Today’s Lecture

- String Matching
  - Notations and Terminology
  - Naive String Matching Algorithm
  - The Rabin-Karp Algorithm
  - Knuth-Morris-Pratt Algorithm
Problem Formulation

- String and Pattern Matching problems are fundamental to any application involving text processing.
- We are given a text \( T[1,\ldots,n] \) (\( n \) characters) and a pattern \( P[1,\ldots,m] \), the problem is to find all occurrences of \( P \) in \( T \).
- We say \( P \) occurs with shift \( s \) in \( T \) (or equivalently, \( P \) occurs at position \( s+1 \) in text \( T \)) if:
  \[
  0 \leq s \leq n - m \quad \text{and} \quad T[s+1\ldots s+m] = P[1\ldots m]
  \]

Notations

- We let \( A \) denote the alphabet and \( A^* \) denote the set of all finite-length strings using the characters in \( A \).
- \( \varepsilon \) will denote the empty string. \( |x| \) will denote the length of string \( x \). The concatenation of strings \( x \) and \( y \) will be denoted by \( xy \).
- We say \( w \) is a prefix (suffix) of string \( x \) if \( x=wy \) (\( x=yw \)) for some string \( y \) in \( A^* \).
- We will denote the \( k \)-character prefix \( P[1,\ldots,k] \) of \( P[1,\ldots,m] \) by \( P_k \). (what is \( P_0 \)?)
Simple Lemma

- Suppose $x$, $y$ and $z$ are strings and $x$ and $y$ are both suffixes of $z$. If $|x| \geq |y|$ then $x$ is a suffix of $y$; if $|y| \geq |x|$, then $y$ is a suffix of $x$, and if $|y| = |x|$ then $x = y$.

Naive Algorithm for String Matching

- The naive algorithm finds all valid shifts using a loop that checks if $P_m$ is suffix of $T_{s+m}$ for each of $n-m+1$ possible values of $s$.

For $s \leftarrow 0$ to $n-m$ do

- If $P[1,\ldots,m] = T[s+1,\ldots,s+m]$ Then
  - Write ("Match at position $s$")
<table>
<thead>
<tr>
<th>Running Time</th>
</tr>
</thead>
</table>

- The total running time: $O(m(n-m))$. For example, the worst case happens when we try to find $P=a^m$ (string of $m$ ‘a’) in $T=a^n$.
- We want to have an algorithm which has a linear running time in terms of $n$ and $m$, more precisely an $O(n+m)$ algorithm.

<table>
<thead>
<tr>
<th>Rabin-Karp Algorithm</th>
</tr>
</thead>
</table>

- The Rabin-Karp algorithm interprets the symbols of the alphabet, $A$, as numbers and considers the strings $T$ and $P$ as numbers to the base $d=|A|$.
- Let $p$ be the number corresponding to $P[1,...,m]$ and $t_s$ be the number corresponding to $T[s+1,...,s+m]$.
- Let us assume for simplicity $d=10$. Our first task is to compute the numbers $p$ and $t_s$. 
Horner’s Rule

- We can compute $p$ in time $O(m)$:
  \[ p = P[m] + 10(P[m-1] + 10(P[m-2] + \cdots + 10(P[2] + 10P[1]) \cdots) \]

- The value of $t_0$ can be similarly computed from $T[1, \ldots, m]$ in $O(m)$ time. We need to compute the values $t_1, t_2, \ldots, t_{n-m}$, in $O(n-m)$ time.

- For this we will use the following recursive relation:
  \[ t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1] \]

Algorithm

- Once we have computed the values for $p$ and $t_0, \ldots, t_{n-m}$, in time $O(n+m)$, we can compare them to find all the occurrences of $P[1, \ldots, m]$ in $T[1, \ldots, n]$.

- But this is only based on the assumption that all the numbers $(p, t_0, \ldots, t_{n-m})$ can be represented in a word, which might not be true!

- In stead of working with these numbers we compute their reminder mod a prime $q$, which will fit in a word.
Example

\[ 2 \ 3 \ 5 \ 9 \ 0 \ 2 \ 3 \ 1 \ 4 \ 1 \ 5 \ 2 \ 6 \ldots \]

\[ 31415 \mod 13 \]

\[ 7 \]

\[ 3 \ 1 \ 4 \ 1 \ 5 \ 2 \]

\[ 7 \ 8 \]

\[ 14152 = (41415 - 3 \times 10000) \times 10 + 2 \pmod{13} \]

\[ (7 - 3 \times 3) \times 10 + 2 \pmod{13} \]

\[ 8 \pmod{13} \]

Spurious Hit

- It may be the case that \( t_s = p \pmod{q} \) but \( T[s+1, \ldots, s+m] \) is not the equal to \( P[l, \ldots, m] \). This is called a spurious hit. In this case we will need to explicitly compare \( P[l, \ldots, m] \) and \( T[s+1, \ldots, s+m] \), which takes \( O(m) \) time. Hence the worst case running time of this algorithm is \( O((n-m)m) \).
The expected Running Time

- In practice, if there are few matches, we will expect big savings (from this algorithm compared to naïve algorithm).
- If we assume that reducing \( \text{Mod } q \) is like a random mapping from \( \mathbb{A}^m \) to \( \mathbb{Z}_p \) (set of numbers from \( 0, \ldots, q-1 \)), i.e. \( \{0, \ldots, q-1\} \), then the expected number of spurious hits is \( O((n-m)/q) \).
- If we denote number of matches between \( P \) and \( T \) by \( v \), then the over all running time is \( O(n) + O(m(v+(n-m)/q)) \).

Knuth-Morris-Pratt (KMP) Algorithm

- This algorithm runs in time liner time in terms of sum of the lengths, \( n+m \).
- The key to this running time is the use of an auxiliary function, called prefix function, which computed based on string \( P \).
- The idea is to avoid many backtrackings which occur in previous two algorithms. Instead of moving the pattern one position at a time, we will use the partial information obtained in previous trials to slid the pattern \( P \) more than one position to right, without bypassing potential matches.
Example

Suppose we are matching $P$ with $T[6,\ldots,16]$:
- There is a mismatch between $P[11]$ and $T[16]$.
- We can slide our match to $T[13]$, why?

Because!

- We know the matching stopped at position $P[10]$, i.e. $T[5,\ldots,15]$ has been matched to $P[1,\ldots,10]$. Let us look at string $P[1,\ldots,10]$.

The longest prefix of this string that appears also as suffix in it is “ABA”, so the last three position of $P_{10}$ are the same as its first three position.

Since the last three position of $P$ are the same as $T[13,\ldots,15]$, by transitivity first three characters of $P$ are the same as $T[13,\ldots,15]$. 


Example


In General

- The amount of shift to which we slide $P$ to depends only on structure of $P$ and how much of $P$ has been matched to $T$.
- Assume that $q$ symbols of $P$ has been already matched to $T[s+1,...,s+q]$, and there is a mismatch between $T[s+q+1]$ and $P[q+1]$, the question is how much can we slid in our algorithm.
Assume that $q$ symbols of $P$ has are already matched, to answer our previous question we need to know the longest proper suffix of $P[1,\ldots,q]$ which is a prefix of $P[1,\ldots,q]$.

We call this quantity $\pi$ and define it as:

$p(q) = \max\{k : k < q \text{ and } P[1,\ldots,k] = P[q-k+1,\ldots,q]\}$

Example

<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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>p</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
KMP Algorithm

KMP - Matcher(T, P)

q ← 0;  i ← 0;
Repeat /* Assume P[1…q] = T[i-q…i-1]*/
{ If p[q+1] = T[i] Then {
    q ← q + 1;  i ← i + 1;
    If q = m Then /* Write "match at:":*/
        q ← p(q)
}
Else /* Mismatch */
{ If q = 0 Then  i ← i + 1
Else  q ← p(q)
}
} Until (i = n + 1)

Running Time of KMP

- The running time of this algorithm is \( O(n) \) assuming that the repeat loop will run in \( O(1) \).
- At each iteration of the algorithm either the value of \( i \) increases by one or the \( P \) slides to the right, but either of this can happen at most \( n \) times.
- Our major assumption is that the value of \( p(q) \) is computed off-line and is present when the algorithm is executing its main loop. Having an \( O(m) \) time algorithm to compute \( p(q) \) for all values of \( q \) between 1 and \( n \), will give us the desired running time of \( O(n+m) \).
Computing $\pi$

- We know $p[1]$ is always zero.
- Suppose we have computed $p[1]$ to $p[i-1]$ and we want to compute $p[i]$:  
  - We know that $P[1,\ldots,p[i-1]]$ is the longest proper prefix that is also a suffix of $P[1,\ldots,i-1]$.
  - Let $q = p[i-1]$. If $P[i] = P[q+1]$, then we know $p[i] = q + 1$

```
Computing $\pi$

- If $P[i]$ is not equal to $P[q+1]$:  
  - We know that next longest prefix of $P[1\ldots i-1]$ which is also a suffix of it is $P[1,\ldots,p[q]]$ (why?).  
  - In this case we compare the value of $P[p[q]+1]$ with $P[i]$ and if they are equal then we will set $p[i] = p[q] + 1$
  - Otherwise, we will repeat this process.
```
Example

<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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>p</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>

Suppose we want to compute p[10]
- We are given p[1] to p[9]. We set q = p[9].
- Since P[10] is not equal to P[5], we will set q = p[q], that is q = p[4] = 2.

Algorithm

p - Function(P)

\[
\begin{aligned}
p & \leftarrow 0; \quad q \leftarrow 0; \quad i \leftarrow 0; \\
\text{Repeat} & \quad \text{/* Assume p[1…i – 1] is known */} \\
\{ & \text{If } p[q + 1] = P[i] \text{ Then } \\
& \quad q \leftarrow q + 1; \quad i \leftarrow i + 1; \quad p[i] \leftarrow q; \}
\text{Else} & \quad \text{/* Mismatch */} \\
\{ & \text{If } q = 0 \text{ Then} \\
& \quad p[i] \leftarrow 0; i \leftarrow i + 1; \}
\text{Else } & q \leftarrow p(q) \\
\} & \text{Until } (i = m + 1)
\end{aligned}
\]
Running Time

- The algorithm is very similar to KMP algorithm in the sense that in each iteration either the $i$ will be increased by one or the string will slide to the right, and this can happen at most $O(m)$ times.
- The main loop of the algorithm will take $O(1)$ time on each iteration, which implies the running time of algorithm is $O(m)$. 