Advanced Algorithm

String Matching

Topics To be Covered

- ✓ String Matching Terminology
- ✓ String Matching Applications
- ✓ Naïve String Matching (Brute-Force Algorithm)
- ✓ Horspool’s Algorithm
- ✓ String Matching Using Finite Automata
- ✓ Rabin-Karp Algorithm ……and others

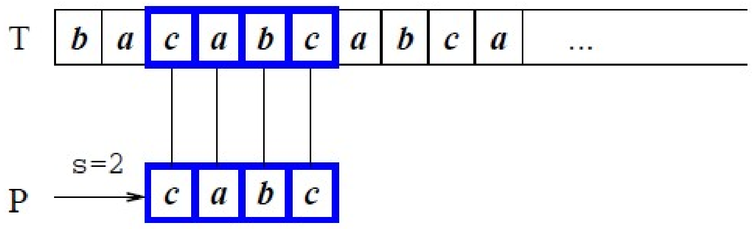
Terminology

Given a text array $T[1 \dots n]$ and a pattern array $P[1 \dots m]$ such that the elements of T and P are characters taken from alphabet Σ . e.g., $\Sigma = \{0, 1\}$ or $\Sigma = \{a, b, \dots, z\}$.

The **String Matching Problem** is to find *all* the occurrence of P in T .

Terminology

A pattern P occurs with **shift** s in T , if $P[1 \dots m]$ = $T[s+1 \dots s+m]$. The String Matching Problem is to find all values of s. Obviously, we must have $0 \le s \le n-m$.



Applications

- Password Verification
- Find a Pattern(word) in the PDF or WORD file
- Searching a text in the web page
- Search a computer virus pattern given in virus database into a newly installed software

Brute-Force Algorithm

Initially, P is aligned with T at the first index position. P is then compared with T from **left-to-right**. If a mismatch occurs, "slide" P to *right* by 1 position, and start the comparison again.

Brute-Force Algorithm

```
BF_StringMatcher(T, P) {
 n = length(T);
 m = length(P);

 // s increments by 1 in each iteration
 // => slide P to right by 1
 for (s=0; s<=n-m; s++) {
  // starts the comparison of P and T again
  i=1; j=1;
  while (j<=m && T[s+i]==P[j]) {
   // corresponds to compare P and T from
   // left-to-right
   i++; j++;
  }
  if (j==m+1)
   print "Pattern occurs with shift=", s
 }
}
```

Brute-Force Algorithm

```
BF_StringMatcher(T, P) {
 n = length(T);
 m = length(P);

 // s increments by 1 in each iteration
 // => slide P to right by 1
 for (s=0; s<=n-m; s++) {
  // starts the comparison of P and T again
  i=1; j=1;
  while (j<=m && T[s+i]==P[j]) {
   // corresponds to compare P and T from
   // left-to-right
   i++; j++;
  }
  if (j==m+1)
   print "Pattern occurs with shift=", s
 }
}
```

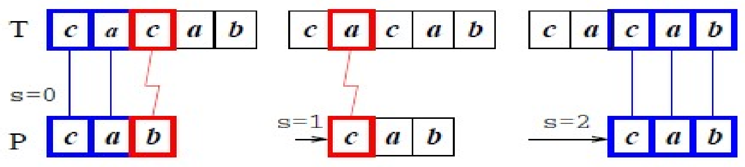
Brute-Force Algorithm

```
BF_StringMatcher(T, P) {
 n = length(T);
 m = length(P);

 // s increments by 1 in each iteration
 // => slide P to right by 1
 for (s=0; s<=n-m; s++) {
  // starts the comparison of P and T again
  i=1; j=1;
  while (j<=m && T[s+i]==P[j]) {
   // corresponds to compare P and T from
   // left-to-right
   i++; j++;
  }
  if (j==m+1)
   print "Pattern occurs with shift=", s
 }
}
```

Brute-Force Algorithm

Initially, P is aligned with T at the first index position. P is then compared with T from **left-to-right**. If a mismatch occurs, "slide" P to *right* by 1 position, and start the comparison again.



Complexity : Θ (m(n-m+1)) is equivalent to Θ(mn)

Horspool Algorithm

- The pattern is check with the text from right to left and progresses left to right through the text.
- Horspool's algorithm shifts the pattern by looking up shift value in the character of the text aligned with the last character of the pattern in table made during the initialization of the algorithm.
-
- Let *c* be the character in the text that aligns with the last character of the pattern. If the pattern does not match there are 4 cases to consider.

Horspool’s Algorithm

The **mismatch occurs at the last character** of the pattern:
Case 1: ***c* does not exist** in the pattern (Not the mismatch occurred here) then shift pattern right the size of the pattern.

```
T[0] ...      S      ... T[n-1]
              |
         LEADER
               LEADER
```

Case 2: The mismatch happens at the last character of the pattern and ***c* does exist** in the pattern then the shift should be to the **right most *c*** in the *m*-1 remaining characters of the pattern.

```
T[0] ...      A      ... T[n-1]
              |
         LEADER
            LEADER
```

Horspool’s Algorithm

The **mismatch happens in the middle** of the pattern:
Case 3: The mismatch happens in the middle (therefore *c* is in pattern) and there are **no other *c* in the pattern** then the shift should be the pattern length.

```
T[0] ...    MER      ... T[n-1]
            |
         LEADER
               LEADER
```

Case 4: The mismatch happens in the middle of the pattern but **there is other *c* in pattern** then the shift should be the **right most *c*** in the *m*-1 remaining characters of the pattern.

```
T[0] ...   EDER      ... T[n-1]
           |
         LEADER
       LEADER
```

Horspool’s Algorithm

```
Algorithm 2.11: Horspool
Input: text $T = T[0 \dots n)$, pattern $P = P[0 \dots m)$
Output: position of the first occurrence of $P$ in $T$
Preprocess:
  (1) for $c \in \Sigma$ do $shift[c] \leftarrow m$
  (2) for $i \leftarrow 0$ to $m-2$ do $shift[P[i]] \leftarrow m-1-i$
```

Horspool's Algorithm

Text: JIMY_HAILED_THE_LEADER_TO_STOP
Pattern: LEADER

```
JIMY_RAN_AND_HAILED_THE_LEADER_TO_STOP
    ||     | |     |     ||  |
LEADER     | |     |     ||  |
```

Horspool's Algorithm

Text: JIMY_HAILED_THE_LEADER_TO_STOP
Pattern: LEADER

```
JIMY_RAN_AND_HAILED_THE_LEADER_TO_STOP
    ||     | |     |     ||  |
LEADER     | |     |     ||  |
      LEADER |     |     ||  |
```

Horspool’s Algorithm

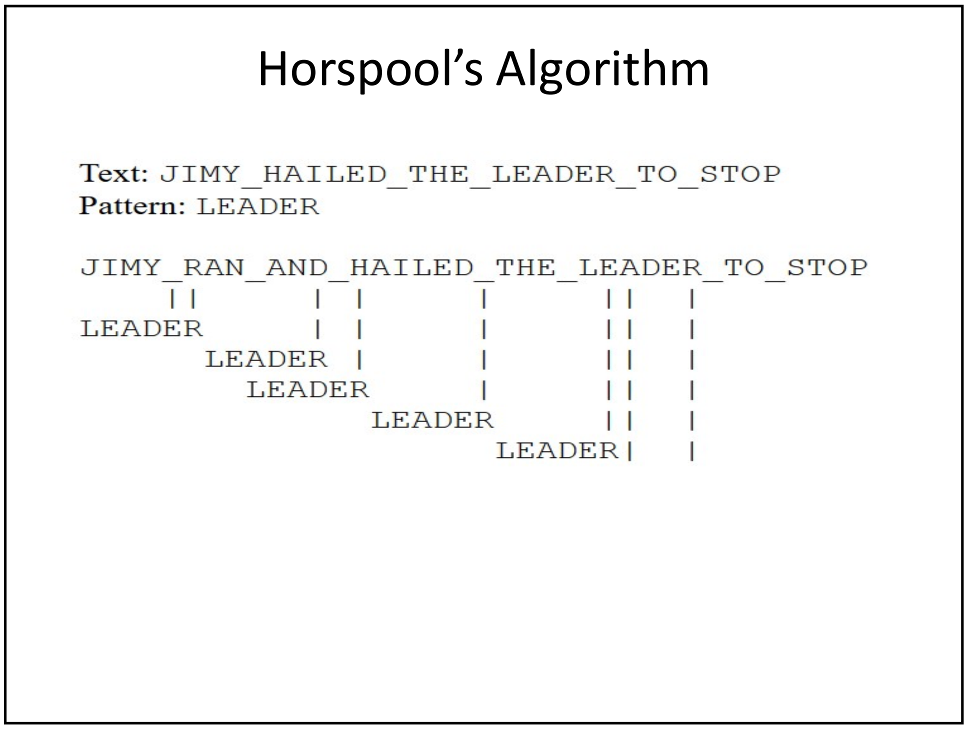
Text: JIMY_HAILED_THE_LEADER_TO_STOP
Pattern: LEADER

```
JIMY_RAN_AND_HAILED_THE_LEADER_TO_STOP
    ||     | |     |     ||  |
LEADER     | |     |     ||  |
      LEADER |     |     ||  |
        LEADER     |     ||  |
```

Horspool’s Algorithm

Text: JIMY_HAILED_THE_LEADER_TO_STOP
Pattern: LEADER

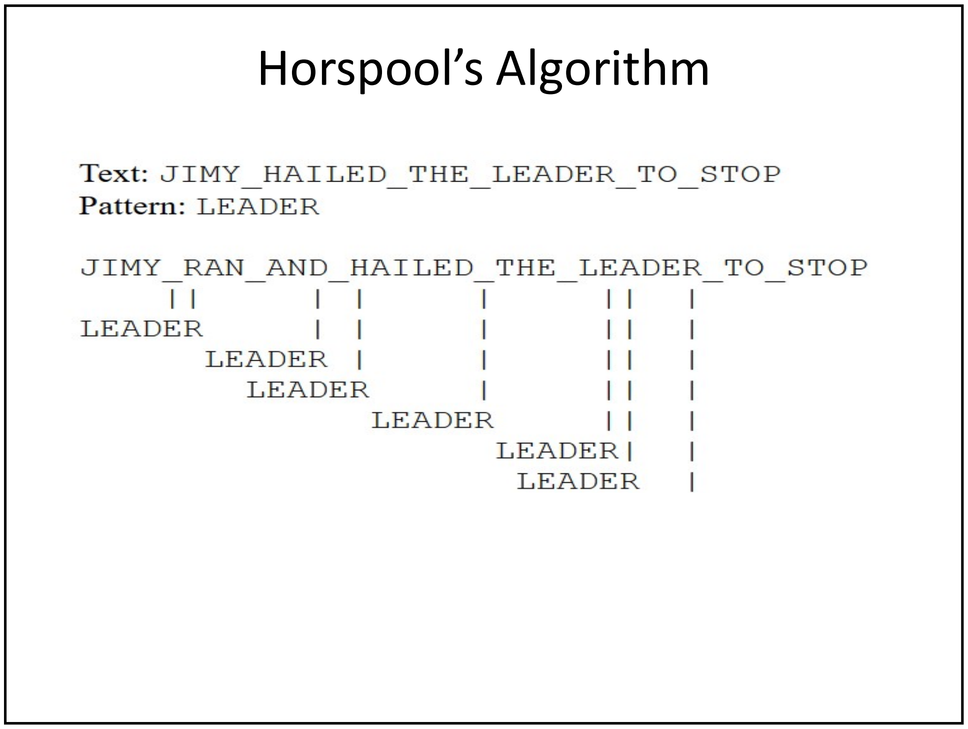
```
JIMY_RAN_AND_HAILED_THE_LEADER_TO_STOP
    ||     | |     |     ||  |
LEADER     | |     |     ||  |
      LEADER |     |     ||  |
        LEADER     |     ||  |
              LEADER     ||  |
```



Horspool's Algorithm

Text: JIMY_HAILED_THE_LEADER_TO_STOP
Pattern: LEADER

```
JIMY_RAN_AND_HAILED_THE_LEADER_TO_STOP
    ||     | |     |     ||  |
LEADER     | |     |     ||  |
      LEADER |     |     ||  |
        LEADER     |     ||  |
              LEADER     ||  |
                    LEADER|  |
```



Horspool's Algorithm

Text: JIMY_HAILED_THE_LEADER_TO_STOP
Pattern: LEADER

```
JIMY_RAN_AND_HAILED_THE_LEADER_TO_STOP
    ||     | |     |     ||  |
LEADER     | |     |     ||  |
      LEADER |     |     ||  |
        LEADER     |     ||  |
              LEADER     ||  |
                    LEADER|  |
                     LEADER  |
```

Horspool’s Algorithm

Text: JIMY_HAILED_THE_LEADER_TO_STOP
Pattern: LEADER

```
JIMY_RAN_AND_HAILED_THE_LEADER_TO_STOP
    ||     | |     |     ||  |
LEADER     | |     |     ||  |
      LEADER |     |     ||  |
        LEADER     |     ||  |
              LEADER     ||  |
                    LEADER|  |
                     LEADER  |
                        LEADER
```

The worst case cost is $\Theta(nm)$, but for random text is $\Theta(n)$.

Try Yourself

Text: **JIM_SAW_ME_IN_A_BARBER_SHOP**

Pattern: **BARBER**