Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

String Matching

Dr. Shylaja S S

String Matching

Given a string of n characters called the **text** and a string of m characters ($m \le n$) called the **pattern**; find a substring of the text that matches the pattern. To put it more precisely, we want to find i - the index of the leftmost character of the first matching substring in the text-such that

$t_i = p_0, \dots, t_{i+j} = p_j, \dots, t_{i+m-1} = p_{m-1}$:

$$\begin{array}{ccccccccccc} t_0 & \dots & t_i & \dots & t_{i+j} & \dots & t_{i+m-1} & & \dots & t_{n-1} & \text{text } T \\ & & \updownarrow & & \updownarrow & & \updownarrow & & & & \\ & & p_0 & \dots & p_j & \dots & p_{m-1} & \text{pattern } P & & & \end{array}$$

If matches other than the first one need to be found, a string-matching algorithm can simply continue working until the entire text is exhausted.

A brute-force algorithm for the string-matching problem is quite obvious: align the pattern against the first m characters of the text and start matching the corresponding pairs of characters from left to right until either all m pairs of the characters match (then the algorithm can stop) or a mismatching pair is encountered. In the latter case, shift the pattern one position to the right and resume character comparisons, starting again with the first character of the pattern and its counterpart in the text. Note that the last position in the text which can still be a beginning of a matching substring is n - m (provided the text's positions are indexed from 0 to n- 1). Beyond that position, there are not enough characters to match the entire pattern; hence, the algorithm need not make any comparisons there.

```
ALGORITHM  BruteForceStringMatch(T[0 .. n -1], P[0 .. m -1])
//Implements brute-force string matching
//Input: An array T[0 .. n - 1]  of n  characters representing a text and an array
//P[0 .. m - 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a matching
//substring or -1 if the search is unsuccessful
for i ← 0 to n-m do
    j ←0
    while j < m and P[j] = T[i + j] do
        j ← j+1
    if j = m return i
return -1
```

Example:

```
N 0 B 0 D Y _ N 0 T I C E D _ H I M
N 0 T
  N 0 T
    N 0 T
      N 0 T
        N 0 T
          N 0 T
            N 0 T
              N 0 T
```

Fig 1: Example of brute-force string matching. (The pattern's characters that are compared with their text counterparts are in bold type.)

Note that for this example, the algorithm shifts the pattern almost always after a single character comparison. However, the worst case is much worse: the algorithm may have to make all m comparisons before shifting the pattern, and this can happen for each of the n-m+1 tries. Thus, in the worst case, the algorithm is in Θ(nm). For a typical word search in a natural language text, however, we should expect that most shifts would happen after very few comparisons (check the example again). Therefore, the average-case efficiency should be considerably better than the worst-case efficiency. Indeed it is: for searching in random texts, it has been shown to be linear, i.e., Θ(n + m) = Θ(n).