String Matching

- The outer-loop is executed at most n-m+1 times,
- The inner loop is executed at most m times.
- The running time of this method is

$$O((n-m+1)m) \Rightarrow O(nm)$$

- Since there is no preprocess, the total *running time* is the same as its *matching time*

Naive-Complexity

- **Worst case: compares pattern to each substring of** text of length M. For example, M=5.

  1) *AAAAAAAAAAAAAAAAAAAAAAAAAAAH*

  AAAAH *5 comparisons made*

  2) *AAAAAAAAAAAAAAAAAAAAAAAAAAAH*

  AAAAH *5 comparisons made*

  3) *AAAAAAAAAAAAAAAAAAAAAAAAAAAH*

  AAAAH *5 comparisons made*

- Total number of comparisons: M (N-M+1)
- Worst case time complexity: O(MN)

Naive-Complexity

Best case if pattern found: Finds pattern in first M positions of text.

1) *AAAAABBBBBBBBBBBBBBBBBBBBH*
AAAAA ***5 comparisons made***

- Total number of comparisons: M
- Best case time complexity: O(M)

Naive-Complexity

- Best case if pattern not found: Always mismatch on first character.

1) *AAAAAAAAAAAAAAAAAAAAAAAAAAAH*

OOOOH *1 comparison made*

2) *AAAAAAAAAAAAAAAAAAAAAAAAAAAH*

OOOOH *1 comparison made*

i) *AAAAAAAAAAAAAAAAAAAAAAAAAAAH*

1 comparison made *OOOOH*

- Total number of comparisons: N
- Best case time complexity: O(N)

Naive String Matching

- Suppose that all characters in the pattern P are different. Show how to accelerate NAIVE-STRING-MATCHER to run in time $O(n)$ on an n-character text T.

Naive String Matching

- Suppose that pattern *P* and text *T* are *randomly* chosen strings of length *m* and *n*, respectively, from the *d*-ary alphabet $\sum_d = \{0, 1, \ldots, d - 1\}$, where $d \geq 2$. Show that the *expected* number of character-to-character comparisons made by the implicit loop in line 4 of the naive algorithm is

$$(n - m + 1)\frac{1 - d^{-m}}{1 - d^{-1}} \leq 2(n - m + 1) .$$

(Assume that the naive algorithm stops comparing characters for a given shift once a mismatch is found or the entire pattern is matched.)

Naive String Matching

- Inefficient because information gained about the text for one value of s is entirely ignored in considering other values of s.

  e.g. if p = aaab and we find that s=0 is valid then none of the shifts 1,2 or 3 are valid

  since T[4] = b.

- How can we improve?

Knuth-Morris-Pratt Algorithm

- The Knuth-Morris-Pratt (KMP) string searching algorithm differs from the brute-force algorithm by keeping track of information gained from previous comparisons.
- Based on shifts of the pattern on itself
- Since the length of pattern is just m, preprocessing cost avoids dependence on Σ
- Thus running time complexity is $O(m+n)$