Lecture 16: String Matching
CLRS- 32.1, 32.4

Outline of this Lecture

- String Matching Problem and Terminology.
- Brute Force Algorithm.
- The Knuth-Morris-Pratt (KMP) Algorithm.
- The Boyer-Moore (BM) Algorithm.
String Matching Problem and Terminology

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

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

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

\[
\begin{array}{ccccccccccc}
T & b & a & c & a & b & c & a & b & c & a & \ldots \\
\end{array}
\]

\[
\begin{array}{ccccccccccc}
P & c & a & b & c \\
\end{array}
\]
String Matching Problem and Terminology

A string $w$ is a prefix of $x$ if $x = wy$, for some string $y$.

Similarly, a string $w$ is a suffix of $x$ if $x = yw$, for some string $y$. 
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
    }
}
The Knuth-Morris-Pratt (KMP) Algorithm

In the Brute-Force algorithm, if a mismatch occurs at $P[j]$ ($j > 1$), it only slides $P$ to right by 1 step. It throws away one piece of information that we’ve already known. What is that piece of information?

Let $s$ be the current shift value. Since it is a mismatch at $P[j]$, we know $T[s+1..s+j-1] = P[1..j-1]$.

How can we make use of this information to make the next shift? In general, $P$ should slide by $s' > s$ such that $P[1..k] = T[s' + 1..s' + k]$. We then compare $P[k+1]$ with $T[s' + k + 1]$.
The Knuth-Morris-Pratt (KMP) Algorithm

When we slide $P$ to right, it should be a place where $P$ could possibly occur in $T$.

$$
\begin{array}{c}
\text{T} & b & a & c & b & a & b & a & b & a & b & a & b & c & b & a & b \\
\text{P} & a & b & a & b & a & c & a & 1 & q \\
\end{array}
$$

$$
\begin{array}{c}
\text{T} & b & a & c & b & a & b & a & b & a & a & b & c & b & a & b \\
\text{P} & a & b & a & b & a & c & a & 1 & k \\
\end{array}
$$

$$
\begin{array}{c}
1 & q \\
\text{P[1..q]} & a & b & a & b & a \\
\text{P[1..k]} & a & b & a \\
1 & k \\
\end{array}
$$

$P[1..k]$ is a suffix of $P[1..q]
Do not shift too much, as it may miss some matched patterns!

Do not shift too much, as it may miss some matched patterns!

It shifts too much! A matched pattern is missed.
The next function

We need to answer the following question: Given $P[1..q]$ match text characters $T[s + 1..s + q]$ , what is the least shift $s' > s$ such that

$$P[1..k] = T[s' + 1..s' + k] ,$$

where $s' + k = s + q$?

In practice, the shift $s'$ can be precomputed by comparing $P$ against itself. Observe that $T[s' + 1..s' + k]$ is a known text, and it is a suffix of $P[1..q]$ . To find the least shift $s' > s$, it is the same as finding the largest $k < q$, s.t.,

$P[1..k]$ is a suffix of $P[1..q]$ .
The *next function*

Given $P[1..m]$, let $\text{next}$ be a function $\{1, 2, \ldots, m\} \rightarrow \{0, 1, \ldots, m - 1\}$ such that

$$\text{next}(q) = \max\{k : k < q \text{ and } P[1..k] \text{ is a suffix of } P[1..q]\}.$$ 

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P[q]$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>next $(q)$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Given $\text{next}(q)$ for all $1 \leq q \leq m$, we can use the KMP algorithm.
The Knuth-Morris-Pratt (KMP) Algorithm

KMP_StringMatcher(T, P) {
    n = length(T); m = length(P);
    compute_Next(P);
    q = 0; // number of characters matched so far
    i=1;
    while (i<=n) {
        // loop until a match is found, or number of characters matched so far is 0;
        // note 'i' is unchanged.
        while (q > 0 and P[q+1] != T[i]) {
            q = next[q];
        }
        // matched character increased by 1
        if (P[q+1] == T[i]) q = q+1;
        if (q == m) {
            print "Pattern occurs with shift=", i-m
            q = next[q];
        }
        i++;
    }
}
The Knuth-Morris-Pratt (KMP) Algorithm

- **q**: 1 2 3 4 5 6 7
- **P[q]**: a b a b a b c
- **next(q)**: 0 0 1 2 3 4 0

```plaintext
i=3, q=0

i=8, q=5; p[q+1]!=T[i] (enters the while loop)
q assigns to next[5] (=3)

i=8, q=1; p[q+1]!=T[i] (exits the while loop)
q assigns to next[1] (=0)

i=8, q=0;
(exits the while loop)
```
How to compute $next$ function

Given $next[1], next[2], \ldots, next[q]$, how can we compute $next[q+1]$?

1. If $P[q+1]=P[next[q]+1]$, then $next[q+1]=next[q]+1$. 

![Diagram showing the next function computation](image)
2. If $P[q+1] \neq P[next[q]+1]$, then do what?

$P$ should slide to a place such that the prefix of $P[1..next[q]]$ occurs as a suffix of $P[q-next[q+1]..q]$; this information is stored in $next[next[q]]$!
How to compute next function

We first set \texttt{next[1]}=0, then compute \texttt{next[q]} with \( q = 2, 3, \ldots m \), one by one in \( m - 1 \) iterations.

```c
compute_Next(P) {
    m = length(P);
    next[1]=0; // initialization
    k = 0; // number of characters matched
            // so far
    q=2;
    while (q<=m) {
        while (k > 0 and P[k+1] != P[q]) {
            k = next[k];
        }
        if (P[k+1]==P[q]) k=k+1;
        next[q]=k;

        q++;
    }
}
```
Running Time of the KMP Algorithm

1. `compute_next`

   (a) \(3q-k = 6\) at the beginning, and \(3q-k \leq 3m\) at all times.

   (b) Note that after each comparison, \(3q-k\) increases *at least* by 1. But the value of \(3q-k\) starts at 6, and the largest possible value is \(3m\), it implies there are \(O(m)\) number of comparisons.

   (c) Hence, the running time of `compute_next` is \(O(m)\).
2. **KMP[StringMatcher]**

   (a) $3i - q = 3$ at the beginning, and $3i - q \leq 3n$ at all times.

   (b) Note that after each comparison, $3i - q$ increases *at least* by 1.

   (c) Hence, the running time of **KMP[StringMatcher]** is $O(n) + O(m) = O(m + n)$. 

---

**Running Time of the KMP Algorithm**
The Boyer-Moore (BM) Algorithm

The Boyer-Moore (BM) algorithm slides $P$ from left to right; however it compares $P$ and $T$ from right to left, i.e., $P[m]$ will first compare with $T[i]$. If they match, it then compares $P[m-1]$ with $T[i-1]$, etc. Else, it slides $P$ to right, and compare $P[m]$ with $T$ again.
One insight of BM algorithm is that, if there is a mismatch between \( P[j] \) and \( T[i] \), and \( T[i] \) does not appear in \( P \). \( P \) should be advanced by \( j \).

\[ T \ldots \_i\_n\_t\_h\_e\_w\_a\_t\_e\_r\_ \ldots \]
\[ P \quad \text{start comparison} \]

\[ T \ldots \_i\_n\_t\_h\_e\_w\_a\_t\_e\_r\_ \ldots \]
\[ P \quad s \rightarrow \text{device} \]

\[ T \ldots \_i\_n\_t\_h\_e\_w\_a\_t\_e\_r\_ \ldots \]
\[ P \quad s+5 \rightarrow \text{device} \]
The BM Algorithm: the bad-character heuristic

If $T[i]$ appears in $P$, shift $P$ such that $T[i]$ is aligned with the rightmost occurrence of $T[i]$ in $P$.

\begin{align*}
\text{T} & \quad \ldots \quad _{a} \quad _{w} \quad \boxed{i} \quad d \quad e \quad _{s} \quad h \quad o \quad t \quad _{...} \quad \ldots \\
\text{P} & \quad \overset{s}{\rightarrow} \quad \boxed{d} \quad e \quad \boxed{v} \quad i \quad \boxed{c} \quad e \\
\text{T} & \quad \ldots \quad _{a} \quad _{w} \quad \boxed{i} \quad d \quad e \quad _{s} \quad h \quad o \quad t \quad _{...} \quad \ldots \\
\text{P} & \quad \overset{s+4}{\rightarrow} \quad \boxed{d} \quad e \quad \boxed{v} \quad i \quad \boxed{c} \quad e
\end{align*}

start comparison
The BM Algorithm: the bad-character heuristic

If it happens the alignment of $T$ and $P$ gives a negative shift value, then just ignore it.

```
T ... _ G r e e c e e _ i s _ l o c ...  
```

```
P  d e v i c e
```

```
T ... _ G r e e c e e _ i s _ l o c ...  
```

negative shift

```
P  d e v i c e
```
The BM Algorithm: the good suffix heuristic

Similar to the KMP algorithm, if the current shift is \( s \), and it is a mismatch at \( P[j] \), then we know \( P[j + 1..m] = T[s + j + 1..s + m] \). Then we can shift \( P \) by \( s' \) such that \( T \) is aligned with the rightmost occurrence of \( P[j + 1..m] \).
The BM Algorithm

The BM Algorithm takes the *larger* shift amount computed by bad-character heuristic and good-suffix heuristic.

![Diagram of BM Algorithm](image)